VisIt User Manual Documentation

Release 3.2.2

LLNL

Nov 10, 2023

CONTENTS

1 Getting Started 1

2 Getting Help 13

3 Introduction to VisIt 15

4 Using VisIt 21

5 Python Scripting 591

6 VisIt Tutorials 841

7 Java Client 1067

8 Getting Data into VisIt 1101

9 Building VisIt from Sources 1239

10 VisIt Developer Manual 1283

11 Search Syntax 1391

12 Acknowledgments 1393

13 Glossary 1395

Index 1397

CHAPTER

ONE

GETTING STARTED

1.1 Installing and Starting VisIt

Pre-built binaries for VisIt are provided on the following platforms:

- Redhat, TOSS, Ubuntu (fully supported)
- Centos, Debian, Fedora (partially supported)
- macOS
- Microsoft Windows

For an explanation of what *fully* and *partially* supported mean, see our section on *supported platforms*.

New versions are usually released every 2-3 months. Users can find releases at the VisIt releases page. See our section on *managing GitHub notifications* to get notified of new releases.

Download a binary release compatible with the machine on which you want to run VisIt. If you are installing VisIt on Linux, you will also need to download and use the `visit-install` script.

Installing VisIt on platforms other than those listed here requires *building VisIt from sources*.

1.1.1 Installing on macOS

VisIt releases include an app-bundle for macOS packaged in a DMG image. Download and open the DMG file and copy the VisIt app-bundle to your applications directory or any other path.

Note: If you have a previous version of VisIt already installed, you may be prompted by macOS to decide if you want to **Keep both** versions or **Replace** the old version with the new version. If you choose **Keep both**, macOS will automatically adjust the name of the new version to something like `VisIt 2`. The space will cause problems and it will have to be removed by changing the name to something without spaces like `VisIt2` or `VisIt-2` or `VisIt-x.y.z` where `x.y.z` is the version number of the installation. If you do not have the necessary privileges to change the name, a system administrator's help may be needed.

To run VisIt double click on the VisIt app-bundle. Alternatively, VisIt can be run on macOS from the `Terminal` using a command of the form:

```
/Applications/VisIt.app/Contents/Resources/bin/visit
```

Note: Running from the `Terminal` may work around issues accessing some folders on local storage or code signing and notarization.

The `visit-install` script can also be used to install tarball packaged macOS binaries. For this case follow the Linux installation instructions.

1.1.2 Installing on Linux

Installing VisIt on Linux (and optionally on macOS) is done using the `visit-install` script. Make sure that the `visit-install` script is executable by entering the following command at the command line prompt:

```
chmod +x visit-install
```

The visit-install script has the following usage:

```
./visit-install version platform directory
```

The **version** argument is the version of VisIt being installed. The **platform** argument depends on the type platform VisIt is being installed for. The platform argument can be one of the following: linux, linux-x86_64, darwin. The **directory** argument specifies the directory to install VisIt into. If the specified directory does not exist then VisIt will create it.

For example, to install an x86_64 version of VisIt 3.2.2, use:

```
./visit-install 3.2.2 linux-x86_64 /usr/local/visit
```

This command will install the 3.2.2 version of VisIt into the `/usr/local/visit` directory. Note that when you enter the above command, the file `visit3_2_2.linux-x86_64.tar.gz` must be present in the current working directory.

The `visit-install` script will prompt you to choose a network configuration. A network configuration is a set of VisIt preferences that provide information to enable VisIt to identify and connect to remote computers and run VisIt in client/server mode. VisIt includes network configuration files for several computing centers with VisIt users.

After running `visit-install`, you can launch VisIt using `bin/visit`. For example, if you installed to `/usr/local/visit`, you can run using:

```
/usr/local/visit/bin/visit
```

We also recommend adding `visit` to your shell's path. For bash users this can usually be accomplished by modifying the `PATH` environment variable in `~/.bash_profile`, and for c-shell users accomplished by modifying the `path` environment variable in `~/.cshrc`.

The exact procedure for this varies with each shell and may be customized at each computing center, so please refer to your shell and computing center documentation.

1.1.3 Installing on Windows

VisIt release binaries for Windows are packaged in an executable installer. To install on Windows run the installer and follow its prompts.

The VisIt installation program adds a VisIt program group to the Windows Start menu and it adds a VisIt shortcut to the desktop. You can double-click on the desktop shortcut or use the Start menu's VisIt program group to launch VisIt. In addition to creating shortcuts, the VisIt installation program creates file associations for `.silo`, `.visit`, and `.session/.vses` files so double-clicking on files with those extensions opens them with VisIt.

You can also run the installer from a command prompt, and pass it certain parameters to override defaults including running in silent mode. Available options are as follows:

```
-ALLUSERS              Install for all users.
                       Must be in admin-mode.
                       Default : install for current user

-SITE <site-name>      Specifies host profiles to be installed (eg llnl).
                       Default: None

-PB <parallel bank>    Specifies parallel bank (FOR LLNL host-profiles).
                       Default: wbronze

-DB <database reader> Specifies a default database reader for VisIt.
                       (eg Silo, FLASH, etc).
                       Default: None

-DEV                   Install plugin development tools.
                       Default: no plugin dev tools

-LIBSIM                Install libsim tools.
                       Default: no libsim tools

-AssociatePython       Associate python files with VisIt.
                       Default: don't set up association

-AssociateCurves       Associate curve files with VisIt.
                       Default: don't set up association

/S                     Make install silent.

/D <installationdir>   Change install directory to <installationdir>.
                       Default is %PROGRAM FILES% for ALL USERS and
                       %HOMEPATH% for single user.
                       MUST BE THE LAST PARAMETER!
```

1.1.4 Installing ffmpeg

`ffmpeg` is a high quality MPEG 4 encoder. The VisIt movie wizard uses `ffmpeg` if it is found in the user's search path. `ffmpeg`'s licensing is incompatible with VisIt's so we do not ship and install `ffmpeg` with VisIt. You can install `ffmpeg` as part of a VisIt installation so that it is available for all user's.

To install `ffmpeg` as part of a VisIt installation you would do the following steps.

1) Get the `ffmpeg` executable for each platform of interest.
2) Copy the `ffmpeg` executable for each platform to the architecture specific bin directory.
3) Set the group and file permissions appropriately for each executable.
4) Create a soft link from `ffmepg` to `frontendlauncher` in the bin directory.

Here we go through an example where we install `ffmpeg` into VisIt 3.3.3, which has two architectures (`linux-intel` and `linux-x86_64`) installed. The `ffmpeg` executables are named `ffmpeg.intel` and `ffmpeg.x86_64`. We will set the group to `visit` and the file permissions to `775`.

```
cp ffmpeg.intel visit/3.3.3/linux-intel/bin/ffmpeg
chgrp visit visit/3.3.3/linux-intel/bin/ffmpeg
chmod 775 visit/3.3.3/linux-intel/bin/ffmpeg
cp ffmpeg.x86_64 visit/3.3.3/linux-x86_64/bin/ffmpeg
```

(continues on next page)

(continued from previous page)

```
chgrp visit visit/3.3.3/linux-x86_64/bin/ffmpeg
chmod 775 visit/3.3.3/linux-x86_64/bin/ffmpeg
ln -s frontendlaucher visit/bin/ffmpeg
```

1.1.5 Startup Options

VisIt has many startup options that affect its behavior (see the *Startup Options* for complete documentation).

1.2 Startup Options

You can get help on starting VisIt with the commands

```
visit -help
visit -fullhelp
```

For convenience, the output from `visit -fullhelp` is shown below.

USAGE: visit [options]:

```
Interface options
------------------------------------------------------------------------------
    -gui                 Run with the Graphical User Interface (default).
    -cli                 Run with the Command Line Interface.

Movie making options
------------------------------------------------------------------------------
    -movie               Run the CLI in a movie making mode. Must be
                         combined with -sessionfile. Will produce a simple
                         movie by drawing all the plots in the specified
                         session for every timestep of the database.

Startup options
------------------------------------------------------------------------------
    -o <filename>        Open the specified data file at startup.
    -s <filename>        Run the specified VisIt script.
    -sessionfile <filename>  Open the specified session file at startup
                         Note that this argument only takes effect with
                         -gui or -movie.
    -config <filename>   Initialize the viewer at startup using the named
                         config file.  If an absolute path is not given,
                         the file is assumed to be in the .visit directory.
    -noconfig            Don't process configuration files at startup.
    -launchengine <host> Launch an engine at startup. The <host> parameter
                         is optional. If it is not specified, the engine
                         will be launched on the local host. If you wish
                         to launch an engine on a remote host, specify
                         the host's name as the <host> parameter.
    -nosplash            Do not display the splash screen at startup.

Window options
------------------------------------------------------------------------------
    -small               Use a smaller desktop area/window size.
    -geometry <spec>     What portion of the screen to use.  This is a
```

(continues on next page)

(continued from previous page)

```
                          standard X Windows geometry specification. This
                          option can be used to set the size of images
                          generated from scripts and movies.

    -viewer_geometry <spec> What portion of the screen the viewer windows
                            will use. This is a standard X Windows geometry
                            specification. This option overrides the
                            -geometry option that the GUI passes to the
                            viewer.

    -window_anchor <x,y> The x,y position on the screen where VisIt's GUI
                          will show its windows (Main window excluded).
    -style <style>       One of: windows,cde,motif,sgi.
    -locale <locale>     The locale that you want VisIt to use when displaying
                          translated menus and controls. VisIt will use the
                          default locale if the -locale option is not
                          provided.
    -background <color>  Background color for GUI.
    -foreground <color>  Foreground color for GUI.
    -nowin               Run with viewer windows off-screen (i.e. OSMesa).
                          This is typically used with the -cli option.
    -stereo              Enable active stereo, also known as the
                          page-flipping, or 'CrystalEyes' mode.
    -nowindowmetrics     Prevents X11 from grabbing and moving a test
                          widget used in calculating window borders. This
                          option can be useful if VisIt hangs when
                          displaying to an Apple X-server.

Version options
------------------------------------------------------------------------------
    -version             Do NOT run VisIt. Just print the current version.
    -git_version         Do NOT run VisIt. Just print the Git version it
                          was built from.
    -beta                Run the current beta version.
    -v <version>         Run a specified version. Specifying 2 digits,
                          such as X.Y, will run the latest patch release
                          for that version. Specifying 3 digits, such as
                          X.Y.Z, will run that specific version.

Other resources for help
------------------------------------------------------------------------------
    run-time:            While running VisIt, look under the "Help" menu.
    on-line:             https://visit-help.llnl.gov


******************************************************************************
                           ADDITIONAL OPTIONS
******************************************************************************

Parallel launch options
------------------------------------------------------------------------------
    Notes: All of these options are ordinarily obtained from host profiles.
    However, the command line options override anything in the profiles.

    When parallel arguments are added but the engine is not the
    component being launched, -launchengine is implied.  Explicitly
    add -launchengine to launch a remote parallel engine.
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------------------------
    -setupenv             Use the VisIt script to set up the environment
                          for the engine on the compute nodes.
    -par                  Run the parallel version. This option is implied
                          by any of the other parallel options listed below.
    -l    <method>        Launch in parallel using the given method.
    -pl   <method>        Launch only the engine in parallel as specified.
    -la   <args>          Additional arguments for the parallel launcher.
    -sla  <args>          Additional arguments for the parallel sub-launcher.
    -np   <# procs>       The number of processors to use.
    -nn   <# nodes>       The number of nodes to allocate.
    -p    <part>          Partition to run in.
    -n    <name>          The parallel job name.
    -b    <bank>          Bank from which to draw resources.
    -t    <time>          Maximum job run time.
    -machinefile <file>   Machine file.
    -expedite             Makes DPCS give priority scheduling.

    -icet                 In scalable rendering mode, use the IceT parallel
                          image compositor (default).
    -no-icet              Do not use the IceT parallel compositor.

Hardware accelerated parallel (scalable) rendering options
------------------------------------------------------------------------------
    Notes: These options should only be used with parallel clusters that
    have graphics cards.  If you are using a serial version of VisIt, you
    are already getting hardware acceleration and these options are not
    needed.  Furthermore, you must be in scalable rendering mode for VisIt
    to utilize a cluster's GPUs.  By default, VisIt is configured to
    switch into scalable rendering mode when rendering complexity exceeds
    a predefined limit.

    VisIt can manage the creation and tear down of X servers for you.  It
    will do this automatically if you specify the -launch-x parameter,
    but you can customize the process with the -x-args and -display
    parameters, which respect %l and %n format specifiers.

------------------------------------------------------------------------------
    -hw-accel             Tells VisIt that it should use graphics cards.
    -n-gpus-per-node <int> Number of GPUs per node of the cluster (1).
    -launch-x             Tell VisIt to manage the X servers
    -no-launch-x          Let the cluster manager X servers [default]
    -display              Tells VisIt which display to use.
    -x-args '<string>'    Extra arguments to X server.

Load balance options
------------------------------------------------------------------------------
    Note: Each time VisIt executes a pipeline the relevant domains for the
    execution are assigned to processors. This list of domains is sorted in
    increasing global domain number. The options below effect how domains
    in this list are assigned to processors. Assuming there are D domains
    and P processors...
------------------------------------------------------------------------------
    -lb-block             Assign the first D/P domains to processor 0, the
                          next D/P domains to processor 1, etc.
    -lb-stride            Assign every Pth domain starting from the first
                          to processor 0, every Pth domain starting from the
```

(continues on next page)

(continued from previous page)

```
                        second to processor 1, etc.
    -lb-absolute        Assign domains by absolute domain number % P. This
                        guarantees a given domain is always processed
                        by the same processor but can also lead to poor
                        balance when only a subset of domains is selected.
    -lb-random          Randomly assign domains to processors.
    -allowdynamic       Dedicate one processor to spreading the work
                        dynamically among the other processors.  This mode
                        has limitations in the types of queries it can
                        perform.  Under development.
    -lb-stream          Similar to -lb-block, but have the domains travel
                        down the pipeline one at a time, instead of all
                        together.  Under development.

Database differencing options
-------------------------------------------------------------------------------
    Use the '-diff <ldb> <rdb>' option to run VisIt in a database
    differencing mode. VisIt will generate expressions to facilitate
    visualization and analysis of the difference between the left-database,
    <ldb>, and right-database, <rdb>. VisIt will open windows to display
    both the left and right databases as well as their difference.

    VisIt uses the Cross-Mesh Field Evaluation (CMFE) expression functions
    to help generate these differences. A CMFE function creates an instance
    of a variable from another (source) mesh on the specified (destination)
    mesh. VisIt can use two variants of CMFE expression functions depending
    on how similar the source and destination meshes are; connectivity-based
    (conn_cmfe) which assumes the underlying mesh(s) for the left and right
    databases have identical connectivity and position-based (pos_cmfe) which
    does not make this assumption. VisIt will attempt to automatically select
    which variant of CMFE expression to use based on some simple heuristics.
    For meshes with identical connectivity, conn_cmfe expressions are
    preferrable because they are higher performance and do not require VisIt
    to perform any interpolation. In fact, the conn_cmfe operation is
    perfectly anti-symmetric. That is <ldb> - <rdb> = -(<rdb> - <ldb>).
    The same cannot be said for pos_cmfe expressions. However, pos_cmfe
    expressions will attempt to generate useful results regardless of the
    similarity of the underlying meshes.

    Note that the differences VisIt will compute in this mode are single
    precision. This is true regardless of whether the input data is itself
    double precision. VisIt will convert double precision to single
    precision before processing it. Although this is a result of earlier
    visualization-specific design requirements and constraints, the intention
    is that eventually double precision will be supported.

    Finally, be sure to bring up Controls->Macros in the GUI to find a set
    of useful operations specifically tailored to database differencing. Also,
    typing 'help()' (including the '()') at the python prompt after starting
    'visit -diff' will generate a more detailed help message.

    -diff <ldb> <rdb>   Indicate you wish to run VisIt in database
                        differencing mode and specify the two databases
                        to difference.

                        Note: All options occurring on the command-line
                        *after* the '-diff' option are treated as options
```

(continues on next page)

(continued from previous page)

```
                          to the differencing script while all options
                          occurring *before* the '-diff' option are treated
                          as options to VisIt.

    -diffsum <ldb> <rdb>  Run only the difference summary method of the
                          'visit -diff' script, in nowin mode so its fast,
                          print the results, and immediately exit.

    -force_pos_cmfe       Force use of position-based CMFE expressions.

Advanced options
------------------------------------------------------------------------------
    -guesshost            Try to guess the client host name from one of
                          the SSH_CLIENT, SSH2_CLIENT, or SSH_CONNECTION
                          environment variables.
    -noloopback           Disable use of the 127.0.0.1 loopback device.
    -sshtunneling         Tunnel all remote connections through ssh.  NOTE:
                          this overrides values set in the host profiles.
    -noint                Disable interruption capability.
    -nopty                Run without PTYs.
    -verbose              Prints status information during pipeline
                          execution.
    -dir <directory>      Run a version of VisIt in the specified directory.
                          The directory argument should specify the
                          path to a VisIt installation directory.
                          /bin is automatically appended to this path.
    -forceversion <ver>   Force the given version.  Overrides all
                          intelligent version selection logic.
    -publicpluginsonly    Disable all plugins but the default ones.
    -compiler <cc>        Require version built with the specified compiler.
    -objectmode <mode>    Require a specific object file mode.
    -forceinteractivecli Force the CLI to behave interactively, even if run
                          with no terminal; similar to python's '-i' flag.
    -fullscreen           Create the viewer window in full screen mode.
                          May not be compatible with all window managers.
    -viewerdisplay <dpy>  Have the viewer use a different display than the
                          current value of DISPLAY.  Can be useful for
                          power wall displays with a separate console.
    -cycleregex <string>  A regex-style regular expression to be used
                          in extracting cycle numbers from file names. It
                          is best to bracket this string in single
                          quotes (') to avoid shell interpretation of
                          special characters such as star (*). The format
                          of the string begins with an opening '<' character,
                          followed by the regular expression itself followed
                          by a closing '>' character, optionally followed by
                          a space ' ' character and sub-expression reference
                          to indicate which part of the regular expression is
                          the cycle number.  Default behavior is as if
                          -cycleregex '<([0-9]+)[^0-9]*\$> \\0'
                          was specified meaning the last sequence of one
                          or more digits before the end of the string found
                          is used as the cycle number. Do a 'man 7 regex'
                          to get more information on regular expression
                          syntax.
    -ui-bcast-thresholds <int1> <int2>
                          Two integers controlling behavior of parallel
```

(continues on next page)

(continued from previous page)

```
                        engine waiting in a broadcast for the next RPC
                        from the viewer. VisIt used to rely solely upon
                        MPI_Bcast for this. However, many implementations
                        of MPI_Bcast use a polling loop that winds up
                        keeping all processors busy and can make them
                        unuseable by other processes. This is particularly
                        bad for SMPs. So, VisIt implemented its own
                        broadcast using MPI's send/receive methods. <int1>
                        specifies the number of nanoseconds a processor
                        sleeps while polling for completion of the
                        broadcast. Specifying a value of zero (0) for <int1>
                        results in falling back to older behavior using
                        MPI's MPI_Bcast. <int1> effectively controls how
                        'busy' processors will be, polling for completion
                        of the broadcast. <int2> specifies the number of
                        seconds all processors should spin, polling as fast
                        as possible, checking for completion of the
                        broadcast BEFORE inserting sleeps into their
                        polling loops. <int2> effectively controls how
                        many seconds VisIt's server will be maximally
                        responsive (although also keeping all processors
                        occupied) before becoming more 'friendly' to
                        other processes on the same node. The defaults
                        are <int1> = 50000000 nanoseconds (1/20th of a sec)
                        and <int2> = 5 seconds meaning VisIt will spin
                        processors maximally for 5 seconds before inserting
                        sleeps such that polling happens at the rate of 20
                        times per second.
    -idle-timeout <int>  An integer representing the number of minutes an
                        engine is allowed to idle (e.g. sit there doing no
                        execution whatsoever, waiting for commands from
                        the viewer). If this timeout is reached, the engine
                        will terminate itself. The default is 480 minutes
                        (8 hours).
    -exec-timeout <int>  An integer representing the number of minutes an
                        executing engine is allowed to remain in the
                        execution of any single command from the viewer.
                        If this timeout is reached, the engine will
                        terminate itself. the default is 30 minutes.
                        Beware that among other things, this timeout
                        effects how long orphaned parallel processes will
                        hang around, tying up parallel compute resources,
                        following an exit-triggering error condition on
                        any one process.

Developer options (most for xml2... tools)
------------------------------------------------------------------------------
    -public             xml2cmake: force install plugins publicly
    -private            xml2cmake: force install plugins privately
    -clobber            Permit xml2... tools to overwrite old files
    -noprint            Silence debugging output from xml2... tools
    -outputtoinputdir   Force xml2... tools to write output files to
                        the directory containing the input XML file
    -arch               print supported architecture(s) and exit

Debugging options
------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
    Note: Debugging options may degrade performance
-------------------------------------------------------------------------------
    -debug <level>          Run with <level> levels of output logging.
                            <level> must be between 1 and 5. This will generate
                            debug logs (called 'vlogs' for ALL components.
                            Note that debug logs are unbuffered. However, if
                            you also specify 'b' immediately after the digit
                            indicating the debug level (e.g. '-debug 3b'), the
                            logs will be buffered. This can substantially improve
                            performance when a lot of debug output is generated.
                            However, also beware that when debug logs are buffered,
                            there isn't necessarily any guarantee they will contain
                            the most recent debug output just prior to a crash.
    -debug_<compname> <level>
                            Run specified component with <level> of output
                            logging. For example, '-debug_mdserver 4' will run
                            the mdserver with level 4 debugging. Multiple
                            '-debug_<compname> <level>' args are allowed.
    -debug_engine_rank <r>
                            Restrict debug output to the specified rank.
    -debug-processor-stride N
                            Have only every Nth processor output debug logs.
                            Prevents overwhelming parallel file systems.
    -clobber_vlogs          By default, VisIt maintains debug logs from the 5
                            most recent invocations or restarts of each VisIt
                            component. They are named something like
                            A.mdserver.5.vlog, A.engine_ser.5.vlog, etc with
                            the leading letter (A-E) indicating most to least
                            recent. The clobber_vlogs flag causes VisIt to remove
                            all debug logs and begin creating them anew.
    -vtk-debug              Turn on debugging of VTK objects used in pipelines.
    -pid                    Append process ids to the names of log files.
    -timing                 Save timing data to files.
    -withhold-timing-output
                            Withhold timing output during execution. Prevents
                            output of timing information from affecting
                            performance.
    -never-output-timings
                            Never output timings files.  This is used when
                            you want the timer to be enabled (for usage by
                            developers to measure inner loops), but you
                            want to avoid blowing memory with the bookkeeping
                            for each and every timing call.
    -timing-processor-stride N
                            Have only every Nth processor output timing info.
                            Prevents overwhelming parallel file systems.
    -env                    Print env. variables VisIt will use when run.
    -dump (dump_dir)        Dump intermediate results from AVT filters,
                            scalably rendered images, and html pages.
                            Takes an optional argument that specifies the
                            directory for -dump output files.
    -info-dump (dump_dir)
                            Dump html pages only.
                            Takes an optional argument that specifies the
                            directory for -info-dump output files.
    -gdb <args> <comp>      Run gdb with <args> on component <comp>.
                            Default <args> is whitespace.
```

(continues on next page)

(continued from previous page)

```
    -break <funcname>     Add the specified breakpoint in gdb.
    -xterm                With -gdb-something, run gdb in an xterm window.
    -newconsole           Run any VisIt component in a new console window.
    -totalview <args> <comp>
                          Run totalview with <args> on component <comp>.
                          Default <args> is whitespace.
    -valgrind <args> <comp>
                          Run valgrind with <args> on component <comp>.
                          Default <args> is --tool=memcheck --error-limit=no
                          --num-callers=50.
    -strace <args> <comp>
                          Run strace with <args> on component <comp>.
                          Default <args> is -ttt -T.

                          In the above, all arguments between the tool name
                          and the VisIt component name are treated as args
                          to the tool.

    -apitrace <args> <comp>
                          Run apitrace with <args> on component <comp>.
                          Default <args> is trace --api gl.

                          In the above, all arguments between the tool name
                          and the VisIt component name are treated as args
                          to the tool.

    -debug-malloc <args> <comp>
                          Run the component with the libMallocDebug library
                          on MacOS X systems. The libMallocDebug library
                          lets the MallocDebug application attach to the
                          instrumented application and retrieve memory
                          allocation statistics. The -debug-malloc flag
                          also sets up the environment for the leaks and
                          heap tools.

                          Printing heap allocations:
                          % visit -debug-malloc gui &
                          % Get the gui's <pid>
                          % heap <pid>

                          Printing memory leaks:
                          % visit -debug-malloc gui &
                          % Get the gui's <pid>
                          % leaks <pid>

                          Run with MallocDebug:
                          Perl does not seem to be happy with libMallocDebug
                          so you can run the GUI like this:
                          % visit -cli
                          >>> OpenGUI('-debug-malloc', 'MallocDebug', 'gui')
                          Connect to the gui with MallocDebug and do your
                          sampling.
    -nsys <args> <comp>
                          Run Nsight Systems cli with <args> on component <comp>.
                          Default <args> are: profile --stats=true --gpu-metrics-
↪device=all
```

(continues on next page)

(continued from previous page)

```
    -numrestarts <#>      Number of attempts to restart a failed engine.
    -quiet                Don't print the Running message.
    -protocol             Print the definitions of the state objects that
                          comprise the VisIt protocol so they can be compared
                          against the values on other computers.
```

CHAPTER

TWO

GETTING HELP

2.1 Methods of Contact

Methods of contact for help differ in *requirements* and *privacy* level. In addition, while we try to accommodate a variety of means of *initial contact*, anything requiring *ongoing dialog* must use only a *preferred* method of communication. Please read about our *communication policies* for more details.

Method	Privacy	Requirements	Preferred
GitHub Discussion	World discoverable / readable	Free GitHub account	Yes
MS Teams	Visible only in llnl.gov	Access to LLNL networks	No
Hotline Telephone Call	Same as ordinary telphone call	Access to LLNL networks	No

In general, **coverage** is during normal West Coast business hours, 8am-12pm and 1-5pm (GMT-8, San Francisco time zone), Monday through Friday excluding LLNL holidays. **Response time** may be as much as four hours due to team member's multi-tasking among many responsibilities.

A GitHub Discussion is the preferred method of contact because it is likely to be seen by more people who can provide a timely response and because any response we produce can be viewed by the widest possible audience.

If you are so inclined, we welcome you to read more about the VisIt project's *Site Reliability Engineering* processes to understand how we provide support.

2.2 Supported Platforms

Note: Supported platforms periodically change due to any of a number of factors outside the control of the VisIt project including demand, developer access, technology obsolescence, and majority stakeholder requirements.

The platforms upon which the majority of contributors regularly develop and run VisIt and for which the resources exist to provide *full support* are determined primarily by our host organization. For these platforms, we are able to reproduce bugs, test bug-fixes, develop work-arounds and in general provide the highest quality support services. These platforms are…

- Redhat, TOSS, Ubuntu
- Windows 7, 8 and 10
- macOS 10.15

We do make an effort to provide pre-compiled binaries as well as perform minimal testing for Centos, Debian, Fedora. However, because we do not yet routinely develop or test on these platforms, we are not able to provide the same level of support as we do for Redhat, TOSS, Ubuntu.

Finally, for other platforms, *building* VisIt from sources is an option. We do try to be responsive to issues building VisIt from sources.

2.3 Supported Versions

A VisIt version number is composed of three digits, `A.B.C`, where `A` is the major version, `B` is the minor version and `C` is the patch version. Patch releases are made approximately 2-4 times a year and are designed to be compatible. Minor releases are made approximately 1-2 times a year and are not compatible with any prior versions. Major releases are made infrequently. Major releases are not compatible with any prior versions and might also not be compatible with older operating systems, older compilers, and/or older primary dependencies (e.g. VTK or GL).

Tip: Use VisIt's GitHub milestones page or reach out to the VisIt team on our GitHub discussions page if you need help planning for upcoming releases of VisIt.

Only in very rare circumstances does the VisIt team have the resources to update a previously released version of VisIt. This does occasionally happen but it is very rare. Instead, we ask that users please try to keep up to date with the most current minor release of VisIt.

Tip: Wherever possible, its best if users can keep a version or two *behind* the current minor release around as a fall back in case the current release introduces any show-stopper behavior.

What this means is that the only *supported* version of VisIt is the *current* minor version which can be identified by browsing our releases page. When users report issues which are reproducible *only* in versions of VisIt older than the *current* minor version, the team may be able to suggest work-arounds but will otherwise ask users to please upgrade to the current version. On the other hand, when issues reported in older versions are reproducible in the current version as well, the VisIt team will try to provide additional support.

2.4 Managing GitHub Notifications

The VisIt project is maintained on GitHub. Users needing support or even to just be notified of new releases need a (free) GitHub account to create and comment on discussions and issues and to watch for releases.

You will receive notifications from GitHub for...

- ...any activity *you initiate* (issue, discussion, pull request, etc.)
- ...any activity *types* you choose to *watch*.
- ...when someone @mentions you.

In addition, GitHub provides controls for the types of activities for which you want to receive notifications. For example, users can configure their watch settings on the main VisIt repository to be notified only when new releases are made.

Every email generated by GitHub includes an `unsubscribe` link at the bottom to turn off notifications for the particular activity (issue, discussion, pull request, etc.) the email is associated with.

Be aware that if you unsubscribe from activity *you initiate* (e.g. submitting an issue), you may miss any follow-up discussion and/or resolution. We will close, possibly without resolution, any issues or discussions awaiting follow-up for more than 21 days.

CHAPTER

THREE

INTRODUCTION TO VISIT

VisIt is a free, open source, platform independent, distributed, parallel, visualization tool for visualizing data defined on two- and three-dimensional structured and unstructured meshes. VisIt's distributed architecture allows it to leverage both the compute power of a large parallel computer and the graphics acceleration hardware of a local workstation. VisIt's user interface is often run locally on a Windows, Linux, or macOS desktop computer while its compute engine component runs in parallel on a remote computer. VisIt's distributed architecture allows VisIt to visualize simulation data where it was generated, eliminating the need to move the data to a visualization server. VisIt can be controlled by its Graphical User Interface (GUI), through the Python and Java programming languages, or from a custom user interface that you develop yourself. More information about VisIt can be found online at https://visit.llnl.gov/.

This manual explains how to use the VisIt GUI. You will be given a brief overview on how VisIt works and then you will be shown how to start and use VisIt.

3.1 Understanding how VisIt works

3.1.1 VisIt's Core Abstractions

VisIt's interface is built around five core abstractions. These include:

- Databases
- Plots
- Operators
- Expressions
- Queries

Databases

Databases read data from files and presents the data in the user interface as variables. VisIt supports many different types of variables including:

- Meshes
- Scalars
- Vectors
- Tensors
- Materials
- Species

Meshes are the foundation of all the other types of variables. They consist of a discretization of space into cells. All the other variables are defined on the cells of the mesh.

Scalars are single valued fields and examples include density, pressure and temperature. Vectors are multi valued fields that have a direction and magnitude. Examples include velocity and magnetic fields. Tensors are multi valued fields that are typically thought of as 2 x 2 matrices in the case of 2D data and 3 x 3 matrices in the case of 3D data. The typical tensor variable is the stress tensor. Materials are a special type of variable that associates one or more materials with a cell. The location of the material is not specified within the cell and in the case of multi material cells, algorithms must be used to determine where the material is located in the cell, typically by looking at the materials in neighboring cells. Species are variables that are associated with each material. For a given material, species are a further breakdown of a material. The distinctive property of a species is that it is uniformly distributed throughout the material. For example, air consists of many different gases such as oxygen, nitrogen, carbon monoxide, carbon dioxide, etc.

Plots

Plots take variables and generate a visual representation of the variable. Some examples include the Mesh plot, which displays the mesh lines of the mesh, the Pseudocolor plot, which maps scalar variables to color, and the Vector plot, which displays vector glyphs indicating the direction and magnitude of a vector field. Plots work on specific types of variables and the graphical user interface limits the display of variables that can be used with a given plot to the appropriate variables.

Operators

Operators take variables and modify them in some way. Operators perform their operations before they are plotted. Multiple operators may be applied to a variable forming a pipeline. For example, a mesh may be subsetted so that all the values fall within a given range, furthermore, the mesh may be subsetted to a portion of the mesh within a user specified box.

Expressions

Expressions perform calculations on variables to generate new variables. Some common expressions consist of the standard mathematical operations such as addition, subtraction, multiplication and division. It also includes more complex operations such as gradient and divergence.

Queries

Queries summarize data and typically take variables as input and generate either a single value or some small number of values. Queries can also create curves, the most common of which is the result of a query over time that creates a curve of a scalar value over time. Some examples of queries include minimum, maximum, spatial extents and volume.

3.1.2 VisIt's Architecture

VisIt has a client-server architecture that consists of one or more clients that connect to a viewer, which connects to one or more parallel servers. The clients and viewer typically run locally on the users desktop system while the parallel servers run on some remote high performance compute platform. This is shown in Figure 3.1. This is the most general case, but the components can also all run on a single system, either on the desktop or on a remote high performance compute platform. The server can also run in serial and for small data sets is completely sufficient.

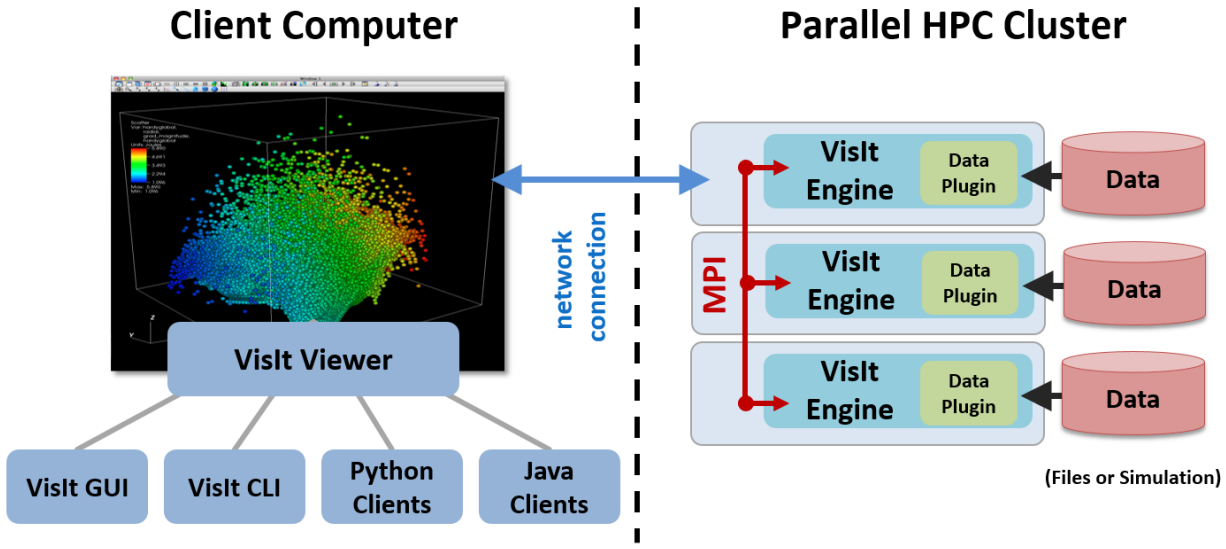


Fig. 3.1: VisIt's architecture

VisIt supports a number of different clients including a Graphical User Interface (GUI), a Python based Command Line Interface (CLI), and a Java programming interface. More than one client can be active at a time and VisIt coordinates the state between them so that they are consistent.

The viewer is responsible for displaying the visual results of the plots and coordinating the state information between the various clients.

The server is responsible for reading the data from disk and performing all the manipulations on the data. The server reads and does all of its processing in parallel when running in parallel. The server can either render the data to be displayed in parallel or send the data to be rendered by the viewer. For small data sets, rendering in the viewer is faster and has less latency. For large data sets it is better to render the data in parallel (using scalable rendering) and then send the rendered image to the viewer for display. The implementation of scalable rendering is shown in Figure 3.2. VisIt is by default configured to automatically switch between shipping data to the viewer and performing scalable rendering based on the amount of geometry to be rendered.

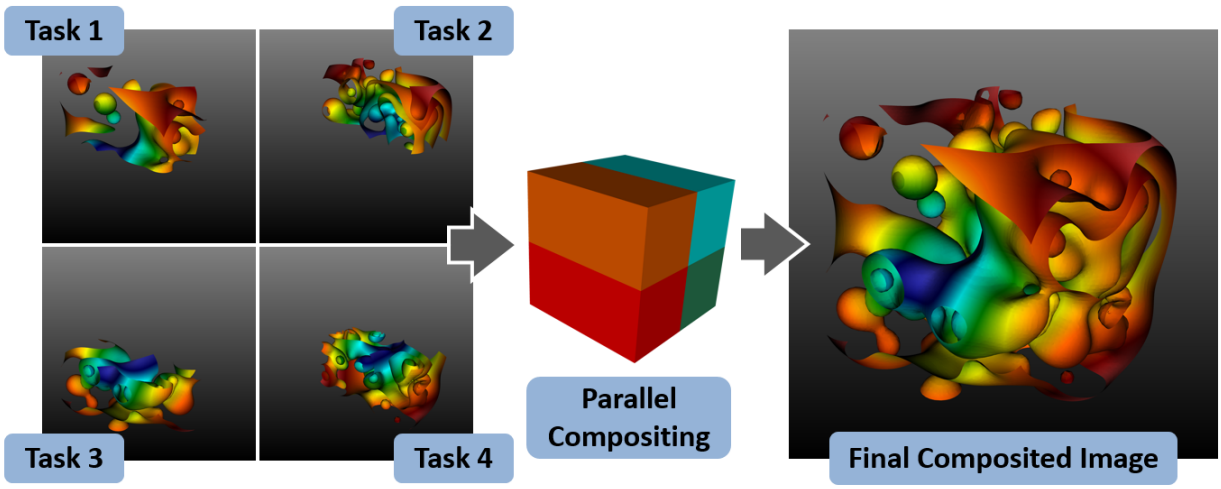


Fig. 3.2: VisIt's scalable rendering

3.1.3 VisIt's Graphical User Interface

When you run the VisIt graphical user interface, you are seeing windows from the Qt based GUI and the viewer. The GUI is a VisIt client that provides the user interface and menus that let you choose what to visualize. The viewer displays all of the visualizations and is responsible for keeping track of VisIt's state and coordinating this state with the other components. Both the GUI and the viewer are meant to run locally to take advantage of the local computer's graphics hardware. The next two components can also be run on a client computer but they are more often run on a remote, parallel computer or cluster where the data files are generated.

The viewer supports up to 16 visualization windows. Each window is independent of the others. VisIt uses an active window concept; all changes made in **Main** window or one of its popup windows apply to the currently active visualization window. The **Main** window and visualization window are shown in Figure 3.3.

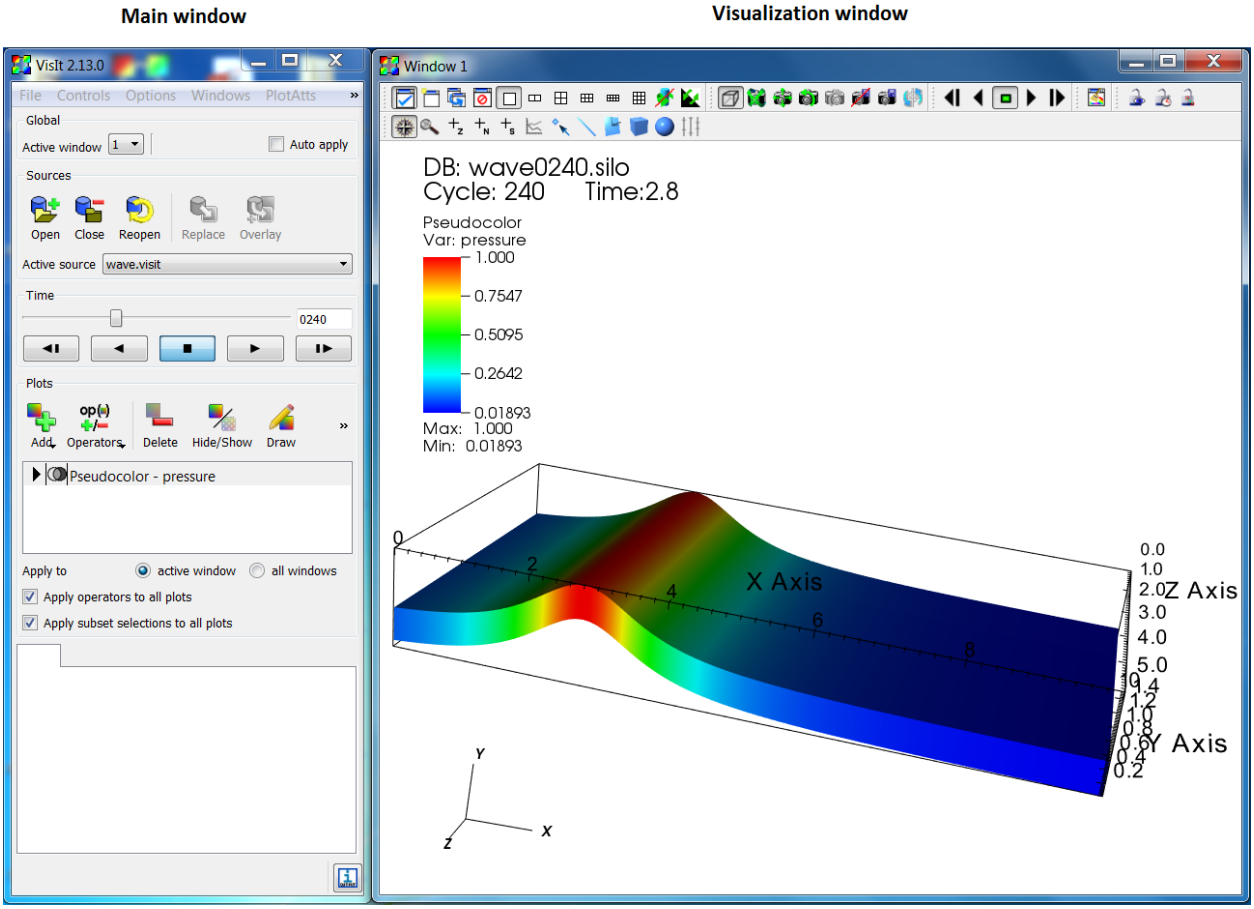


Fig. 3.3: VisIt's graphical user interface

Servers are launched on each machine where data to be visualized is located. Servers are launched on demand, typically when a database is opened. If there is more than one host profile on a system, VisIt will pop up a window asking which profile to use and additional properties such as the number of processors and nodes to use. The **Host Profiles** window is used to specify properties about the servers for different machines, such as the number of processors to use by default when running the server. The status of a compute engine is displayed in the **Compute Engines** window.

3.2 Getting Started

The rest of this manual details the ins and outs to using VisIt, but you can also very quickly visualize your data by opening a database and creating plots. You must first select databases to visualize. Sample data files are usually installed with VisIt in a data directory in the directory in which VisIt was installed. If you are running VisIt on the Windows platform, you can double-click on one of the sample Silo data files to open it in VisIt or you can run VisIt and open the **File open** window from the **Main** window's **File** menu. Using the **File open** window, navigate to the appropriate directory, highlight a file, and click the **Ok** button. If the database was successfully opened, the **Add** menu will be enabled.

Once you have opened a database, you can use it to create a plot by selecting a plot type and database variable from the **Add** menu. Once a plot is created, the **Active plot** list will show that the new plot has been added by displaying a description of the plot drawn in green text. The color green indicates that the plot is in the new state and has not been drawn yet. To draw the plot, click the **Draw** button in the middle of the **Main** window. That's all there is to creating a plot using VisIt. For more detailed information on creating plots and performing specific actions in VisIt, refer to the other chapters in this book.

CHAPTER

FOUR

USING VISIT

Contents:

4.1 The Main Window

VisIt's **Main** window, shown in Figure 4.1, contains three main areas: the file area, the plot area and the notepad area. The file area contains controls for working with sources and selecting the current time state. The plot area contains controls for creating and modifying plots and operators. The notepad area is a region where frequently used windows may be posted for quick and convenient access.

4.1.1 Posting a window

Each time a window posts to the notepad area, a new tab is created in the notepad and the posted window's contents are added to the new tab. Clicking on a tab in the notebook displays a posted window so that it can be used.

Postable windows have a **Post** button to post the window. Clicking on the **Post** button hides the window and adds its controls to a new tab in the notepad area. Posting windows allows you to have several windows active at the same time without cluttering the screen. When a window is posted, its **Post** button turns to an **UnPost** button that, when clicked, removes the posted window from the **Notepad** area and displays the window in its own window. Figure 4.2 shows an example of a window with a **Post** button and also shows the same window when it is posted to the notepad area.

4.1.2 Using the main menu

VisIt's **Main** menu contains seven menu options that allow you to access many of VisIt's most useful features. Each menu option displays a submenu when you click it. The options in the submenus perform an action such as saving an image. Menu options that contain a name followed by ellipsis open another VisIt window. Some menu options have keyboard shortcuts that activate windows. The **File** menu contains options that deal with files and simulations. The **Controls** menu contains options that open VisIt windows that, for the most part, set the look and feel of VisIt's visualization windows. The **Options** menu contains options that allow you to set the appearance of the GUI, manage host profiles, manage VisIt plugins, set various preferences and save VisIt's settings to a configuration file. The **Windows** menu contains controls that manage visualization windows. The **PlotAtts** and **OpAtts** menus allow access for setting the attributes of all the plots and operators. The **Help** menu provides options for viewing online help, VisIt's copyright agreement, and release notes which describe the major enhancements and fixes in each new version of VisIt. The options for each menu except for the plot and operator attribute menus are shown in Figure 4.3 and will be described in detail later in this manual.

The **Main** menu and the **Plots** and **Operators** menus are merged in the macOS version of VisIt because macOS applications always have all menus in the system menu along the top of the display.

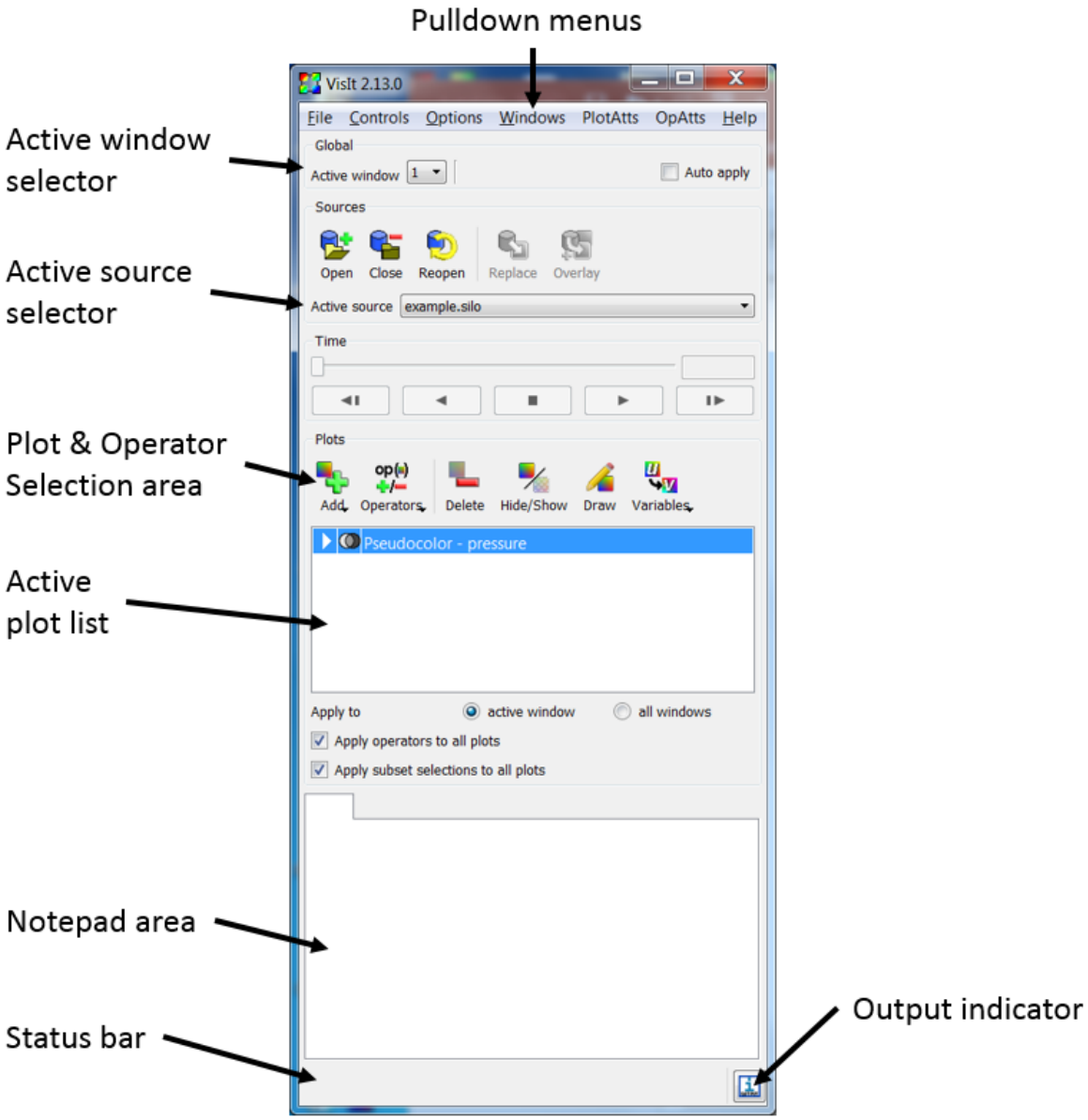


Fig. 4.1: VisIt's Main window

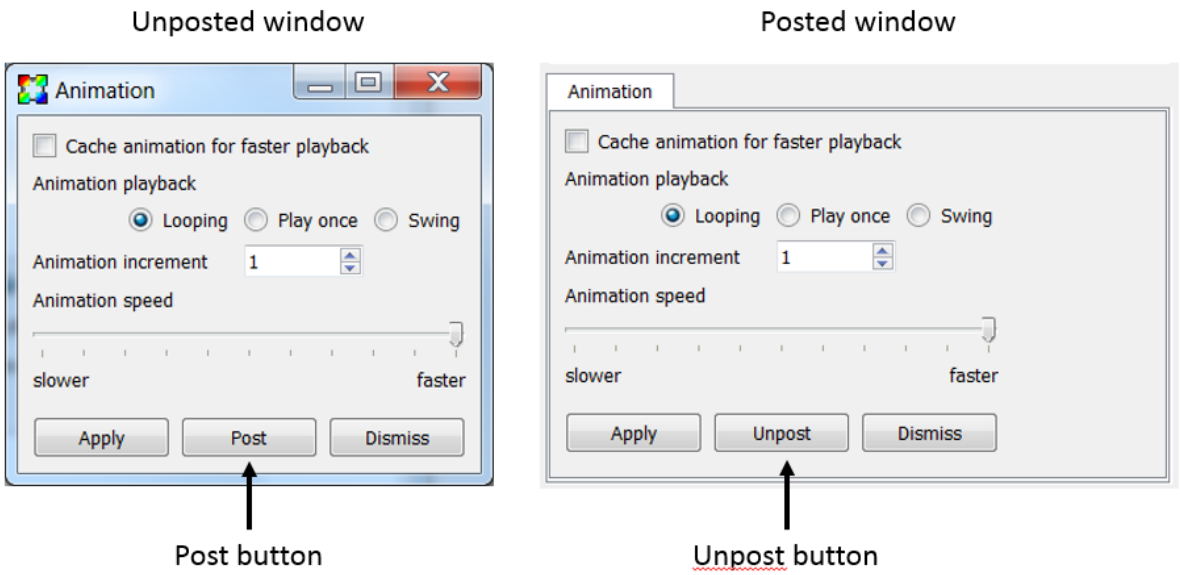


Fig. 4.2: An unposted and posted window

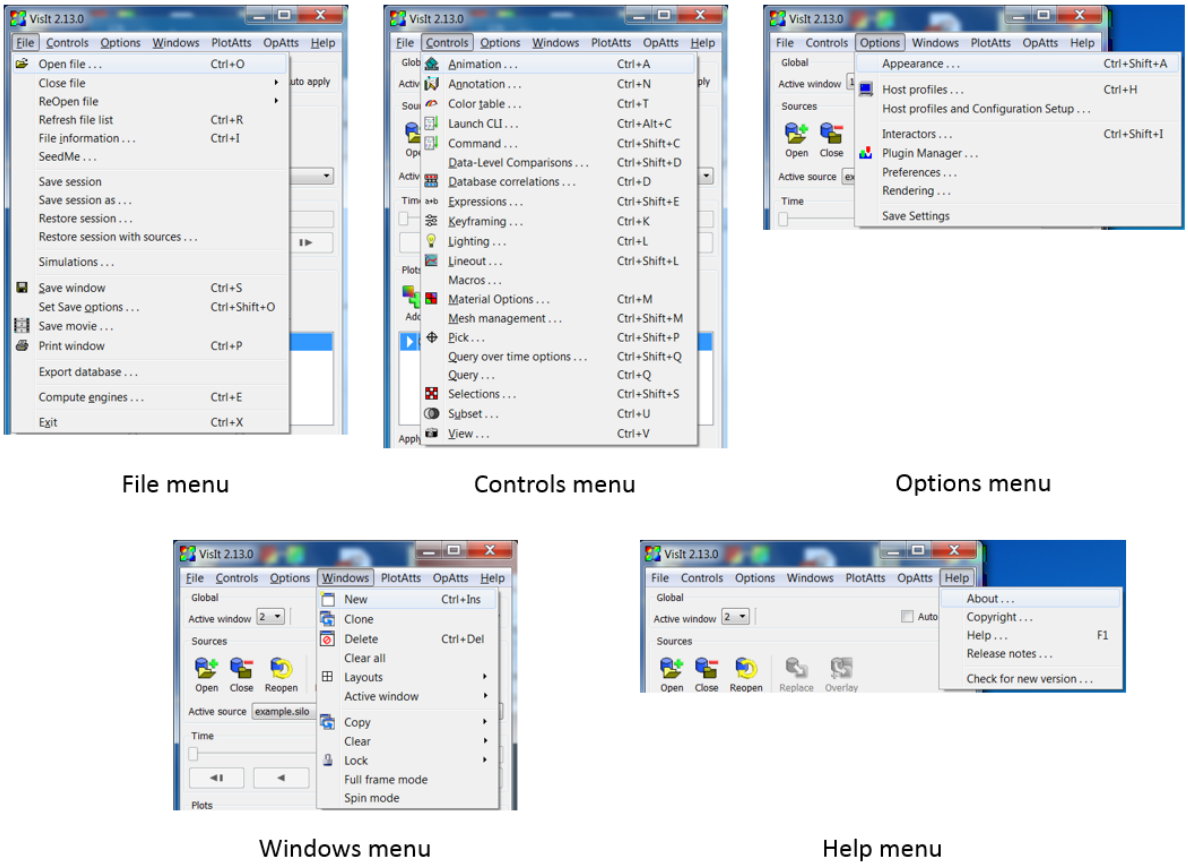


Fig. 4.3: VisIt's main menus

4.1.3 Viewing status messages

VisIt informs the user of its progress as it creates a visualization. As work is completed, status messages are displayed in the bottom of the **Main** window in the status bar. In addition to status messages, VisIt sometimes displays error or warning messages. These messages are displayed in the **Output** window, shown in Figure 4.4. To open the **Output** window, click the **Output** indicator in the lower, right hand corner of the **Main** window. When the **Output** window contains an unread message, the **Output** indicator changes colors from blue to red.

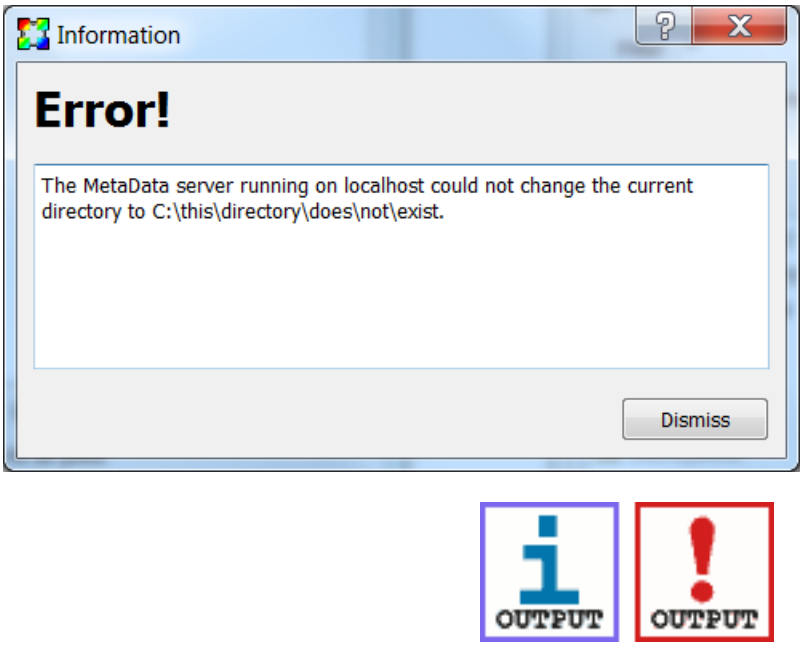
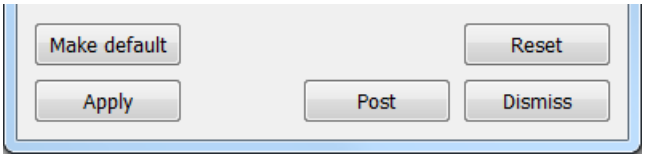


Fig. 4.4: The output window and output indicator

4.1.4 Applying settings

When using one of VisIt's control windows, you must click the **Apply** button for the new settings to take effect. All control windows have an **Apply** button in the lower left corner of the window. By default, new settings are not applied until the **Apply** button is clicked because it is more efficient to make several changes and then apply them at once. VisIt has a mode called **Auto apply** that makes all changes in settings take place immediately. **Auto apply** is not enabled by default because it can cause plots to be regenerated each time settings change and for the database sizes for which VisIt is designed, auto apply may not always make sense. If you prefer to have new settings apply immediately, you can enable auto apply by clicking on the **Auto apply** check box in the upper, right hand corner of the **Main** window. If **Auto apply** is enabled, you do not have to click the **Apply** button to apply changes.



4.2 Working with Databases

In this chapter, we will discuss how to work with databases in VisIt. A database can be either a set of files on disk or a running simulation. You can manage both types of databases using the same VisIt windows. First we'll learn about

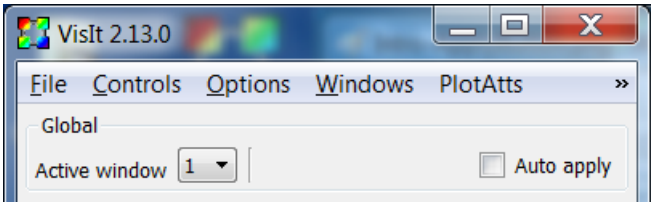


Fig. 4.5: The Apply button and Auto apply check box

Supported File Types, then the *File Open Window* which allows you to browse the local system or a remote host to find your files. Next, we'll learn how to open databases for visualization using the *Sources Pane*. After that we'll learn how to control animation in the *Time Pane* before learning how to examine information about a database using the *File Information Window*.

4.2.1 Supported File Types

VisIt can create visualizations from databases that are stored in many types of underlying file formats. VisIt has a database reader for each supported file format and the database reader is a plugin that reads the data from the input file and imports it into VisIt. If your data format is not listed in File formats supported by VisIt then you can first translate your data into a format that VisIt can read (e.g. Silo, VTK, etc.) or you can create a new database reader plugin for VisIt. For more information on developing a database reader plugin, refer to the Getting Data Into VisIt manual or contact us via Getting help.

File extensions

VisIt uses file extension matching to decide which database reader plugin should be used to open a particular file. Each database reader plugin has a set of file extensions that are used to match a filename to it. When a file's extension matches (case sensitive except on MS Windows) that of a certain plugin, VisIt attempts to load the file with that plugin. If the plugin cannot load the file, then VisIt attempts to open the file with the next plugin that matches the extension.

If you have a file with a common extension like `.hdf5` or `.h5`, there can be *many* VisIt plugins that match those extensions. VisIt will use the *first* plugin it tries that appears to successfully open the file. Sometimes, the first plugin that can read the file isn't the one you really wanted. In that case, your options are to explicitly select the plugin or to add it to the list of *preferred* plugins.

To explicitly select the plugin, use **File –> Open file...** and select the plugin you want from the **Open file as type** pull down list. To add a plugin to the list of *Preferred Database Plugins*, go to **Options –> Plugin Manager...** and then the **Databases** tab. Select the plugin from the list on the left and then hit the **Add to preferred list** button. Be sure to go to **Options –> Save settings...** if you want your selections to persist across VisIt sessions. If VisIt finds it is unable to open a file either because there are no plugins matching the extension or all the matching plugins failed to open the file, it will begin trying preferred plugins in the order from *top* to *bottom* of the list.

VisIt also supports the `-fallback_format` command-line option. This option adds the specified plugin to the list of preferred plugins. For example, `-fallback_format VTK` adds the VTK plugin to the list of preferred plugins. More than one `-fallback_format` option can be specified on the command-line and earlier encountered options take precedence over later ones.

Finally, you can also specify the plugin to use to open a file as part of the `-o` command-line option. For example, to open the file `foobar.gorfo` as a Silo file, you can specify `-o foobar.gorfo,Silo_1.0` on the command-line when starting VisIt. Note this feature of the `-o` option *requires* the plugin name (in correct case) followed by an underscore (_) and then its version number which is almost always `1.0`. If you want to see the plugin name options as well as their version numbers, go to **Options –> Plugin Manager...** and then the **Databases** tab. Adding plugins to the list of *Preferred Database Plugins* will display their names and version numbers.

Example Data Files

As part of VisIt's regular testing, a number of example data files VisIt reads can be found in VisIt's data subdirectory of the main code repository. In particular, if you are looking for examples of various of the human readable ASCII formats VisIt reads so that you can produce a compatible file, you may find examples there that help.

More Details of ASCII Formats

Here we describe more details specific to some of the ASCII formats VisIt reads.

Creating `.visit` Files

To create a `.visit` file, simply make a new text file that contains the names of the files that you want to visualize and save the file with a `.visit` extension.

- Visit will take the first entry in the `.visit` file and attempt to determine the appropriate plugin to read the file.
- Not all plugins can be used with `.visit` files. In general, **MD** or **MT** formats sometimes do not work.
  - An **MT** file is a file format that provides multiple time steps in a single file. Thus, grouping multiple **MT** files to produce a time series may not be supported.
  - An **MD** file is one that provides multiple domains in a single file. Thus, grouping multiple **MD** files to produce a view of the whole may not be supported.

Here is an example `.visit` file that groups time steps together. These files should contain 1 time step per file.

```
timestep0.silo
timestep1.silo
timestep2.silo
timestep3.silo
...
```

Here is an example `.visit` file that groups various smaller domain files into a whole dataset that VisIt can visualize. Note the use of the `!NBLOCKS` directive and how it designates the number of files in a time step that constitute the whole domain. The `!NBLOCKS` directive must be on the first line of the file. In this example, we have 2 time steps each composed of 4 domain files.

```
!NBLOCKS 4
timestep0_domain0.silo
timestep0_domain1.silo
timestep0_domain2.silo
timestep0_domain3.silo
timestep1_domain0.silo
timestep1_domain1.silo
timestep1_domain2.silo
timestep1_domain3.silo
...
```

You may also explicitly indicate the *time* associated with a file (or group of block files) using the `!TIME` directive like so...

```
!NBLOCKS 4
!TIME 1.01
timestep0_domain0.silo
timestep0_domain1.silo
```

(continues on next page)

(continued from previous page)

```
timestep0_domain2.silo
timestep0_domain3.silo
!TIME 2.02
timestep1_domain0.silo
timestep1_domain1.silo
timestep1_domain2.silo
timestep1_domain3.silo
...
```

Point3D Files

Point3D files are four or fewer columns of ASCII values with some header text to indicate the variable names associated with each column and a `coordflag` entry to indicate how to interpret the columns of data as coordinates. Point3D files can be used to define discrete points in 1, 2 and 3 dimensions having a single scalar value associated with each point. Some examples are below. The Point3D file…

```
x y z value
0 0 0 0
0 0 1 1
0 1 0 2
0 1 1 3
1 0 0 4
1 0 1 5
1 1 0 6
1 1 1 7
```

Defines a collection of 8 points in 3 dimensions have a scalar variable named *value*. Below, the `#coordflag` directive is used to define the same collection of 8 points in 3 dimensions as the previous example except where the columns holding the z-coordinate and the scalar variable are interleaved.

```
 x y value z
 #coordflag xyvz
 0 0 0 0
 0 0 1 1
 0 1 2 0
 0 1 3 1
 1 0 4 0
 1 0 5 1
 1 1 6 0
 1 1 7 1
```

In the example below, the `#coordflag` directive is used to define a collection of points in *two dimensions* where each point has a velocity magnitude value associated with it.

```
x y velocity
#coordflag xyv
0 0 1
0 1 1.01
1 0 2.02
```

Likewise, for a collection of points in just *one dimension*, we would have

```
x y velocity
#coordflag xv
```

(continues on next page)

(continued from previous page)

```
0 1
1 1.01
2 2.02
```

There are some additional examples of Point3D files on the VisIt wiki pages.

4.2.2 File Open Window

The **File Open Window** allows you to select files and simulations by browsing file system either on your local computer or the remote computer of your choice. You can open the **File Open Window** by choosing the **Open** option from the **Sources** section of the main GUI panel (shown in Figure 4.6), or by Choosing the **Open File** option from the **File** dropdown menu. When the window opens, its current directory is set to the current working directory or a directory from VisIt's preferences. See Figure 4.7.

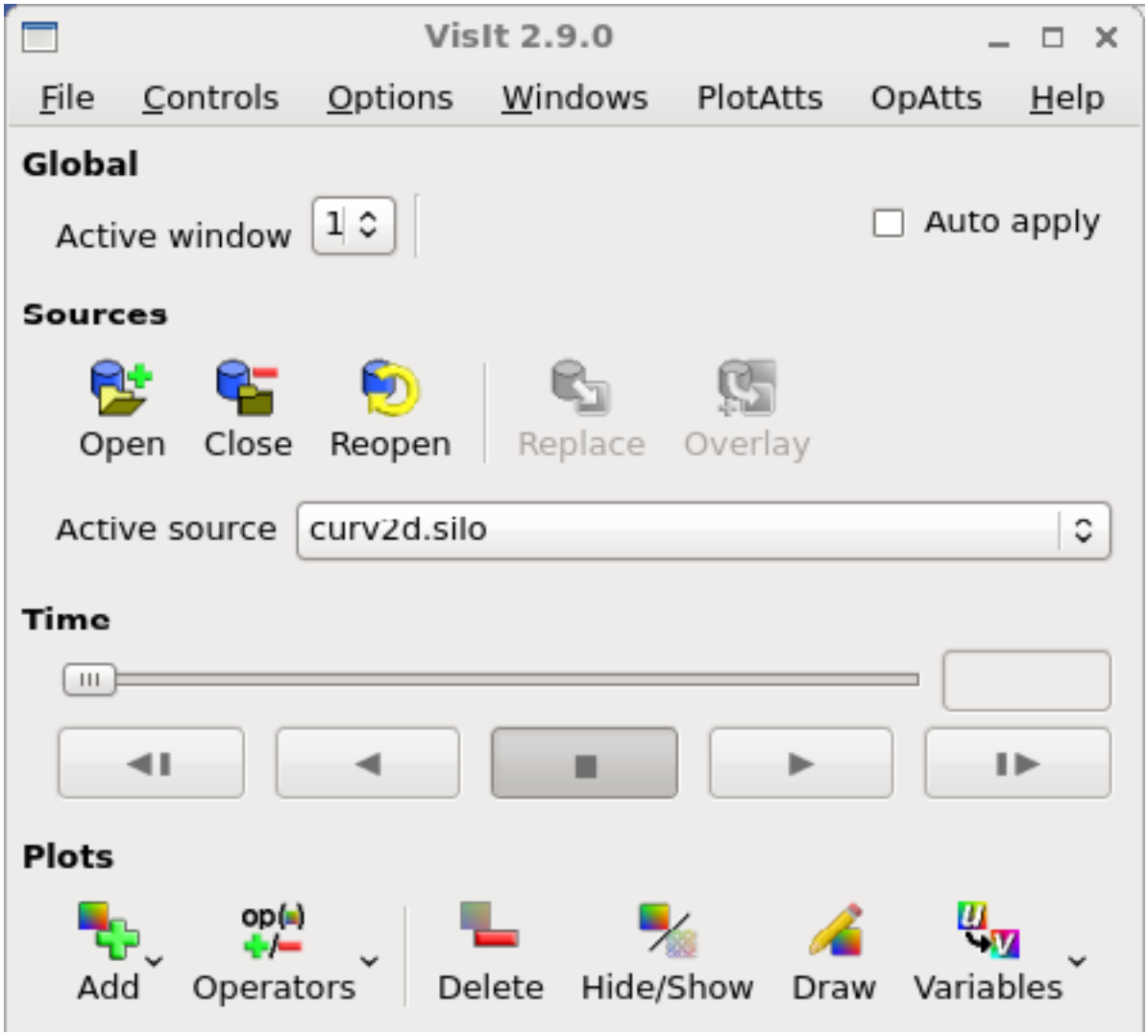


Fig. 4.6: Main gui panel showing Sources section

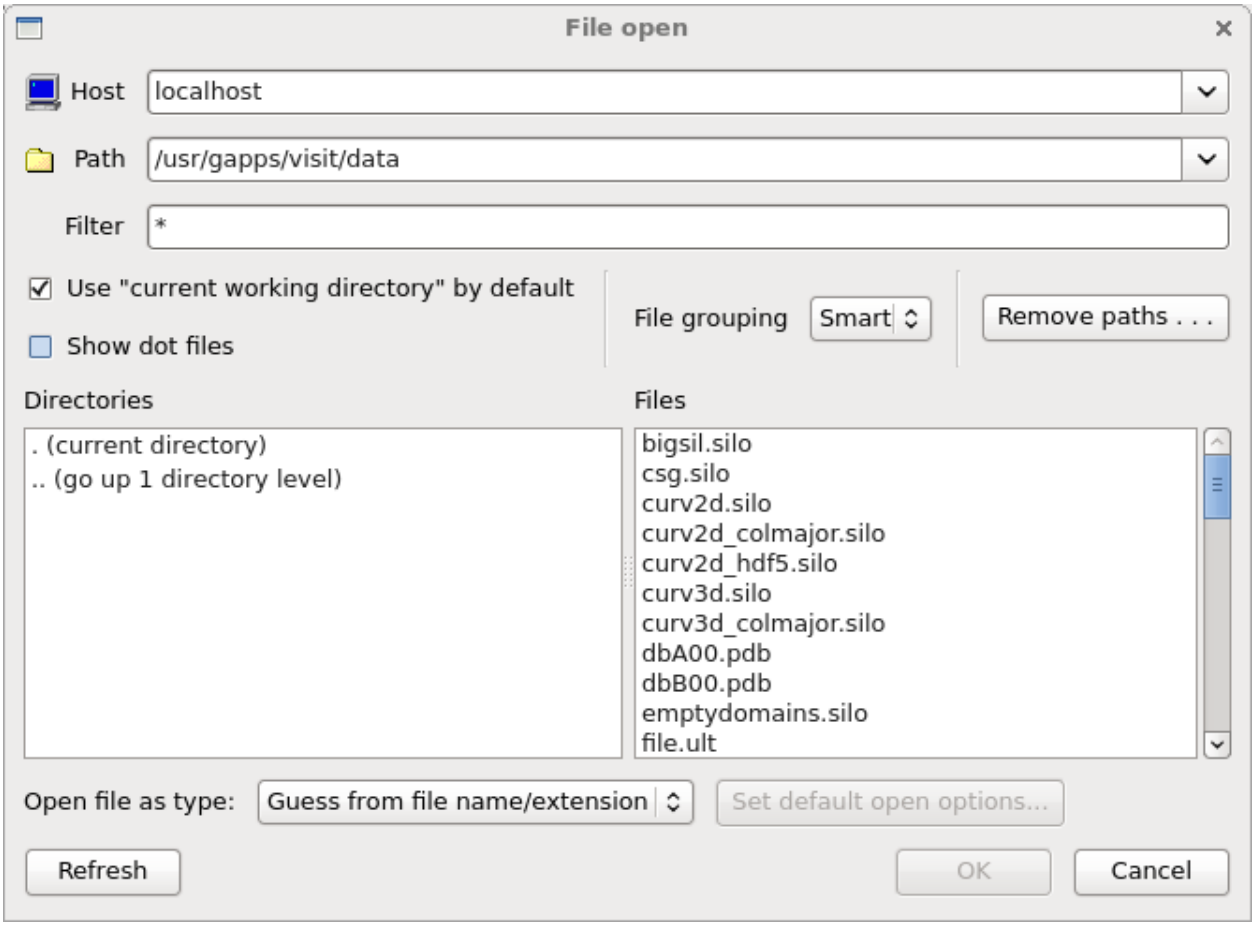


Fig. 4.7: File Open Window

Changing hosts

One of VisIt's strengths is its ability to operate on files that exist on remote computers. The default host is: "localhost", which is a name understood by the system to be the name of your local computer. To access the files on a remote computer, you must provide the name of the remote computer in the **Host** text field by either typing the name of a remote computer and pressing the Enter key or by selecting a remote computer from the list of recently visited hosts. To access the list of recently visited hosts, click on the down-arrow at the far right of the **Host** text field.

Changing the host will cause VisIt to launch a database server on the specified computer so you can access files there. Note that if you do not have an account on the remote computer, or if VisIt is not installed there, you will not be able to access files. Also note that VisIt may prompt you for a password to authenticate your access to the remote computer. To set up password-less access to remote computers, refer to *Setting Up Password-less SSH*.

Once a database server is running on the remote computer, its file system appears in the directory and file lists. The host name for each computer you access is added to the list of recently visited computers so that you may switch easily to computers you have recently accessed. If you installed VisIt with the provided network configurations then the list of recently visited computers also contains the hosts from the host profiles, which are covered later in this document.

Changing directories

To select data files, you must often change the active directory. This can be done in two ways. The first way is to enter the entire directory path into the **Path** text field and press Enter. You can use UNIX shell symbols, like the "~" for your home directory, or the "../" to go up one directory from your current directory. The directory conventions used depend on the type of computer being accessed. A MS Windows computer expects directories to be specified with a disk drive and a path with back slashes (e.g. `C:\temp\data`) while a UNIX computer expects directories with forward slashes (e.g. `/usr/local/data`). Keep the type of computer in mind when entering a path. After a path has been typed into the **Path** text field, VisIt will attempt to change directories using the specified path. If VisIt cannot change to the specified directory, the **Output Window** will appear with an error message and the **Path** text field will revert to the last accepted value. Another way to change directories is to double click the mouse on any of the entries in the directory list. Note that as you change directories, the contents of the **File list** change to reflect the files in the current directory. You can immediately return to any recently visited directory by selecting a directory from the **Path** text field's pull-down menu.

Default directory

By default, VisIt looks for files in the current directory. This is often useful in a UNIX environment where VisIt is launched from a command line shell in a directory where database files are likely to be located. When VisIt is set to look for files in the current directory, the **Use "current working directory" by default** check box is set. If all of your databases are located in a central directory that rarely changes, it is worthwhile to uncheck the check box, change directories to your data directory, and save settings so the next time VisIt runs, it will look for files in your data directory.

Changing filters

A filter is a pattern that is applied to the files in the **File list** to determine whether or not they should show up in the list. This mechanism allows the user to exclude many files from the list based on a naming convention, which is useful since VisIt's data files often share some part of their names.

The **Filter** text field controls the filter used to display files in the file list. Changing the filter will often change the **File list** as files are shown or hidden. The **Filter** text field accepts standard UNIX C-Shell pattern matching, where, for example, a "*" matches filter ("*") shows all files in the **File list**. Note that you can specify more than one filter provided you separate them with a space.

Virtual databases

A virtual database is a time-varying database that VisIt artificially creates out of smaller, single time step databases that have related filenames. Virtual databases allow you to access time-varying data without having to first create a `.visit :ref:`Need a reference to .visit files` file. The files that are grouped into a virtual database are determined by the file filter. That is, only files that match the file filter are considered for grouping into virtual databases. You can change the definition of a virtual database by changing the file filter. A virtual database appears in the file list as a set of filenames that are grouped under a single filename that contains the "*" wildcard character. (Figure 4.8) When you click on any of the filenames in the virtual database, the entire database is selected.

You can tell VisIt to not automatically create virtual databases by selecting the `Off` option in the **File grouping** pull-down menu. When automatic file grouping is turned off, no files are grouped into virtual databases and groups of files that make up a time-varying database will not be recognized as such without a `.visit` file. See Figure 4.9.

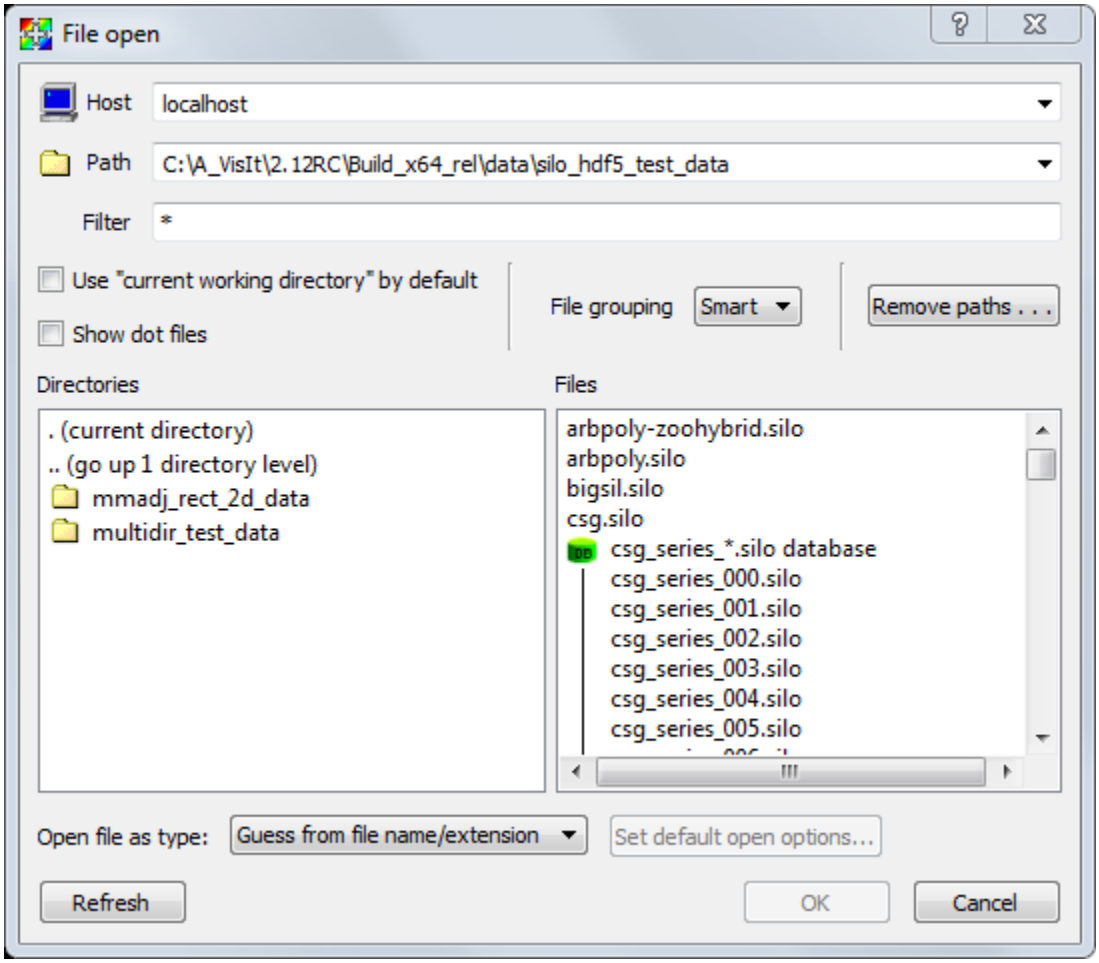


Fig. 4.8: File grouping turned on (Smart setting)

VisIt has two levels of automatic file grouping. The default level is Smart file grouping, which enables automatic file grouping but has extra rules that prevent certain groups of files from being grouped into virtual databases. If you find that Smart file grouping does not provide the virtual databases that you expect, you can back the file grouping mode down to On or turn it off entirely.

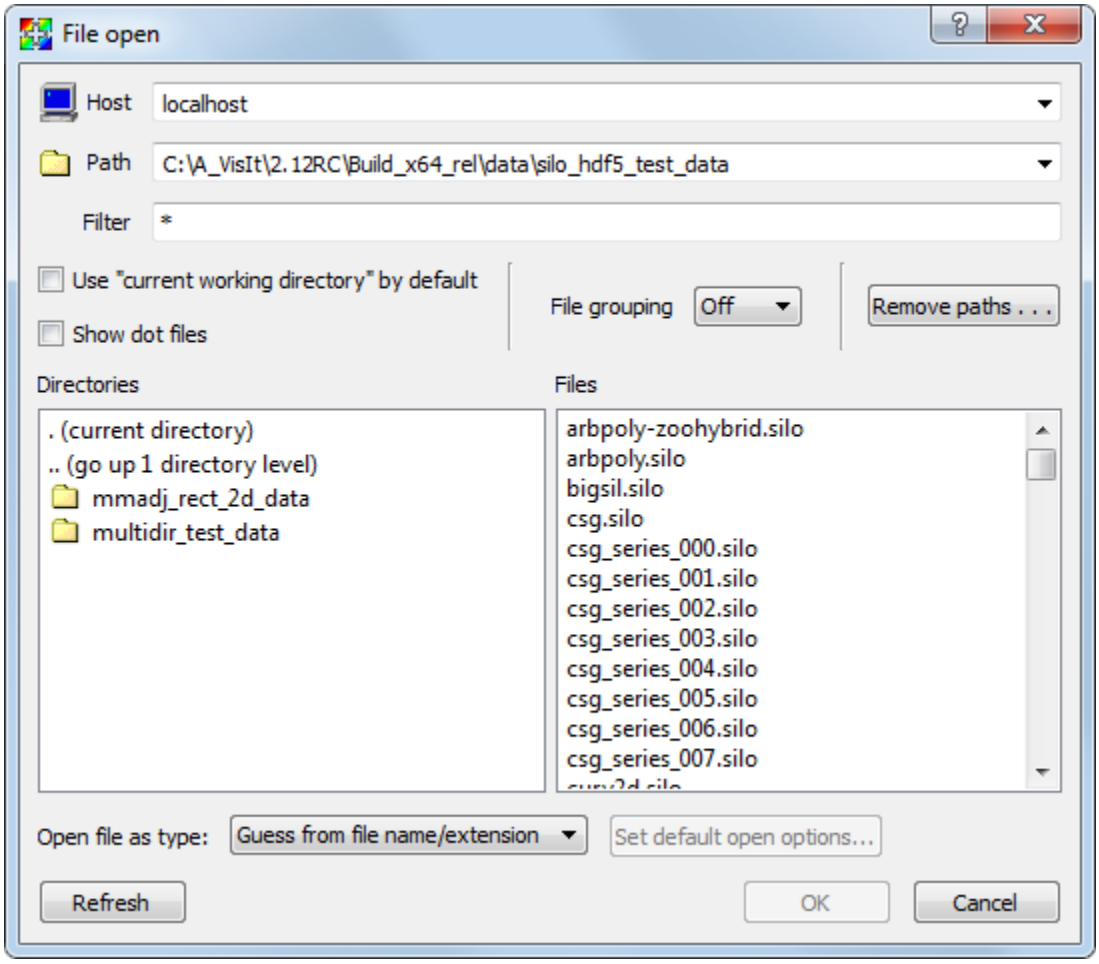


Fig. 4.9: File grouping turned off

Refreshing the file list

Scientific simulations often write out new data files as they run. The **Refresh** button makes VisIt re-read the current directory to pick up any new files added by a running simulation. If the active source is a virtual database whose definition was changed by refreshing the file list, then VisIt will close and reopen the active source so information about new time states is made available.

Clearing out recently visited paths

The **File Open Window** maintains a list of all of the paths that have ever been visited and adds those paths to the recently visited paths list, which can be accessed by clicking on the down-arrow at the far right of the **Paths** text field. When you click on a path in the recently visited paths list, VisIt sets the database server's path to the selected path retrieves the list of files in that directory. If you visit many paths, the list of recently visited paths can become quite long. Click the **File Open Window's Remove Paths** button to activate the **Remove Recent Paths** window. The **Remove Recent Paths** window allows you to select paths from the recently visited paths list and remove them from the list. The **Remove Recent Paths** window is shown in Figure 4.10.

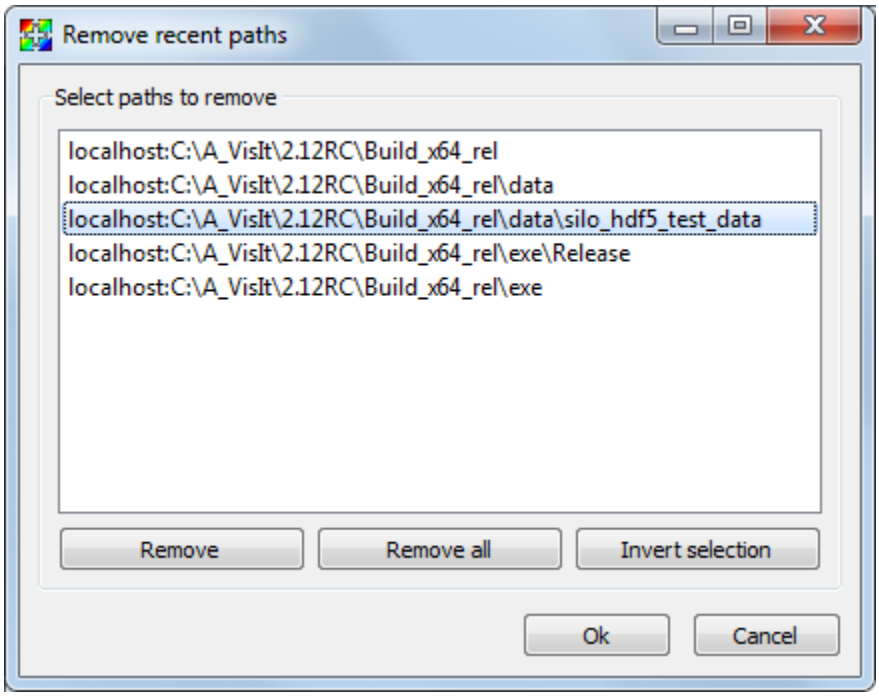


Fig. 4.10: Remove recent paths window

Connecting to a running simulation

Computer simulations often take weeks or months to complete and it is often necessary to visualize data from the simulation before it has completed in order to diagnose potential problems. VisIt comes with a simulation interface library that can be linked into your serial or parallel simulation application in order to provide hooks so VisIt can plot data from your running simulation. When instrumented with the VisIt simulation interface library, your simulation can periodically check for incoming VisIt connections. When VisIt successfully connects to your simulation, all of your simulation variables are available for plotting without having to write plot files to disk. During the time that VisIt is connected, your simulation acts as a VisIt compute engine in addition to its regular responsibilities. You can pause the simulation while using VisIt to interact with the data or you can choose to have the simulation continue and push new

data to VisIt for plotting. For more information about instrumenting your simulation code with the VisIt simulation library interface, see the Getting Data Into VisIt manual.

VisIt currently treats simulations as though they were ordinary files. When the VisIt simulation interface library is enabled in your application, it writes a special file with a `.sim2` extension to the `.visit/simulations` directory in your home directory (`%Documents%\VisIt\simulations` on Windows). Each `.sim2` file encodes the time and date it was created into the file name so you can distinguish between multiple simulations that VisIt can potentially open. A `.sim2` file contains information that VisIt needs in order to connect via sockets to your simulation. If you want to connect to a simulation, you must select the `.sim2` files corresponding to the simulations to which you want to connect. (Figure 4.11). Once that is done, connecting to a simulation is the same as opening any other disk file.

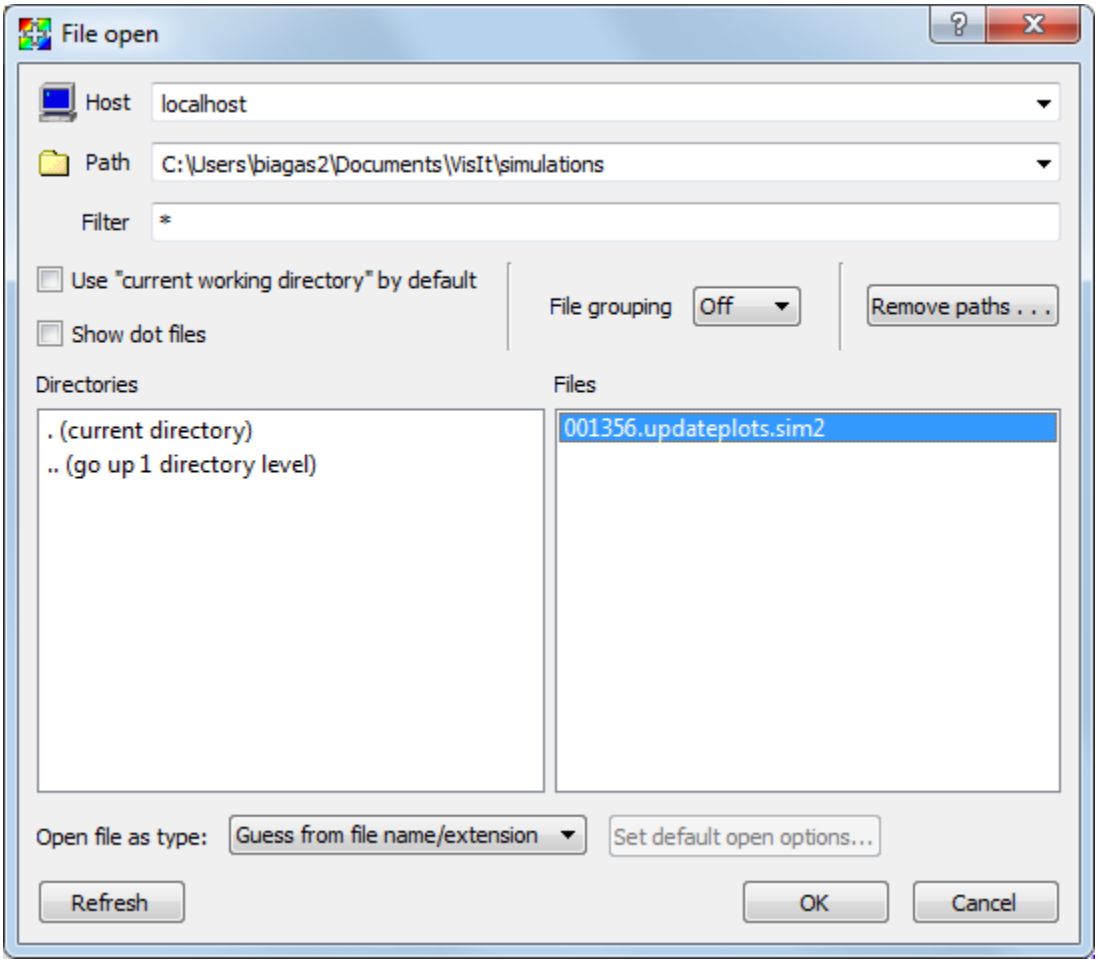


Fig. 4.11: Accessing a simulation using the File Open Window

4.2.3 Database Read Options

Several database plugins have options that affect reading and/or writing with that format. These are described in this section. Alternatively, in some cases, the behavior of a database plugin may be affected by enviornment variables.

Blueprint

MFEM LOR Setting

This option allows the user to select a MFEM Low-Order-Refinement Scheme. The two options are Legacy LOR and MFEM LOR. The Legacy setting was previously VisIt's only LOR method in the Blueprint Plugin. It produces discontinuous output, while the new option produces a continuous low order mesh. This new option is the default.

MFEM

MFEM LOR Setting

This option allows the user to select a MFEM Low-Order-Refinement Scheme. The two options are Legacy LOR and MFEM LOR. The Legacy setting was previously VisIt's only LOR method in the MFEM Plugin. It produces discontinuous output, while the new option produces a continuous low order mesh. This new option is the default.

Exodus

Detect Compound Variables

Checking this option will cause the plugin to try to guess that similarly named variables are the scalar components of an aggregate type such as a vector, tensor or array variable. The plugin will then automatically define expressions for these aggregate typed variables. For example, it will cause the plugin to combine three scalar variables with names such as `velx`, `vely` and `velz` into a *vector* expression `vel` defined as `{velx, vely, velz}`. Note that this is just a convenience to free users from having to define expressions manally within their VisIt session.

Use Material Convention

With this option, the user can cause the plugin to recognize standard or custom *material conventions*. The Exodus file format does not define any specific standards for handling advecting and *mixing* materials. Different data producers have defined different conventions. A few pre-defined conventions for handling mixed materials from Exodus files are supported. In addition, users can define their own custom conventions as well. For a custom convention, the user must define the *namescheme* that will produce the names of the scalar variables holding material volume fractions. Optionally, users can specify a namescheme to produce the names of the scalar variables holding material-specific values for an associated non-material-specific variable.

The namescheme used here are identical to those described in the Silo user's manual with one extension. The conversion specifier `%V` is used to denote the basename (non-material-specific) name of a set of scalar variables holding material specific values.

The ALEGRA nameschemes for volume fraction and material specific variables are `"@%s@n?'&VOLFRC_%d&n&':'VOID_FRC':@"` and `"@%V_%d@n"`.

The CTH nameschemes are `"@%s@n?'&VOLM_%d&n&':'VOID_FRC':@"` and `"@%V_%d@n"`.

Finally, in all cases it is assumed materials are identified starting from index one (1). The special material id of zero (0) is used to denote void.

Material Count

Ordinarily, the plugin will determine the material count from the material convention nameschemes. However, if it is having trouble getting the correct count, users can specify it manually with this option.

ffp

The ffp plugin can optionally use the STRIPACK library to improve its behavior and performance. It will do so by loading the STRIPACK library as a dynamicaly loaded shared library *when* VisIt can find it. VisIt will find the STRIPACK library if it is available in the VisIt installation's top-level `lib` directory (typically something like `/foo/bar/visit/3.1.1/linux-x86_64/lib`) or if the enviornment variable `VISIT_FFP_STRIPACK_PATH` is set specifying a path to the shared library as in

```
setenv VISIT_FFP_STRIPACK_PATH /foo/bar/lib/libstripack.so
```

for csh (and friends) or for sh (and friends)...

```
export VISIT_FFP_STRIPACK_PATH=/foo/bar/lib/libstripack.so
```

In client/server mode, the STRIPACK library must be installed on both the client and the server.

The *build_visit* tool can be used to download, build and install the STRIPACK library. Here is an example bash shell `build_visit` command-line...

```
env FCFLAGS="-fdefault-real-8 -fdefault-double-8 -shared -fPIC" \
STRIPACK_INSTALL_DIR=/usr/local/visit/lib ./build_visit --fortran \
--no-visit --no-thirdparty --thirdparty-path /dev/null --no-zlib --stripack
```

Because STRIPACK is non-BSD licensed software, part of the `build_visit` process for installing it is to accept the STRIPACK license terms.

NASTRAN

Num Materials

This option allows the user to indicate that the NASTRAN plugin should look for and try to define a material object. If the user knows the *number* of materials in the input database, it is best to specify it here because that will avert the plugin having to read all lines of the input before understanding the material configuration. However, if the user does not know the number of materials, enter `-1` here and the plugin will search for all information related to the material configuration during the *open*. This will lead to longer open times. A value of `0` here means to ignore any material information if present.

PLOT3D

Overview

PLOT3D is a computer graphics program designed to visualize the grid and solutions of structured computational fluid dynamics (CFD) datasets. It is developed and maintained by NASA. PLOT3D is not a self describing format. Therefore VisIt does not know if the file it should read is:

- 2D or 3D
- Binary or ASCII
- Fortran-style (record based or not) or C-style
- Has Iblanking or not
- Single block or multiblock

To get VisIt to read your file, you need to give it hints. You do this with a text file with extension *.vp3d*, which describes the variant of Plot3D being used, or through the Read options that can be set when opening the file.

VisIt will perform some amount of auto-detection for binary files. If auto-detection fails, then VisIt will fall back to settings from '.vp3d' if used, or Read options otherwise. If VisIt doesn't display your data as expected, some of these options may need to be tweaked. Auto-detection will most likely fail for non-record based Fortran binary files.

Please Note: If your single-grid data file has the 'nblocks' field, you will need to tell VisIt it is a 'MultiGrid' file. VisIt will then correctly read 'nblocks' and create single-grid output.

Example vp3d file

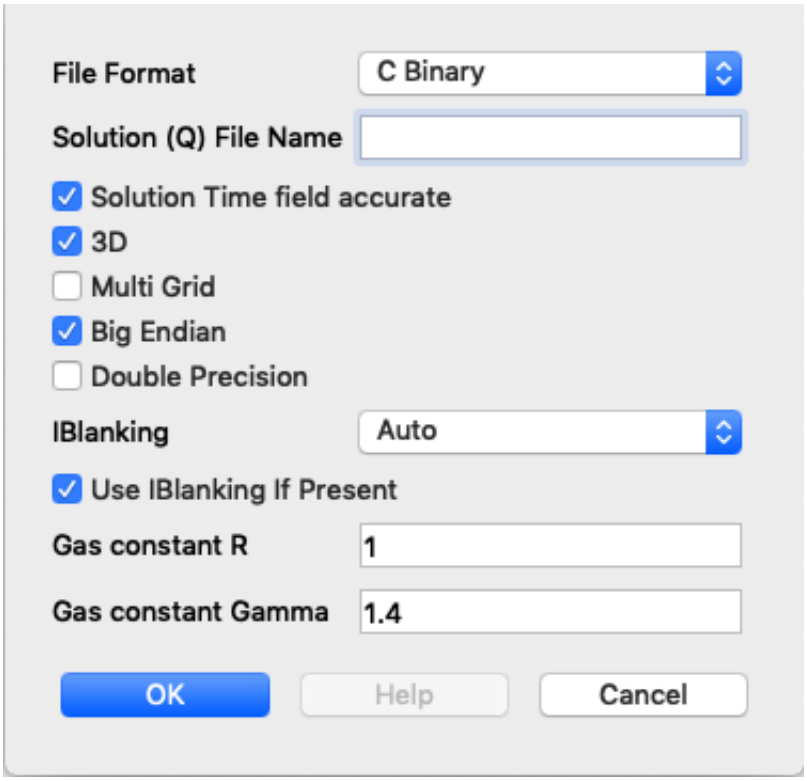
```
# Files:
#
# Note: the Grid file and Solution file fields do not have to be specified.
# If they do not appear, VisIt will assume that the .vp3d should be replaced
# with ".x" for the grid file and ".q" for the solution file.
#
# Support for time-series solution files added in VisIt 2.10.0.
# VisIt will look for '*' and '?' wildcards in the solution name
#
GRID NHLP_2D.g
# SOLUTION NHLP_2D.q
# Time-series example, requesting all time steps
# SOLUTION NHLP_2D_*.q
# Time-series example requesting subset of time steps
# SOLUTION NHLP_2D_??3?.q

# Single/Multi Grid. Single grid will be assumed if no value is specified.
#
# Options:
MULTI_GRID
# SINGLE_GRID

# Data encoding, ASCII or Binary. Binary will be assumed if no value is
# specified.
#
# Options:
#BINARY
ASCII

# Endianness. This only applies to BINARY files. Native endian will
# be assumed if no value is specified.
#
# Options:
#LITTLE_ENDIAN
#BIG_ENDIAN

# OBSOLETE, Structured assumed, due to lack of unstructured sample data
# Structured grid vs unstructured grids. Structured grids will be assumed
# unless stated otherwise.
#
# Options:
# STRUCTURED
# UNSTRUCTURED

# Iblanking in the file. No iblanking is assumed unless stated otherwise.
```

(continues on next page)

(continued from previous page)

```
#
# Options:
# NO_IBLANKING
# IBLANKING

# Ignore iblanking. If there is iblanking in the file, you can opt to ignore it.
#
# Options:
# IGNORE_IBLANKING

# 2D vs 3D. 3D will be assumed unless stated otherwise.
#
# Options:
2D
# 3D

# Precision. Single precision is assumed unless stated otherwise.
#
# Options:
SINGLE_PRECISION
# DOUBLE_PRECISION

# Compression. This only applies to ASCII files. Some codes compress
# repeated values as 4*1.5 as opposed to 1.5 1.5 1.5 1.5. It is assumed
# the data is not compressed unless stated otherwise.
#
# Options:
# COMPRESSED_ASCII
# UNCOMPRESSED_ASCII

# C vs Fortran Binary. C-Binary is assumed.
# VisIt 2.10.0, added FORTRAN_BINARY_STREAM, to differentiate between
# record-based (FORTRAN_BINARY) and non record based (FORTRAN_BINARY_STREAM)
# Options:
# C_BINARY
# FORTRAN_BINARY
# FORTRAN_BINARY_STREAM

# Time. Tells VisIt whether or not the 'Time' field in the solution file is accurate.
# If set to '1', VisIt will use this as the 'time' value displayed in plots for time-
↪series data. (Default)
# If set to '0', and this is time-series data, VisIt will attempt to parse the 'time'␣
↪from the solution file name.
SOLUTION_TIME_ACCURATE 1

# R, Gamma values (used for computing functions like Temperature, Pressure, Enthalpy,␣
↪Entropy)
# Defaults are shown.
# R 1.0
# GAMMA 1.4
```

Read Options via GUI



Read Options via CLI

```
# MDServer must be started in order grab the default Open options for the reader
OpenMDServer("localhost")
# Grab the default options
opts = GetDefaultFileOpenOptions("PLOT3D")
# and change a couple of things
# specify sub-selection of time slices.
opts["Solution (Q) File Name"] = r"Jespersen.1/????3?"
opts["Solution Time field accurate"] = 0
SetDefaultFileOpenOptions("PLOT3D", opts)
OpenDatabase(data_path("./TaperedCylinder/grid.p3d"), 0, "PLOT3D_1.0")
```

Or, you can create your own subset of the options:

```
opts = {'Multi Grid':1, "Solution (Q) File Name":"wbtr.bin"}
SetDefaultFileOpenOptions("PLOT3D", opts)
OpenDatabase(data_path("./WingBodyTail/wbtg.bin"), 0, "PLOT3D_1.0")
```

Here are the defaults:

```
>>> opt = GetDefaultFileOpenOptions("PLOT3D")
>>> print opt
{
  'File Format': 'C Binary # Options are: ASCII, C Binary, Fortran binary, Fortran␣
↪binary stream',
```

(continues on next page)

(continued from previous page)

```
  'Solution (Q) File Name': '',
  'Solution Time field accurate': 1,
  '3D': 1,
  'Multi Grid': 0,
  'Big Endian': 1,
  'Double Precision': 0,
  'IBlanking': 'Auto # Options are: Auto, Always, Never',
  'Use IBlanking If Present': 1,
  'Gas constant R': 1.0,
  'Gas constant Gamma': 1.4
}
```

Silo

Ignore Extents

The Silo database plugin has the ability to load spatial and data extents for Silo multi-block (e.g. multiple domain) objects. This feature is an optional *acceleration* feature that enables VisIt to cull domains based on knowledge of downstream operations. For example, it can avoid reading domains known not to intersect a slice plane. However, if the data producer creates buggy extents data, this can lead to problems during visualization. So, the Silo plugin has read options to disable spatial and data extents. The options for each are `Always`, `Auto`, `Never` and `Undef`(ined) where `Always` and `Never` mean to always *ignore* or never *ignore* the extents data and `Auto` means to ignore extents data for files written by data producers known to have issues with extents data in the past. The `Undef` setting is to deal with cases where users may have *saved settings* with very old versions of these options.

Force Single

The `Force Single` check box enables the Silo library's DBForceSingle() method. This can potentially be useful when reading double precision data and running out of memory.

Search for ANNOTATION_INT (and friends)

The `ANNOTATION_INT` (and friends) objects are generic containers sometimes used to store mesh-specific data using Silo's compound array. However, because there is no multi-block analog for Silo compound arrays, in order to handle them VisIt needs to be forced to go searching for their existance in all the files comprising a multi-block database. Thus, enabling this option can result in much slower database *open* times.

ZipWrapper

TMPDIR

Specifies the directory to be used for temporary, decompressed files. Defaults to `$TMPDIR` which will then resolve to the `$TMPDIR` environment variable which if either not defined or not a writable directory will then default to either `/usr/tmp` or `/var/tmp` and finally `$HOME` environment variable.

Don't atexit()

Ordinarily, when VisIt exits, it will remove any decompressed files it left around from invocations of ZipWrapper's decompression logic. This disables removal of decompressed files upon exit from VisIt.

Max. # decompressed files

Specifies the maximum number of decompressed files that can be in existance at any one time. Default is 50. In parallel, this is a total summed over all processors unless a negative number is specified in which case it is the total per processor (useful for processor local tmp directories).

Unique moniker for dirs made in $TMPDIR

An arbitrary string designed to be highly *unique* among all possible processes that can write to `TMPDIR`. Defaults to `$USER` which will then resolve to the `$USER` enviornment variable.

Decompression command

Specifies the decompression command to use to decompress files. Default is to use file extension to determine command according to table below

File Extension	Decompression Command
.gz	gunzip -f
.bz	bunzip -f
.bz2	bunzip2 -f
.zip	unzip -o

4.2.4 Sources Pane

The **Sources pane** , near the top of the **Main Window,** displays the currently active source, and contains controls to open, close, reopen, and and overlay sources. Sources are most frequently database files.

Opening a file

To open a file, you want to visualize, click on the **Open** button. This opens the *File Open Window*. Once a file is open, the **Close** and **Reopen** buttons become enabled.

If you have opened multiple files, the **Active source** drop-down menu allows you to switch between the files.

When the **ReOpen** button is clicked, all cached information about the open database is deleted, the database is queried again for its information, and any plots that use that database are regenerated using the new information. This allows VisIt to access data that was added to the database after VisIt first opened it.

Reopening a database

Sometimes it is useful to begin visualizing simulation data before the simulation has finished writing out data files for all time steps. When you open a database in VisIt and create plots and later want to visualize new time steps that have been generated since you first opened the database, you can reopen the database to force VisIt to get the data for the new time steps. To reopen a database, click the **ReOpen** button in the **Sources pane**. When VisIt reopens a database,

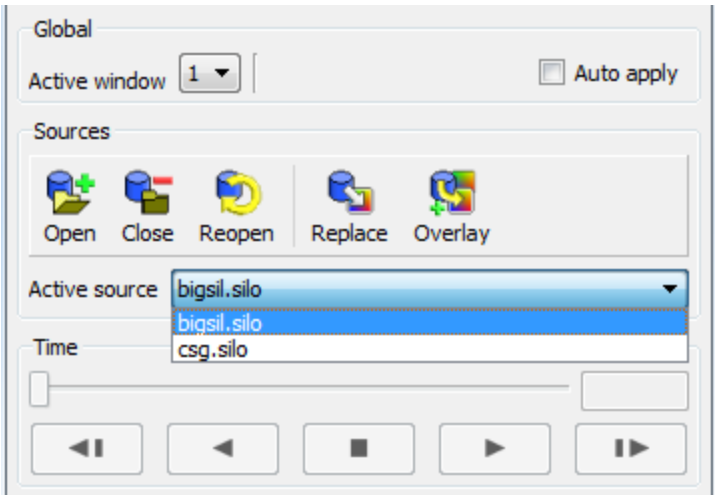


Fig. 4.12: Controls for setting the active source

it clears the geometry for all plots that used that database and cached information about the database is erased so that when VisIt reopens the database, plots are regenerated using the new data files.

Replacing a database

If you have created a plot with one database and want to see what it looks like using data from another database, you can replace the database using the **File panel's Replace** button. To replace a database, first select a new database by clicking on a file in the **File panel's Selected files list** and then click the **Replace** button. This will make VisIt try to replace the databases used in the plots with the new database. If the replace operation is a success, the plots are regenerated using the new database and they are displayed in the visualization window.

Overlaying a database

Overlaying a database is a way to duplicate every plot in the plot list using a new database. To overlay plots, select a new database from the **Active sources** dropdown, then click the **Overlay** button. This copies each plot in the **Active plot list** and replaces the database with the specified database. If the operation succeeds, the plots are generated and displayed in the visualization window. It is important to remember that each time the **Overlay** button is clicked, the number of plots in the plot list doubles.

4.2.5 Time Pane

The **Time Pane** contains controls for setting the active timestep, and VCR controls for playing animations.

Setting the active time step

When a time-varying database is open, the animation controls are activated so any time step in the database can be used. Note that the animation controls are only active when visualizing a time-varying database or when VisIt is in keyframe animation mode.

Time-varying databases are composed of one or more time steps which contain data to be visualized. The active time step is the time step within a time-varying database that VisIt uses to generate plots. The **Time pane** is located just below the **Sources pane** and contains controls that allow you to set the active time step used for visualization. The

Animation slider and the **Animation text field** show the active time step. To set the active time step, you can drag the **Animation slider** and release it when you get to the desired time step, or you can type in a cycle number into the **Animation text field** . If you type in a cycle number that is not in the database, the active time step will be set to the time step with the closest cycle number to the cycle that was specified.

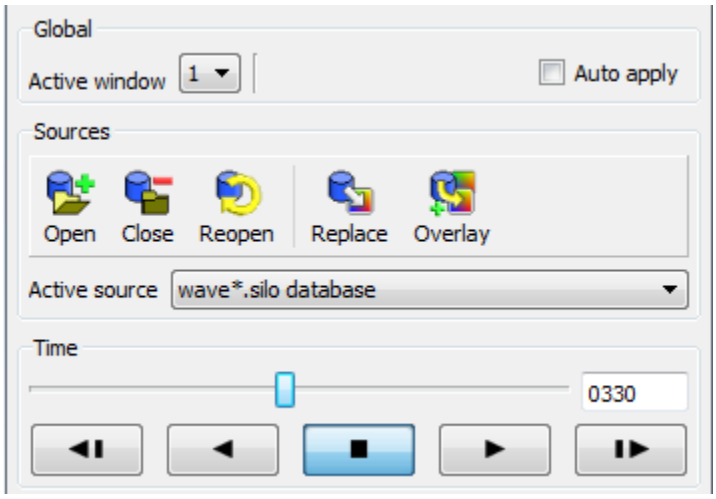


Fig. 4.13: Controls for setting the active time step

Playing animations

The **Time pane** also contains a set of **VCR buttons** that allow you to put VisIt into an animation mode that plays your visualization using all of the time steps in the database. The **VCR buttons** are only active when you have a time varying database. The leftmost VCR button moves the animation back one frame. The VCR button second from the left plays the animation in reverse. The middle VCR button stops the animation. The VCR button second from the right plays the animation. The VCR button farthest to the right advances the animation by one frame. As the animation progresses, the **Animation Slider** and the **Animation Text Field** are updated to reflect the active time step.

4.2.6 File Information Window

This **File Information Window**, shown in Figure 4.14, displays information about the currently open file. The **File Information Window** is opened by choosing the **Files information** option from the **Main Window's File** menu. The window displays the names and properties of the open file's meshes, scalar variables, vector variables, and materials. The window updates each time the active file changes such as when switching between plots in the **Active plot list** or opening a new file using the controls in the **File panel**.

4.3 Plots

This chapter explains the concept of a plot and goes into detail about each of VisIt's different plot types.

4.3.1 Working with Plots

A plot is a viewable object, created from a database, that can be displayed in a visualization window. VisIt provides several standard plot types that allow you to visualize data in different ways. The standard plots perform basic visualization operations like contouring, pseudocoloring as well as more sophisticated operations like volume rendering.

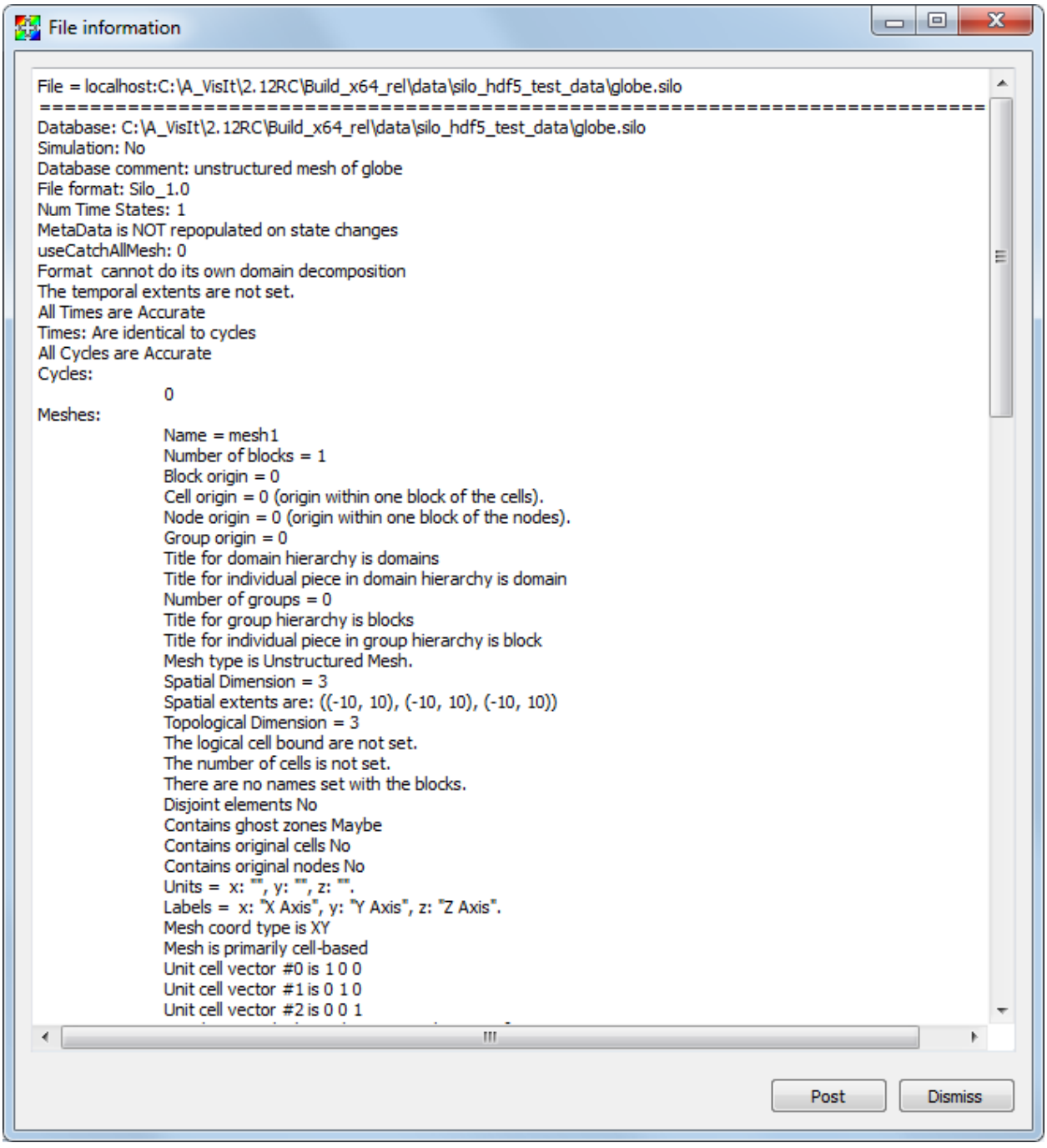


Fig. 4.14: File Information Window

All of VisIt's plots are plugins so you can add new plot types by writing your own plot plugins. For information on creating new plot plugins contact us via Getting help.

Managing Plots

To visualize your data, you will iteratively create and modify many plots until you achieve the end result. Since plots may be created and deleted many times, VisIt provides controls in its **Main Window** to handle these functions. The **Plots** area, shown in Figure 4.15, contains the controls for managing plots.

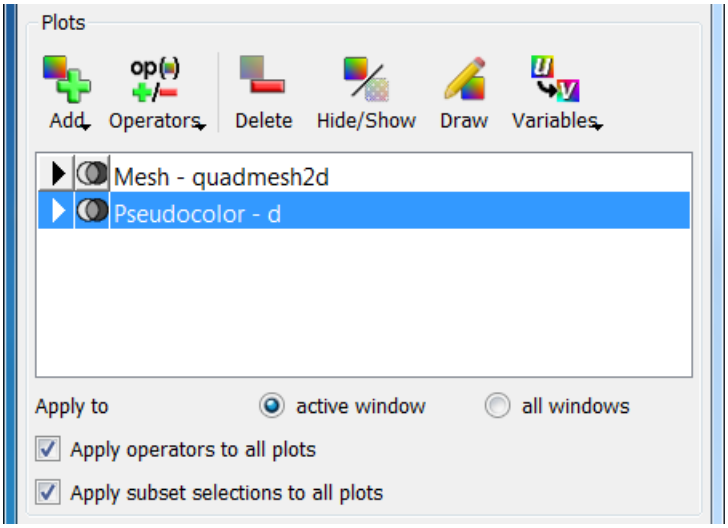


Fig. 4.15: The active plots area

The most prominent feature of the **Plots** area, the plot list contains a list of the plots that are in the active visualization window. The entries in the plot list contain the plot name and variable. Plot list entries change colors depending on the state of the plot. When plots are initially created, their plot list entries are green indicating that they are new and have not been submitted to the compute engine for processing. When a plot is being created on the compute engine, its plot list entry is yellow. When a plot has finished generating on the compute engine, its plot list entry turns black to indicate that the plot is done. If the compute engine cannot generate a plot, the plot's plot list entry turns red to indicate an error with the plot.

The plot list displays more then just the names of the visualization window's plots. The plot list also allows you to set the active plots, that is, those plots that can be modified. Highlighted plot entries are active.

The **Add** menu, an important part of the **Plots** area, contains the options that create new plots.

Creating a plot

To use any of VisIt's capabilities, you must know how to create a plot. First, make sure you have opened a database. Once you have an open database, use the **Add** menu to create a plot.

Selecting the **Add** menu pops up a list of VisIt plot types. Plots for which the open database has no data are disabled. If a plot type is enabled, pulling the mouse toward the right while holding down the left button shows which variables can be plotted. Release the mouse button when the mouse cursor is over the variable that you want to plot, and a new plot list entry will appear in the plot list. The new plot list entry will be colored green in the plot list until VisIt is told to draw when you click the **Draw** button. The **Add** menu is disabled until a database is open.

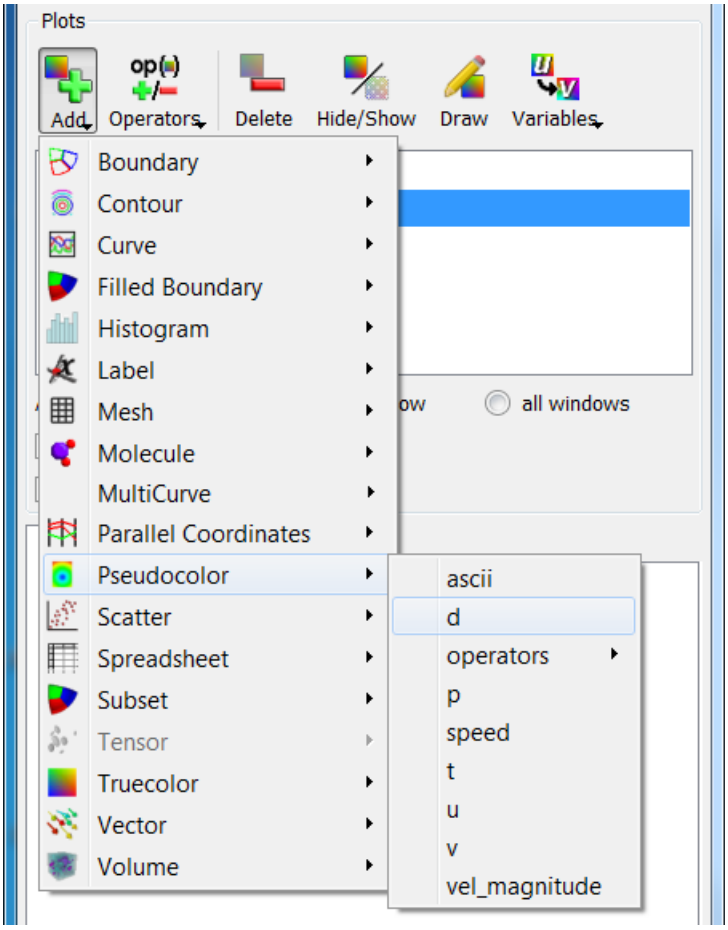


Fig. 4.16: The Add menu

Deleting a plot

VisIt deletes all the selected plots when you click the **Delete** button. If the plot list has keyboard focus, you can also delete a plot using the **Delete** key.

Selecting a plot

Since VisIt will only let you modify active plots, you must be able to select plots. To select a plot, click on its entry in the plot list. Multiple plots can be selected by holding down the `Ctrl` key and clicking plot entries one at a time. Alternatively, groups of plot entries can be selected by clicking on a plot entry and then clicking another plot entry while holding down the `Shift` key.

Drawing a plot

When you add a plot to the plot list, it won't be drawn until you click the **Draw** button. Once you do, the new plot's plot list entry switches from green to yellow in the plot list to indicate that its results are pending and the compute engine starts generating the plot. Clicking the **Draw** button causes all new plots to be drawn.

Hiding a plot

When you are visualizing your data, you will often have many different plots in the same visualization window. Sometimes you might want to temporarily hide plots from view to more easily view the other plots in the window. To hide the selected plots, click the **Hide/Show** button in the **Plots** area. When a plot is hidden, its plot list entry is gray and contains the word `hidden` to indicate that the plot is hidden. To show a hidden plot, select the hidden plot and click the **Hide/Show** button again. Note that plots must exist for the **Hide/Show** button to be enabled.

Setting plot attributes

Each plot type has its own plot attributes window used to set attributes for that plot type. Plot attributes windows are activated by double-clicking a plot entry in the plot list. You can also open a plot attribute window by selecting a plot type from the **PlotAtts** (Plot Attributes) menu shown in Figure 4.17,

Changing plot variables

When examining a plot, you might want to look at another variable. For example, you might want to switch from looking at density to pressure. VisIt allows the plot variable to be changed without having to delete and recreate the plot. To change the plot variable, first make sure the plot is active, then select a new variable from the available variable names in the **Variables** menu. The **Variables** menu contains only the variables from the database that are compatible with the plot.

4.3.2 Standard Plot Types

VisIt comes with eighteen standard plots: Boundary, Contour, Curve, FilledBoundary, Histogram, Label, Mesh, Molecule, `MultiCurve`, ParallelCoordinates, Pseudocolor, Scatter, Spreadsheet, Subset, Tensor, Truecolor, Vector, and Volume. This section explains each plot in detail.

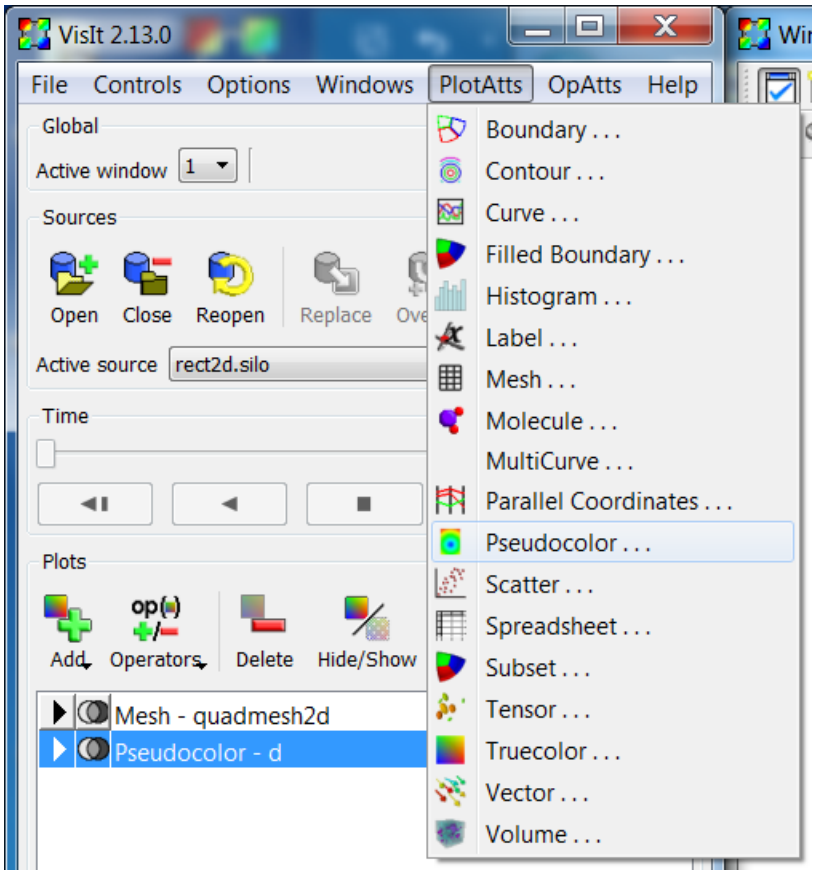


Fig. 4.17: The PlotAtts menu

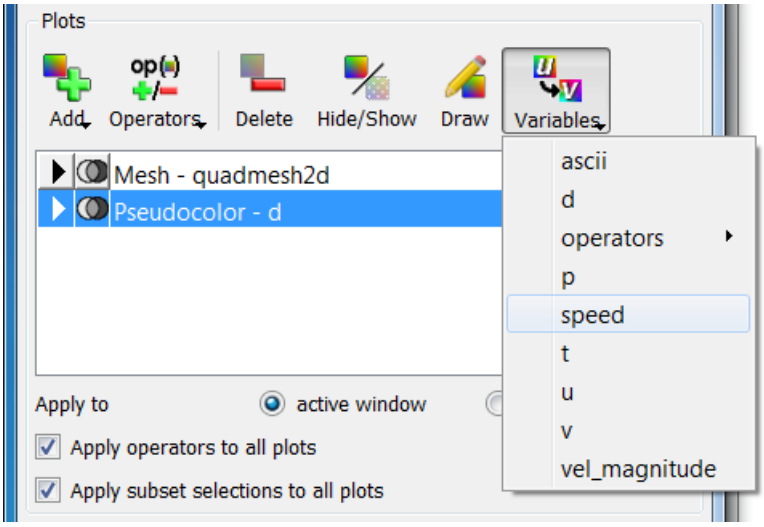


Fig. 4.18: The Variables menu

Common Controls

There are a number of attributes of plots that are common to many, if not all plots. These include such things as **Color table**, **Foreground** and **Background** colors, **Opacity**, **Line width** and **Point type**, **Log** or **Linear** scaling, the **Legend** checkbox, the **Lighting** checkbox and others. These common plot attributes are described here first using the **Pseudocolor plot** as an example.

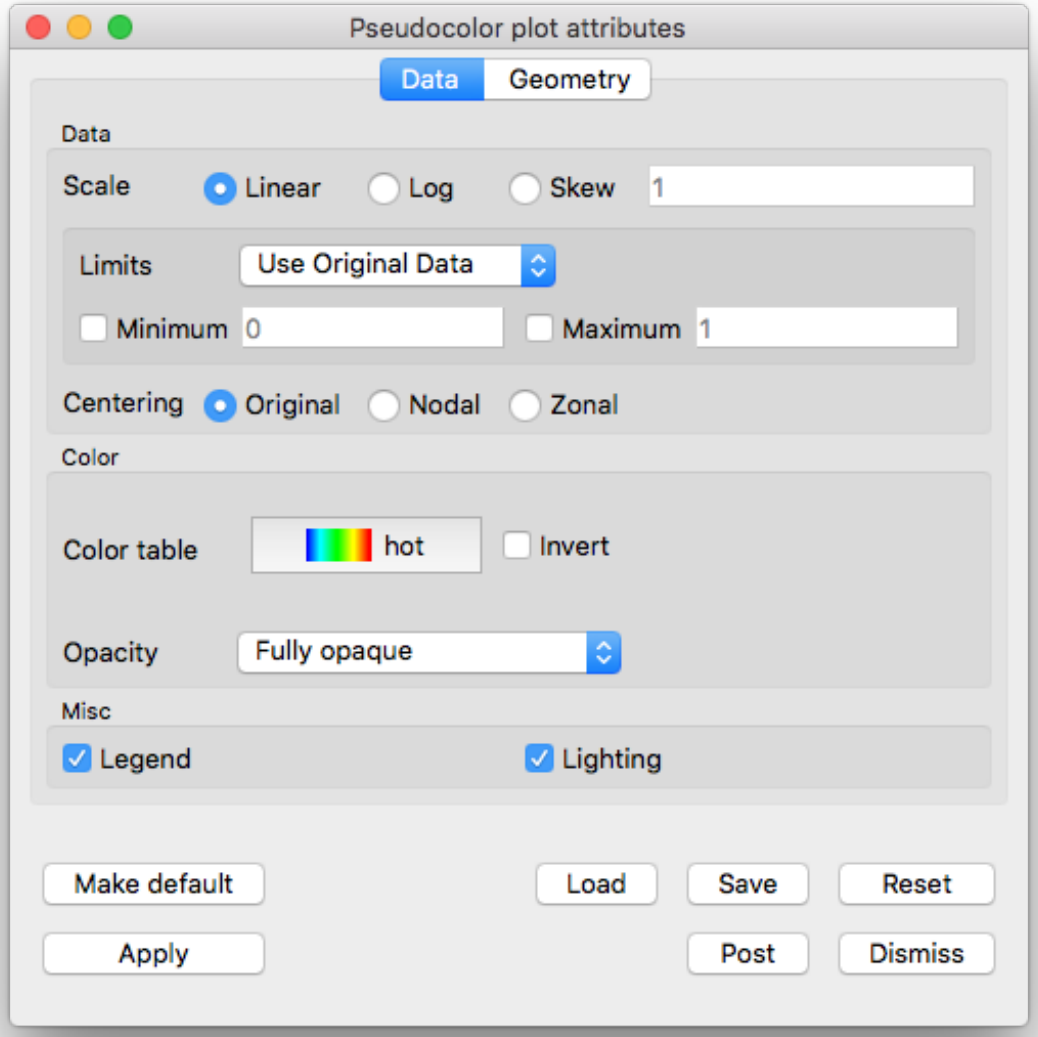


Fig. 4.19: Example of **Pseudocolor plot attribute window**

Then, attributes specific to each plot type are described in the remaining sections.

Plot buttons

All plot attribute windows have several buttons at the bottom for common operations. Use the **Apply** after you have changed one or more attributes of a plot to make the new settings take effect. The **Make default** button is used to take the current settings and make those the default for the remainder of the VisIt session. Each time a new plot of that type is created, it will be created with whatever the current defaults are for that plot. If you want these settings to persist across VisIt sessions, you can either **Save session**, and then restart from this saved session later, or **Save settings** and then all VisIt sessions will use those defaults. For more about saving sessions and settings, see *How to Save Settings*. The **Save** and **Load** buttons give you the option of saving and loading plot attributes using their own separate XML. This allows users to easily share individual plot attributes. The reset button will return the plot's attributes to whatever the current defaults are. The **Dismiss** button will dismiss the window. The **Post** button will place the window in the **Notepad area** (see *Posting a window*).

Plot colors

By default, VisIt uses the **Hot** color table which maps values at the minimum of the data range to blues, values at the maximum of the data range to reds with transitions from blue to violet, to green, to yellow in between. However, many plots offer the option of selecting a specific color table. In the picture of the **Pseudocolor plot attributes** window, above, the color table may be changed by selecting the currently named table. A pull-down list will appear from which you can select a different table. For more information about **Color tables**, see *Color Tables*.

In addition, many plots have options to control colors and transparency (opacity) of individual plot elements such as lines on the **Mesh plot** or contours on the **Contour plot**.

Point type and size

The **Pseudocolor**, **Mesh** and **Scatter** plots can use eight different point types for drawing point meshes (see Figure 4.20). The default option of **Point** is fastest and forces the plot to draw all of its points as tiny points. The **Sphere** option applies textures to the points so it is nearly as fast as **Point**. Any of the other options place a glyph at each point, taking longer to render. To set the point type choose an option from the **Point type** menu. Setting the **Point type** to anything other than **Point** will have no effect if the plotted mesh is not a point mesh.

If you choose any of the point types except **Point**, then you can also specify a point size by typing a new value into the **Point size** text field. The point size is used to determine the size of the glyph. For example, if you choose **Box**, and you enter a **Point size** of 0.1, then the length of all of the edges on the Box glyphs will be 0.1. If you use **Point**, then the **Point size** text field becomes the **Point size (pixels)** text field and you can set the point size in terms of pixels.

For **Mesh** and **Pseudocolor** plots, the point size can also be scaled by a scalar variable if you check the **Scale point size by variable** check box and select a new scalar variable from the **Variable** menu. The value `default` must be replaced with the name of another scalar variable if you want VisIt to scale the points with a variable other than the one being plotted.

Lighting

Various plots include a **Lighting** checkbox. When the box is checked, it means the plot will obey all *active light sources*. When the box is **not** checked, this does not mean the plot will not be lit at all. Instead, it means that the plot will be lit by **Ambient** lighting only.

Boundary and FilledBoundary Plots

The Boundary plot and FilledBoundary plot are discussed together because of their similarity. Both plots concentrate on the boundaries between materials but each plot shows the boundary in a different way. The Boundary plot, shown

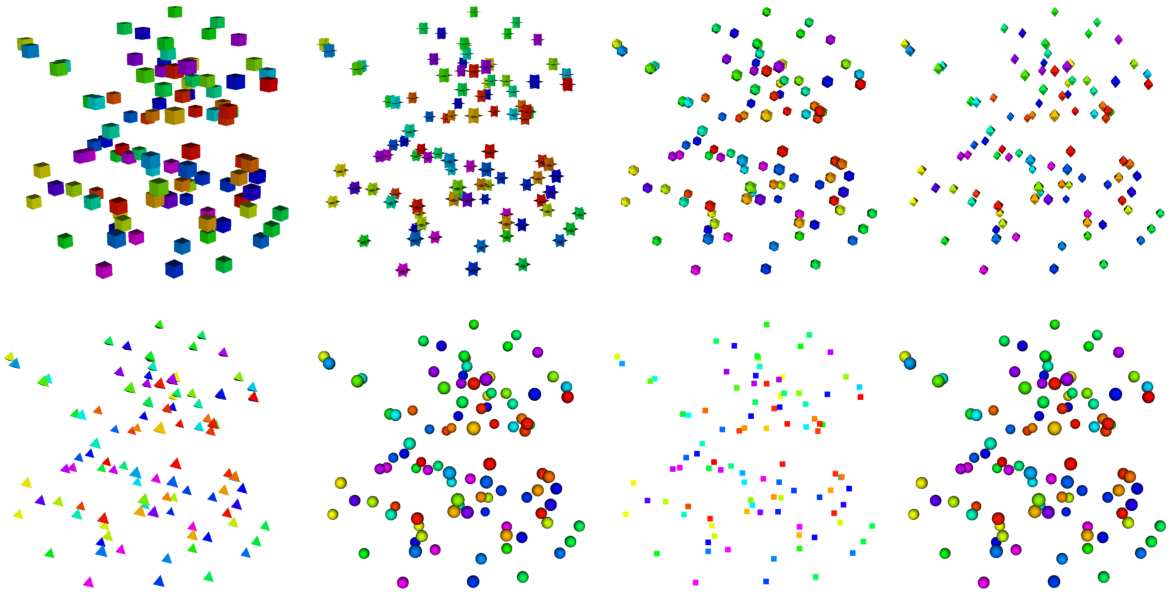


Fig. 4.20: Point types: Box, Axis, Icosahedron, Octahedron, Tetrahedron, Sphere Geometry, Point, Sphere

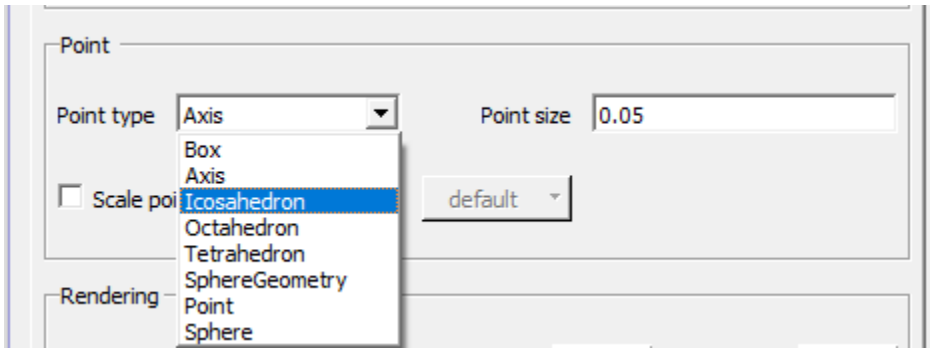


Fig. 4.21: Point type menu, expanded

in Figure 4.22, displays the surface or lines that separate materials.

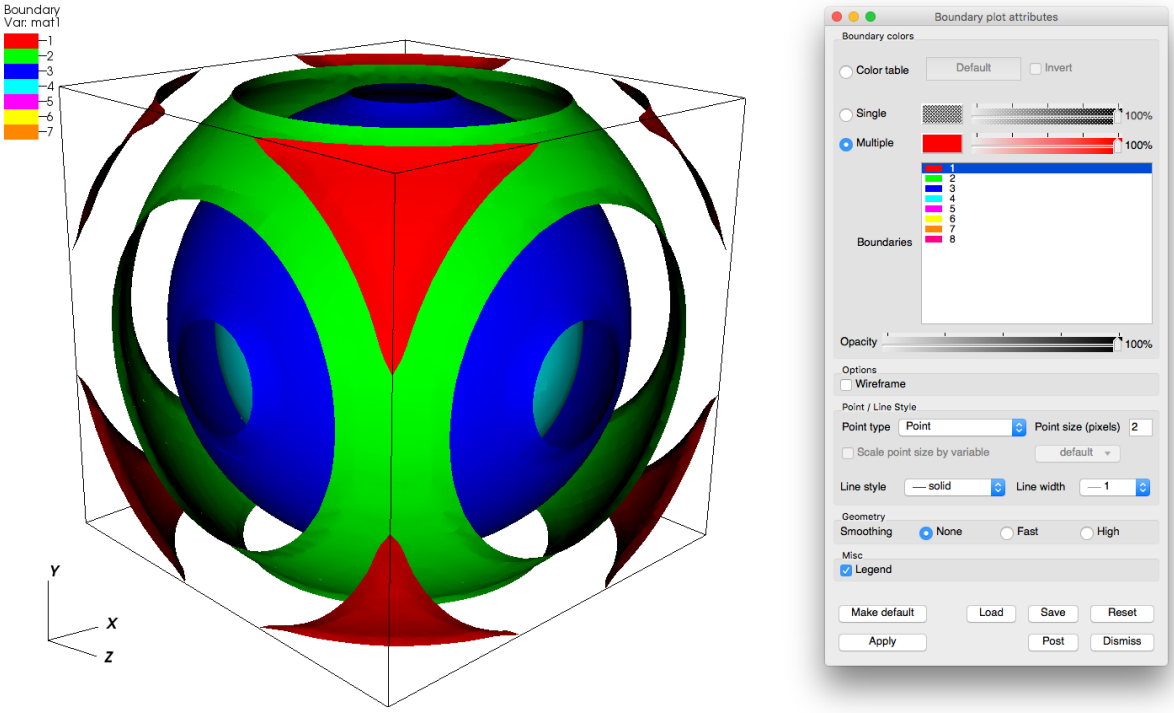


Fig. 4.22: Boundary plot and its plot attributes window

The FilledBoundary plot (see Figure 4.23) shows the entire set of materials, each using a different color. Both plots perform material interface reconstruction on materials that have mixed cells, resulting in the material boundaries used in the plots.

Combining the FilledBoundary plot with subsets (see Figure 4.24) can provide a insight into where each material is inside the mesh by turning off materials in a particular domain. For more information about subsets, see the **Subsetting** chapter. .

Changing colors

The main portion of the **Boundary plot attributes window** and **FilledBoundary plot attributes window**, also known as the **Boundary colors area**, is devoted to setting material boundary colors. The **Boundary colors area** contains a list of material names with an associated material color. Boundary plot and FilledBoundary plot colors can be assigned three different ways, the first of which uses a color table. A color table is a named palette of colors that you can customize to suite your needs. When the Boundary plot or FilledBoundary plot use a color table to color subsets, they selects colors that are evenly spaced through the color table based on the number of subsets. For example, if you have three materials and you are coloring them using the "xray" color table, three colors are picked out of the color table so your material boundaries are colored black, gray, and white. To color a Boundary plot or FilledBoundary plot with a color table, click on the **Color table radio button** and choose a color table from the **Color table menu** to right of the **Color table radio button**.

If you want all subsets to be the same color, click the **Single** radio button at the top of the **Boundary plot attributes window** and select a new color from the **Popup color menu** that is activated by clicking on the **Single color button**. The opacity slider next to the **Single color button** sets the opacity for the single color.

Clicking the **Multiple** radio button causes each material boundary to be a different, user-specified color. By default, multiple colors are set using the colors of the discrete color table that is active when the Boundary or FilledBoundary

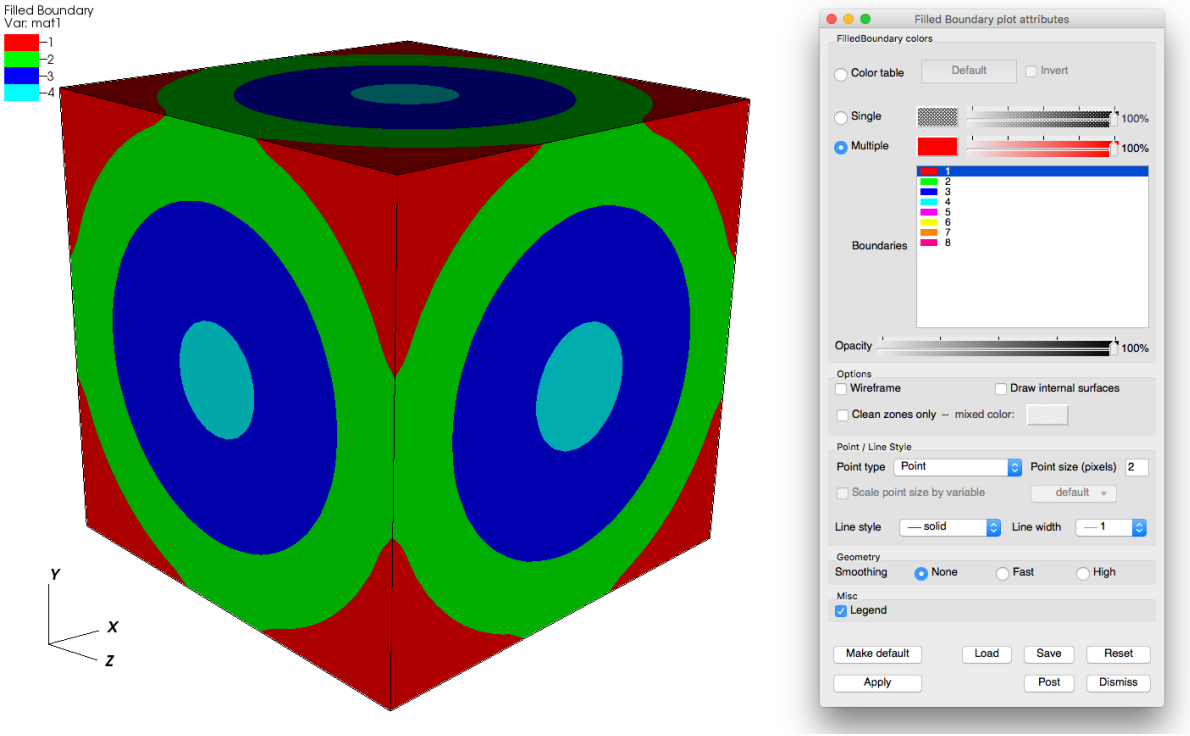


Fig. 4.23: FilledBoundary plot and its plot attributes window

plot is created. To change the color for any of the materials, select one or more materials from the list of materials and click on the **Color button** to the right of the **Multiple** radio button and select a new color from the **Popup color menu**. To change the opacity for a material, move **Multiple** opacity slider to the left to make the material more transparent or move the slider to the right to make the material more opaque.

The **Boundary plot attributes window** contains a list of material names with an associated color. To change a material's color, select one or more materials from the list, click the color button and select a new color from the popup color menu.

Opacity

The Boundary plot's opacity can be changed globally as well as on a per material basis. To change material opacity, first select one or more materials in the list and move the opacity slider next to the color button. Moving the opacity slider to the left makes the selected materials more transparent and moving the slider to the right makes the selected materials more opaque. To change the entire plot's opacity globally, use the **Opacity** slider near the bottom of the window.

Wireframe mode

The Boundary plot and the FilledBoundary plot can be modified so that they only display outer edges of material boundaries. This option usually leaves lines that give only the rough shape of materials and where they join other materials as seen in. To make the Boundary or FilledBoundary plots display in wireframe mode, check the **Wireframe** check box near the bottom of the window.

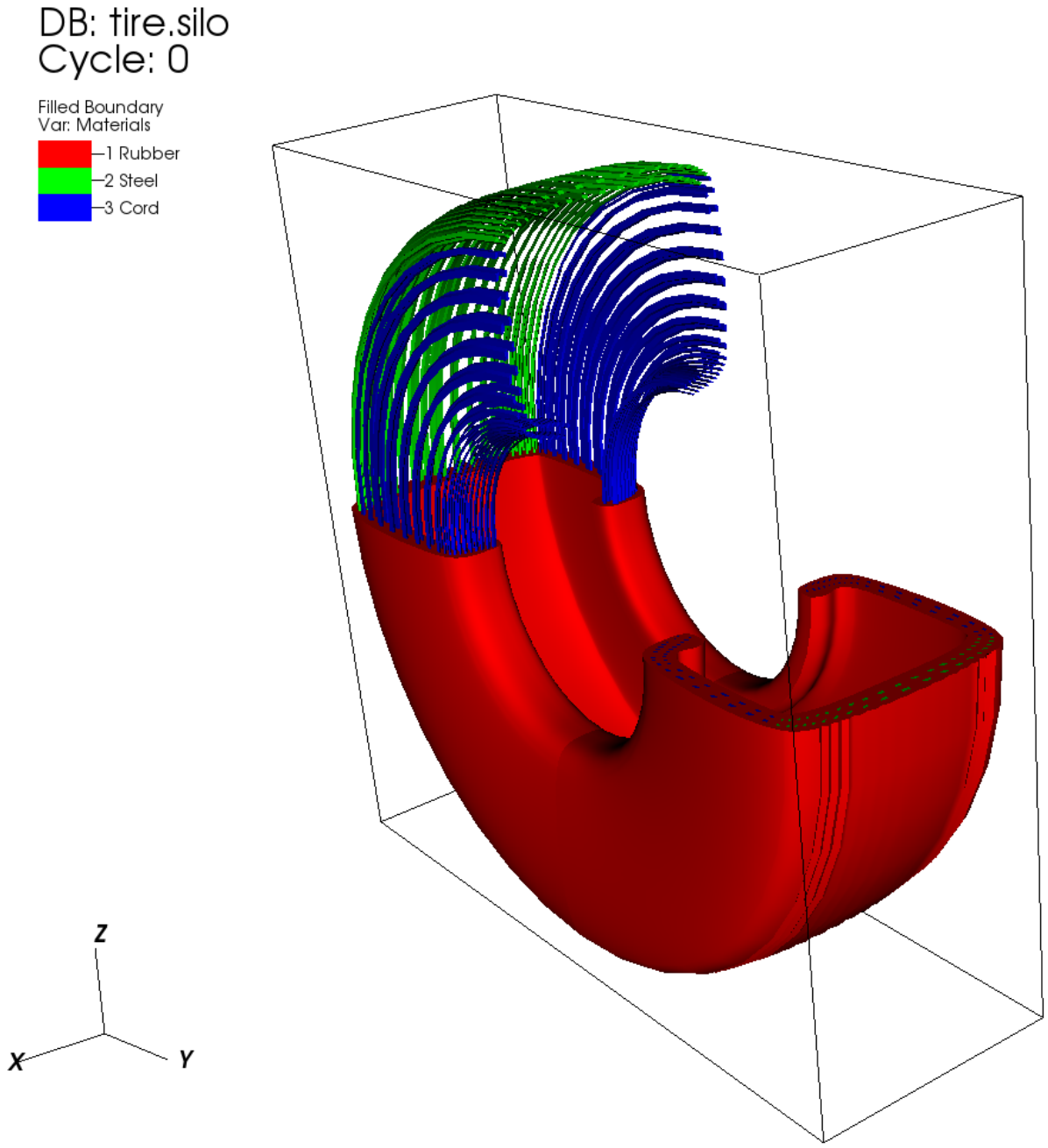


Fig. 4.24: FilledBoundary plot combined with subsets

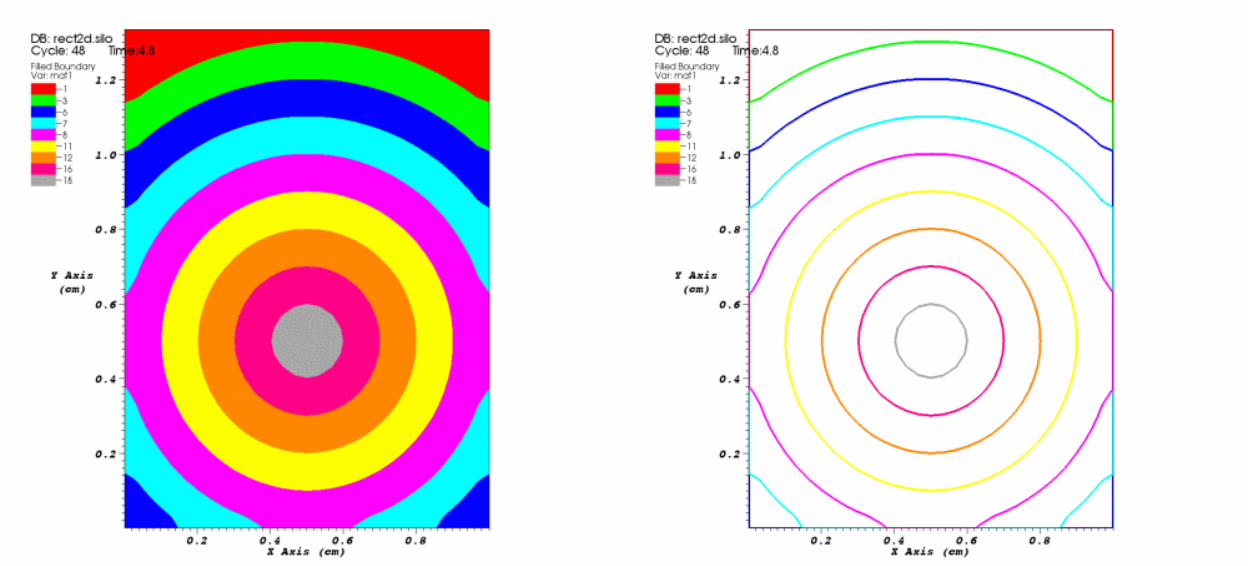


Fig. 4.25: Filled mode and wireframe mode

Geometry smoothing

Sometimes visualization operations such as material interface reconstruction can alter mesh surfaces so they are pointy or distorted. The Boundary plot and the FilledBoundary plot provide an optional Geometry smoothing option to smooth out the mesh surfaces so they look better when the plots are visualized. Geometry smoothing is not done by default, you must click the **Fast** or **High** radio buttons to enable it. The **Fast** geometry smoothing setting smooths out the geometry a little while the **High** setting works produces smoother surfaces.

Drawing only clean zones

The FilledBoundary plot, since it deals almost exclusively with plotting materials, has an option to only draw clean zones, which are zones that contain a single material. When only clean zones are drawn, all clean cells are drawn normally but all zones that contained more than one material are drawn with a color that can be set to match the vis window's background color (see). Drawing clean zones is primarily used to examine how materials mix in 2D databases. To make VisIt draw only the clean zones, click the **Clean zones only** check box. After that, you can set the mixed color by clicking on the **Mixed color** color button and selecting a new color from the popup color palette.

Setting point properties

Albeit rare, the Boundary and FilledBoundary plots can be used to plot points that belong to different materials. Both plots provide controls that allow you to set the representation and size of the points. You can change the points' representation using the different **Point Type** radio buttons. The available options are:

- **Box**
- **Axis**
- **Icosahedron**
- **Octahedron**

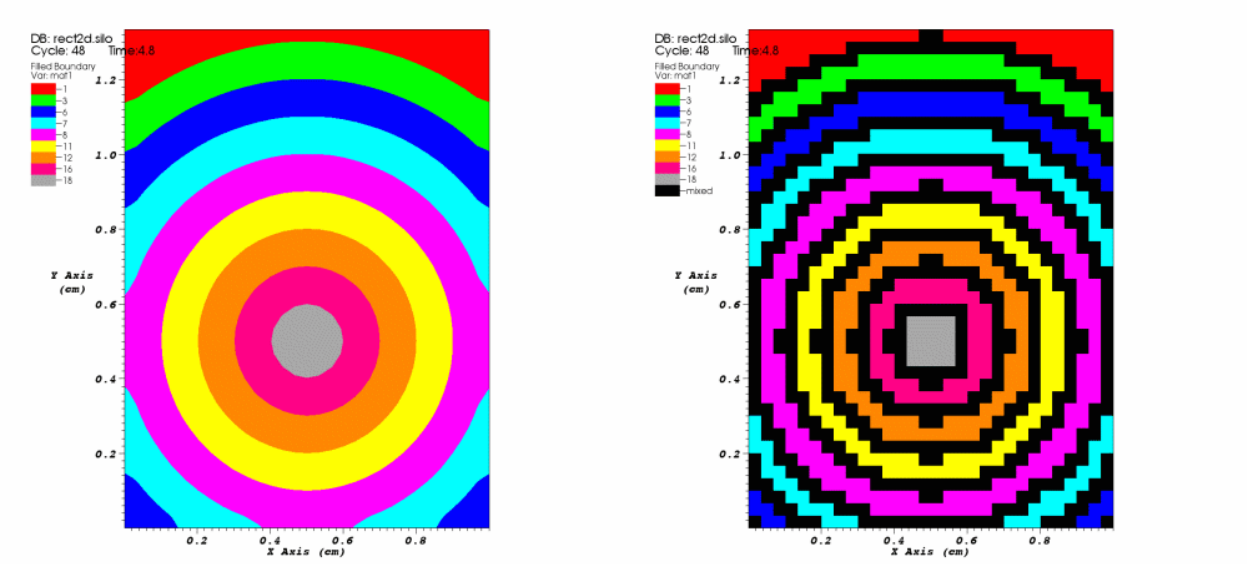


Fig. 4.26: All zones and clean zones

- **Tetrahedron**
- **Point**
- **Sphere**

The default point type is **Point** because that is the fastest to draw, followed by **Sphere**. The other point types create additional geometry and can take longer to appear on the screen and subsequently draw. To change the size of the points when the point type is set to **Box**, **Axis**, or **Icosahedron**, you can enter a new floating point value into the **Point size** text field. When the point type is set to **Point** or **Sphere**, the **Point size** text field becomes the **Point size (pixels)** text field and you should enter your point size in terms of pixels. Finally, you can opt to scale the points' glyphs using a scalar expression by turning on the **Scale point size by variable** check box and by selecting a scalar variable from the **Variable** button to the right of that check box. Note that point scaling does not occur when the point type is set to **Point** or **Sphere**.

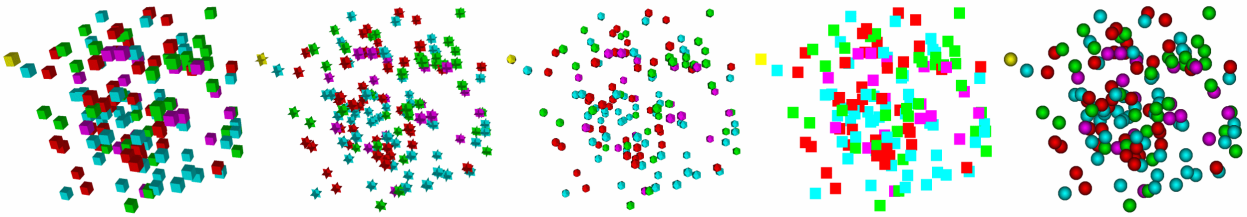


Fig. 4.27: Point types (left-to-right): Box, Axis, Icosahedron, Point, Sphere

Contour Plot

This plot, shown in Figure 4.28, displays the location of values for scalar variables like density or pressure using lines for 2D plots and surfaces for 3D plots. In visualization terms, these plots are isosurfaces. VisIt's Contour plot allows you to specify the number of contours to display as well as the colors and opacities of the contours.

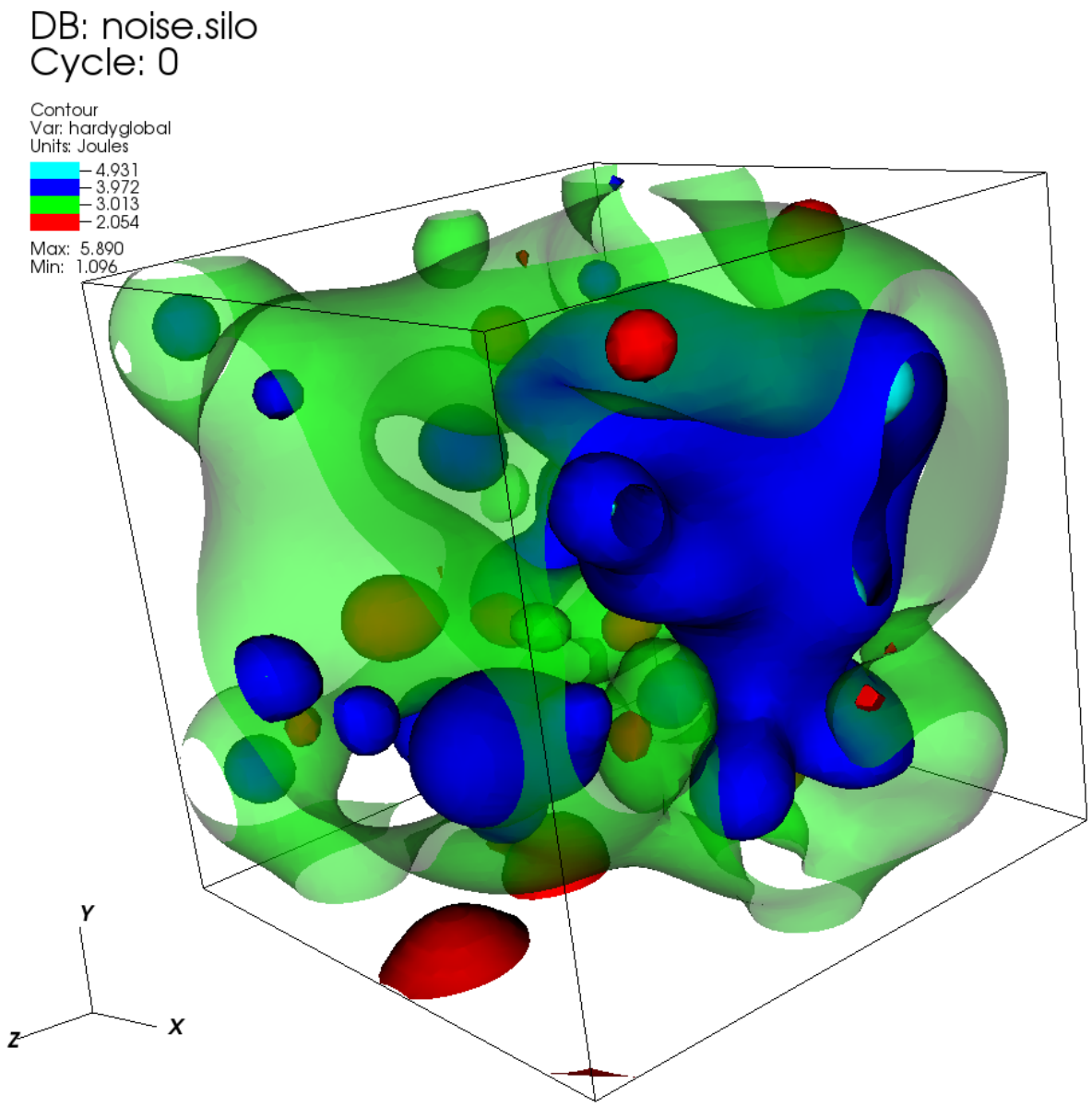


Fig. 4.28: Example of Contour plot

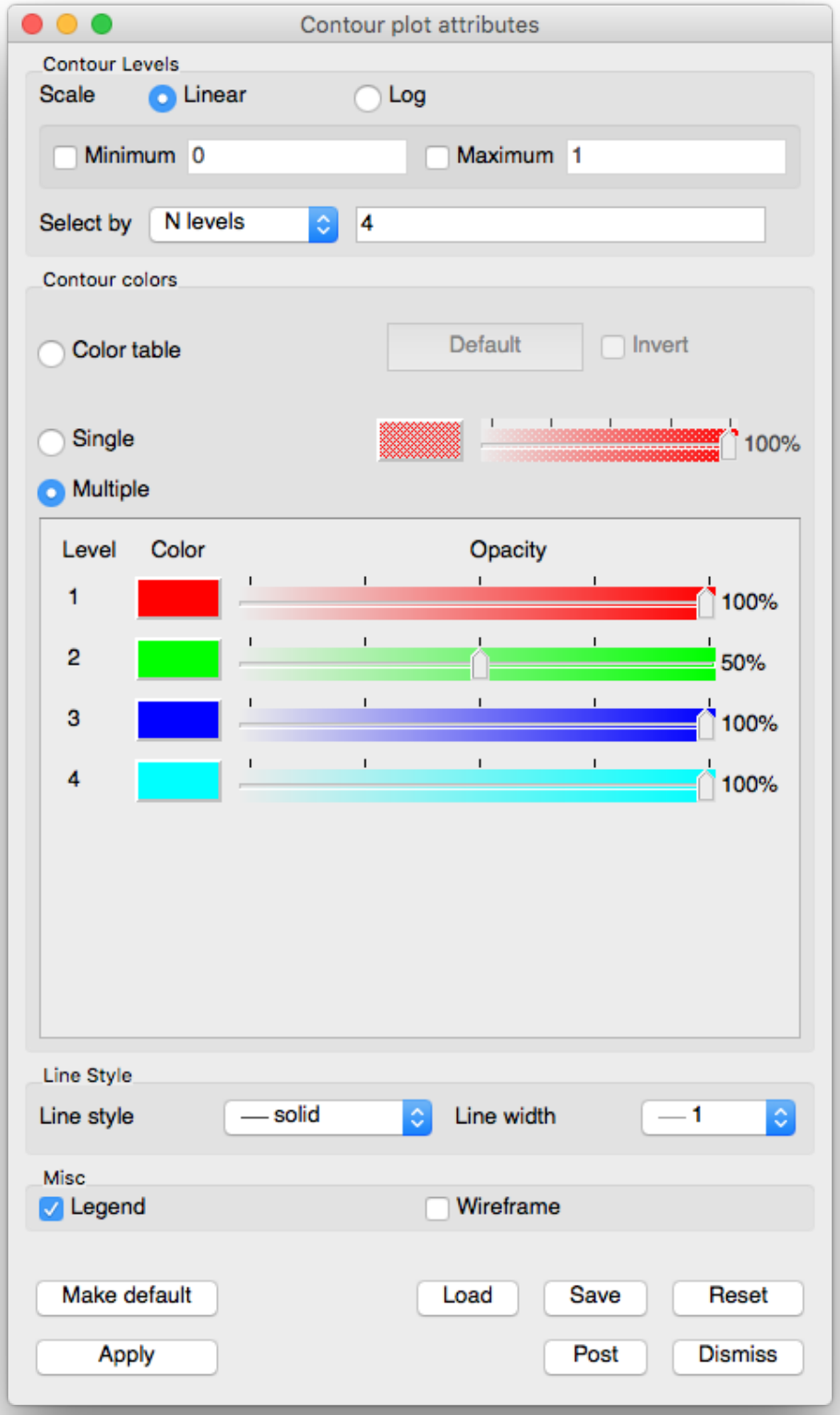


Fig. 4.29: **Contour plot attributes window**

Setting the number of contours

By default, VisIt constructs 10 levels into which the data fall. These levels are linearly interpolated values between the data minimum and data maximum. However, you can set your own number of levels, specify the levels you want to see or indicate the percentages for the levels.

To choose how levels are specified, make a selection from the **Select by** menu. The available options are: **N levels, Levels**, and **Percent**. **N levels**, the default method, allows you to specify the number of levels which will be generated, with 10 being the default. **Levels** requires you to specify floating point numbers for the levels you want to see. **Percent** takes a list of percentages like 50.5, 60, and 40.0. Using the numbers just mentioned, the first contour would be placed at the value which is 50.5% of the way between the minimum and maximum data values. The next contour would be placed at the value which is 60% of the way between the minimum and maximum data values, and so forth. You specify all values for setting the number of contours by typing into the text field to the right of the **Select by** menu.

Setting Limits

The **Contour plot attributes window** provides controls that allow you to specify artificial minima and maxima for the data in the plot. This is useful when you have a small range of values that are of interest and you only want the contours to be generated through that range. To set the minimum value, click the **Min** check box to enable the **Min** text field and then type a new minimum value into the text field. To set the maximum value, click the **Max** check box to enable the **Max** text field and then type a new maximum value into the text field. Note that either the min, max or both can be specified. If neither minimum nor maximum values are specified, VisIt uses the minimum and maximum values in the database.

Scaling

The Contour plot typically creates contours through a range of values by linearly interpolating to the next value. You can also change the scale to a logarithmic function to get the list of contour values through the specified range. To change the scale, click either the **Linear** or **Log** radio buttons in the **Contour plot attributes window**.

Setting contour colors

The main portion of the **Contour plot attributes window**, also known as the **Contour colors area**, is devoted to setting contour colors. Contour plot colors can be assigned three different ways, the first of which uses a color table. A color table is a named palette of colors that you can customize to suite your needs. When the Contour plot uses a color table to color the levels, it selects colors that are evenly spaced through the color table based on the number of levels. For example, if you have five levels and you are coloring them using the "rainbow" color table, the Contour plot picks five colors out of the color table so your levels are colored magenta, blue, cyan, green, yellow, and red. The colors change when increasing or decreasing the number of levels when you use a color table because VisIt uses the new number of levels to sample different locations in the color table. As a rule, increasing the number of levels results in coloration that is closer to the color table because more colors from the color table are represented. To color a Contour plot with a color table, click on the **Color table radio button** and choose a color table from the **Color table menu** to right of the **Color table radio button**.

If you want all levels to be the same color, click the **Single** radio button at the top of the **Contour plot attributes window** and select a new color from the **Popup color menu** that is activated by clicking on the **Single color button**. The opacity slider next to the **Single **color button** sets the opacity for the single color.

Clicking the **Multiple** radio button causes each level to be a different, user-specified color. By default, multiple colors are set using the colors of the discrete color table that is active when the Contour plot is created. To change the color for any of the levels, click on the level's **Color button** and select a new color from the **Popup color menu**. To change

the opacity for a level, move its opacity slider to the left to make the level more transparent or move the slider to the right to make the level more opaque.

Wireframe view

The **Contour plot attributes window** provides a **Wireframe** toggle button used to draw only the lines along the edges of the contour. This option only has an effect on 3D Contour plots.

Curve Plot

The Curve plot, shown in Figure 4.30, displays a simple group of X-Y pair data such as that output by 1D simulations or data produced by Lineouts of 2D or 3D datasets. Curve plots are useful for visualizations where it is useful to plot 1D quantities that evolve over time.

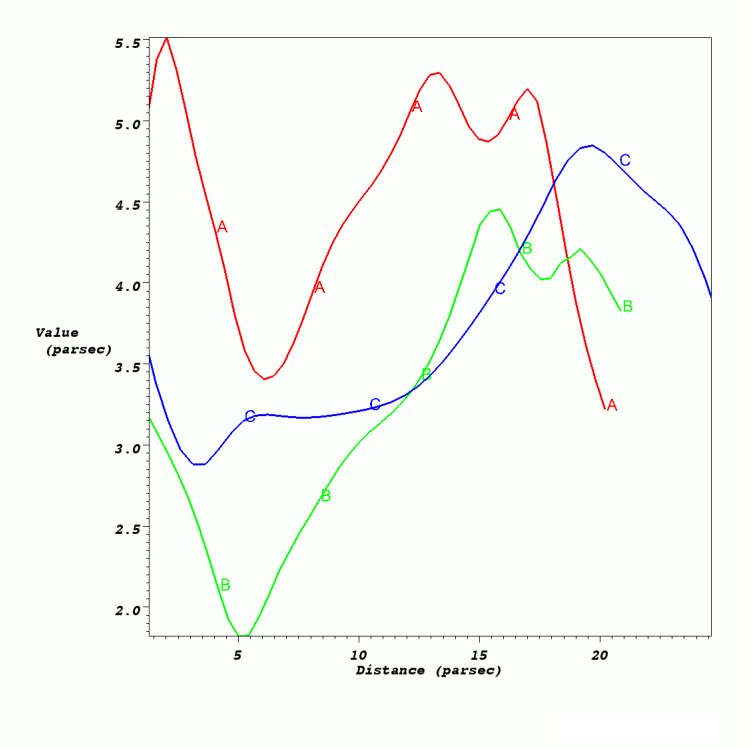


Fig. 4.30: Curve plot

Setting curve color

The Curve plot's color is set up to **Cycle** by default. In other words, each new curve created will be a different color. This can be turned off by selecting the **Custom** radio button, and a new color can be chosen by clicking on the **Color button** and making a selection from the **Popup color menu**.

Showing curve labels

Curve plots have a label that can be displayed to help distinguish one Curve plot from other Curve plots. Curve plot labels are on by default, but if you want to turn the label off, you can uncheck the **Labels** check box.

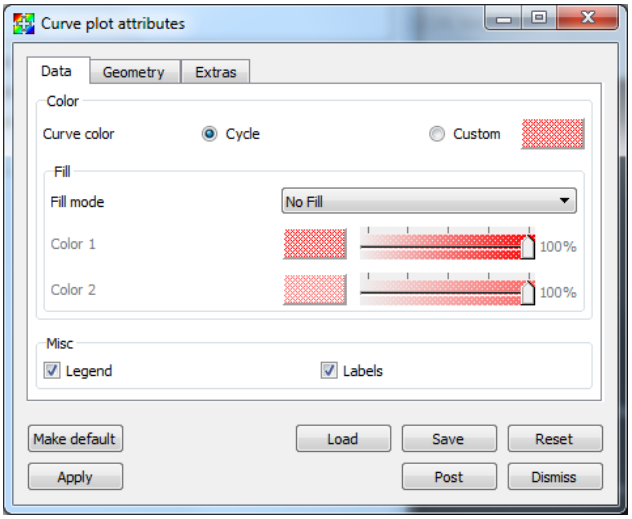


Fig. 4.31: Curve plot attributes, data tab

Space-filled curves

The space below a curve can be filled with color by changing **Fill mode** to either **Solid, Horizontal Gradient or Vertical Gradient**, then choosing one or two colors based upon the mode chosen.

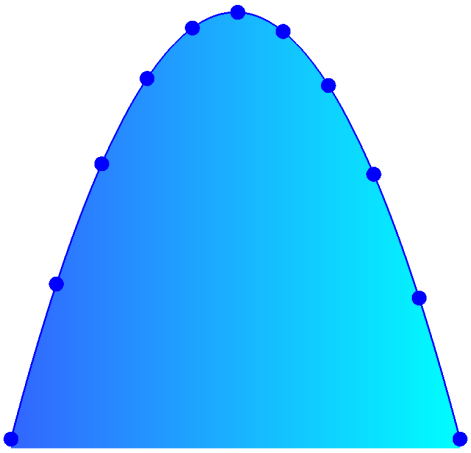


Fig. 4.32: Curve, space-filled with points

Setting line style and line width

Several Curve plots are often drawn in the same visualization window so it is necessary that Curve plots can be distinguished from each other. Fortunately, VisIt provides controls to change the line style and line width so that Curve plots can be told apart. Line style is a pattern used to draw the line and it is solid by default but it can also be dashed, dotted, or dash-dotted. You choose a new line style by making a selection from the **Line Style** combo box on the **Geometry tab** (see Figure 4.33). The line width, which determines the boldness of the curve, is set by making a

selection from the **Line Width** combo box.

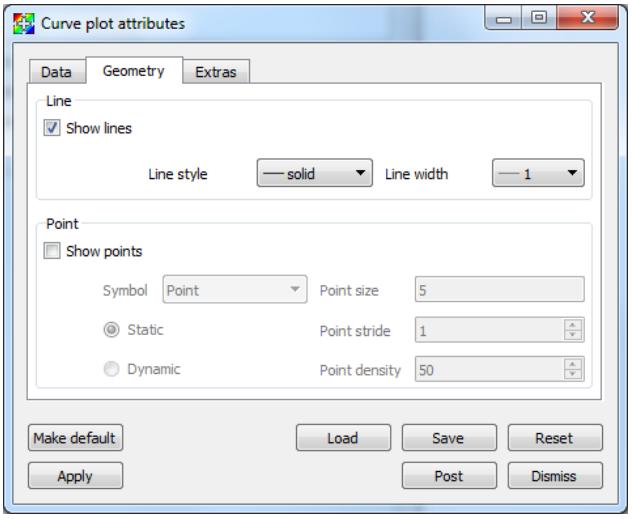


Fig. 4.33: Curve plot attributes, geometry tab

Drawing points on the Curve plot

The Curve plot is composed of a set of (X,Y) pairs through which line segments are drawn to form a curve. To make VisIt draw a point glyph at the location of each (X,Y) point, click the **Show points** check box on the **Geometry** tab. You can control the size of the points by typing a new point size into the **Point size** text field. You can choose the type of symbol used to represent the points by using the **Symbol** combo box.

The number of points drawn can be controlled by the **Static** or **Dynamic** radio buttons. For **Static** mode, points are drawn at regular intervals controlled by the value of the **Point stride** text box. For **Dynamic** mode, the number of points drawn is view-dependent, with density controlled by the **Point density** text box.

Adding Time Cues

Time cues are most often used in conjunction with movie making. They allow for markers to be placed at certain positions along a curve, and/or for the curve to be cropped at the specified position. Time cues make it easier to see the current time position along a curve. Though most often created and controlled via scripting, the **Extras** tab in the **Curve attributes** window can also be used (see Figure 4.34). There are two types of markers: Ball and Line. They are controlled by the **Add Ball** and **Add Line** check boxes. They have separate color and size controls. To crop the line, select the **Crop** check box. The **Position of cue** text box controls the location along the curve where the ball and line are placed and where the cropped curve ends. Figure 4.35 shows examples of curves created using different time cue settings.

Polar coordinate system conversion

If the curve data is in Polar instead of Cartesian coordinates, you can tell VisIt to convert by selecting the **Polar to Cartesian** option on the **Extras** tab. You can choose the **Order** to be **R_Theta** or **Theta_R** and choose **Radians** or **Degrees** for the **Units**. Figure 4.36 shows an example.

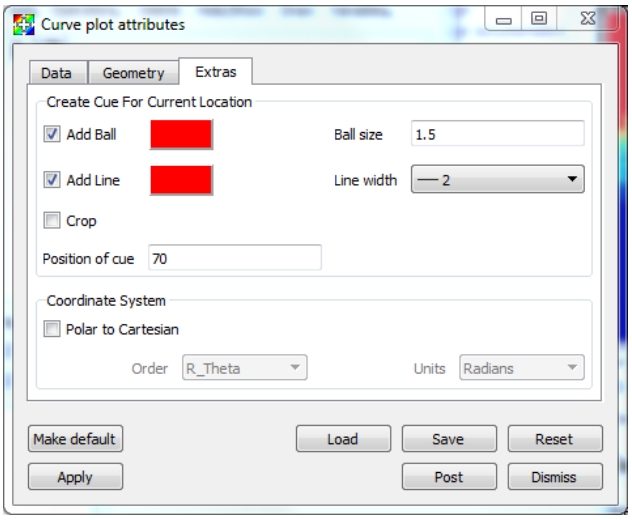


Fig. 4.34: Curve plot attributes, extras tab

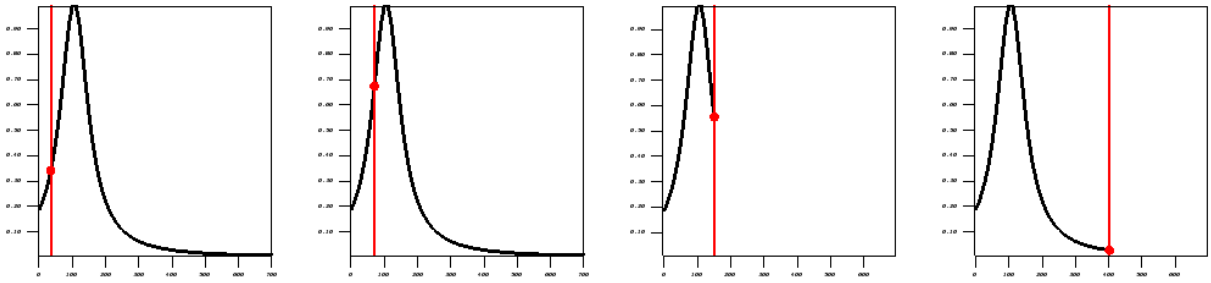


Fig. 4.35: Curve plot with time cues added at different positions, both uncropped and cropped.

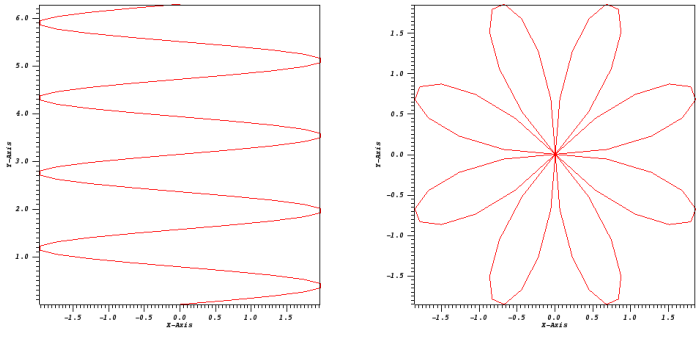


Fig. 4.36: Curve plot before and after Polar coordinate transform (R-theta, radians)

Histogram Plot

The Histogram plot divides the data range of a scalar variable into a number of bins and groups the variable's values into different bins. The values can be based on frequency, they can be weighed by the area/volume of the cells, or they can be weighed by a variable. The values in each bin are then used to create a bar graph or curve that represents the distribution of values throughout the variable's data range. The Histogram plot can be used to determine where data values cluster in the range of a scalar variable. The Histogram plot is shown in Figure 4.37.

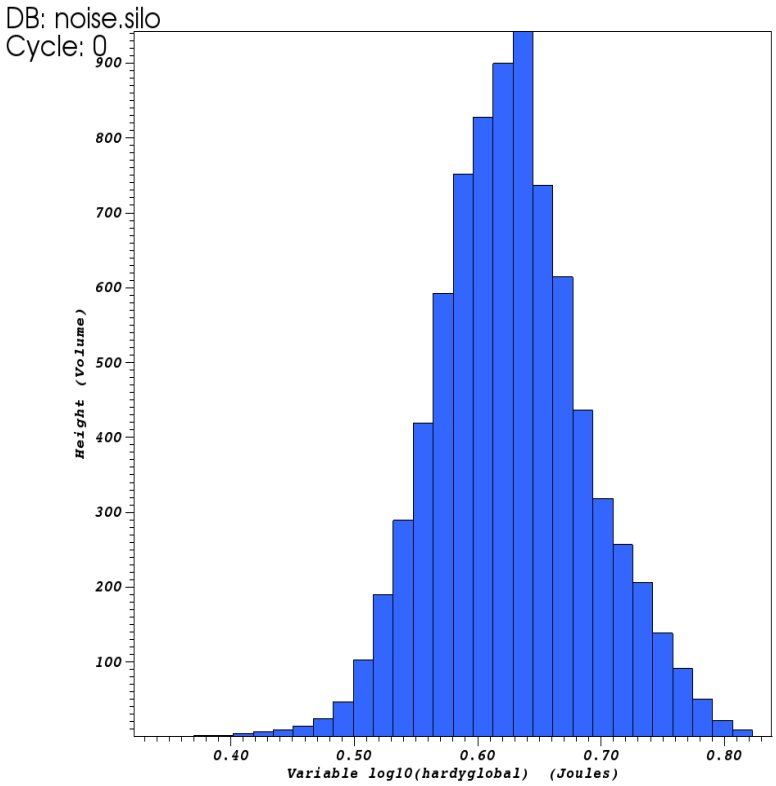


Fig. 4.37: Histogram plot

Setting the histogram data range

By default, the Histogram plot profiles a variables entire data range. If you want to restrict the Histogram plot so it only takes a subset of a variable's data range into consideration when assigning values to bins, you can set the minimum and maximum values that will be considered by the Histogram plot. To specify a data range, click the **Minimum** and/or **Maximum** check box and then type in floating point numeric values into the **Minimum** and **Maximum** text fields in the **Histogram plot attributes window** (see Figure 4.38) before clicking the **Apply** button. Once the data range is set, the Histogram plot will restrict the values that it considers to the specified data range.

Setting the type of graph

The Histogram plot has two mode in which it can appear: curve and block. When the Histogram plot is drawn as a curve, it looks like the Curve plot. When the Histogram plot is drawn in block mode, it is drawn as a bar graph where each bin is plotted along the X-axis and the height of each bar corresponds to the number of values that were assigned to that bin. You can set change the Histogram plot's appearance by clicking the **Curve** or **Block** radio buttons.

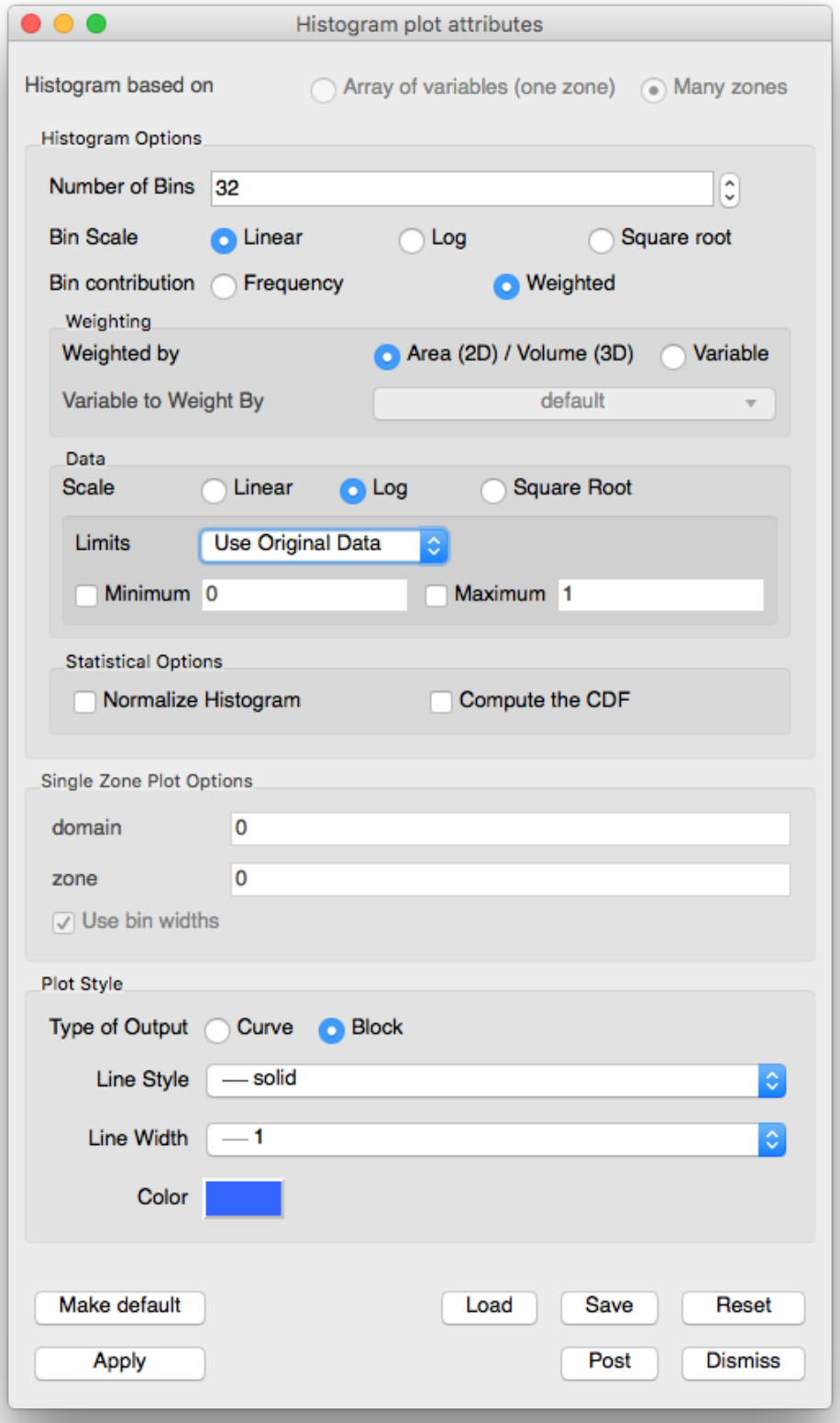


Fig. 4.38: Histogram attributes

Setting the number of bins

The Histogram plot divides a variable's data range into a number of bins and then counts the values that fall within each bin. The bins and the counted data are then used to create a graph that represents the distribution of data within the variable's data range. As the Histogram plot uses more bins, the graph of data distribution becomes more accurate. However, the graph can also become rougher because as the number of bins increases, the likelihood that no data values fall within a particular bin also increases. To set the number of bins for the Histogram plot, type a new number of bins into the **Number of Bins** text field and click the **Apply** button in the **Histogram plot attributes window**.

Setting the histogram calculation method

The data values can be based on frequency, they can be weighed by the area/volume of the cells, or they can be weighed by a variable. By default, **Frequency** is selected under bin contribution. Selecting **Weighted** will enable the **Weighting** options, from which one can select **Area (2D) / Volume (3D)** or **Variable** to determine the type of weighing.

Data scaling

There are three radio buttons that controls how the data values are scaled. The three options are:

- **Linear**: no scaling is applied. This is the default option.
- **Log**: the logarithms of all the scalars are binned.
- **Square Root**: the square roots of all scalars are binned.

Statistical Options

The Histogram binning results can be processed further to calculate statistical distributions. The **Statistical Options** controls support this:

- **Normalize Histogram**: Will bin the data and then divide each bin by the sum of the values across all bins, to create a Probability density function.
- **Compute the CDF**: Will bin the data, normalize it, and then create a Cumulative distribution function from the normalized binning.

Label Plot

The Label plot, shown in Figure 4.39, can display mesh information, scalar fields, vector fields, tensor fields, array variables, subset names, and material names. The Label plot is often used as a debugging device for simulation codes since it allows the user to see labels containing the exact values at the computational mesh's nodes or cell centers. Since the **Label** plot's job is to display labels representing the computational mesh or the fields defined on that mesh, it does not convey much information about the actual mesh geometry. Since having a **Label** plot by itself does not usually give enough information to understand the plotted dataset, the **Label** plot is almost always used with other plots.

Choosing the Label plot's variable

You can choose the **Label** plot's variable using the **Variable** menu under the **Plot list** the same way as you would with any other type of plot. One special property that distinguishes the **Label** plot from some of VisIt's other plots is that it

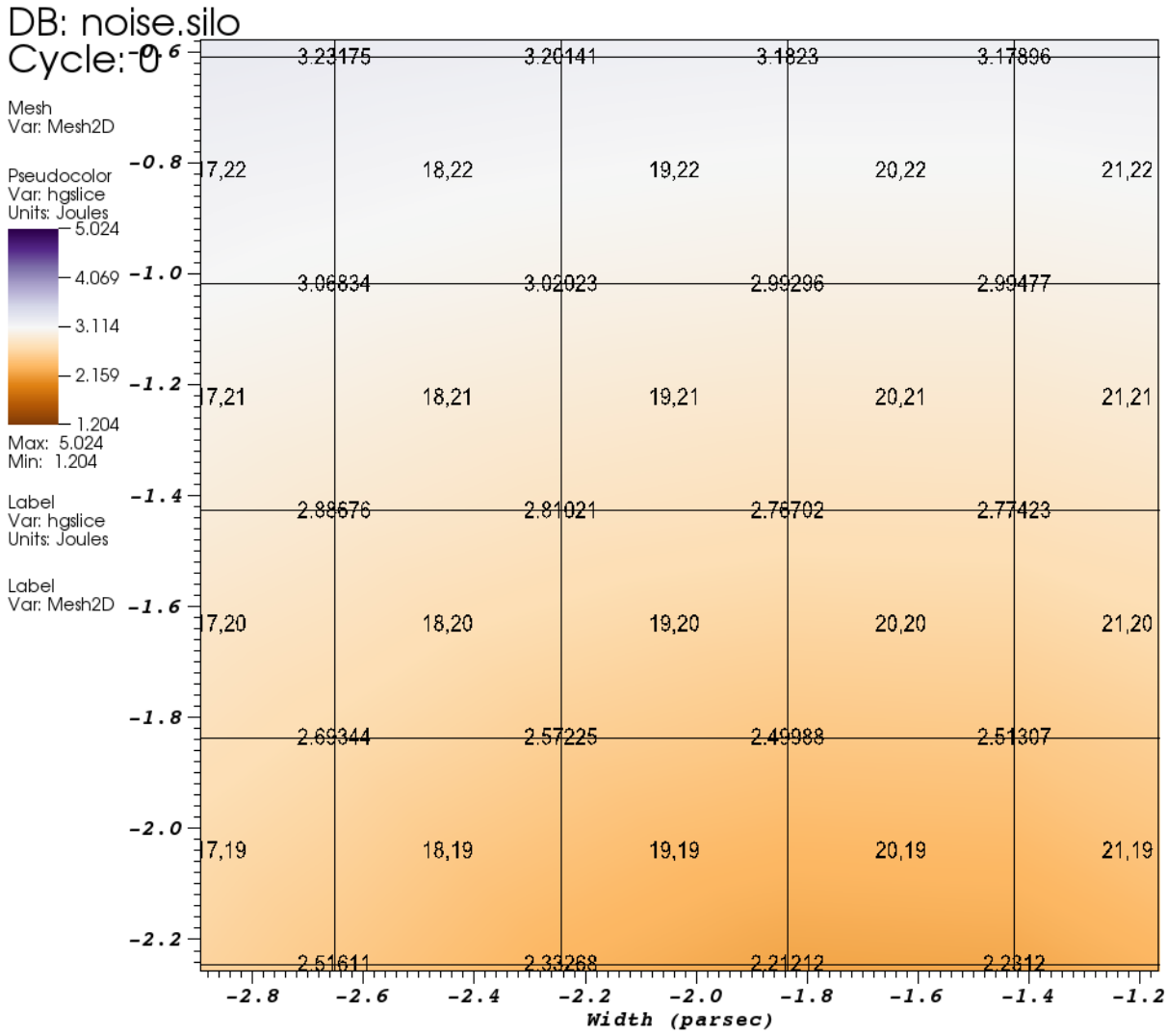


Fig. 4.39: Label plot of the mesh overlayed on Pseudocolor and Mesh plots

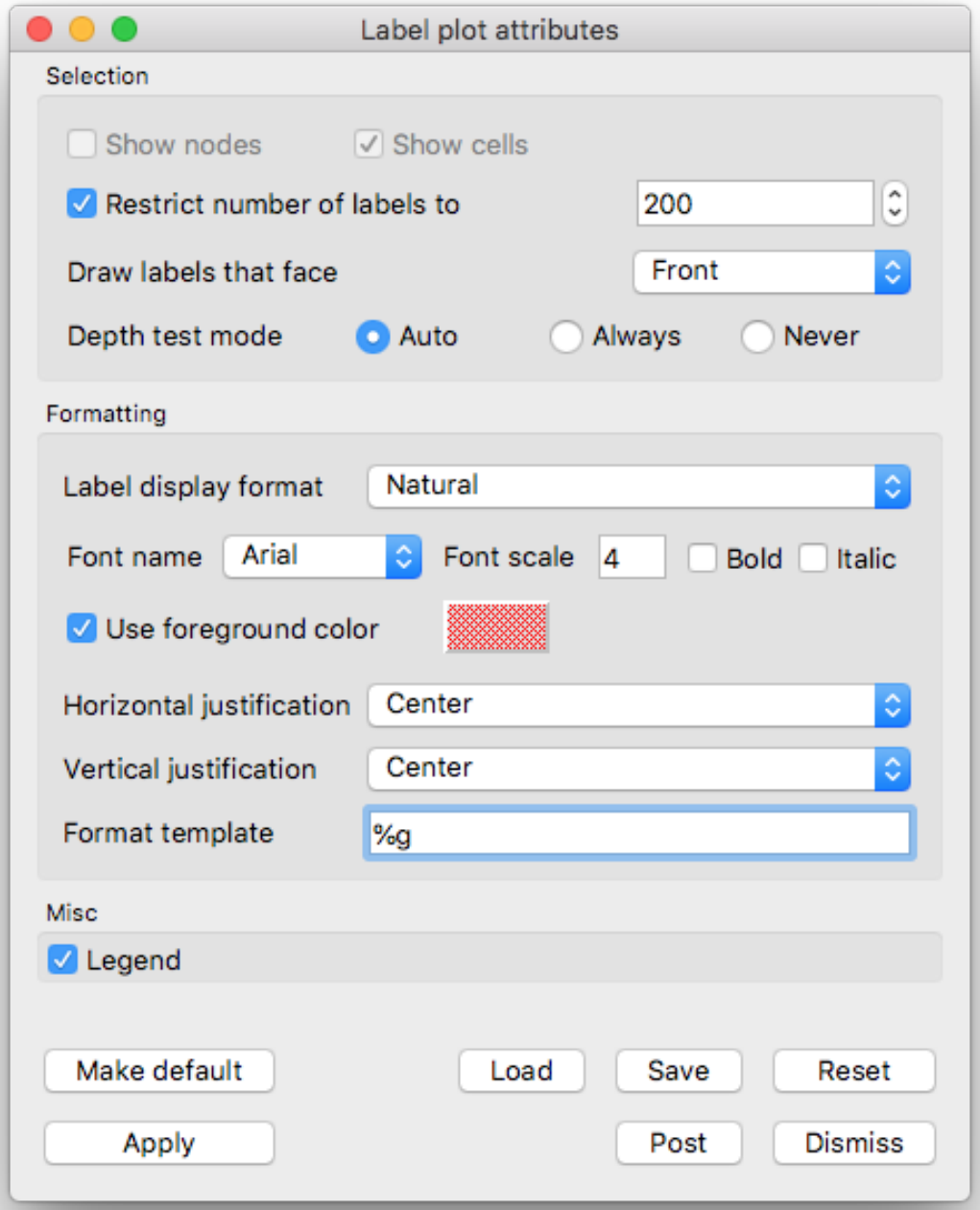


Fig. 4.40: Label plot attributes window

can plot multiple types of variables. The **Label** plot can display information for meshes, scalars, vectors, tensors, array variables, subsets, and materials so you will typically find more variables available for the **Label** plot than you would for other plots. When you choose a mesh variable for the **Label** plot, you can display both the mesh node numbers and cell numbers otherwise you are limited to displaying only the variable being plotted.

Showing node and zone numbers

The **Label** plot can display the node and cell numbers for the computational mesh if you have selected a mesh variable to plot. By default, the **Label** plot will display cell numbers only. The cell numbers will be displayed in the format most natural to the underlying mesh representation, which means that unstructured meshes will have cell numbers that are displayed as single integers while structured meshes will be displayed in i,j,k format when possible. If you want the **Label** plot to show a mesh's node numbers in addition to its cell numbers, you can click on the **Show nodes** check box. If you no longer want the **Label** plot to show the mesh's cell numbers, you can turn off the **Show cells** check box.

Restricting the number of labels

Most computational meshes contain many thousands, millions, or even billions of nodes and cells. Adding that many labels would quickly become burdensome on the computer and would result in a **Label** plot so dense that individual labels could no longer be read or even associated with their cell or node.

VisIt's **Label** plot restricts the number of labels by default to some user-settable number of labels that can comfortably fit on the screen. The method used to restrict the number of labels differs for 2D and 3D plots. For 2D plots, the viewable portion of world space is periodically subdivided, based on the zoom level, into some number of bins to which labels are then assigned. As you zoom in on the **Label** plot, labels that go beyond the viewport are no longer drawn and new labels that were previously hidden take their place. This allows the **Label** plot to efficiently draw many labels without crowding the labels on top of each other. For 3D plots, the **Label** plot divides up the screen into a user-settable number of bins. All label coordinates are transformed so that they can be assigned to a screen bin and the label wins the screen bin if it is closer than the label that was previously in the bin. This ensures that a small subset of all possible labels is drawn and that they do not usually overlap on the screen. If you find that the labels appear to be from the back of the mesh instead of from the front, it's quite possible that the normals generated for your mesh were inverted for some reason. To combat this problem, select **Back** or **Front or Back** from the **Draw labels that face** menu.

If you want to set the number of labels that the **Label** plot will draw, you can type in a new value into the spin box next to the **Restrict number of labels to** check box or use the up and down arrows on the spin box. If you want to force the **Label** plot to draw all labels, you can turn off the **Restrict number of labels to** check box. Sometimes making the **Label** plot draw all of the labels can be faster than drawing a subset of labels.

Depth testing for 3D Label plots

When VisIt draw plots in the visualization window, the plots' geometries often correspond to only the outer surfaces of the originating datasets when those datasets are 3D. This means that the majority of plots consist of convex geometry and the normal test for only drawing labels that face front is often adequate to remove any labels that appear on faces that point away from the current camera. Some plots have geometries that consist of many concave regions, which the afore-mentioned test does not handle well. Plots with concave geometries will often have various pieces be incorrectly visible because though the surfaces may face the camera, they may be obscured by other geometry. When VisIt's **Label** plot draws 3D geometry, it tries to enable additional depth testing to prevent front-facing labels in back of other surfaces from being drawn. Depth testing can degrade performance so, by default, it is allowed only when you are running VisIt on your local workstation. You can set the **Label** plot's depth test mode to tell VisIt when to enable depth testing. To change the values for the depth test mode, click on one of the **Auto**, **Always**, **Never** radio buttons to the right of the **Depth test mode** label. If VisIt wants to use depth testing but is not allowed to then a warning message will be issued and you can set the depth test mode to **Always**.

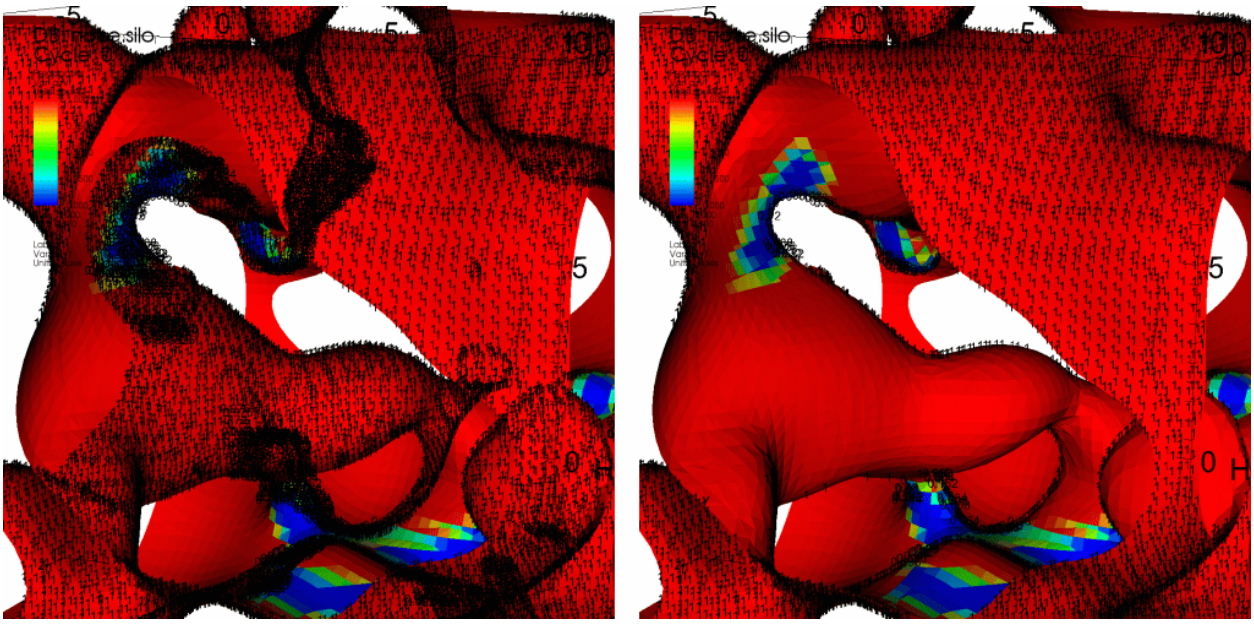


Fig. 4.41: Removing extra labels (left) with depth test (right)

Formatting labels

The **Label** plot provides several options for setting label format. First and foremost, you can set the label display format, which is how mesh node and cell numbers are displayed. By default, the **Label** plot will display labels in their most appropriate format with cell and node numbers for structured meshes displayed as logical i,j,k indices. Setting the label format is only possible for **Label** plots of structured meshes. To change the label format, select a new option from the **Label display format** menu.

The **Label** plot's default behavior is to use the vis window's foreground color but if you want labels to be a specific color, you can turn off the **Use foreground color** check box and select a new label color by clicking on the **Label color** color button.

The **Label** plot also allows control over the font used for the labels. **Font name** menu allows you to choose from among **Arial**, **Courier** and **Times** options. The labels can be **bold** or *italic* by checking the appropriate check boxes. **Font scale** is used to control the font size.

Note that when you are plotting a mesh variable, VisIt will make more controls in the **Label plot attributes window** so you can set color and font options for cells and nodes independently (see Figure 4.42).

Finally, the **Label plot attributes window** provides controls to determine the horizontal and vertical text justification used when drawing each label. To change the horizontal text justification, select a new value from the **Horizontal justification** menu. To change the vertical text justification, select a new value from the **Vertical justification** menu.

Labeling subset names and material names

The **Label** plot can label subset names and material names in addition to meshes and fields defined on those meshes. To add subset names or material names to your visualization, be sure to create a **Label** plot using a variable of either of those types. An example of a **Label** plot of material names is presented in Figure 4.44.

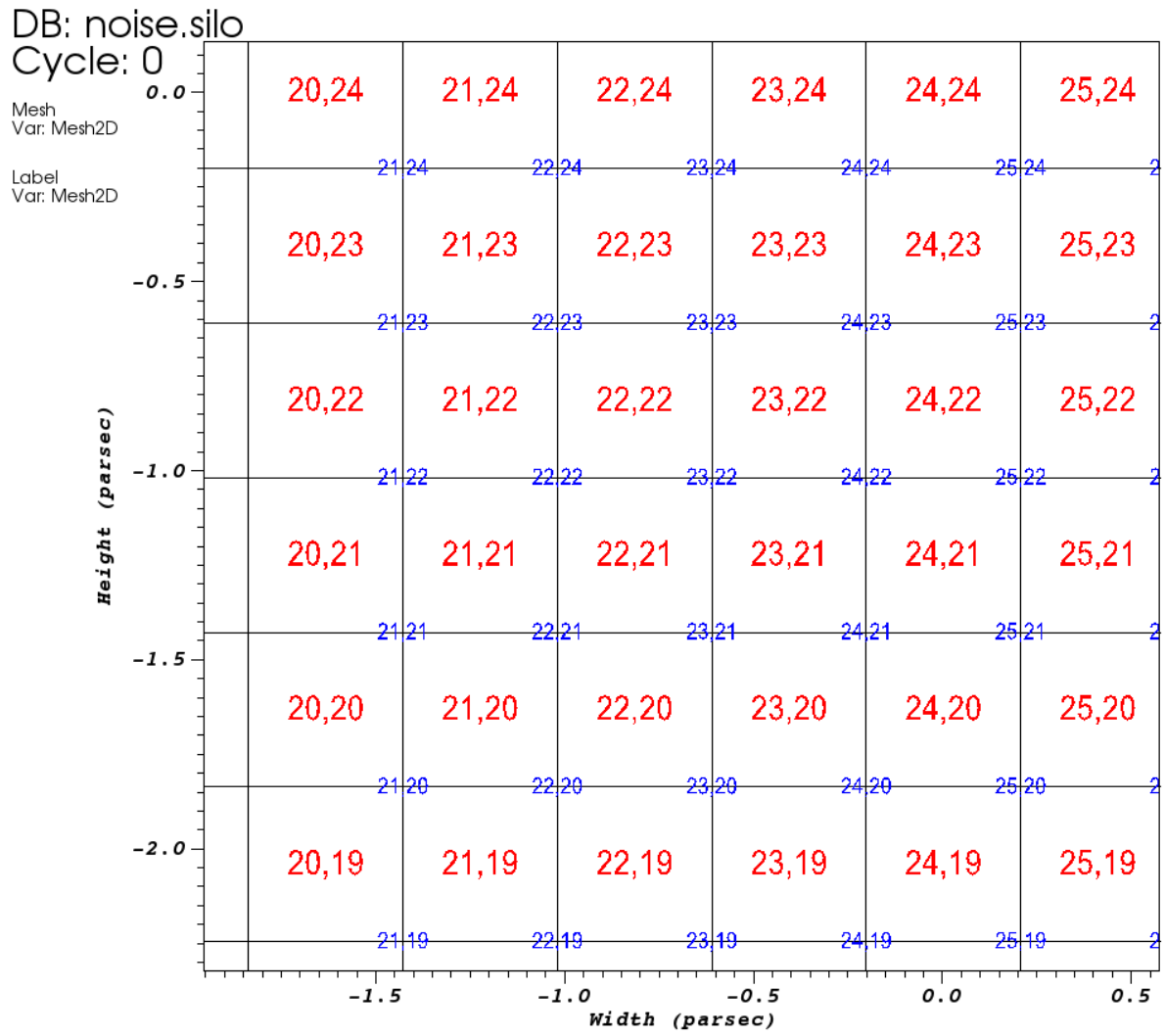


Fig. 4.42: Displaying cell and node labels with different colors

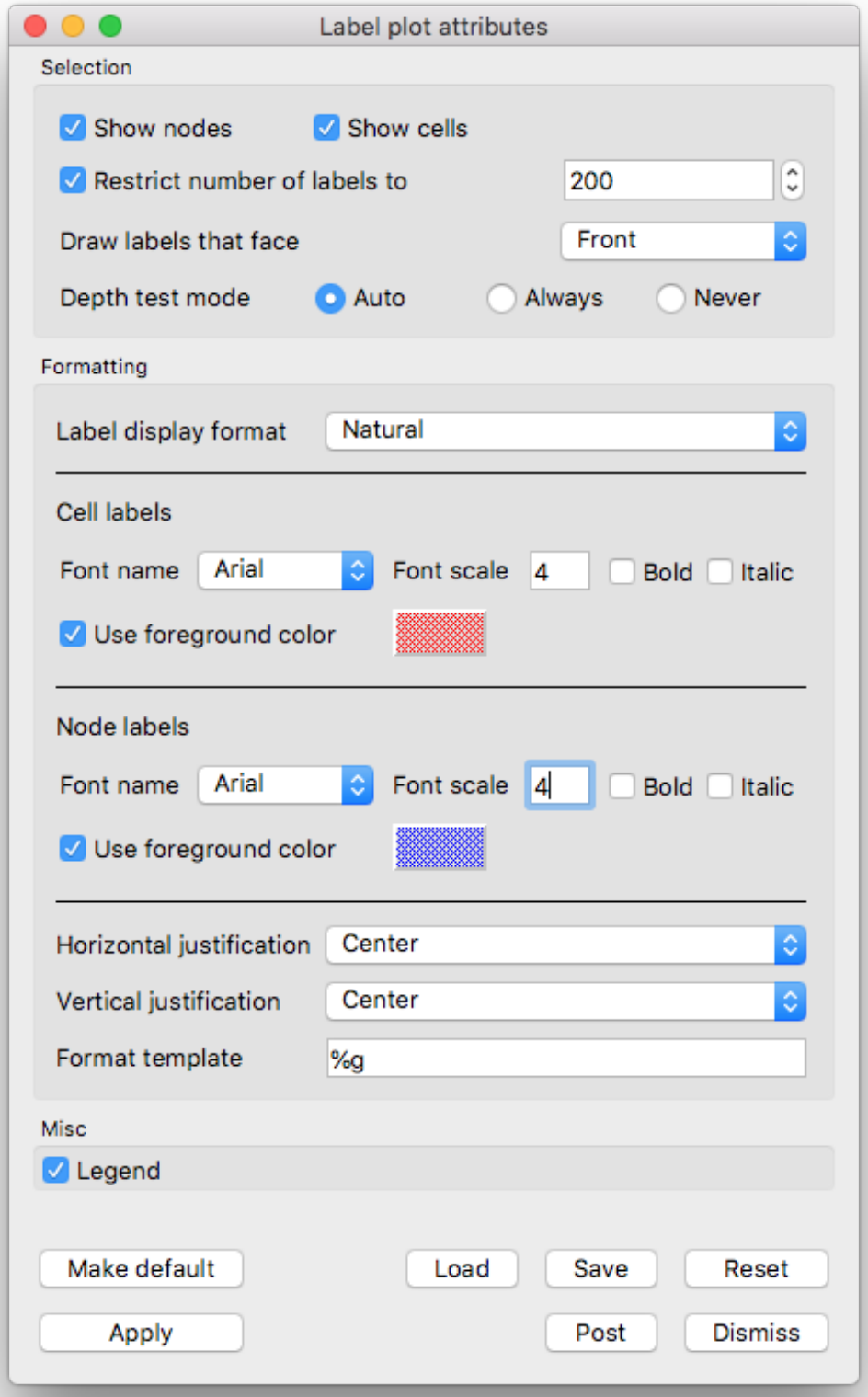


Fig. 4.43: The Label plot attributes window specifying different colors for cell and node labels

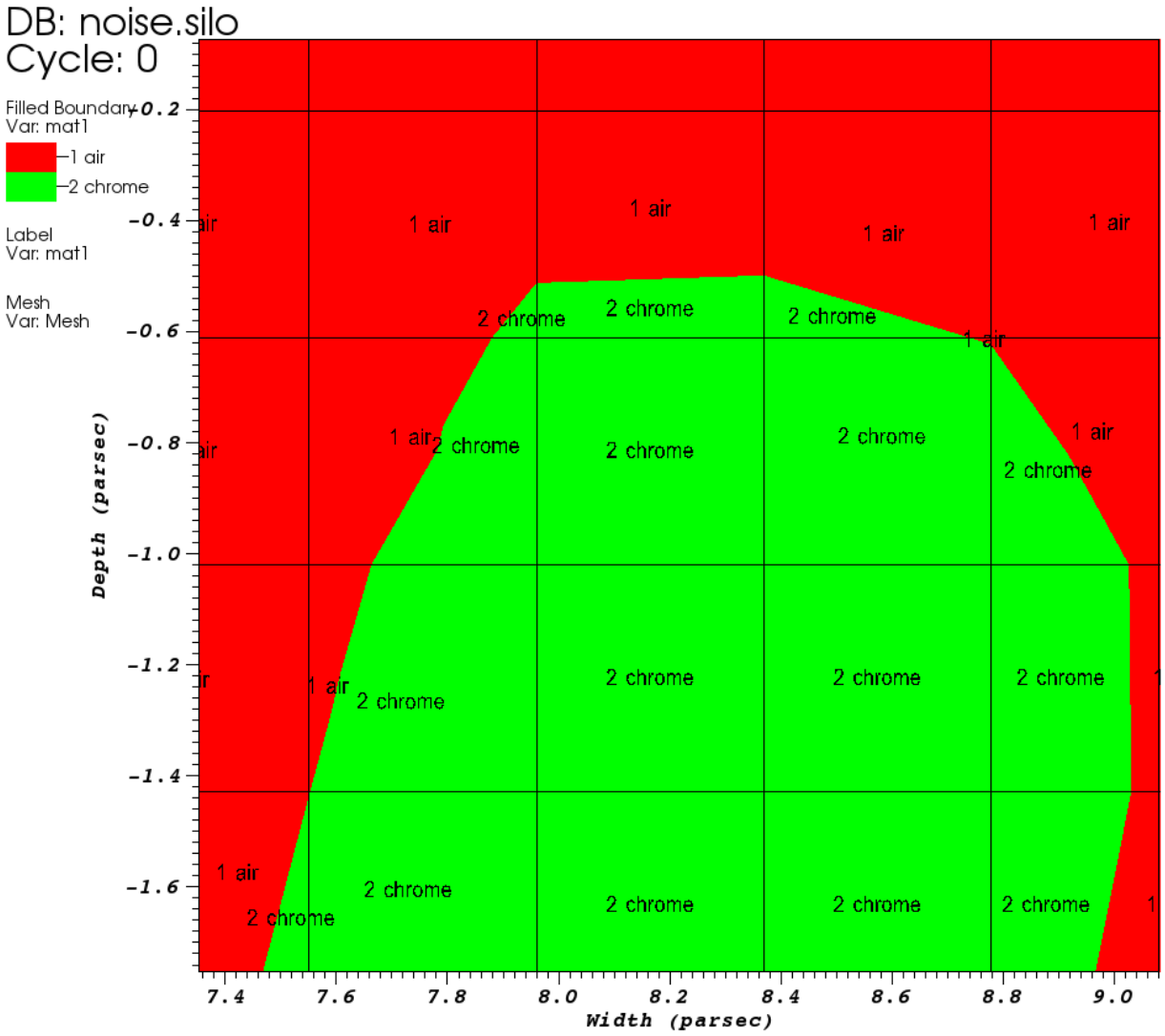


Fig. 4.44: Label plot of materials

Mesh Plot

The **Mesh** plot, shown in Figure 4.45, displays the computational mesh over which a database's variables are defined. The mesh plot is often added to the visualization window when other plots are visualized to allow individual cells to be clearly seen.

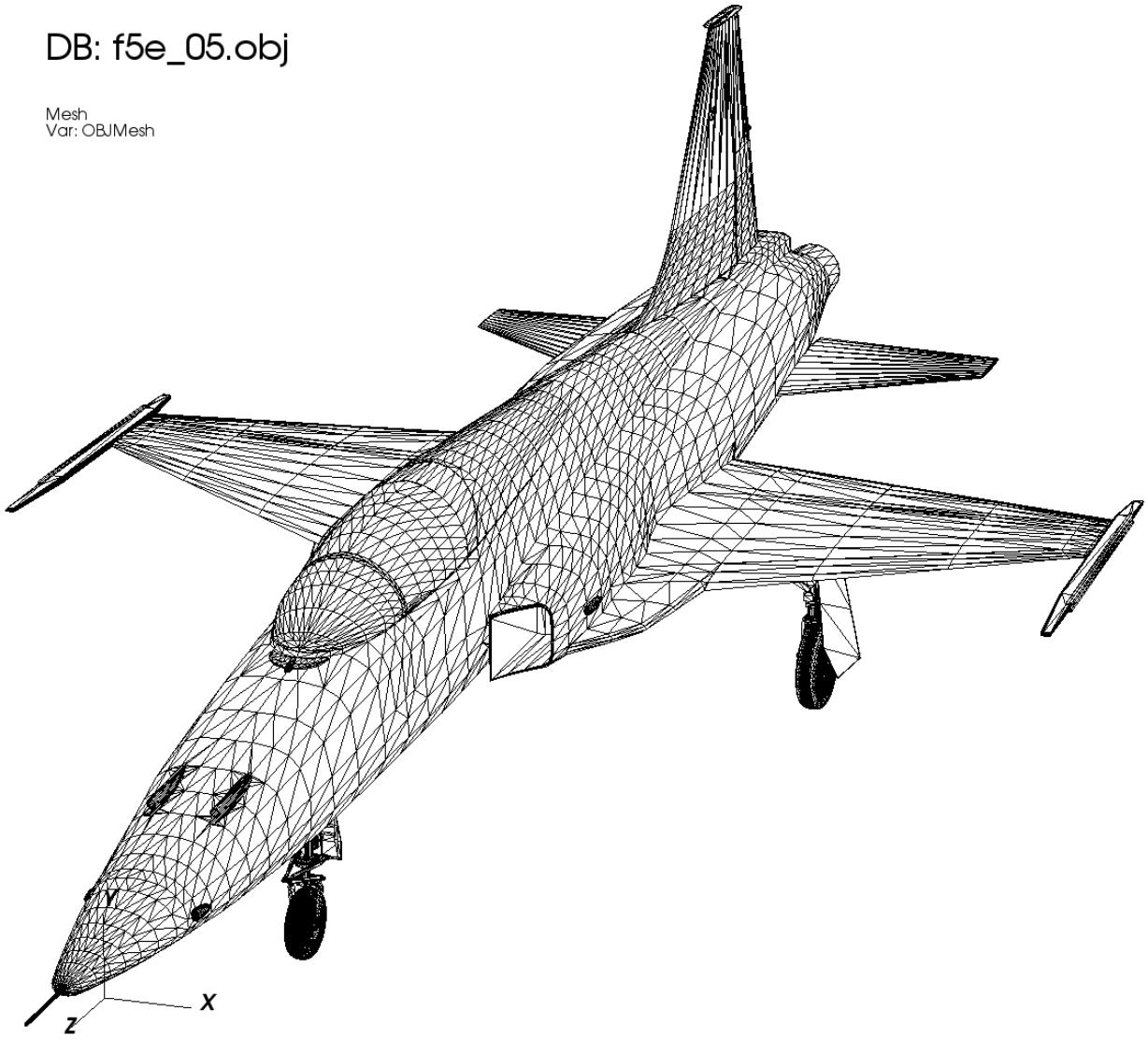


Fig. 4.45: Mesh plot

Mesh plot opaque modes

By default, VisIt's **Mesh** plot draws in opaque mode so that hidden surface removal is performed when the plot is drawn and each face of the externally visible cells are outlined with lines. When the **Mesh** plot's opaque mode is set to automatic, the **Mesh** plot will be drawn in opaque mode unless it is forced to share the visualization window with other plots, at which point the **Mesh** plot is drawn in wireframe mode. When the **Mesh** plot is drawn in wireframe

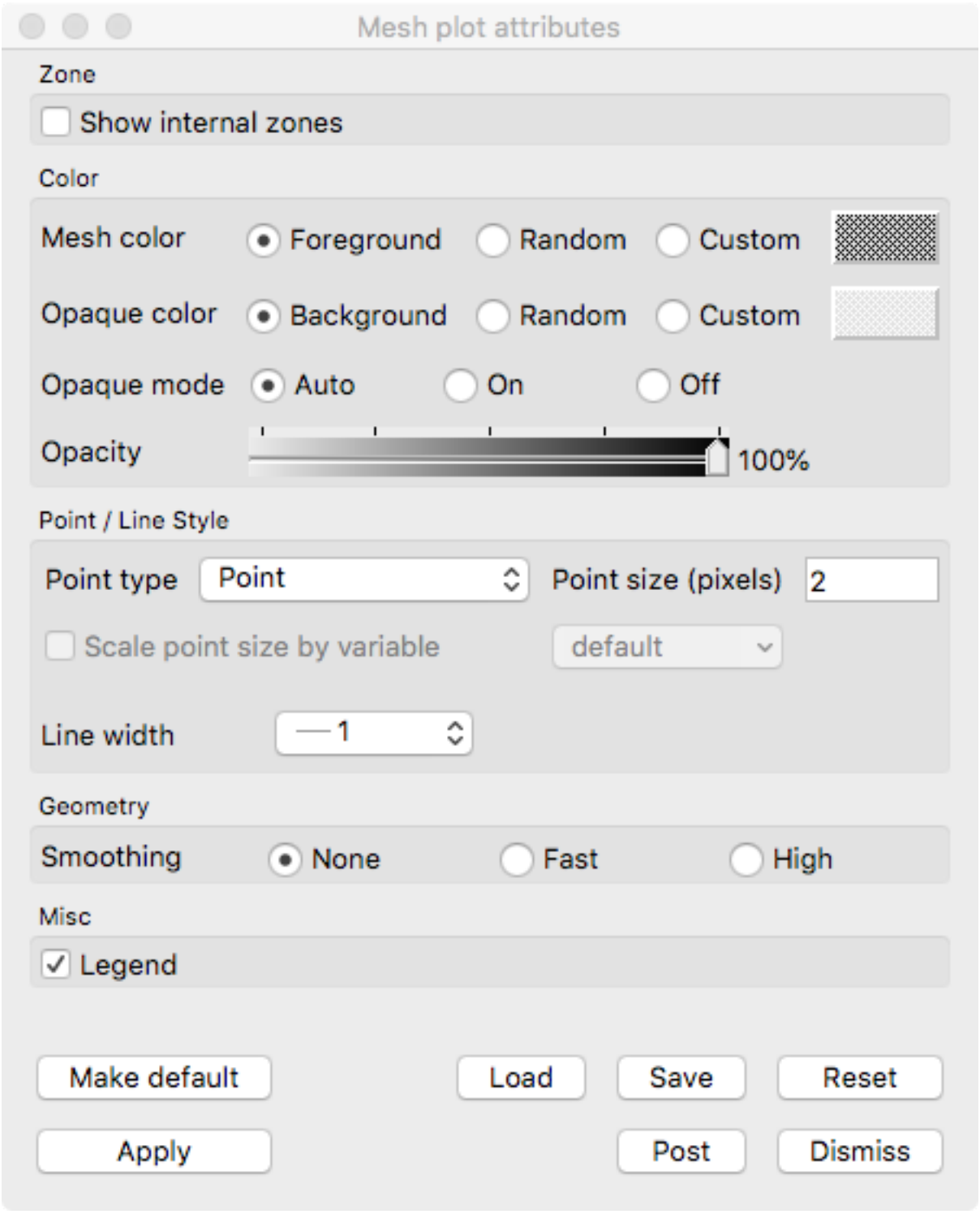


Fig. 4.46: Mesh plot window

mode, only the edges of each externally visible cell face are drawn, which prevents the **Mesh** plot from interfering with the appearance of other plots. In addition to having an automatic opaque mode, the **Mesh** plot can be forced to be drawn in opaque mode or wireframe mode by clicking the **On** or **Off** Radio buttons to the right of the **Opaque mode** label in the **Mesh plot attributes window**.

Showing internal zones

Sometimes it is useful to create mesh plot that shows all internal zones for a 3D database. Rather then plotting just the externally visible zones, which is the **Mesh** plot's default behavior, you can click the **Show internal zones** check box to force the **Mesh** plot to draw the edges of every internal zone.

Changing colors

There are two color controls for a **Mesh** plot. One, the *mesh* color, controls the color of mesh edge lines while the other, the *opaque* color, controls the color of mesh surface (areal) facets. For each color option, there are three choices

- A custom color chosen by the user.
- A random color chosen by VisIt.
- The *Foreground* (for mesh lines) or *Background* (for opaque facets) color.

The default is to use *Foreground* color for the mesh and *Background* color for the opaque color. In this mode, when these colors are changed via the *Annotation* controls, the **Mesh** plot obeys the newly selected colors. Otherwise, the **Mesh** plot maintains its chosen color (either *custom* or *random*).

The random color option is useful when displaying multiple meshes and the user simply needs to be able to easily distinguish among them.

Changing mesh line attributes

The **Mesh** plot's mesh lines have two user-settable attributes that control their width and line style. You can set the line width and line style are set by selecting new options from the **Line style** or **Line width** menus at the top of the **Mesh plot attributes window**.

Changing point type and size

Controls for points are described in *Point type and size*.

Geometry smoothing

Sometimes visualization operations such as material interface reconstruction can alter mesh surfaces so they are pointy or distorted. The **Mesh** plot provides an optional Geometry smoothing option to smooth out the mesh surfaces so they look better when the mesh is visualized. Geometry smoothing is not done by default, you must click the **Fast** or **High** radio buttons to enable it. The **Fast** geometry smoothing setting smooths out the geometry a little while the **High** setting works produces smoother surfaces.

Molecule Plot

The **Molecule** plot takes as input data with atoms and bonds (stored internally as Vertices and Lines in a VTK PolyData structure) and renders it as spheres and lines/cylinders

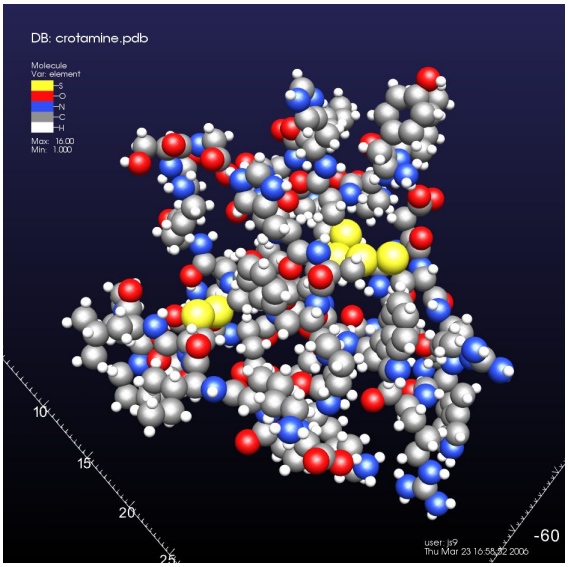


Fig. 4.47: Molecule plot of crotamine, colored by element type, atoms shown with covalent radius, and no bonds.

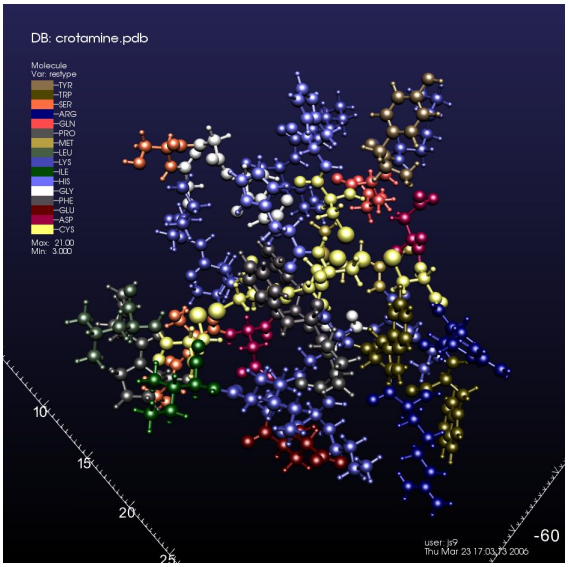


Fig. 4.48: Molecule plot of crotamine, colored by residue type, atoms proportional to covalent radius emulating the CPK syle, bonds colored with adjacent atom color.

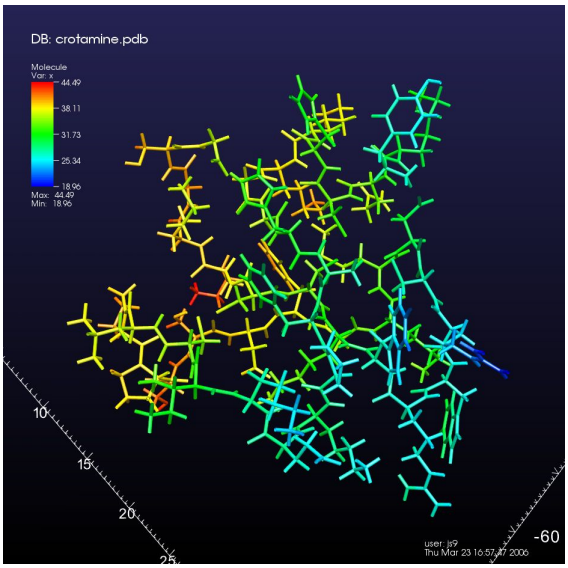


Fig. 4.49: Molecule plot of crotamine, colored by a scalar quantity, no atoms shown, and bonds drawn as cylinders.

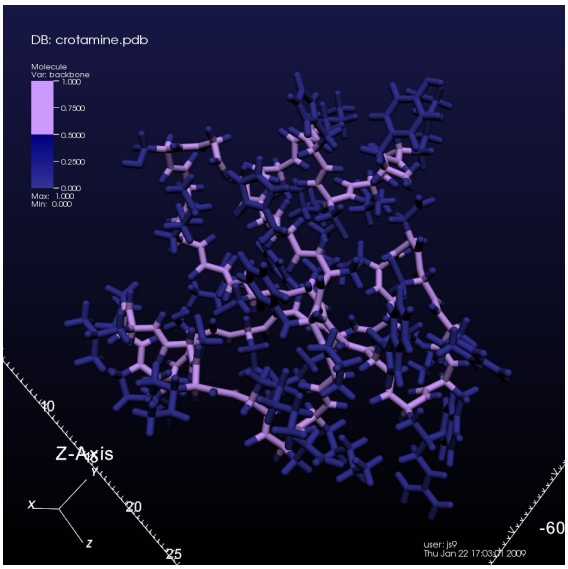


Fig. 4.50: Molecule plot of crotamine, colored by *backbone*, atom at same width as thicker cylinder-shaped bonds.

Controlling how atoms are drawn

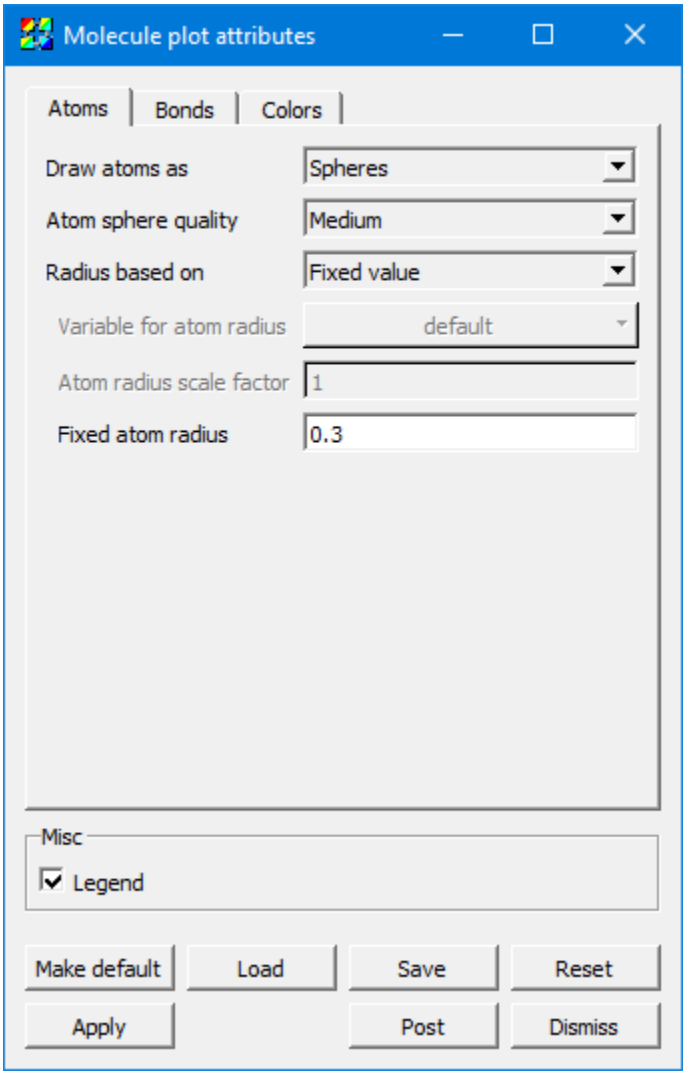


Fig. 4.51: Molecule plot window Atoms tab

The value *Spheres* for **Draw atoms as** means to draw spheres using 3D geometry. *Sphere Impostors* means to draw them using a single flat polygon with an image of a sphere – this requires support from graphics hardware and can introduce some minor graphical artifacts, but it is very fast. The value *None* means that you are only interested in seeing the bonds, and you would like the atoms themselves not to be drawn.

When rendering *Spheres*, **Atom sphere quality** determines the number of polygons used to draw the atom geometry. *Low* corresponds to about a dozen polygons per sphere, *Medium* is several dozen, *High* a couple hundred, and *Super* is about a thousand.

Radius based on determines how the atoms are sized. *Scalar variable* uses a nodal variable on the data set to determine radius. *Covalent radius* and *Atomic radius* are the atomic properties, and they are calculated using a built-in lookup table in VisIt. *Fixed value* simply uses the value in the text field below as the radius. Note that *Covalent radius* and *Atomic radius* require a discrete nodal field called *element* to exist and contain the atomic number. Also, note that some default values are set due to much molecular data being in units of Angstroms. Depending on your data, you may need to change the atomic/bond radii.

When **Radius based on** is set to *Scalar variable*, the **Variable for atom radius** field becomes active and determines which variable shall be used (and multiplied by the scale factor below) as the value for the radius of the rendered atoms.

Atom radius scale factor applies when **Radius based on** is not set to *Fixed value*. This value multiplies the other value used for radius, whether it is the atomic/covalent radius or based on a scalar variable. Note that the atomic and covalent radii used are in Angstroms, so if your data is in other units, you should apply the appropriate conversion factor here.

Fixed atom radius only applies when **Radius based on** is set to *Fixed value*. It is the actual radius you want to use to draw the atoms in world coordinate units.

Controlling how bonds are drawn

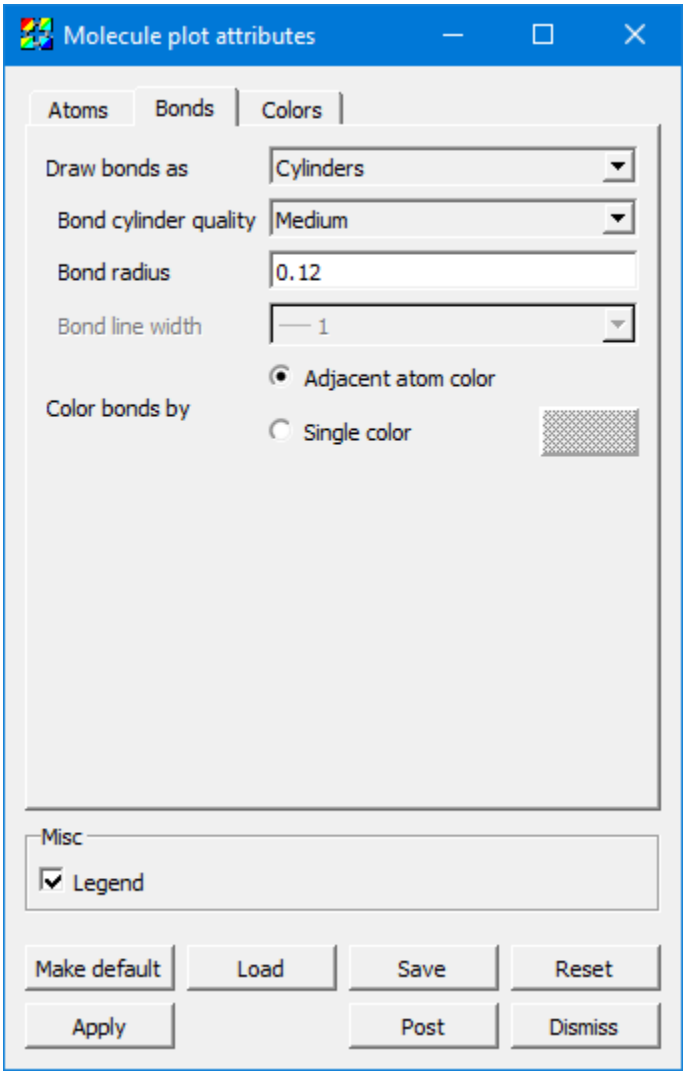


Fig. 4.52: Molecule plot window Bonds tab

The value *None* for **Draw bonds as** means you are only interested in seeing the atoms and would like any bonds to be hidden. *Lines* uses geometric lines with no 3D shading, and *Cylinders* uses 3D geometry with 3D shading. *Lines* is much faster but *Cylinders* looks better.

When **Draw bonds as** is set to *Cylinders*, **Bond cylinder quality** determines the number of polygons used to draw the bonds. *Low* is about three polygons, and *High* is about twenty.

When **Draw bonds as** is set to *Cylinders*, **Bond radius** determines the thickness of the cylinder in world coordinate units. Note that defaults for these values were chosen due to molecular data commonly being in units of Angstroms. Depending on your data, you may need to change the radius used for rendering atoms and bonds.

When **Draw bonds as** is set to *Lines*, **Bond line width** determines the thickness of the line used to draw the bonds in terms of a number of pixels.

Color bonds by can be set to *Adjacent atom color*, which means that each half of the bond is drawn using the color of the atom to which it is attached. Or, it can be set to a *Single color* chosen at the color selector just to the right of this checkbox.

Controlling colors

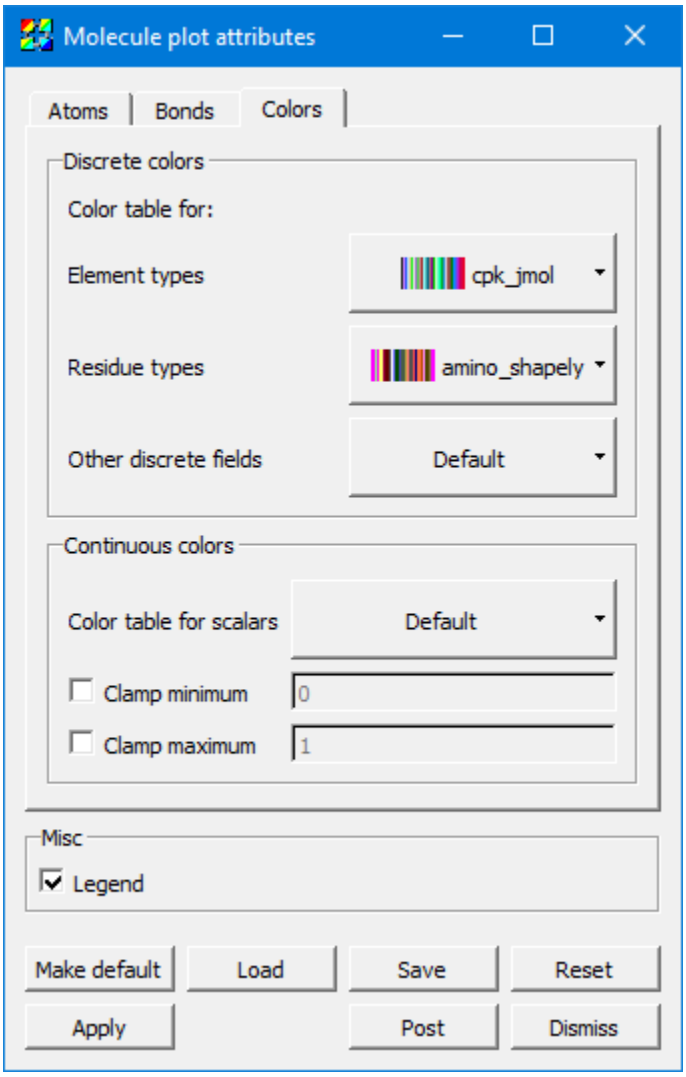


Fig. 4.53: Molecule plot window Colors tab

The **Discrete colors group** is for values which take on integral values. When VisIt encounters a discrete-valued variable, it determines which one of these color tables to use based on the variable name (*element* and *restype*, specifically).

Element types and **Residue types** are specific examples, and they are separate because there are conventional color tables widely used. VisIt provides some of these color tables. **Other discrete fields** catches anything which is not an element or residue type.

The **Continuous colors group** is for values which take on real values. **Color table for scalars** can be set to any color table, typically a continuous one. The **Clamp minimum** and **Clamp maximum** check boxes, along with their values, toggle whether to clamp the continuous field to narrow the range to a specific range of values of particular interest, making full use of the color table within that range and clamping anything outside that range to the colors at the min/max extrema of the selected color table.

Examples in use

See *Molecular data features* for examples of the Molecule plot in use.

Pseudocolor plot

The **Pseudocolor** plot, shown in Figure 4.54, maps a scalar variable's data values to colors and uses the colors to "paint" values onto the variable's computational mesh. The result is a clear picture of the database geometry painted with variable values that have been mapped to colors. You might try this plot first when examining a scientific database for the first time since it reveals so much information about the plotted variable.

Data tab options

VisIt's **Pseudocolor plot attributes window Data tab** allows you to change the data scaling, limits and centering, as well as change colors, opacity and control the plot Legend and lighting. (shown in Figure 4.55)

Scaling the data

The scale maps data values to color values. VisIt provides three scaling options: **Linear**, **Log**, and **Skew**. **Linear**, which is the default, uses a linear mapping of data values to color values. **Log** scaling is used to map small ranges of data to larger ranges of color. **Skew** scaling goes one step further by using an exponential function based on a skew factor to adjust the mapping of data to colors. The function used in skew scaling is **(s^d-1)/(s-1)** where **s** is a skew factor greater than zero and **d** is a data value that has been mapped to a range from zero to one. The mapping of data to colors is changed by changing the skew factor. A skew factor of one is equivalent to linear scaling but values either larger or smaller than one produce curves that map either the high or low end of the data to a larger color range. To change the skew factor, choose **Skew** scaling and type a new skew factor into the **Skew factor** text field.

Limits

Setting limits for the plot imposes artificial minima and maxima on the plotted variable. This effectively restricts the range of data used to color the **Pseudocolor** plot. You might set limits when you are interested in only a small range of the data or when data limits need to be maintained for multiple time steps, as when playing an animation. In fact, we recommend setting the limits when producing an animation so the colors will correspond to the same values instead of varying over time with the range of the plotted variable. Setting limits often highlights a certain range in the data by assigning more colors to that data range.

To set the limits for the **Pseudocolor** plot, you must first select the limit mode. The limit mode determines whether the original data extents (data extents before any portions of the plot are removed), are used or the actual data extents (data extents after any portions of the plot are removed), are used. To select the limit mode, choose either **Use Original Data** or **Use Actual Data** from the **Limits** menu.

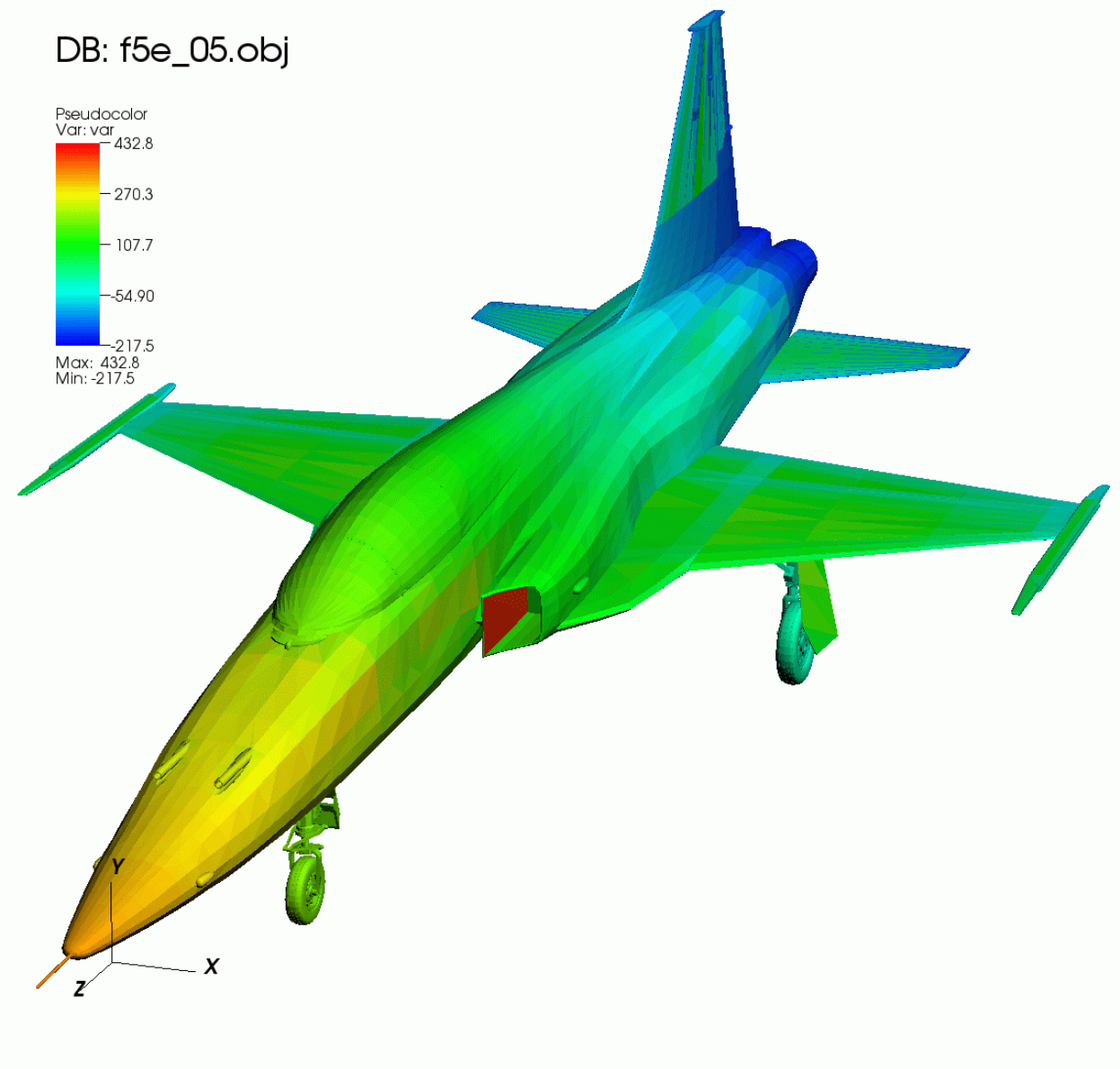


Fig. 4.54: Pseudocolor plot

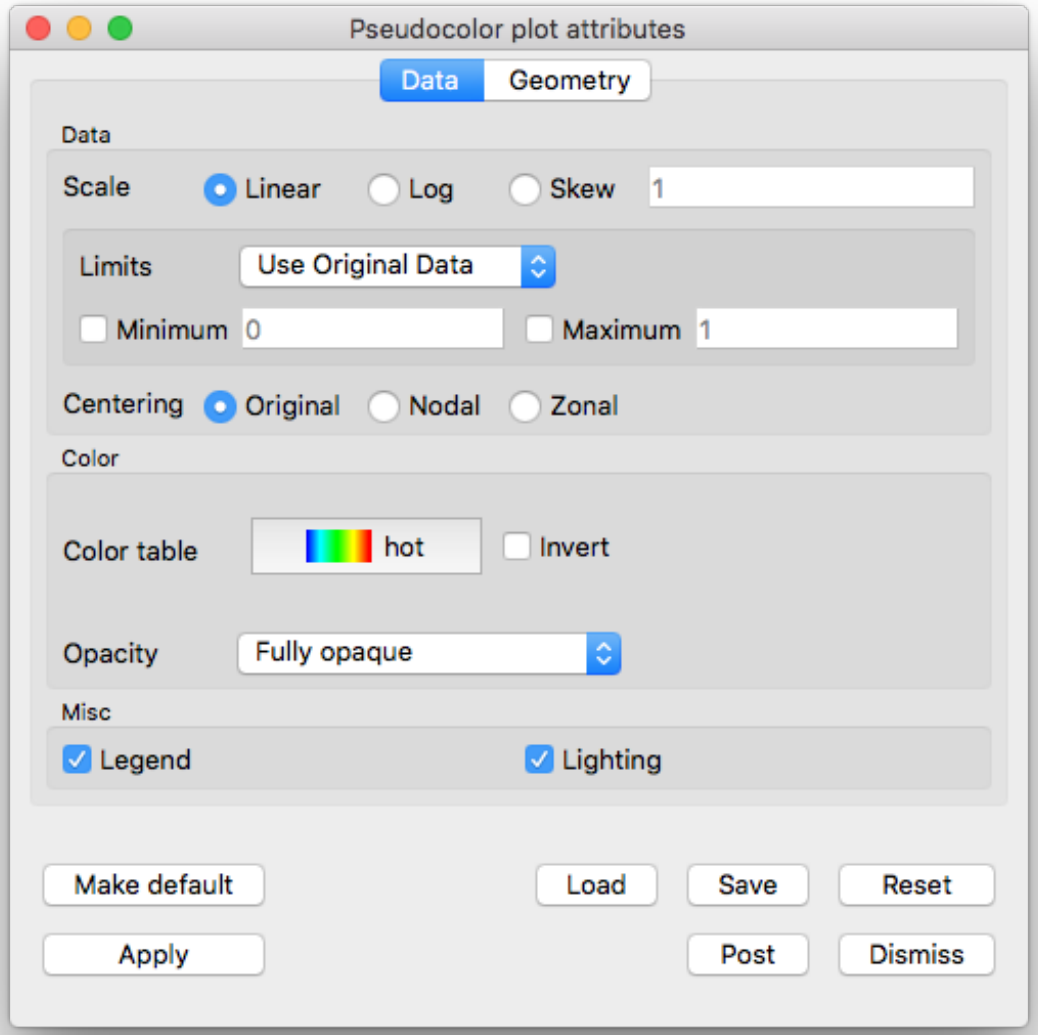


Fig. 4.55: Pseudocolor plot attributes window Data tab

The limits for the **Pseudocolor** plot consist of a minimum value and a maximum value. You may set these limits, and turn them on and off, independently of one another. That is, the use of one limit does not require the use of the other. To set a limit, check the **Min** or **Max** check box next to the **Min** or **Max** text field and type a new limit value into the **Min** or **Max** text field.

Variable centering

Variables in a database can be associated with a mesh in various ways. Databases supported by VisIt allow variables to be associated with a mesh's zones (cells) or its nodes. When a variable is associated with a mesh's zones, the variable field consists of one value for each zone and is said to be *Zone-centered* . When a variable is associated with a mesh's nodes, there are values for each vertex making up the zone and the variable is said to be *Node-centered*.

There are three settings for variable centering: **Natural**, **Nodal**, and **Zonal**. **Natural** variable centering displays the data according to the way the variable was centered on the mesh. This means that node-centered data will be displayed at the nodes with colors being linearly interpolated between the nodes, and zone-centered data will be displayed as zonal values, giving a slightly "blocky" look to the picture. If **Nodal** centering is selected, all data is displayed at the nodes regardless of the variable's natural centering. This will produce a smoother picture, but for variables which are actually zone-centered, you will lose some data (local minima and maxima). If you select **Zonal** centering, all data is displayed as if they were zone-centered. This produces a blockier picture and, again, it blurs minima/maxima for node-centered data.

Changing the color table

The **Pseudocolor** plot can specify which VisIt color table is used for colors. To change the color table, click on the **Color table** button, shown in Figure 4.56, and select a new color table name from the list of color tables. The list of color tables always represents the list of available VisIt color tables. If you do not care which color table is used, choose the Default option to use VisIt's active continuous color table. New color tables can be defined using VisIt's **Color table window** which is described later in this manual.

Opacity

You can make the **Pseudocolor** plot transparent by changing its opacity using the **Opacity** menu. There are four options:

1. **Fully opaque:** (the default), no transparency is applied.
2. **From color table:**, opacity values are obtained from the active color table for the plot. If the color table doesn't support opacities, the plot will be fully opaque.
3. **Constant:** A constant opacity is applied everywhere. A slider is provided to modify the opacity value. Moving the opacity slider to the left makes the plot more transparent while moving the slider to the right makes the plot more opaque.
4. **Ramp:** Opacity is applied on a sliding scale ranging from fully transparent (applied to the lowest values), to the opacity value chosen on the slider. If the the slider is fully to the right, then the maximum values being plotted will be fully opaque.

Legend Behavior

The legend for the **Pseudocolor** plot is a color bar annotated with tick marks and numerical values. Below the color bar the minimum and maximum data values are also displayed. Setting the limits for the plot changes *only* the color-bar portion of the plot's legend. It *does not change* the *Min* and *Max* values printed just below the color bar. Those

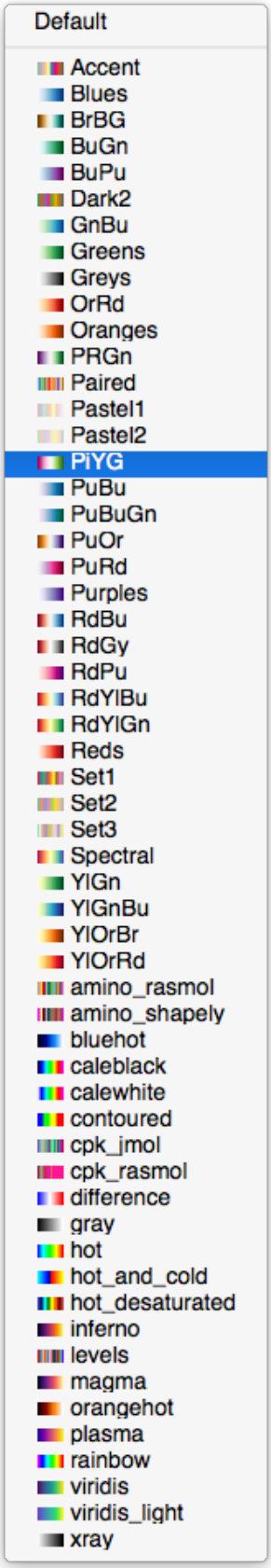


Fig. 4.56: Color table button

values will always display the original data's minimum and maximum values, regardless of the limits set for the plot or the effect of any operators applied to the plot.

Lighting

Lighting adds detail and depth to the **Pseudocolor** plot, two characteristics that are important for animations. The **Lighting** check box in the lower part of the **Pseudocolor plot attributes window** turns lighting on and off. Since lighting is on by default, uncheck the **Lighting** check box to turn lighting off.

Geometry tab options

VisIt's **Pseudocolor plot attributes window Geometry tab** allows you to modify the appearance of lines and points, and change rendering options (shown in Figure 4.57)

Lines

The lines section can be useful when visualizing the results from an :ref: *integral curve system <Integral_Curve_System>* operation.

There are three options for **Line type**: **Lines** (default), **Tubes**, and **Ribbons**.

The width of **Lines** can be changed by choosing an option from the **Line width** menu. The **Tubes** type has a **Resolution** option which represents the roundness of the tube. The higher the resolution, the rounder the tube.

Both the **Tubes** and **Ribbons** type have various methods for affecting the radius. The **Radius** option can be expressed either as an **Absolute** quantity or **Fraction of the Bounding Box** (default) by choosing one of these via the menu. A Variable can be chosen for the radius by checking the **Variable radius** checkbox, and choosing a variable from the menu.

Lines can also have glyphs at their head and tail. Glyph options are **None** (default), **Sphere**, and **Cone**. You can also specify **Resolution** and **Radius** for the glyphs.

Point

Controls for points are described in *Point type and size*.

Representation

By default, the **Pseudocolor** plot renders as a **Surface**. It can also render in **Wireframe** or **Points** mode. Choose the representation by checking one or any combination of the three. **Wireframe** and **Points** will be rendered in the color specified by their corresponding Color buttons.

Geometry smoothing

Sometimes visualization operations such as material interface reconstruction can alter mesh surfaces so they are pointy or distorted. The **Pseudocolor** plot provides an optional Geometry smoothing option to smooth out the mesh surfaces so they look better when the plot is visualized. Geometry smoothing is not done by default, you must click the **Fast** or **High** radio buttons to enable it. The **Fast** geometry smoothing setting smooths out the geometry a little while the **High** setting produces smoother surfaces.

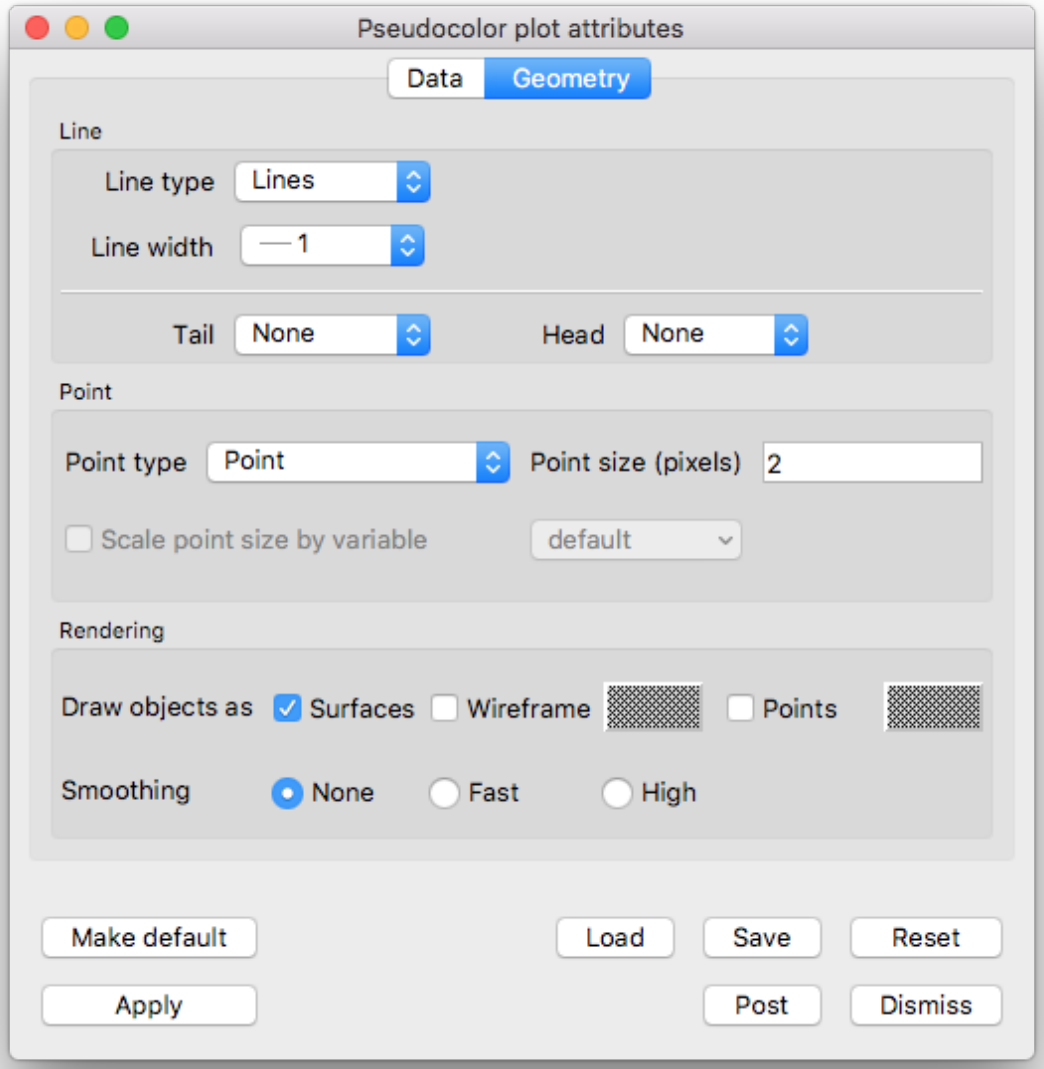


Fig. 4.57: Pseudocolor plot attributes window, geometry tab

Scatter Plot

The **Scatter** plot (see Figure 4.58) allows you to combine multiple scalar fields into a point mesh so you can investigate the relationships between multiple input variables. You might, for example, want to see the behavior of pressure vs. density colored by temperature. The **Scatter** plot can take up to four scalar fields as input and can use up to three of them as coordinates for the created point mesh while one input variable can be used to assign colors to the point mesh. The **Scatter** plot provides individual controls for setting the limits of each input variable and also allows each input variable to be scaled so that all of the resulting points from disparate data ranges fit in a unit cube.

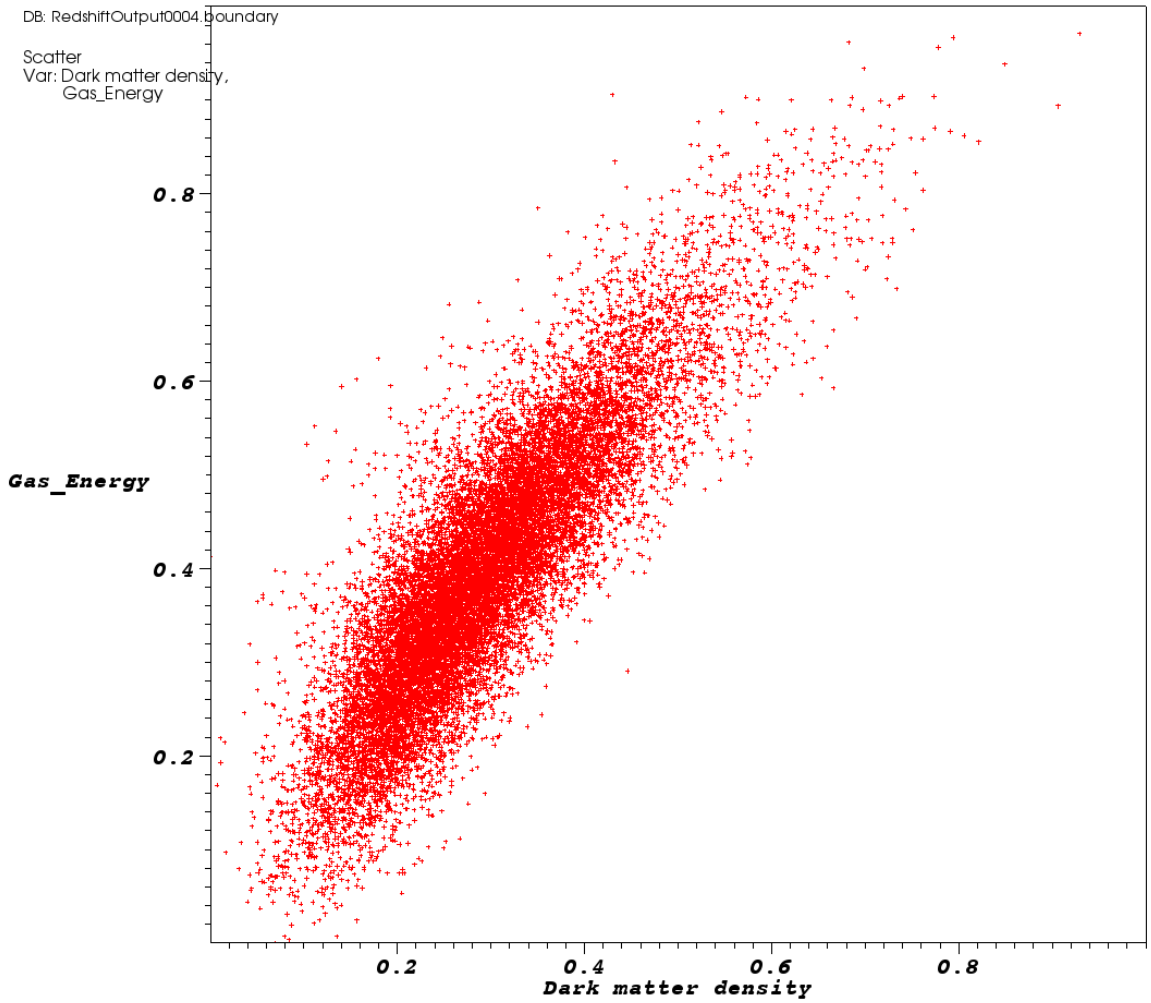


Fig. 4.58: Example of Scatter plot

The **Scatter plot attributes window** is divided into two tabs: **Inputs** and **Appearance**. The **Inputs** tab is further subdivided into tabs for each input variable. Each tab for an input variable contains controls that pertain to selecting the input variable, settings its limits, or setting the role that the input variable will perform within the **Scatter** plot. Each input variable can have one of five roles that will be covered later. The **Appearance** tab contains controls for changing the **Scatter** plot's appearance. Under the two main tabs, the **Scatter plot attributes window** features a small

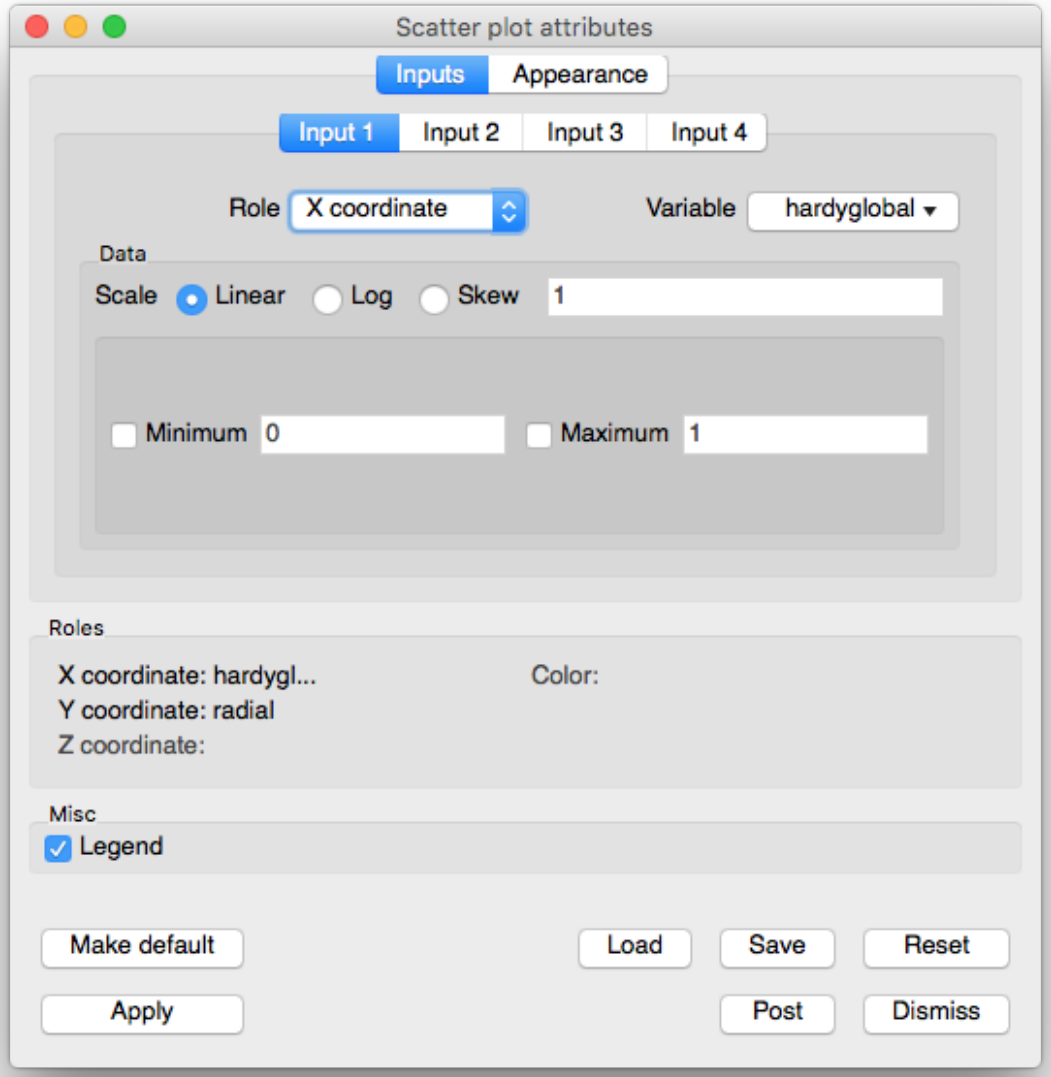


Fig. 4.59: Scatter plot attributes window

section that lists the roles that are used in the plot and which input variables are assigned to each role.

Scatter plot wizard

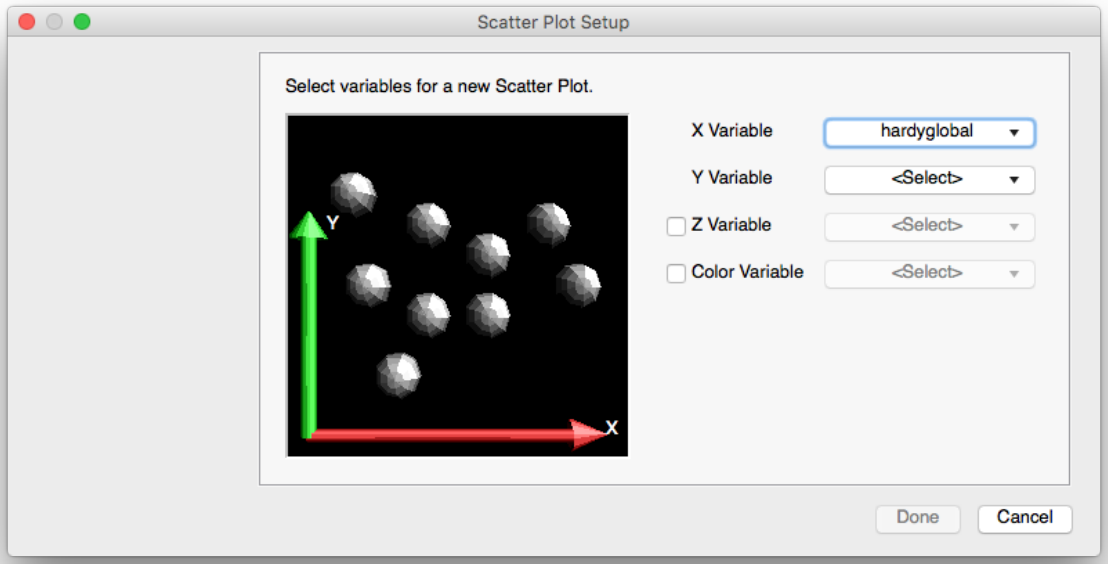


Fig. 4.60: Example of the Scatter plot wizard

Plots are typically created in VisIt when you choose a variable from one of the **Plot menus**. Since the **Scatter** plot takes as input up to four input variables and typical plot creation only initializes one variable, you can imagine that if a **Scatter** plot was created the usual way, only one of its many input variables would be initialized. Furthermore, to initialize the plot, you would have to open the **Scatter plot attributes window** and select the other variables. Since that would not be a very straightforward way to create a **Scatter** plot, VisIt now has support for plot wizards. A plot wizard is a simple dialog window that pops up when you select a variable to plot. A plot wizard leads you through a series of questions that allow VisIt to more fully initialize a new plot. The **Scatter plot wizard** prompts you for the scalar variable to use for the Y-Axis, the variable to use for the Z-Axis (optional), and the variable to use for the plot's colors (optional).

Selecting a variable

Three of the **Scatter** plot's four input variables can be set in the **Scatter plot attributes window**. The first input variable cannot be changed from within the **Scatter plot attributes window** because that is the default variable used by the plot. If you want to change the first input variable, you can use the **Variables** menu under the **Plot list** . If you want to select a different variable for any of the other input variables, you would first click on the input variable's tab and then you would select a new variable by making a selection from the tab's **Variable** button. Note that any combination of nodal and cell-centered variables can be chosen. The **Scatter** plot will recenter any input variables whose centering does not match the first input variable's centering.

Setting an input variable's role

Each of the **Scatter** plot's input variables has a role that you can set which determines how the input variable is used by the **Scatter** plot. An input variable can be used for the X, Y, Z coordinates, for the color, or it can have no role. The role of the input variable is not fixed because you might want to change roles many times and it is much less work to change only the roles instead of reselecting variables, limits, and scaling for an input variable. The flexibility of selecting a role for an input variable makes it convenient to turn off colors or the Z coordinate with little effort. To change the role for an input variable, select a new role from the input variable's **Role** combo box. If you select a role that is already played by another input variable, VisIt will give the current input variable the selected role and set the input variable that previously had the selected role so that it has no role.

Each of the **Scatter** plot roles and their associated input variables are listed in the bottom of the **Scatter plot attributes window** . Roles that have an input variable have the name of the input variable printed next to the name of the role so looking through all of the input variable tabs to determine what the **Scatter** plot should look like is not required. Roles that have no assigned input variable are grayed out.

Setting the minimum and maximum values

The **Scatter** plot allows you to set minimum and maximum limits on the values considered for inclusion into the created point mesh. If an input variable's data value does not lie in the specified minimum/maximum value data range then the point is not included in the created point mesh. Note that setting limits does not cause points to be removed when data values in the color role fall outside of the specified limits. To set the minimum value to be allowed in the created point mesh, click on the **Min** check box and type a new minimum value into the **Min** text field. To set the maximum value to be allowed in the created point mesh, click on the **Max** check box and type a new value into the **Max** text field.

Scaling an input variable

Sometimes input variable data values are clustered in a certain range of the data. When this is the case, the points in the **Scatter** plot will bunch up in one or more dimensions. For more uniformly spaced points, you might try scaling one or more input variables. Each input variable can be scaled in the three common ways: Linear, Log, and Skew. To set the scaling method used for the input variable, click on the **Linear**, **Log**, or **Skew** radio buttons. If you choose the Skew scaling method then you should also enter a value greater than zero into the **Skew factor** text field to determine the function used for skew scaling.

Since the **Scatter** plot's input variables are likely to have wildly different data ranges, the **Scatter** plot provides an option to independently scale each input variable so it is in the range [0,1] so the resulting plot fits entirely in a cube. If you prefer to see the **Scatter** plot without this corrective scaling, you can turn off the Scale to cube check box on the **Scatter plot attribute window's Appearance** tab.

Setting the colors

The **Scatter** plot can map scalar values to colors like the Pseudocolor plot (*Pseudocolor plot*) does or it can color all points using a single color. If you have set one of the input variables to have a color role then the **Scatter** plot will map that input variable's data values to colors using the specified color table. To change the color table used by the **Scatter** plot, click on the **Color table** button and select a new color table from the list of available color tables. If the **Scatter** plot has been configured such that none of the input variables is playing the color role then the **Scatter** plot's points will be drawn using one color. When the **Scatter** plot draws its points using a single color, its default behavior is to color the points using the vis window's foreground color. If you want to instead use a different color, turn off the **Use foreground** check box and click on the **Single color** color button to select a new color.

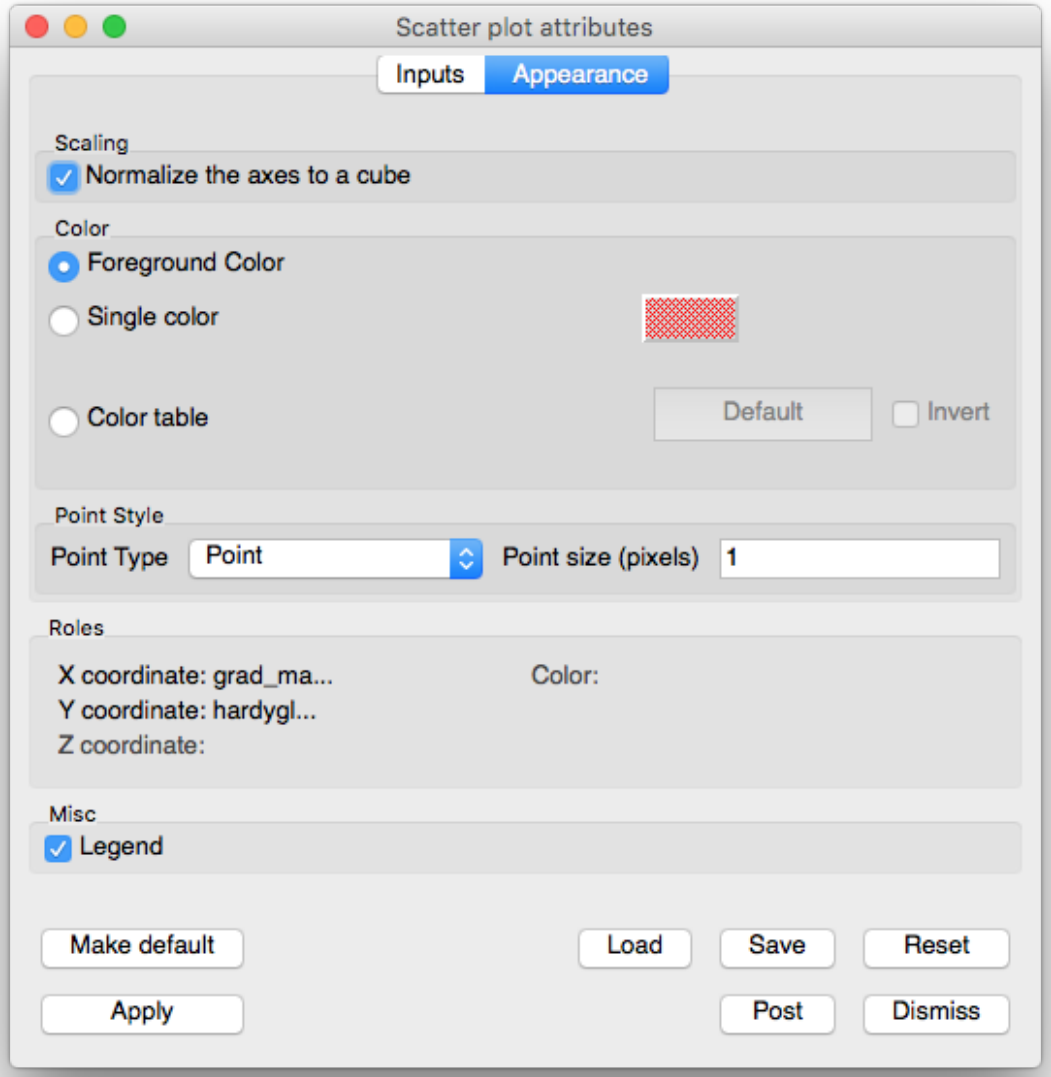


Fig. 4.61: Scatter plot attributes window's Appearance tab

Setting point properties

Controls for points are described in *Point type and size*.

Subset Plot

The Subset plot (example in Figure 4.62) is used to display subsets. The typical scientific database can be decomposed into many different subsets. Frequently a database is decomposed into non-material subsets such as domains or groups. In AMR meshes, subsets can consist of levels or patches. The Subset plot draws the database with its various subsets color coded so they can be distinguished. For more information about subsets, see the **Subsetting** chapter.

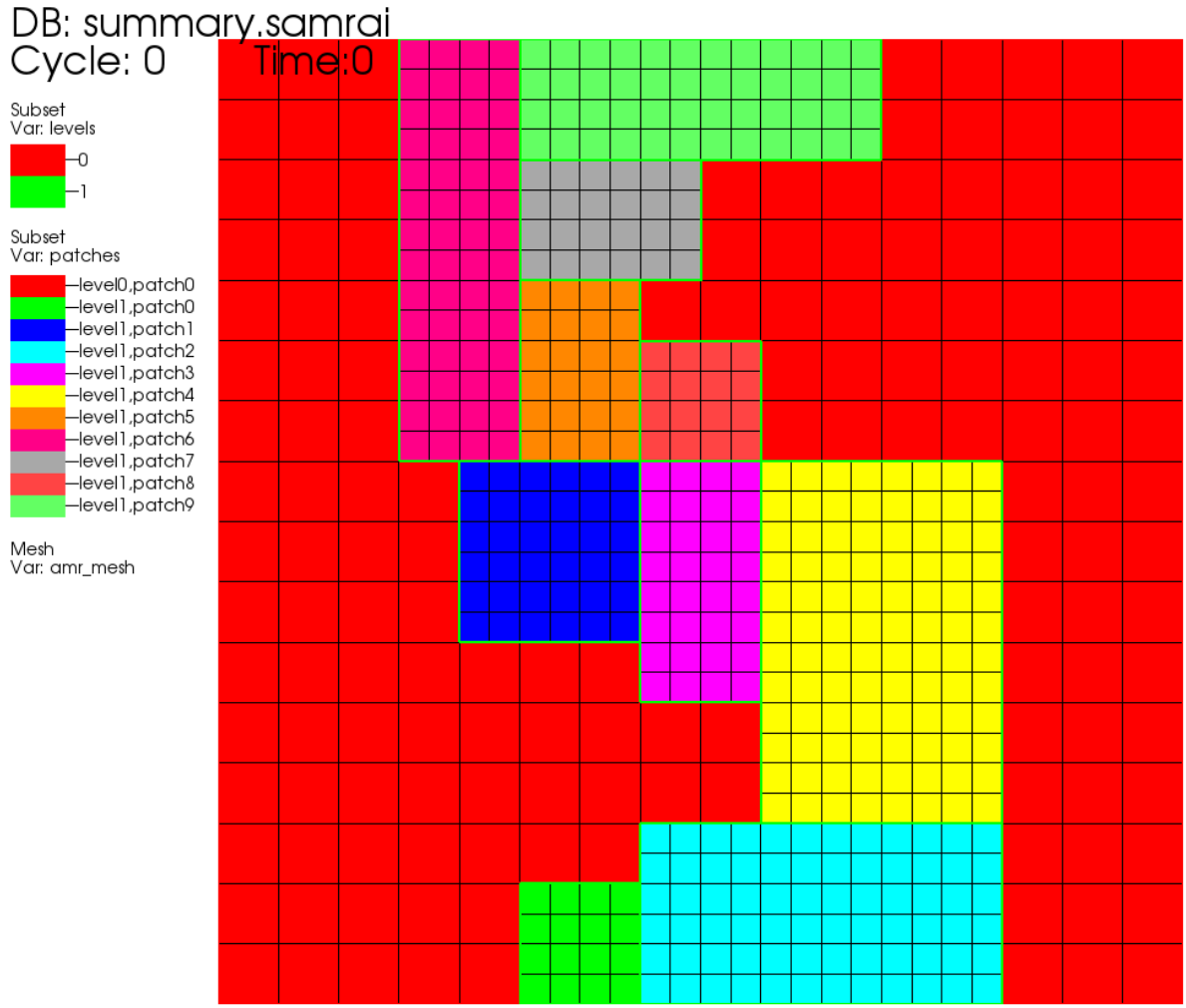


Fig. 4.62: Example of Subset plot of an AMR Mesh

Changing colors

The main portion of the **Subset plot attributes window** , also known as the **Subset colors area**, is devoted to setting subset colors. The **Subset colors area** contains a list of subset names with an associated subset color. Subset plot colors can be assigned three different ways, the first of which uses a color table. A color table is a named palette of

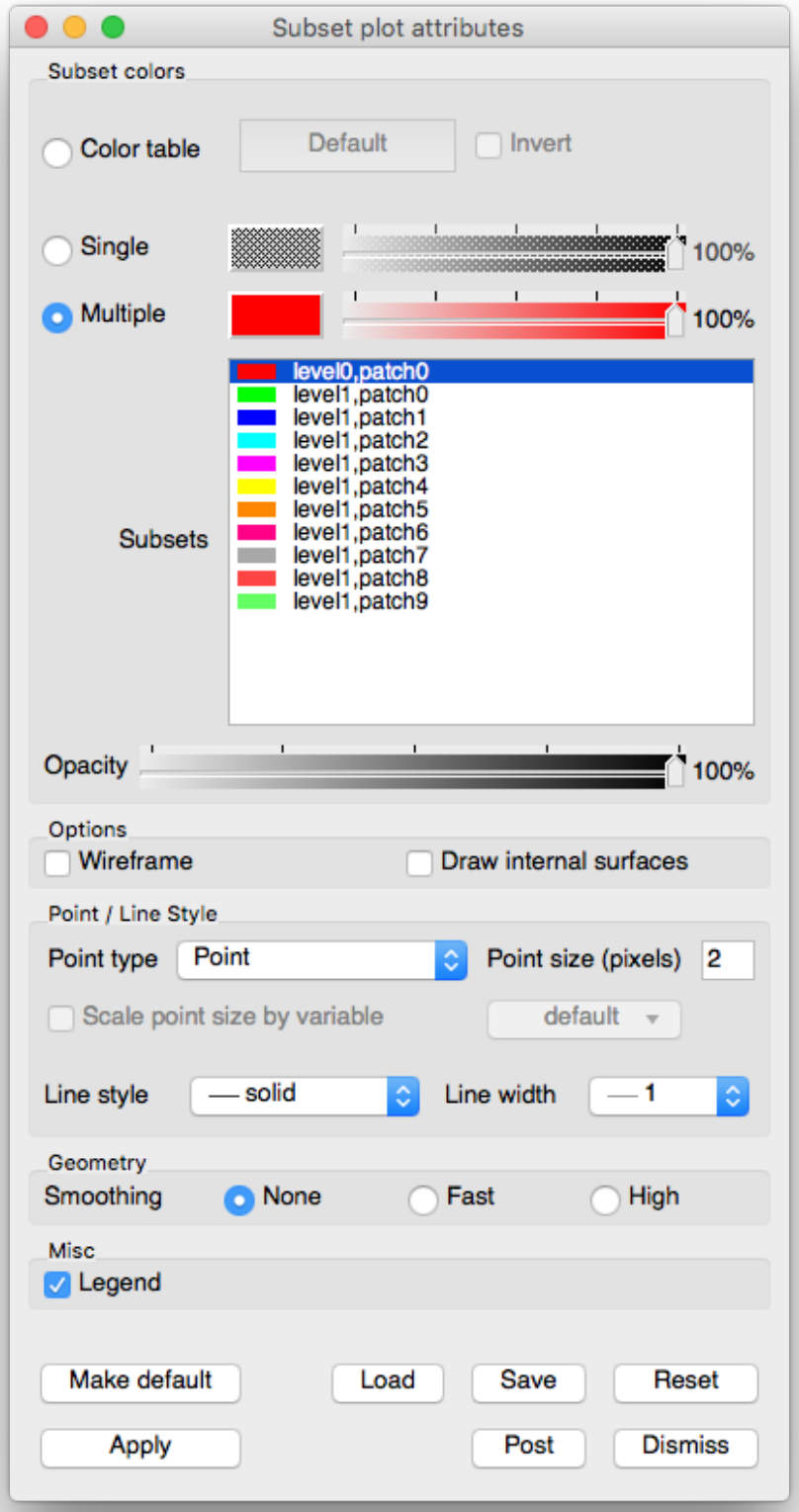


Fig. 4.63: Subset plot attributes window

colors that you can customize to suite your needs. When the Subset plot uses a color table to color subsets, it selects colors that are evenly spaced through the color table based on the number of subsets. For example, if you have three subsets and you are coloring them using the "xray" color table, the Subset plot picks three colors out of the color table so your levels are colored black, gray, and white. To color a Subset plot with a color table, click on the **Color table radio button** and choose a color table from the **Color table menu** to right of the **Color table radio button**.

If you want all subsets to be the same color, click the **Single** radio button at the top of the **Subset plot attributes window** and select a new color from the **Popup color menu** that is activated by clicking on the **Single color button**. The opacity slider next to the **Single color button** sets the opacity for the single color.

Clicking the **Multiple** radio button causes each subset to be a different, user-specified color. By default, multiple colors are set using the colors of the discrete color table that is active when the Subset plot is created. To change the color for any of the subsets, select one or more subsets from the list of subsets and click on the **Color button** to the right of the **Multiple** radio button and select a new color from the **Popup color menu**. To change the opacity for a subset, move **Multiple** opacity slider to the left to make the subset more transparent or move the slider to the right to make the subset more opaque.

The **Subset plot attributes window** contains a list of subset names with an associated subset color. To change a subset's color, select one or more subsets from the list, click the color button and select a new color from the popup color menu.

Opacity

The Subset plot's opacity can be changed globally as well as on a per subset basis. To change subset opacity, first select one or more subsets in the subset list and move the opacity slider next to the color button. Moving the opacity slider to the left makes the selected subsets more transparent and moving the slider to the right makes the selected subsets more opaque. To change the entire plot's opacity globally, use the **Opacity** slider near the bottom of the window.

Setting point properties

Albeit rare, the Subset plot can be used to plot points that belong to different subsets so the **Subset plot attributes window** provides controls that allow you to set the representation and size of the points. You can change the points' representation using the **Point Type** combo box. The available options are: **Box**, **Axis**, **Icosahedron**, **Point**, and **Sphere**. To change the size of the points, you can enter a new floating point value into the **Point size** text field. Finally, you can opt to scale the points' glyphs using a scalar expression by turning on the **Scale point size by variable** check box and by selecting a scalar variable from the **Variable** button to the right of that check box.

Wireframe mode

The Subset plot can be modified so that it only displays outer edges of subsets. This option usually leaves lines that give only the rough shape of subsets and where they join other subsets. To make the Subset plot display in wireframe mode, check the **Wireframe** check box near the bottom of the **Subset plot attributes window**.

Drawing internal surfaces

When you make one or more subsets transparent, you might want to make the Subset plot draw internal surfaces. Internal surfaces are normally removed from Subset plots to make them draw faster. To make the Subset plot draw internal surfaces, check the **Draw internal surfaces** check box near the bottom of the **Subset plot attributes window**.

Geometry smoothing

Sometimes visualization operations such as material interface reconstruction can alter mesh surfaces so they are pointy or distorted. The Subset plot provides an optional Geometry smoothing option to smooth out the mesh surfaces so they look better when the plot is visualized. Geometry smoothing is not done by default, you must click the **Fast** or **High** radio buttons to enable it. The **Fast** geometry smoothing setting smooths out the geometry a little while the **High** setting works produces smoother surfaces.

Tensor plot

The Tensor plot, shown in Figure 4.64, displays tensor variables using ellipsoid glyphs to convey information about a tensor variable's eigenvalues. Each glyph's scaling and rotation is controlled by the eigenvalues/eigenvectors of the tensor as follows: for each tensor, the eigenvalues (and associated eigenvectors) are sorted to determine the major, medium, and minor eigenvalues/eigenvectors. The major eigenvalue scales the glyph in the x-direction, the medium in the y-direction, and the minor in the z-direction. Then, the glyph is rotated so that the glyph's local x-axis lies along the major eigenvector, y-axis along the medium eigenvector, and z-axis along the minor.

Changing the tensor colors

The Tensor plot can be colored by a solid color or by the corresponding to the largest eigenvalue. To color the Tensor plot by eigenvalues, click the **Eigenvalues** radio button and then select a color table name from the color table button to the right of the **Eigenvalues** radio button. To make all tensor glyphs be the same color, click the **Constant** radio button and choose a color by clicking on the **Constant color button** and selecting a new color from the **Popup color menu**.

Setting the tensor scale

The Tensor plot's tensor scale affects how large the ellipsoidal glyphs that represent the tensor are drawn. By default, VisIt computes an automatic scale factor based on the length of the bounding box's diagonal to multiply by the user-specified scale factor. This ensures that the tensors are some reasonable size independent of the size of the mesh. To change the tensor scale, type a new floating point number into the **Scale** text field and click the **Apply** button in the **Tensor plot attributes window**. If you want to turn off automatic scaling so the size of the tensors is solely determined by the scale in the Scale text field, turn off the Auto scale check box. Yet another scaling option for tensors is scaling by magnitude. When the **Scale by magnitude** check box is checked, the magnitude of the tensor's longest eigenvector is used as a scale factor that is multiplied into the scale determined by the user-specified scale and the automatic scale factor.

Setting the number of tensors

When visualizing a large database, a Tensor plot will often have too many tensors to effectively visualize so the Tensor plot provides controls to reduce the number of tensors to a number that looks appealing in a visualization. You can accomplish this reduction by setting a fixed number of tensors or by setting a stride. To set a fixed number of tensors, select the **N tensors** radio button and enter a new number of tensors into the **N tensors** text field. To reduce the number of tensors by setting the stride, select the **Stride** radio button and enter a new stride value into the **Stride** text field.

Truecolor plot

The Truecolor plot, shown in Figure 4.66, is used to plot images of observational or experimental data so they can be compared to other plots, possibly of related, simulated data, in the same visualization window. The Truecolor plot

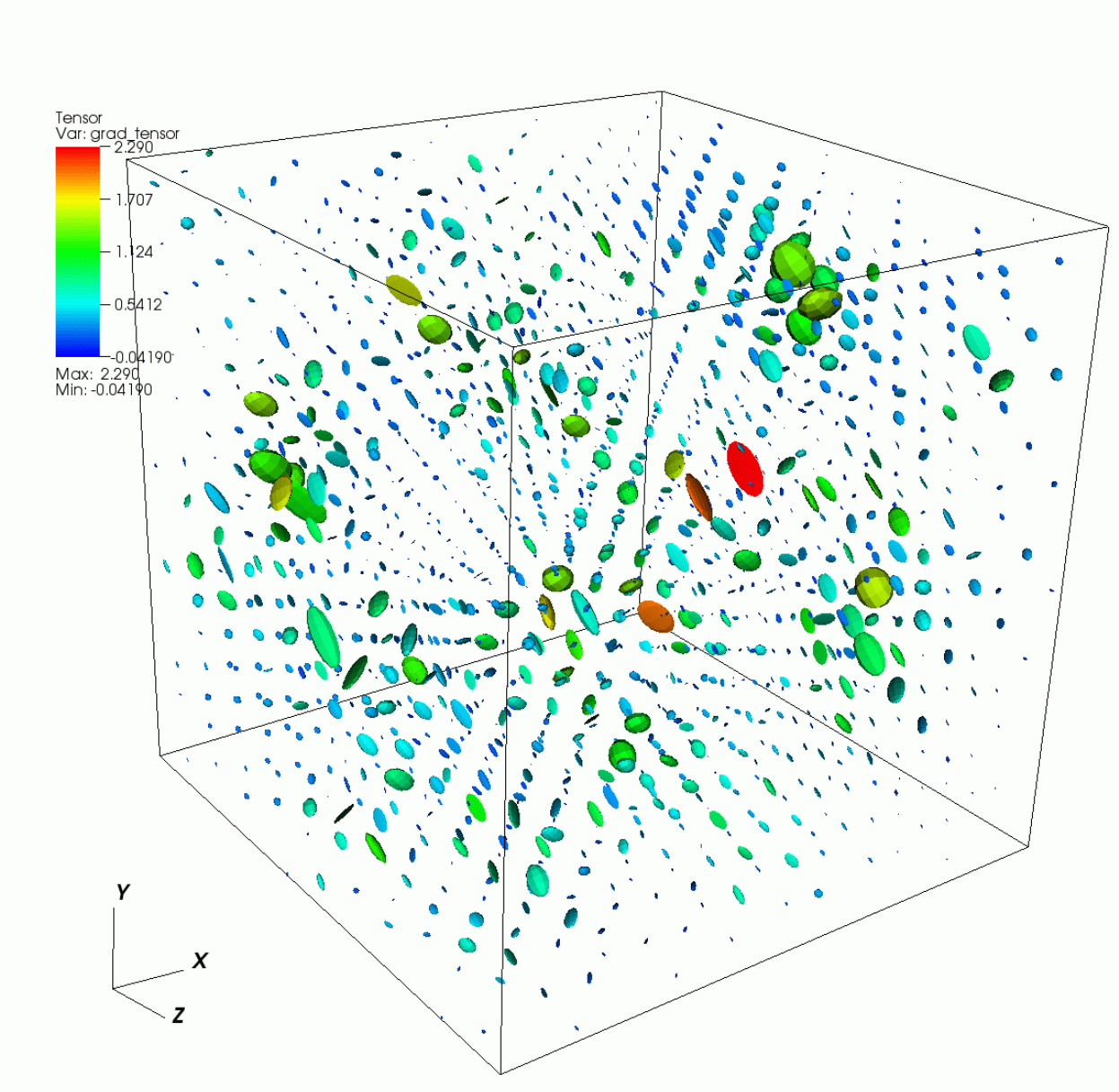


Fig. 4.64: Example of Tensor plot

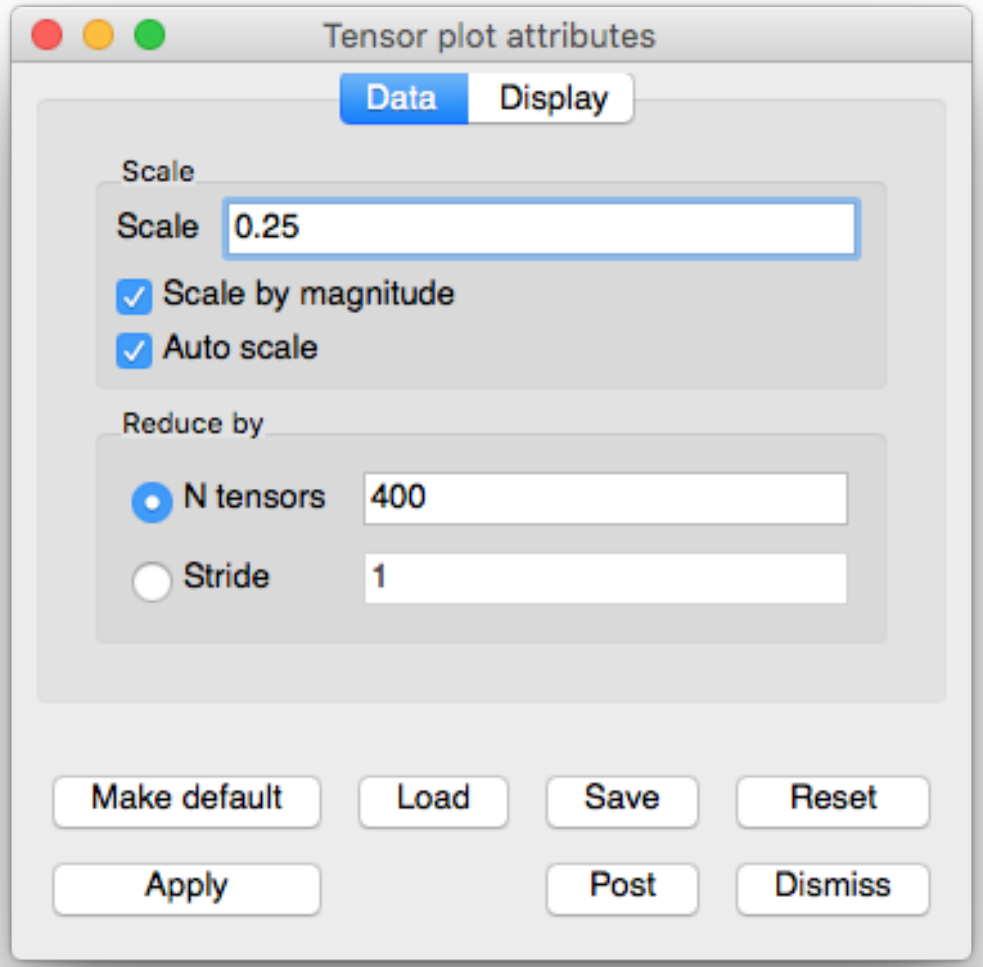


Fig. 4.65: Tensor plot attributes window

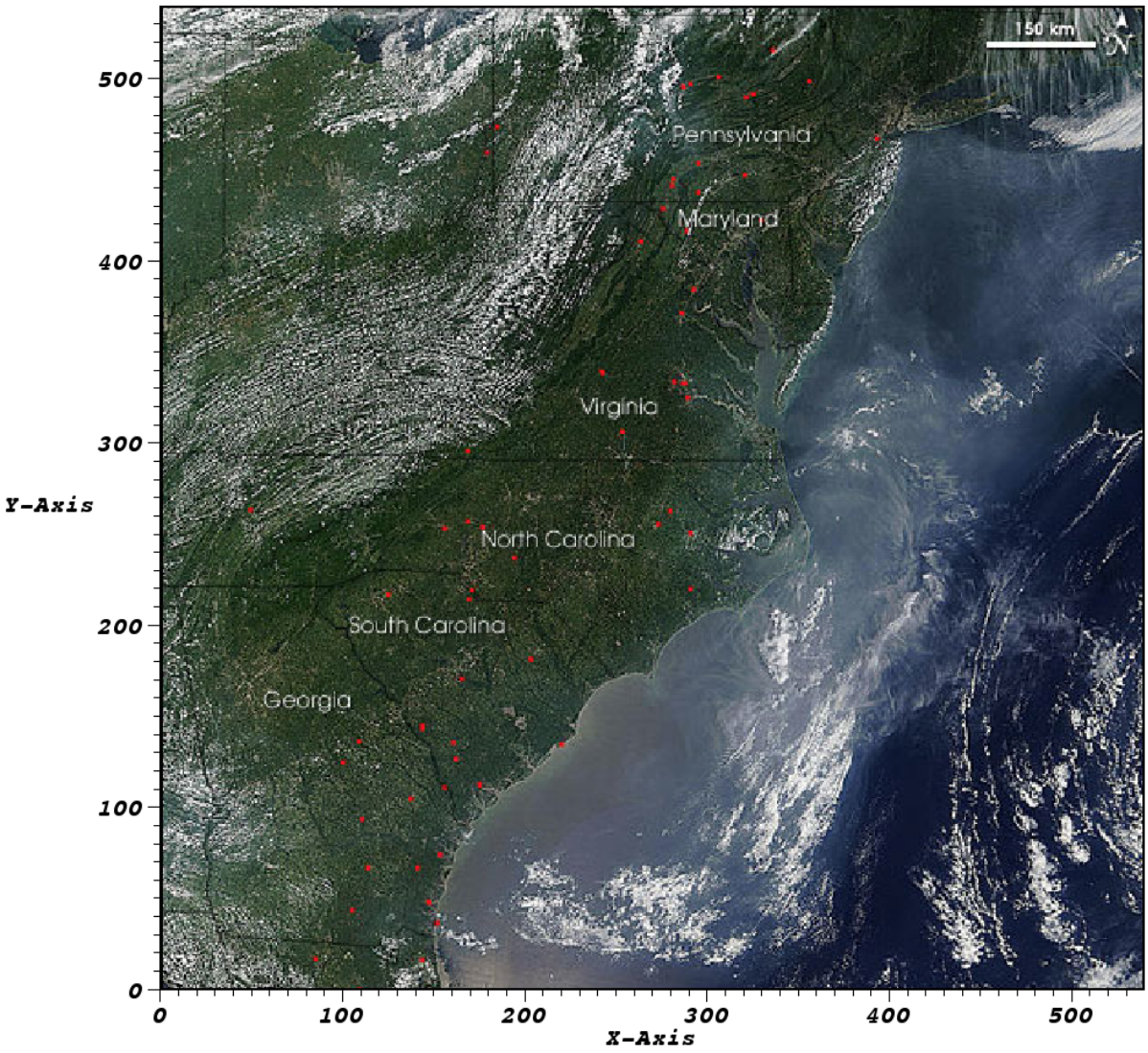


Fig. 4.66: Truecolor Plot

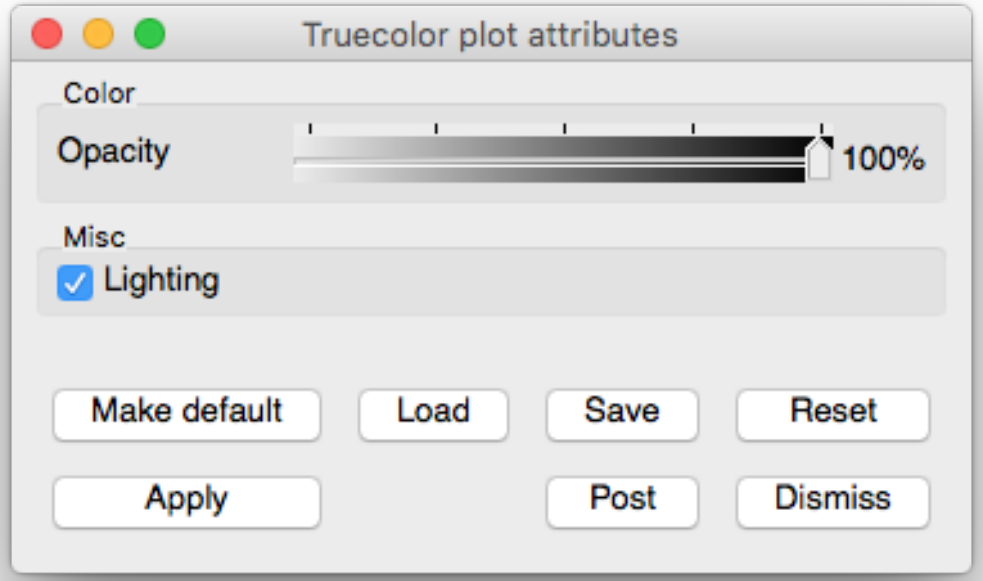


Fig. 4.67: Truecolor Plot Attributes

takes in a color variable, represented in VisIt as a three or four component vector, and uses the vector components as the red, green, blue, and alpha values for the plotted image. This allows you access to many more colors than other plots like the Pseudocolor plot, which can be used only to plot a single color component of an image.

Vector plot

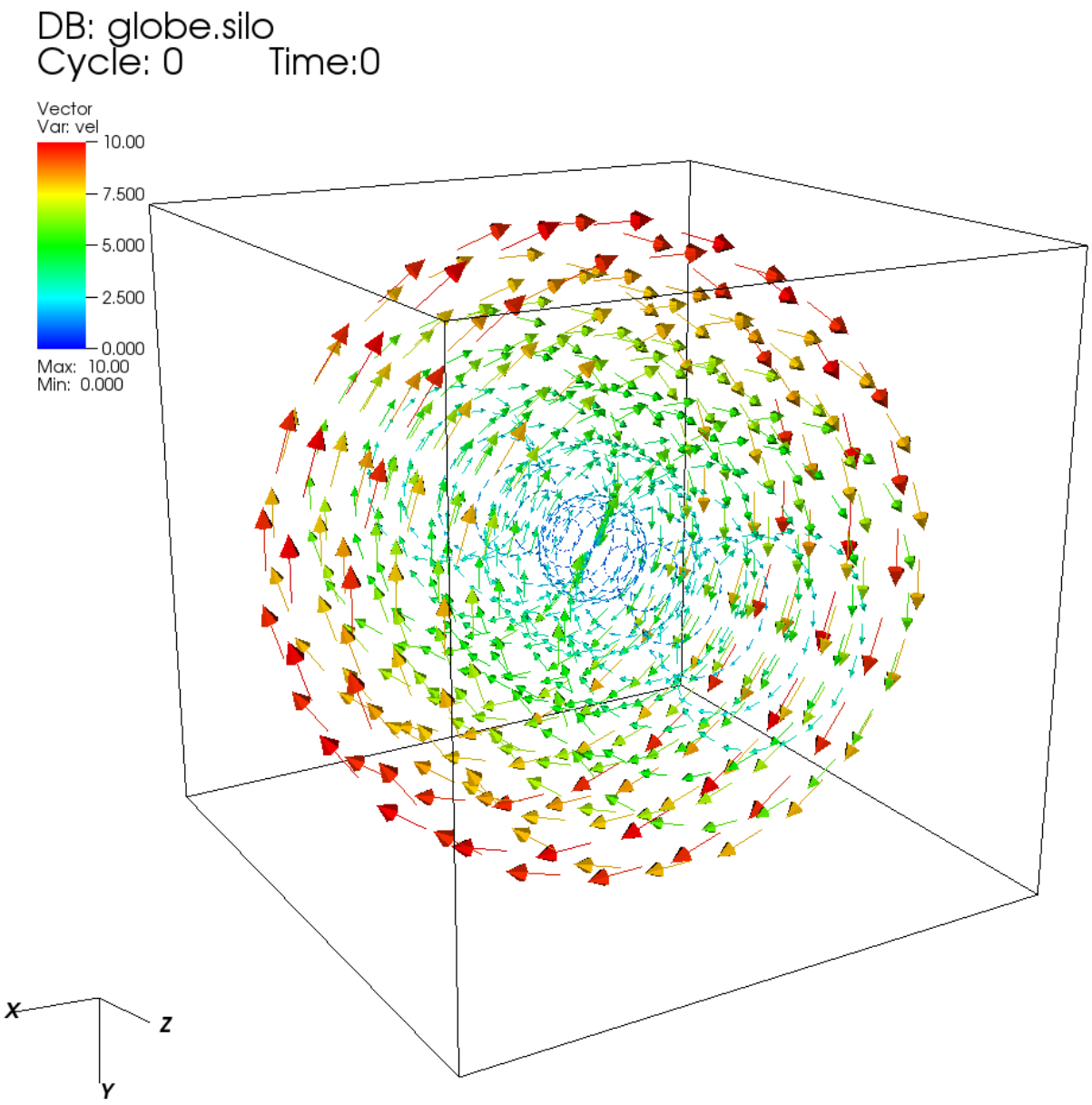


Fig. 4.68: An example vector plot

The Vector plot (example shown in Figure 4.68) displays vector variables using glyphs that indicate the direction and magnitude of vectors in a vector field.

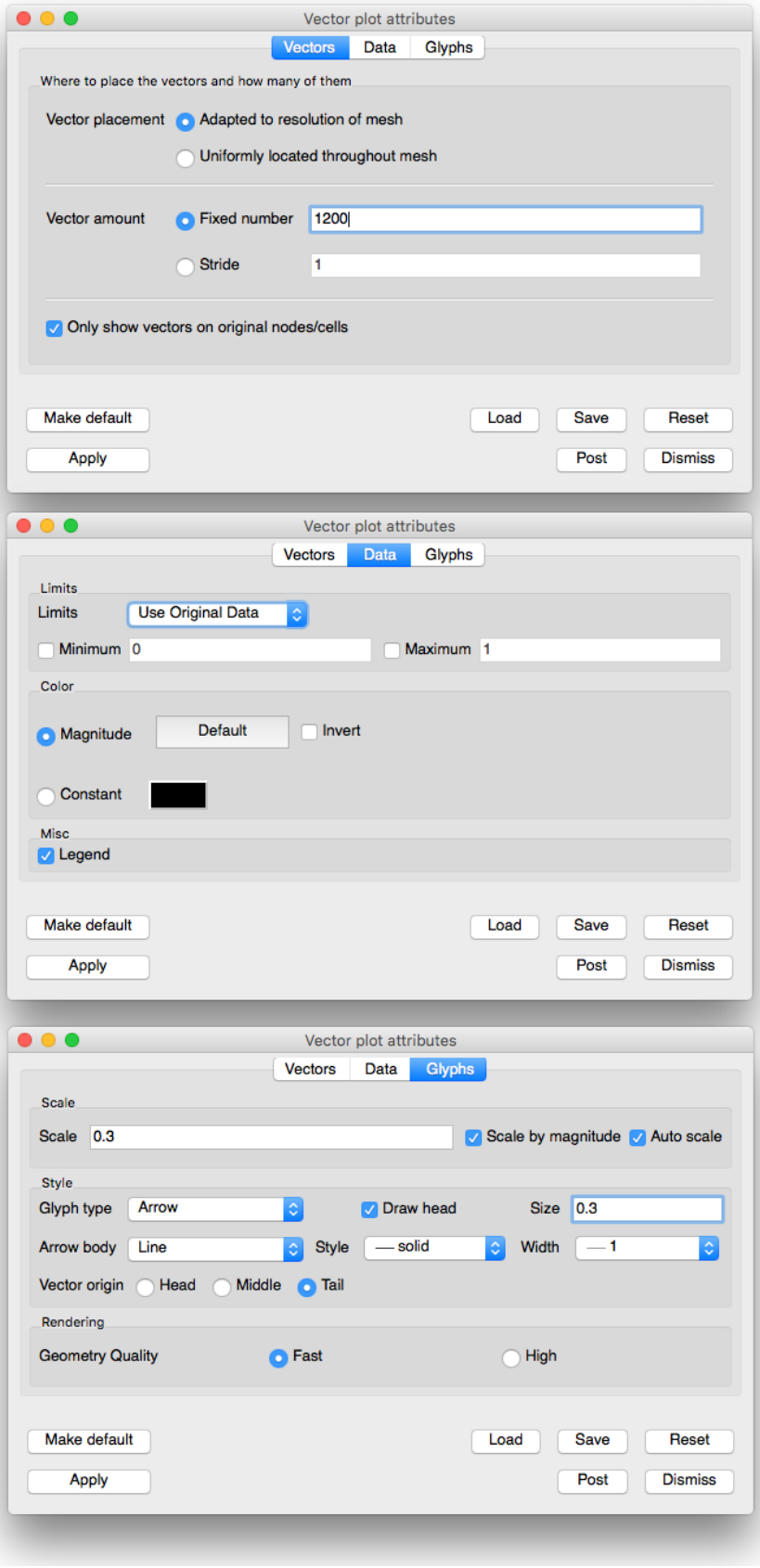


Fig. 4.69: The corresponding Vector plot attributes

Setting vector color

The vectors in the Vector plot can be colored by the magnitude of the vector variable or they can be colored using a constant color. Choose the coloring method by clicking on either the **Magnitude** radio button or the **Constant** color button. When vectors are colored by a constant color, you can change the color by clicking on the color button next to the **Constant** radio button and choosing a new color from the **Popup color menu**. When vectors are colored by magnitude, the color is determined by one of VisIt's color tables, which can be chosen from the **Color table** button next to the **Magnitude** radio button.

If you choose to color the vectors by their magnitudes, you have the option of also specifying minimum and maximum values to aid in the mapping of vector magnitude to color. The options that are used to aid coloring are collectively known as limits. Limits can apply to all vectors that exist in the dataset or just the vectors that have been drawn by the Vector plot. To specify which, choose the appropriate option from the **Limits** combo box. When you specify a minimum value all vectors with magnitudes less than the minimum value are colored using the color at the bottom of the color table. When you specify a maximum value all vectors with magnitudes larger than the maximum value are colored using the color at the top of the color table. To provide a minimum value, check the **Min** check box and type a new minimum value into the **Min** text field. To provide a maximum value, check the **Max** check box and type a new maximum value into the **Max** text field.

Vector scaling

The size of the vector glyphs has a tremendous effect on the Vector plot's readability. VisIt uses an automatically computed scaling factor based on the diagonal of the bounding box as the size for the largest vector. You can augment this size by entering a new scale factor in to the **Scale** text field. It is also possible to turn off automatic scaling by turning off the **Auto scale** check box. When automatic scaling is turned off, the vectors in the Vector plot are the length specified in the **Scale** text field.

If you want each vector to be further scaled by its own magnitude, you can turn on the **Scale by magnitude** check box. When the **Scale by magnitude** check box is off, all vectors are the same length as determined by the automatically computed scale factor and the user-specified scale.

Heads on the vector glyph

You can control the vector head size by typing a new value into the **Head size** text field, which is the fraction of the entire vector's length that will be devoted to the vector's head. Vectors in the Vector plot can be drawn without vector heads so that only the line part of the vector glyph is drawn. This results in cleaner plots, but the vector direction is lost. To turn off vector heads, uncheck the **Draw head** check box at the bottom of the **Vector Attributes Window**.

Tails on the vector glyph

The length of the tails on the vector glyph are determined by the vector scaling factors that have been enabled. You can also set properties that determine the location and line properties used to draw a vector glyph's tail. First of all, you can set the line style used to draw the vector glyph's tail by choosing a line style from the **Line style** combo box. You can choose a new line width for the vector glyph's tail by choosing a new line width from the **Line width** combo box. Finally, you can determine where the origin of the vector is on the vector glyph. The vector origin is a point along the length of the vector that is aligned with the node or cell center where the vector glyph will be drawn. The available options are: Head, Middle, and Tail. You can choose a new Vector origin by clicking on one of the **Head**, **Middle**, or **Tail** radio buttons.

Setting the number of vectors

When visualizing a large database, a Vector plot will often have too many vectors. The Vector plot becomes incomprehensible with too many vectors. VisIt provides controls to thin the number of vectors to a number that looks appealing in a visualization. You can accomplish this reduction by setting a fixed number of vectors or by setting a stride. To set a fixed number of vectors, select the **Fixed vectors** radio button and enter a new number of vectors into the corresponding text field. To reduce the number of vectors by setting the stride, select the **Stride** radio button and enter a new stride value into the **Stride** text field.

Volume plot

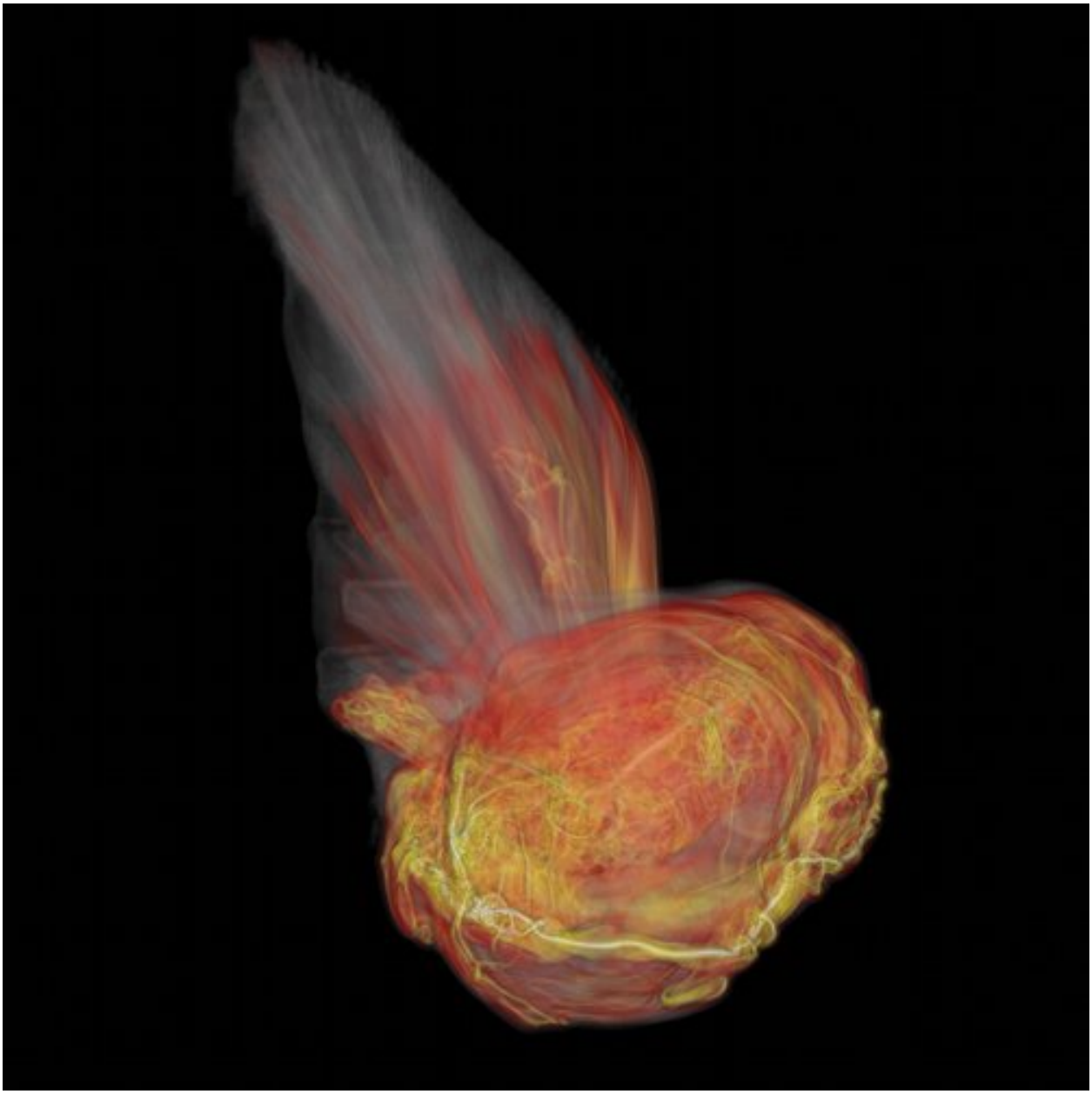


Fig. 4.70: Type Ia Supernova (Image Credit: Blue Waters visualization staff, Rob Sisneros and Dave Semeraro)

The Volume plot uses a visualization technique known as volume-rendering, which assigns color and opacity values to a range of data values. The colors and opacities are collectively known as a volume transfer function. The volume transfer function determines the colors of the plot and which parts are visible. The plot shown in (Figure 4.70) uses volume-rendering for the magnitude of vorticity. The magnitude of vorticity is a measure of turbulence that helps identify a *bubble* within the supernova.

The **Volume Plot Attributes Window**, shown in (Figure 4.71), is divided into two main tabs. The **Rendering Options** tab controls the rendering setting. Each volume rendering method has a different set of inputs. Additionally, the **Rendering Options** tab contains controls for lighting. The **Transfer function** tab has controls for how the data is mapped to colors and the opacities to use for different scalar values.

Rendering Options

The volume plot uses hardware-accelerated graphics by default (Serial). Though this mode is faster, the image resolution is typically lower. Images drawn by software volume rendering, Parallel, Compositing, Integration, SLIVR typically have a higher resolution and thus are more accurate. Note that software volume rendering can be a compute intensive process if the database or the visualization window is large. Shrinking the size of the visualization window before using a software rendering method will reduce the time and resources required to draw the plot.

It is worth noting that if the dataset is large with intricate details, the software volume rendering method is preferred because it scales well in parallel. Using a parallel compute engine can greatly speed up the rate at which software volume rendering operates as long as the dataset is domain-decomposed into roughly equal-sized pieces.

The software volume rendering modes use a technique called ray-casting. In ray-casting, a ray is followed in reverse from the computer screen into the dataset. As a ray progresses through the dataset, sample points are taken and the sample values are used to determine a color and opacity value for the sample point. Each sample point along the ray is composited to form a final color for the screen pixel. Rays are traced from closest to farthest to allow for early ray termination which stops the sampling process when the pixel opacity gets above a certain threshold. This method of volume-rendering yields superior pictures at the cost of speed and memory use.

Rendering Method: Serial Rendering (Figure 4.73).

Rendering Method: Parallel Rendering (Figure 4.74).

Serial and Parallel Rendering Options:

When rendering in Serial and Parallel the data must be defined on a rectilinear grid, which often requires the data to be resampled. The user may select one the following options:

`No Resampling`: Do not resample the data.

`Only if required`: Automatically resample the data but only if needed.

`Single Domain`: Resample the data on to a single rectilinear grid on rank zero (0).

`Parallel Redistribute`: Resample the data over all ranks on to a rectilinear grid and redistribute the results over all ranks.

`Parallel Per Rank`: Resample the data on each rank on to a rectilinear grid. Does not account for multiple samples on boundaries.

Two other resampling options:

`Number of Samples`: The number of samples, may be over all ranks or per rank.

`Centering`: Native, Nodal, or Cell. Resample on a native, nodal or cell basis. The centering should typically be Native. Note: when using an opacity variable it must have the same centering as the color data.

The user may also select OSPRay rendering <https://www.ospray.org>‘_ which is an Open source, Scalable, and Portable Ray tracing engine for volume rendering on Intel Architecture CPUs.

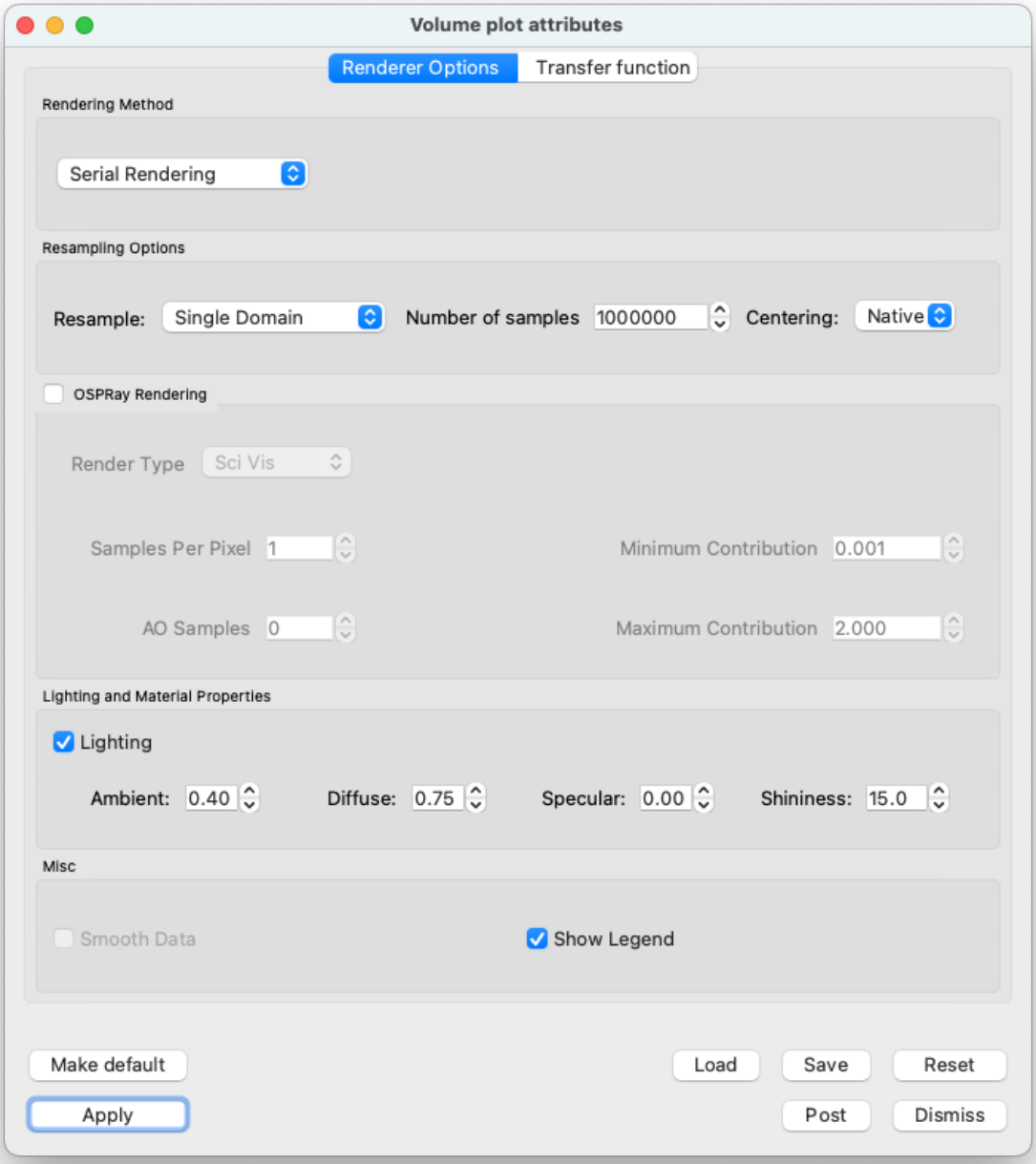


Fig. 4.71: The rendering attributes for the Volume plot

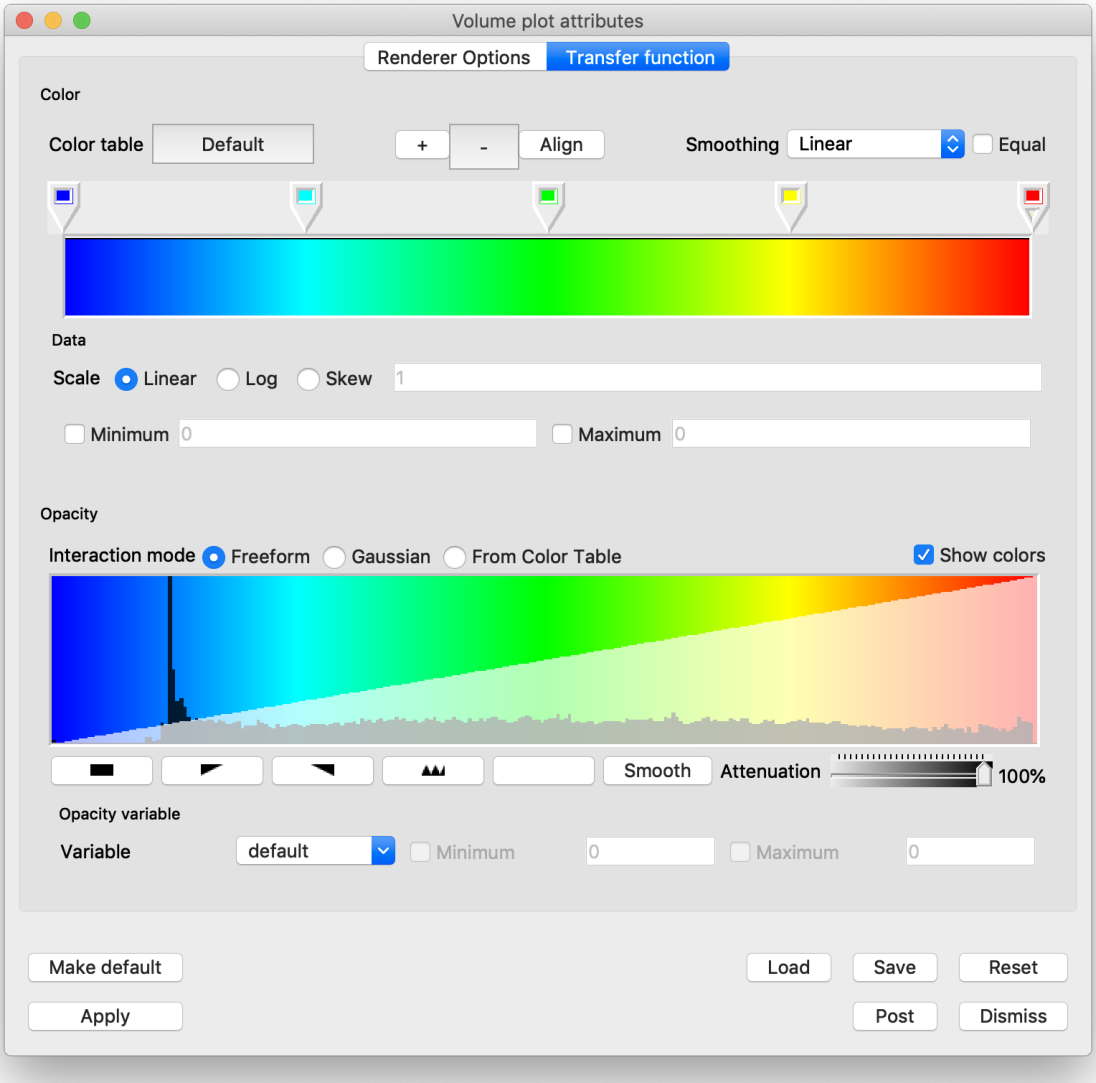


Fig. 4.72: The transfer function editor for the Volume plot

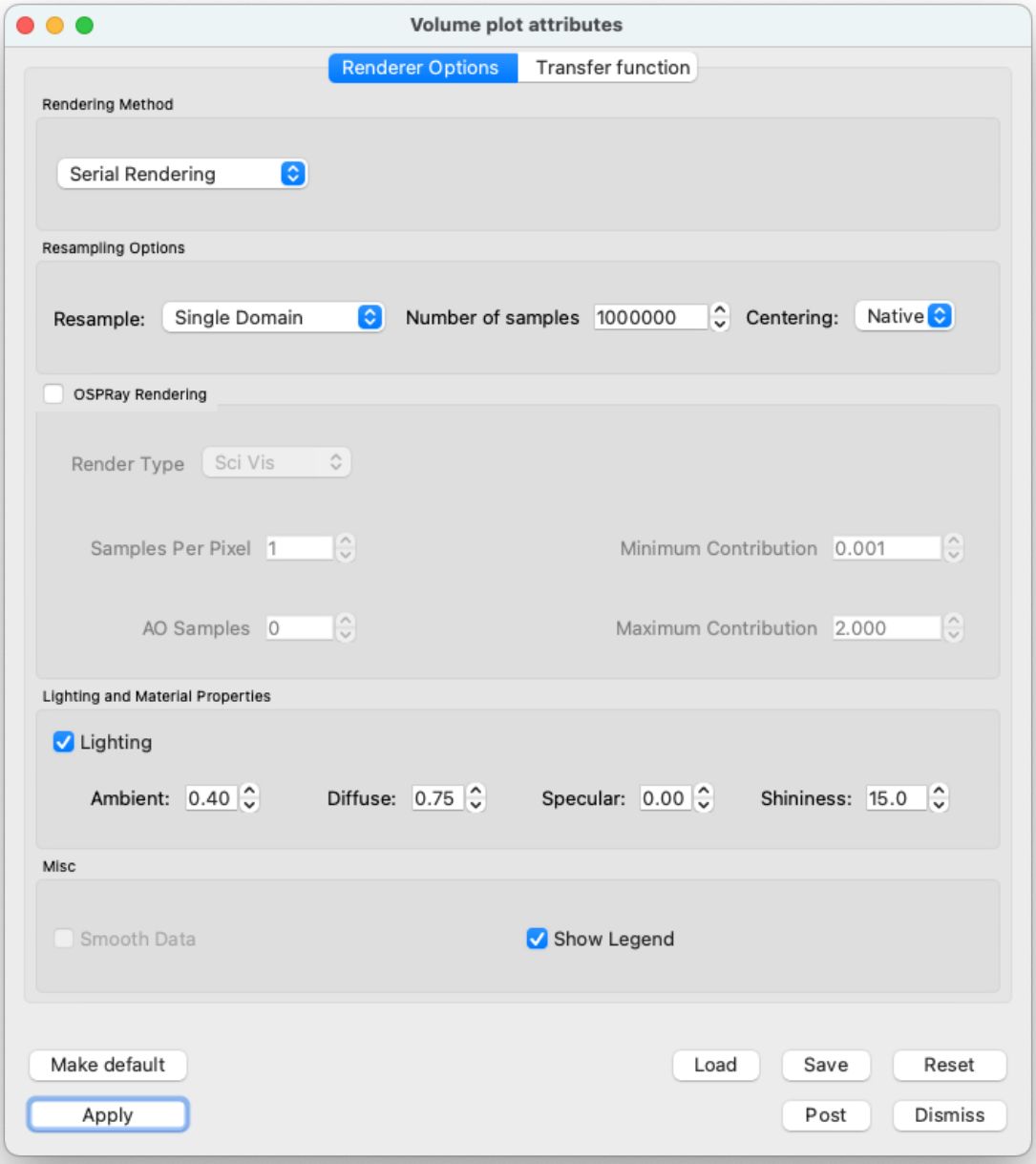


Fig. 4.73: Serial Rendering options

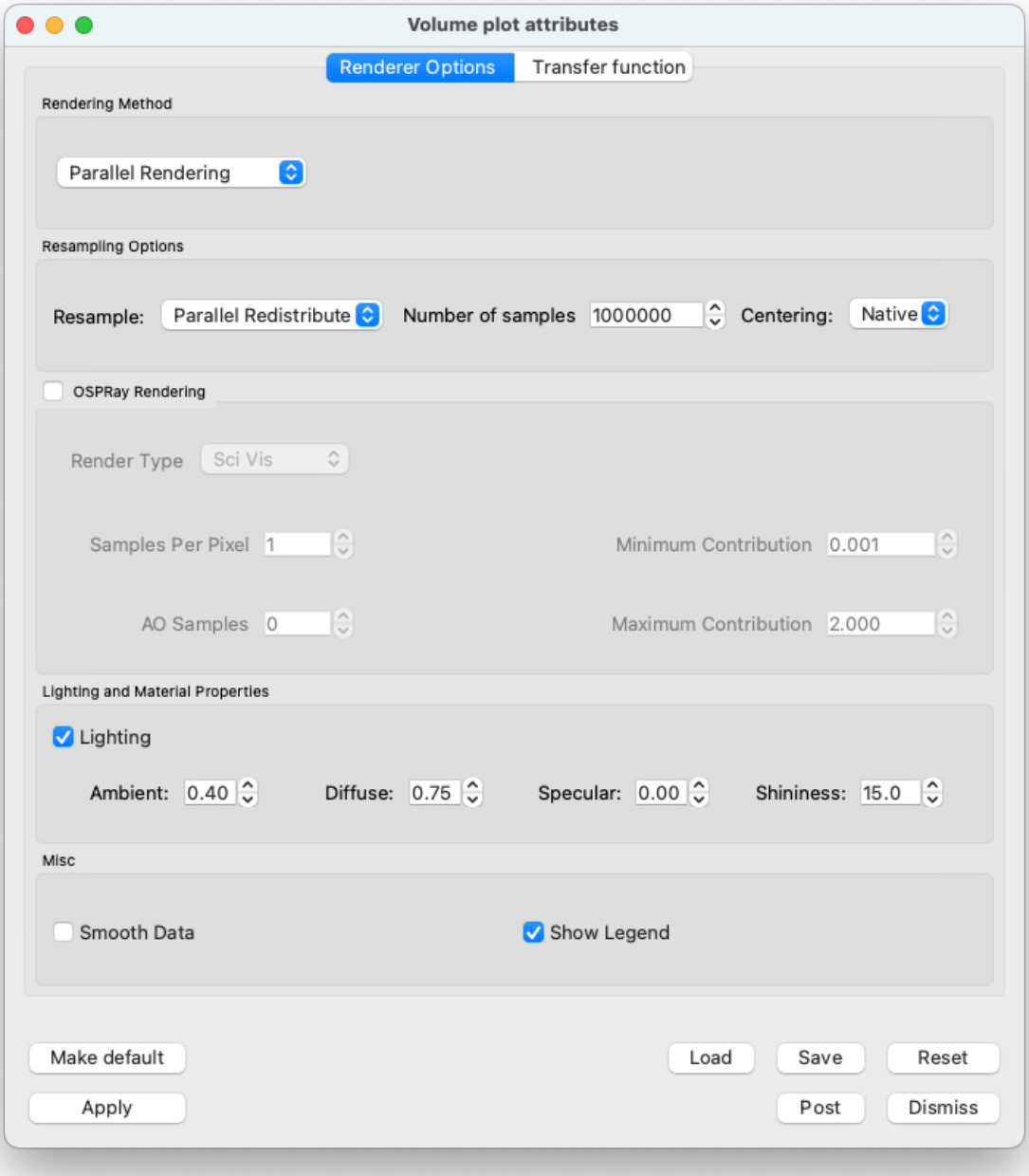


Fig. 4.74: Parallel Rendering options

`Rendering Type`: Sets rendering type to be either "scivis" or "path traced" photo-realism.

`AO Samples`: determines the number of rays per sample to compute ambient occlusion.

`AO Distance`: determines the maximum distance to consider for ambient occlusion.

`Minimum Contribution`: The minimum sample contribution.

`Maximum Contribution`: The maximum sample contribution.

The volume plot can use lighting to enhance the look of the plot. Lighting is enabled by default but the user can disable it by unchecking the **Lighting** check box near the bottom of the window.

`Ambient`: ambient light weight in [0-1]

`Diffuse`: diffuse reflection weight in [0-1]

`Specular`: specular reflection/transmission weight in [0-1]

`Shininess`: Phong exponent, usually in [2-10^4]

Rendering Method: Compositing (Figure 4.75)

Rendering Method: Integration (grey scale) (Figure 4.76)

Rendering Method: SLIVR (Figure 4.77)

Transfer Function

You can design the color component of the volume transfer function using the controls in **Transfer function** tab of the **Volume Plot Attributes Window**. The controls are similar to the controls for the **Color Table Window**. There is a color spectrum that has color control points which determine the final look of the color table. Color control points are added and removed using the + and − buttons. Dragging control points with the mouse moves them and changes their order. Right-clicking on a color control point displays a popup color menu from which a new control point color can be chosen.

The **Transfer function** tab provides controls for setting the limits of the variable being plotted. Limits are artificial minima or maxima that are specified by the user. Setting the limits to a smaller range of values than present in the database cause the plot's colors to be distributed among a smaller range of values, resulting in a plot with more color variety.

To set the limits you first click the **Min** or **Max** check box next to the **Min** or **Max** text field. Clicking a check box enables a text field into which the user can type a new minimum or maximum value.

Like VisIt's other plots that map scalar values to colors, the Volume plot allows for the data values to be scaled using Linear, Log, and Skew functions. To select a scaling function other than linear where values in the data range are mapped 1:1 to values in the color range, click on the **Log** or **Skew** radio buttons.

Setting opacities

The **Transfer function** tab provides several controls that allow the user to define the opacity portion of the volume transfer function. The opacity portion of the volume transfer function determines what can be seen in the volume-rendered image. Data values with a lower opacity allow more to be seen and give the plot a gel-like appearance, while data values with higher opacity appear more solid and occlude objects behind them. The controls for setting opacities are located at the button of the window in the **Opacity** area.

You can set opacity three ways. You can hand-draw an opacity map, create it by designing curves that specify the opacity when they are added together, or use the opacities in the color table, if present. All methods use the controls shown in Figure 4.71.

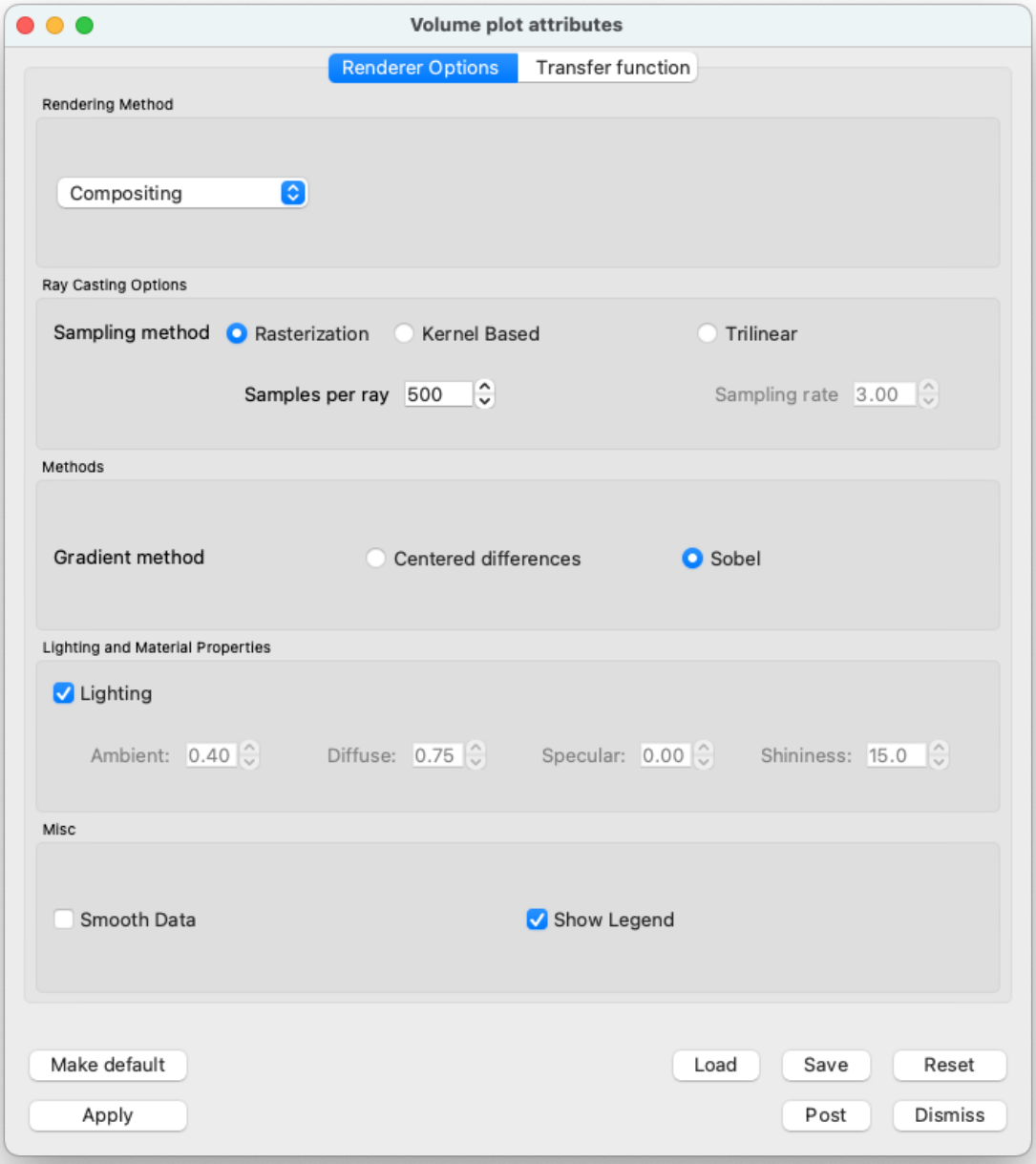


Fig. 4.75: Compositing options

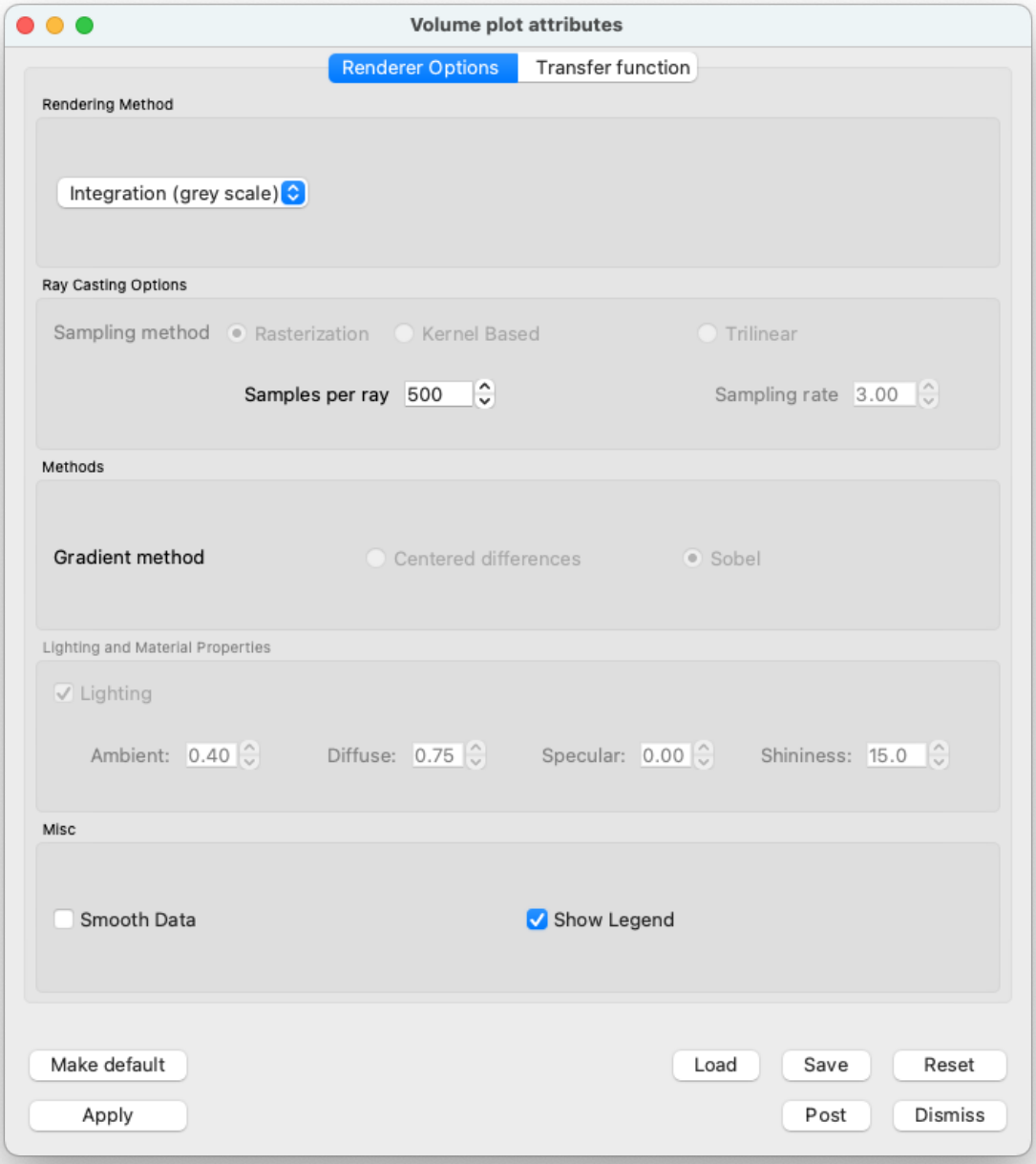


Fig. 4.76: Integration (grey scale) options

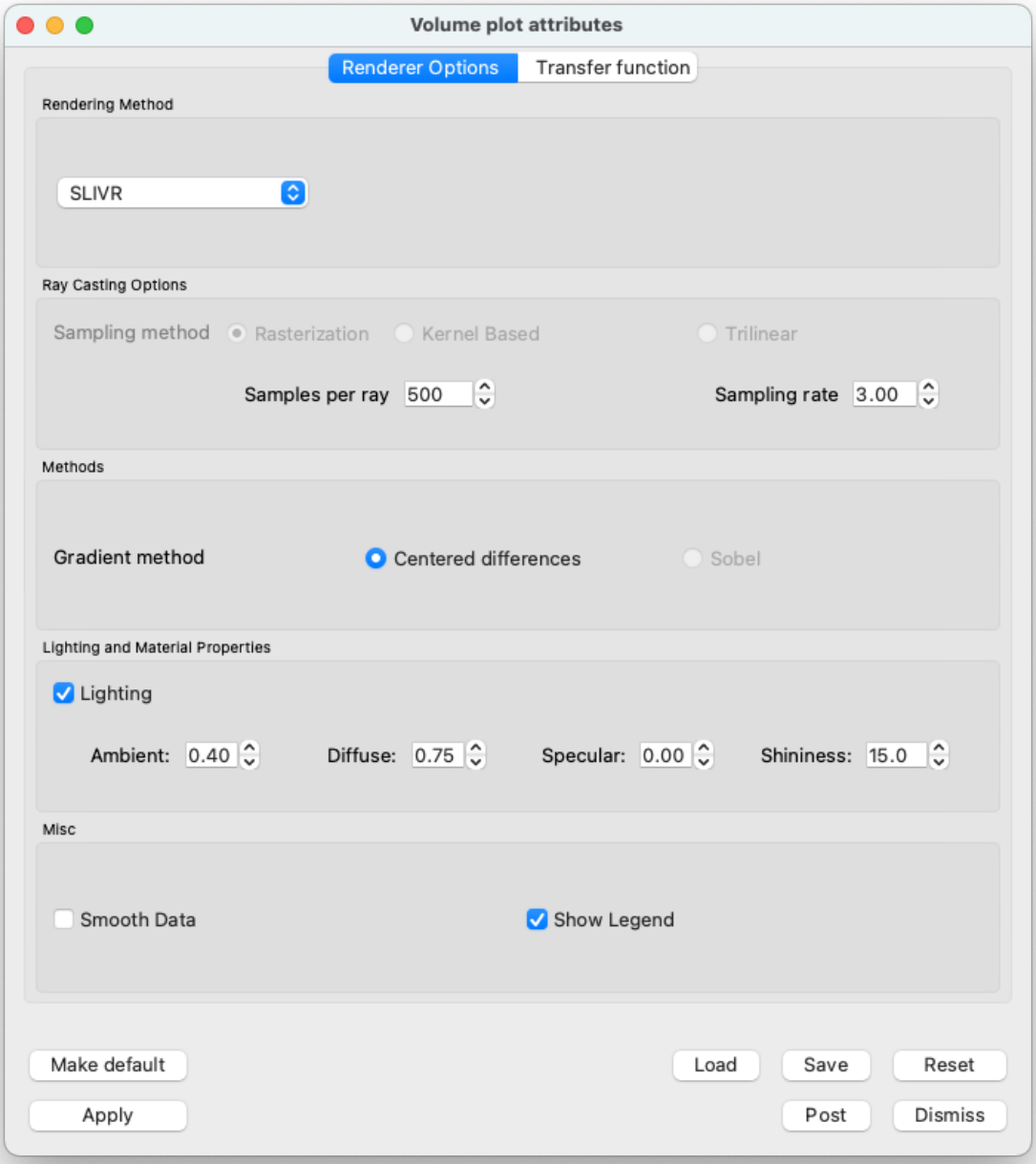


Fig. 4.77: SLIVR options

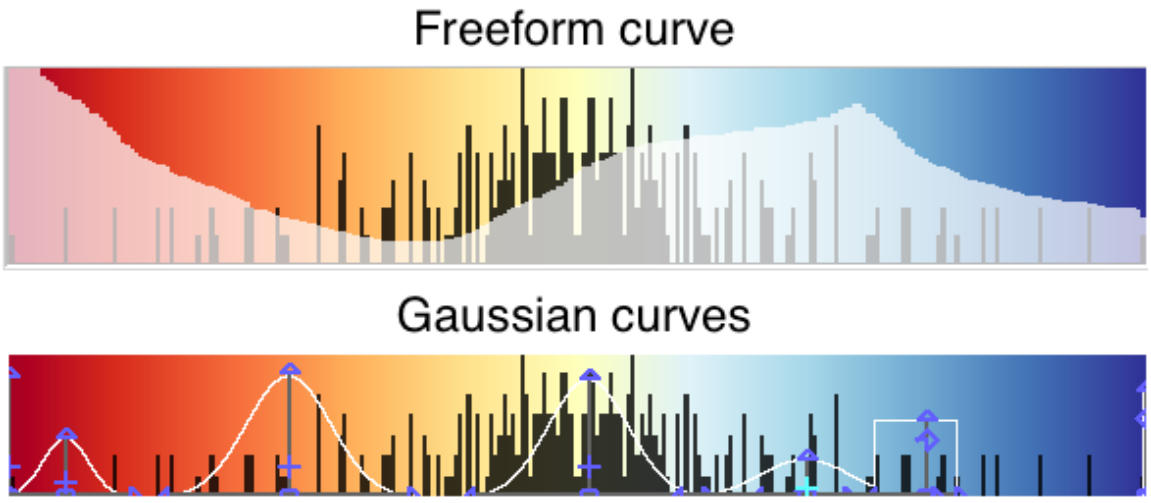


Fig. 4.78: Volume Plot Opacity Options

The interaction mode determines how opacity is set. Clicking on the **Freeform** or **Gaussian** radio buttons selects the interaction mode. If the interaction mode switches from **Gaussian** to **Freeform**, the shape constructed by the **Gaussian** controls is copied to the **Freeform** control. Both controls pretend that the plot's data range is positioned horizontally such that the values on the left of the control correspond to the low data values while the values on the right of the control correspond to high data values. In addition to the color map, there is a histogram of the current data to aide in setting opacity of interesting values. The vertical direction corresponds to the opacity for the given data value. Taller curves are more opaque while shorter curves are more transparent.

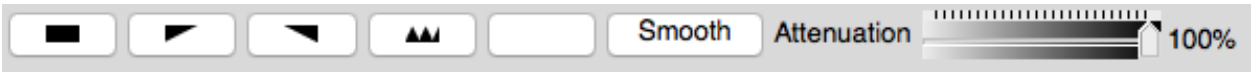


Fig. 4.79: Volume Plot Freeform Opacity Options

To design an opacity map using the **Freeform** control, position the mouse over it and click the left mouse button while moving the mouse. The shape traced by the mouse is entered into the **Freeform** control so that the user can draw the desired opacity curve. Immediately under the **Freeform** control, there are four buttons, shown in (Figure 4.79), which can be used to manipulate the curve. The first three buttons initialize a new curve. The black button makes all data values completely transparent. The ramp button creates a linear ramp of opacity that emphasizes high data values. The white button makes all data values completely opaque. The **Smooth** button smooths out small bumps in the opacity curve that occur when drawing the curve by hand.

Fig. 4.80: Volume Plot Gaussian Opacity Options

The **Gaussian** control used during Gaussian interaction mode is complex but it provides precise control over the shape of a curve. The basic paradigm followed by the **Gaussian** control is that new curves are added and reshaped to yield the desired opacity curve. You add new curves by clicking and dragging in the control. Right clicking with the mouse on an existing curve removes the curve. Each curve has five control points which can change the curve's position

and shape. The control points are shown along with the shapes that a curve can assume. A control point changes color when it becomes active so there the user knows which control point is used. Curves start as a smooth Gaussian shape but they can change between the shapes shown in by moving the shape control point up and down or left and right. Opacity maps are typically created by adding several curves to the window and altering their shapes and sizes until the desired image is obtained in the visualization window. The **Attenuation slider**, the final control involved in creating an opacity map, controls the opacity of the entire opacity map defined by the **Freeform** or **Gaussian** controls. It provides a knob to scale all opacities without having to modify the opacity map.

Changing the opacity variable

The variable used to determine opacity does not have to be the plotted variable. Having a different opacity variable than the plotted variable is useful for instances in which the user wants to determine the opacity using a variable like density while coloring the plot by another variable such as pressure. To change the opacity variable, select a new variable from the **Opacity variable** variable menu. By default, the plotted variable is used as the opacity variable. This is implied when the **Opacity variable** variable button contains the word default. Even when "default" is chosen, it is possible to set artificial data limits on the opacity variable by entering new values into the **Min** or **Max** text fields.

Controlling image quality

When the Volume plot is drawn with graphics hardware, the database is resampled onto a rectilinear grid that is used to place the polygons that are drawn to produce the image. You can control the coarseness of the resampled grid with the **Number of samples** text field. To increase the number of sample points, enter a larger number into the **Number of samples** text field.

When the Volume plot is drawn in ray casting mode, the number of samples along each ray that is cast through the data becomes important. Having too few sample points along a ray gives rise to sampling artifacts such as rings or voids. The user should adjust this number until satisfied with the image. More samples generally produce a better image, though the image will take longer to produce. To change the number of samples per ray, enter a new number of samples per ray into the **Samples per ray** text field.

When using lighting, the gradient calculation method that the Volume plot uses influences the quality of the images that are produced. By default, VisIt uses the Sobel operator, which uses more information from adjacent cells to calculate a gradient. When the Sobel operator is used to calculate the gradient, lighting usually looks better. The alternative gradient calculation method is centered-differences and while it is much less compute intensive than the Sobel operator, it also produces lesser quality gradient vectors, which results in images that are not lit as well. To change the gradient calculation method, click on either the **Centered diff** or **Sobel** radio buttons.

4.4 Operators

This chapter explains the concept of an operator and goes into detail about each of VisIt's operators.

4.4.1 Working with Operators

An operator is a filter applied to a database variable before the compute engine uses that variable to generate a plot. VisIt provides several standard operator types that allow various operations to be performed on plot data. The standard operators perform data restriction operations like planar slicing, spherical slicing, and thresholding, as well as more sophisticated operations like peeling off mesh layers. All of VisIt's operators are plugins and new operators can be written to extend VisIt in new ways. For help creating new operator plugins contact us via our getting help page.

Managing operators

When an operator is applied to a plot, it modifies the data that the plot uses to generate a visualization. Any number of operators can be applied to a plot. Each operator added to a plot restricts or modifies the data that is supplied to the plot. Very sophisticated visualizations can be created by using a series of operators.

The controls for the operators are found in the same location as the plot controls. The plot list, which displays the list of plots found in the current visualization window, also displays the operators applied to each plot. Each entry in the plot list displays the database name (when there is more than one open source), the plot type, the variable, and all operators that are applied to the plot. When an operator is applied to a plot, the name of the operator is inserted in front of the plot variable. If multiple operators are applied to a plot, the most recently added operator appears first when reading left to right while the operator that was applied first appears just to the left of the variable name. Plot list entries can also be expanded to allow the user to add, remove, reorder, and change the attributes of operators.

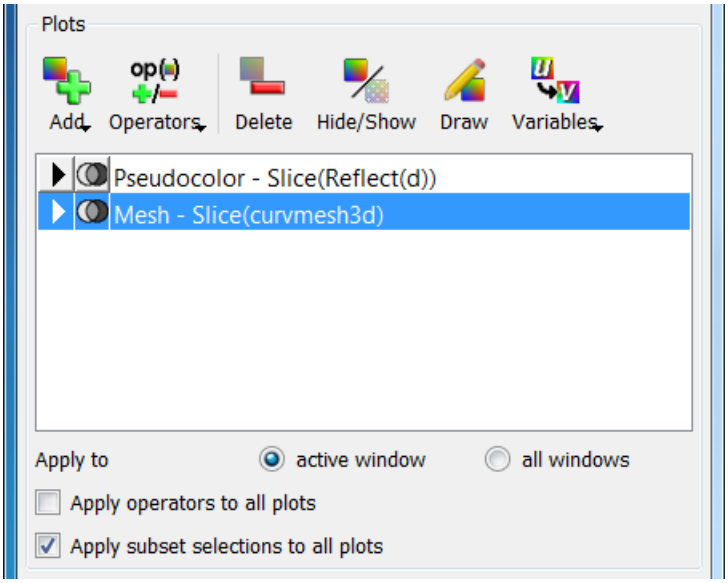


Fig. 4.81: The plots area

Adding an operator

Operators are added by selecting an operator from the **Operators** menu, shown in Figure 4.82. If an operator listed in this chapter is not listed in the **Operators** menu then the operator might not be loaded by default. To enable additional operators, use the *Plugin Manager Window*. When an operator is added, it applies the operator to the selected plots in the plot list unless the **Apply operators to all plots** check box is checked, in which case, the selected operator is applied to all plots in the plot list. By default, operators are applied to all plots in the plot list.

When an operator is added to a plot, the name of the operator appears in the plot list entry to the left of the variable or any previously applied operator. When an operator is added to an already generated plot, the plot is reset back to the new state to allow the user an opportunity to set the operator's attributes before the plot is regenerated. To regenerate the plot with the newly added operator, press the **Draw** button. It is also possible to apply an operator by clicking an operator attributes window's **Apply** button. When this occurs, a dialog window appears asking the user if the operator should be applied to the selected plots (see Figure 4.83).

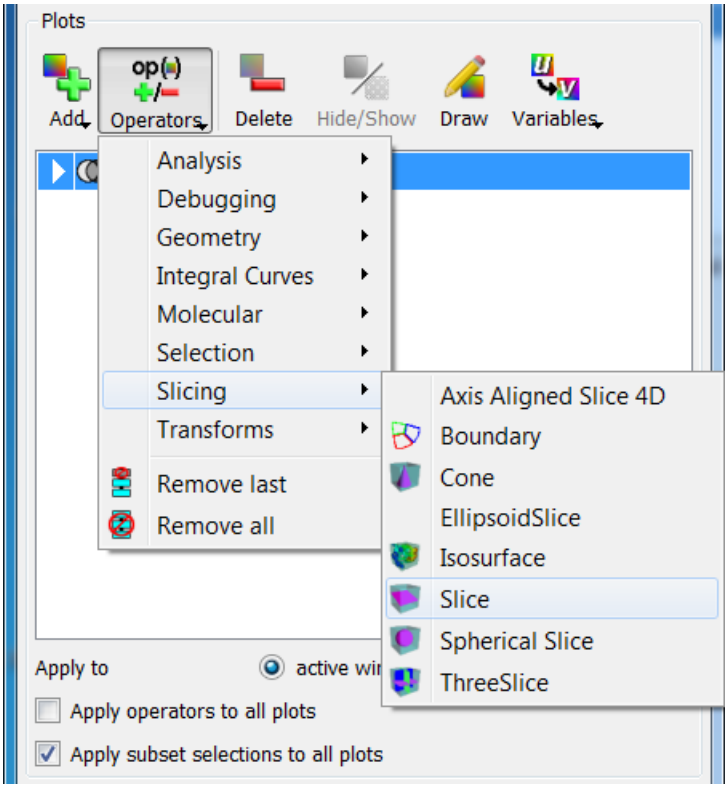


Fig. 4.82: The operators menu

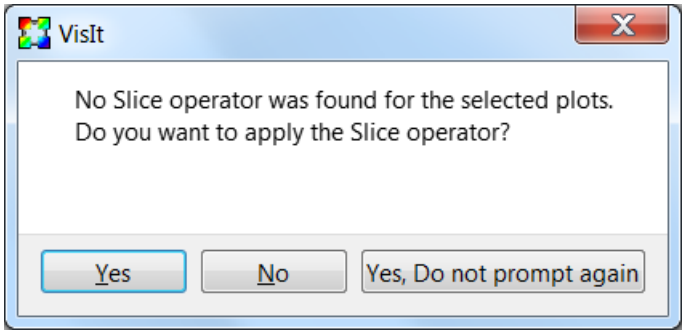


Fig. 4.83: The add operator dialog

Expanding plots

Plot list entries are normally collapsed by default with the operators applied to the plots shown in the plot list as a series of nested operators, which finally take a variable as an argument. The plot list allows plot list entries to be expanded on a per-plot basis so the user can get to each individual operator that is applied to a plot. To expand a plot list entry, click on its expand button, shown in Figure 4.84. When a plot list entry is expanded, the plot's database (if there is more than one open source), the variable, all the operators, and finally the plot get their own line in the plot list entry. This is significant because it allows operators to have additional controls to let you reposition them in the pipeline or remove them from the middle of the pipeline without having to first remove other operators.

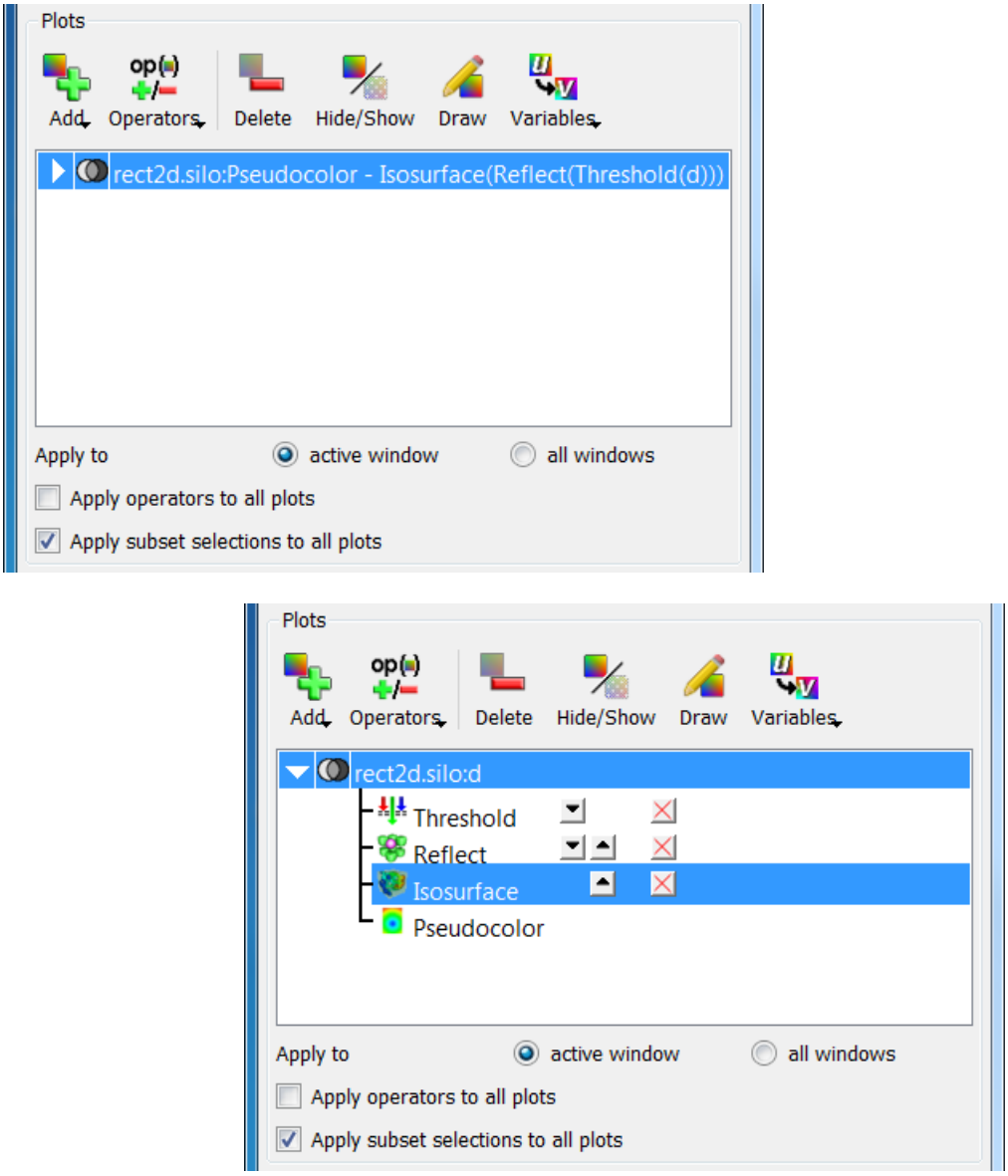


Fig. 4.84: A plot list entry before and after being expanded

Changing the order of operators

Sometimes with several operators applied, it is useful to change the order of the operators. For example, the user might want to apply a Slice operator before a Reflect operator instead of after it to reduce the amount of data that VisIt must process in order to draw your plot. The order in which operators are applied often has a significant impact on the visualization. Using the previous example, suppose a plot is sliced before it is reflected. The resulting visualization is likely to have a reflected slice of the original data. If the order of the operators was reversed so that the Reflect operator came first, the Slice operator's slice plane might not intersect the reflected data in the same way, which could result in a totally different looking visualization.

The plot list entry must be expanded in order to change the order of its operators. Once the plot list entry is expanded, each operator is listed in the order in which they were applied and each operator has small buttons to the right of its name that allow the operator to be moved up or down in the pipeline. To move an operator closer to the database so it is executed before it would have been executed before, click on the **Up** button next to an operator's name. Moving the operator closer to the database in the pipeline is called demoting the operator. Clicking the **Down** button next to an operator's name moves the operator to a later stage of the pipeline. Moving an operator to a later stage of the pipeline is known as promoting the operator since the operator appears closer to the plot in the expanded plot entry. Operators in the plot list entry that can only be moved in one direction have only the **Up** button or the **Down** button while operators in the middle of the pipeline have both the **Up** button and the **Down** button.

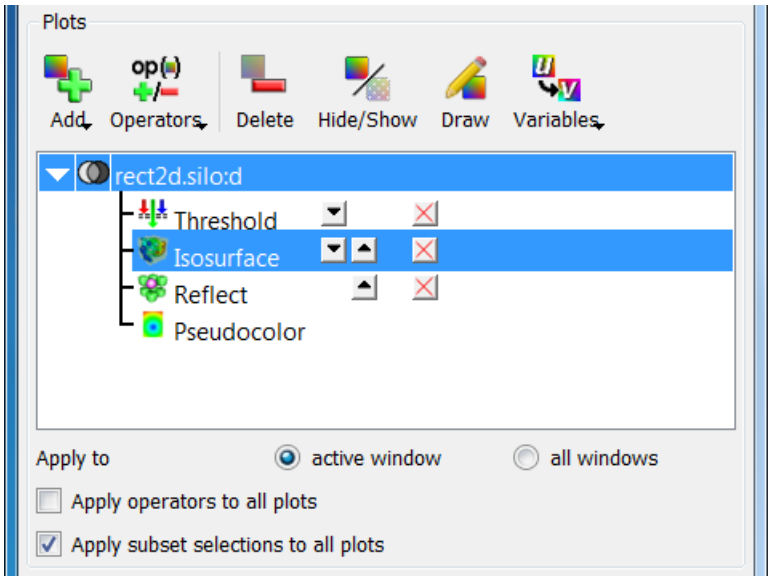


Fig. 4.85: The controls for changing operator order

Removing operators

There are two ways to delete an operator from a plot. The last two entries in the **Operators** menu have options that remove one or more operators. To remove only the last applied operator, select the **Remove last** option from the **Operators** menu. To remove all the operators applied to a plot, select the **Remove all** option from the **Operators** menu. Unless the **Apply operator to all plots** check box is checked, operators are only removed from selected plots. When an operator is removed in this manner and the plot has already been generated, it is immediately regenerated.

The **Operators** menu has controls that allow the last operator applied to a plot to be removed or all of a plot's operators to be removed. VisIt also provides controls that let you remove specific operators from the middle of a plot's operator list. First expand the plot list entry by clicking its **Expand** button and then click on the red **X** button next to the operator to be deleted. When an operator is deleted using the red **X** buttons, the plot is reset back to the new state so

the **Draw** button must be clicked to regenerate the plot. See Figure 4.86 for an example of deleting an operator from the middle of a plot's operator list.

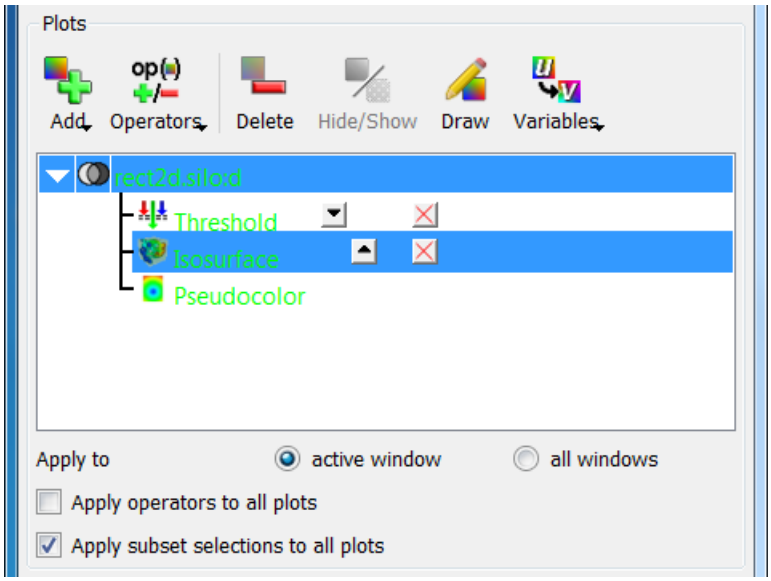


Fig. 4.86: After removing an operator from the middle of the pipeline

Setting operator attributes

Each operator type has its own attributes window used to set attributes for that operator type. Operator attribute windows are brought up by selecting the operator type from the **OpAtts** (Operator attributes) menu shown in Figure 4.87.

When there is only one operator of a given type in a plot's operator list, setting the attributes for that operator type will affect that one operator. When there are multiple instances of the same type of operator in a plot's operator list, only the active operator's attributes are set if the active operator is an operator of the type whose attributes are being set. The active operator is the operator whose attributes are set when using an operator attributes window and can be identified in an expanded plot entry by the highlight that is drawn around it (see Figure 4.88). To set the active operator, expand a plot entry and then click on an operator in the expanded plot entry's operator list.

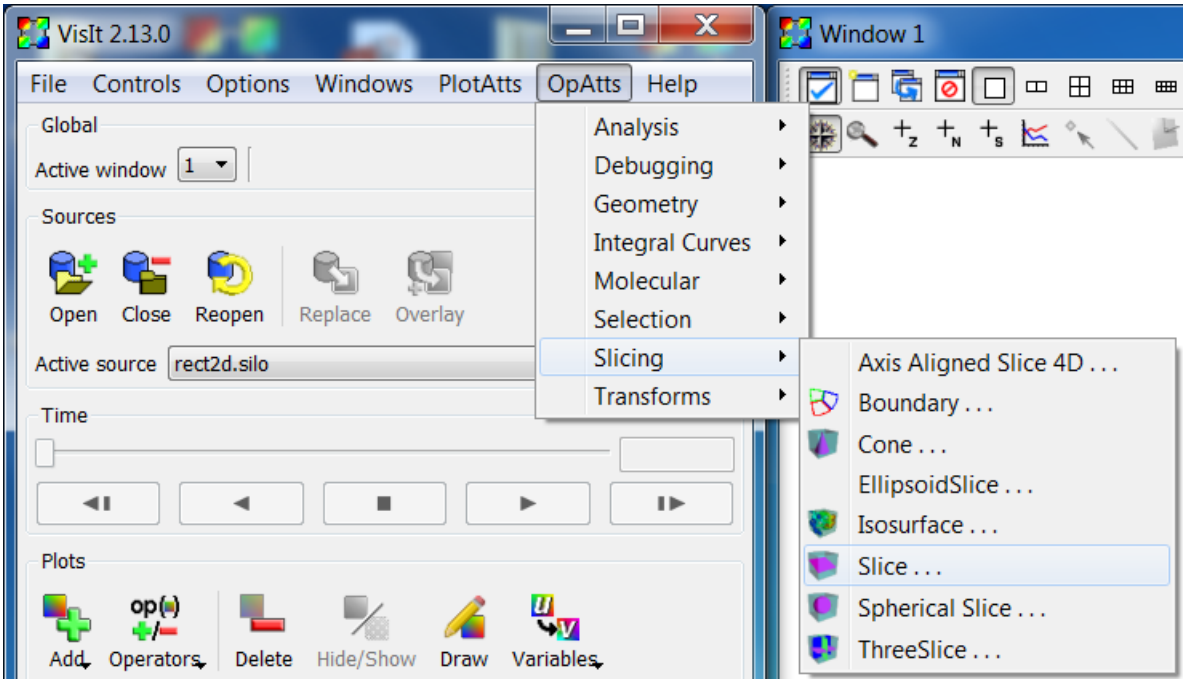
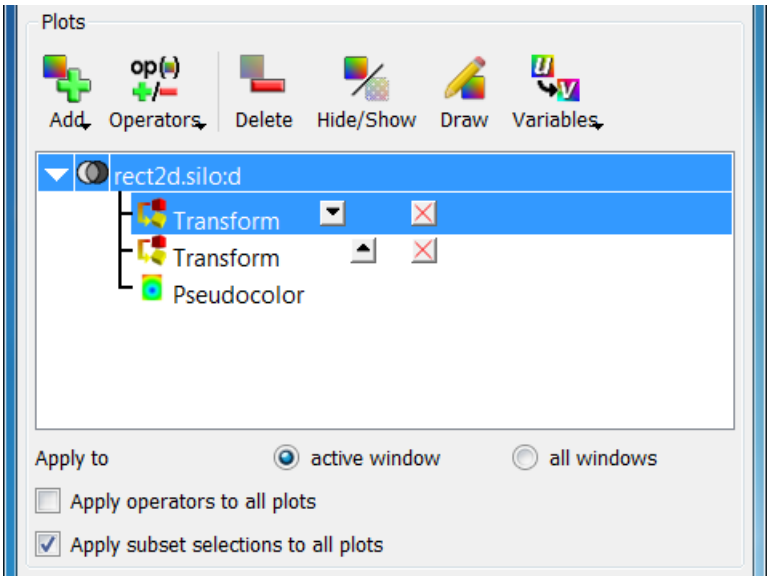


Fig. 4.87: The operator attributes menu



Setting the active operator is useful when there are multiple operators of the same type applied to the same plot. For example, there might be two Transform operators applied to a plot in order to scale a plot with one operator and then rotate the plot with the second Transform operator. In this case the user could add two Transform operators, make the first Transform operator active, set the scaling attributes, make the second Transform operator active, and set the rotation attributes.

4.4.2 Operators that Generate New Variables

Some of VisIt's operators act more like expressions in that they generate new variables that can be plotted. The variable type they output does not necessarily match the variable type they accept as input. For example, the IntegralCurve

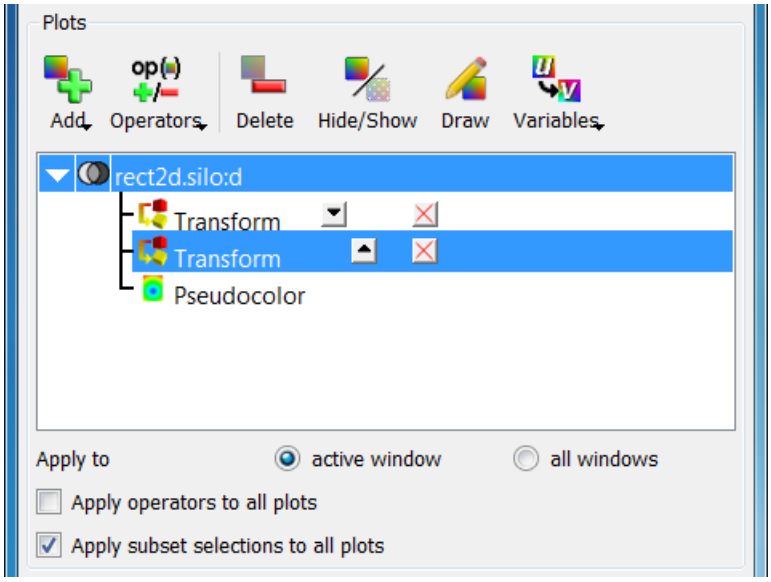


Fig. 4.88: Setting the active operator

operator accepts a Vector and outputs a Scalar, while the ConnectedComponents operator accepts a Mesh and outputs a Scalar.

Most of the operators that generate new variables are best applied using the **operators** submenu of a particular plot's **variable** menu. See Figure 4.89,

It is probably best after applying an operator in this fashion to open the Operator's attributes window to ensure good settings for your data before clicking **Draw**.

Operators that generate Scalars:

1. Connected Components
2. DataBinning
3. Flux
4. *Integral Curve operator*
5. *Lagrangian Coherent Structure (LCS) operator*
6. *Limit Cycle operator*
7. ModelFit
8. *Poincaré operator*
9. StatisticalTrends

Operators that generate Vectors:

1. *Lagrangian Coherent Structure (LCS) operator*
2. SurfaceNormal

Operators that generate Curves:

1. DataBinning
2. Lagrangian
3. LimitCycle

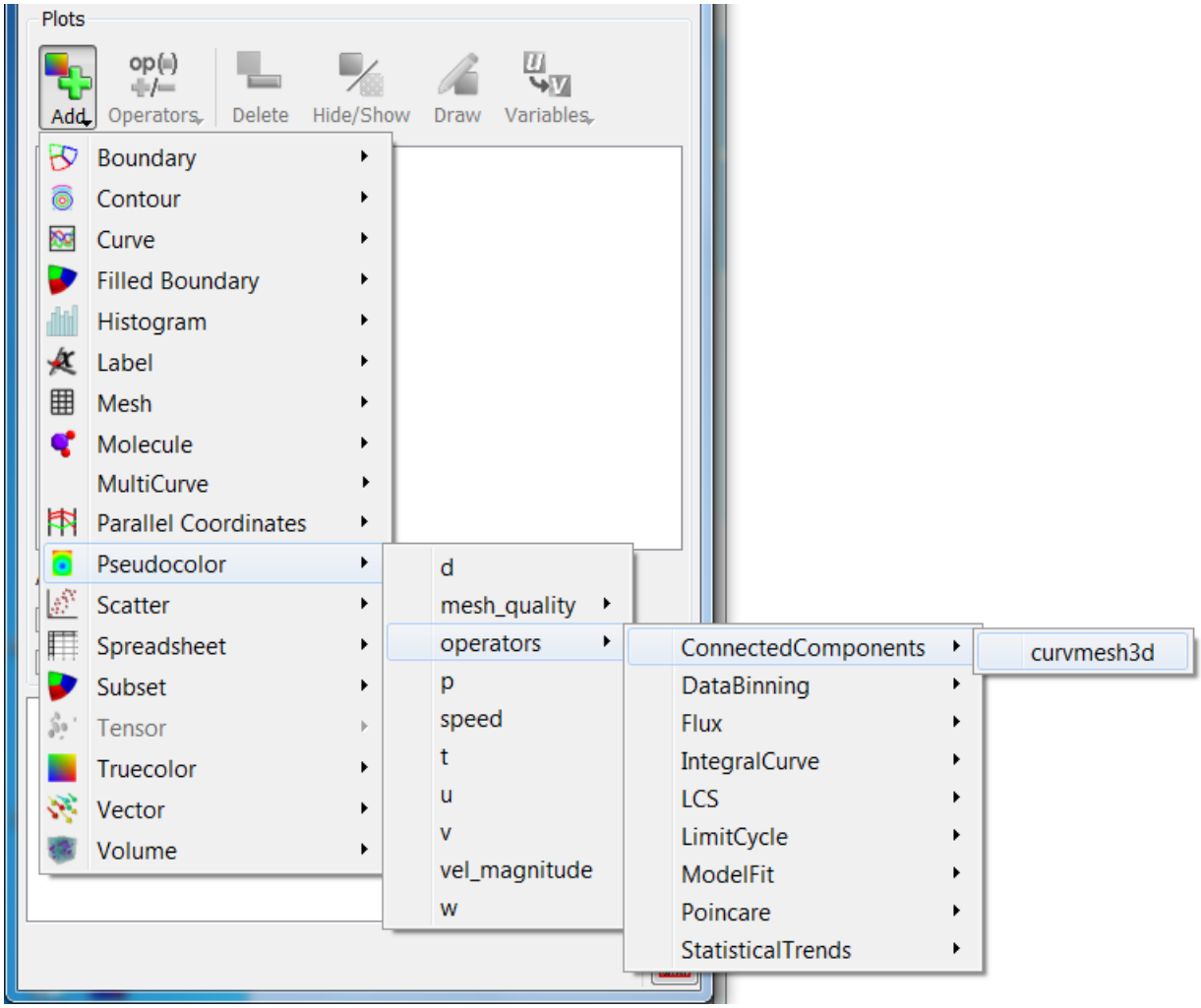


Fig. 4.89: The menu for applying an operator that generates a new variable.

4. Lineout

4.4.3 Operator Types

VisIt is installed with operator plugins, which perform a wide variety of functions. Some of the operators are not be enabled by default so they do not show up in the **Operator** menu. Use the *Plugin Manager Window*, which can be opened by clicking on the **Plugin Manager** option in the **Main Window's Preferences menu**, to enable additional operators or disable operators that you rarely use.

Box operator

The Box operator, which is mostly intended for use with 3D datasets, removes areas of a plot that are either partially or completely outside of the volume defined by an axis-aligned box. The Box operator does not clip cells that straddle the box boundary, it just removes the cells from the visualization leaving jagged edges around the edges of the box where cells were removed.

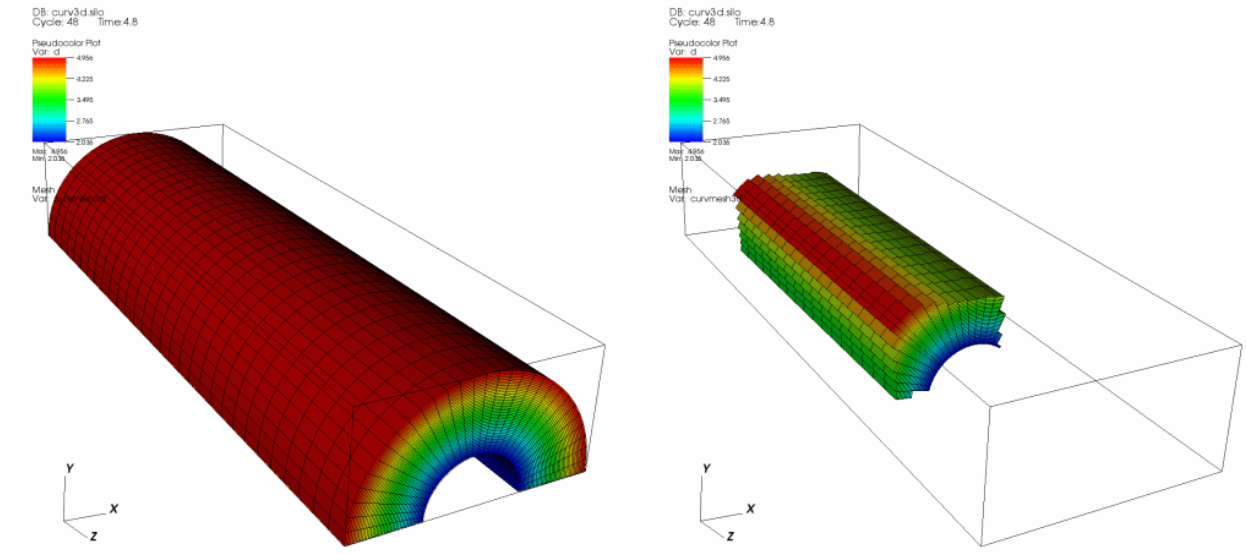


Fig. 4.90: Box operator example (original on left, with Box operator applied on right)

Setting how cells are removed

The Box operator can either remove cells that are totally outside of the box or it can remove those cells outside of the box and cells that are only partially outside of the box. By default, the Box operator only removes cells that are completely outside of the box. To make the Box operator also remove cells that are partially outside of the box, you click the **All** radio button in the **Box attributes window** (shown in Figure 4.90). Selecting the **Inverse** option will return everything in the mesh except those cells bounded by the selected box.

Resizing the box

The Box operator uses an axis aligned box to remove cells from the visualization so the box can be specified as a set of minimum and maximum values for X, Y, and Z. To set the size of the box using the **Box operator attributes**

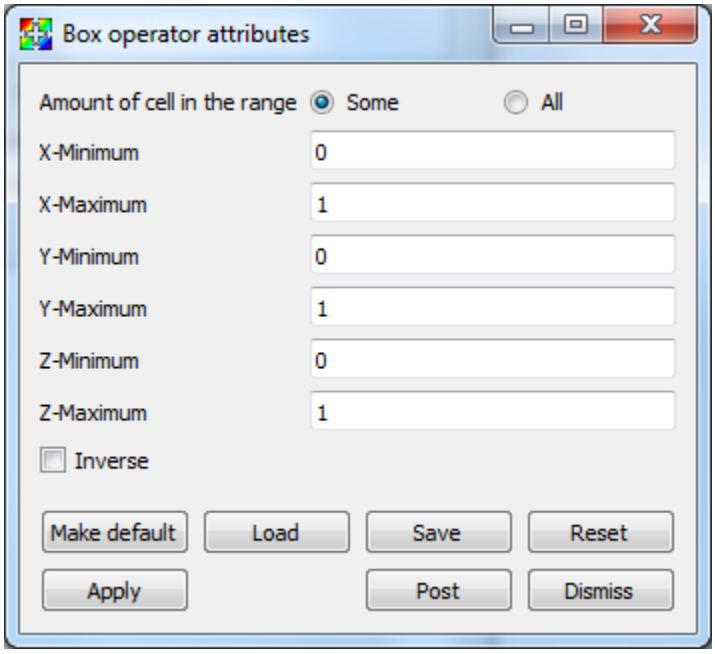


Fig. 4.91: Box attributes window

window, you type new coordinates into the **X Minimum, X Maximum, Y Minimum, Y Maximum, Z Minimum, or Z Maximum** text fields.

The Box operator can also be resized interactively with VisIt's Box tool (for more information, see the *Interactive Tools* chapter). If you want to use the Box tool to resize the Box operator's box, first make sure to select the plot that uses the Box operator in the Plot list and then enable the Box tool. When the Box tool appears, it uses the same box as the Box operator. Moving or resizing the Box tool causes the Box operator to also move or be resized and the plots in the visualization window get regenerated with the new box.

Clip operator

The Clip operator can remove certain shapes from a dataset before it is plotted. More specifically, the Clip operator can clip away box- or sphere-shaped regions from a database. The database remains in its original dimension after being clipped by the Clip operator and since the Clip operator manipulates the database before it is plotted, the surfaces bounding the removed regions are preserved in the final plot. While being geared primarily towards 3D databases, the Clip operator also clips 2D databases. When applied to 2D databases, the Clip operator can remove rectangular or circular regions from the database. Figure 4.92 shows a Pseudocolor and Mesh plots with a Clip operator.

Removing half of a plot

The Clip operator uses up to three planes to define the region that is clipped away. Each plane is specified in origin-normal form where the origin is a point in the plane and the normal is a vector that is perpendicular to the plane. When a plane intersects a plot, it serves as a clipping boundary for the plot. The plane's normal determines which side of the plane is clipped away. The region on the side of the plane pointed to by the normal is the region that the Clip operator clips away. If more than one plane is active, the region that is left as a result of the first clip operation is clipped by the next plane, and so on.

Only one plane needs to be used to remove half of a plot. Find the center of the database by inspecting the 3D axis annotations in the visualization window. Type the center as the new plane origin into the **Origin** text field for plane

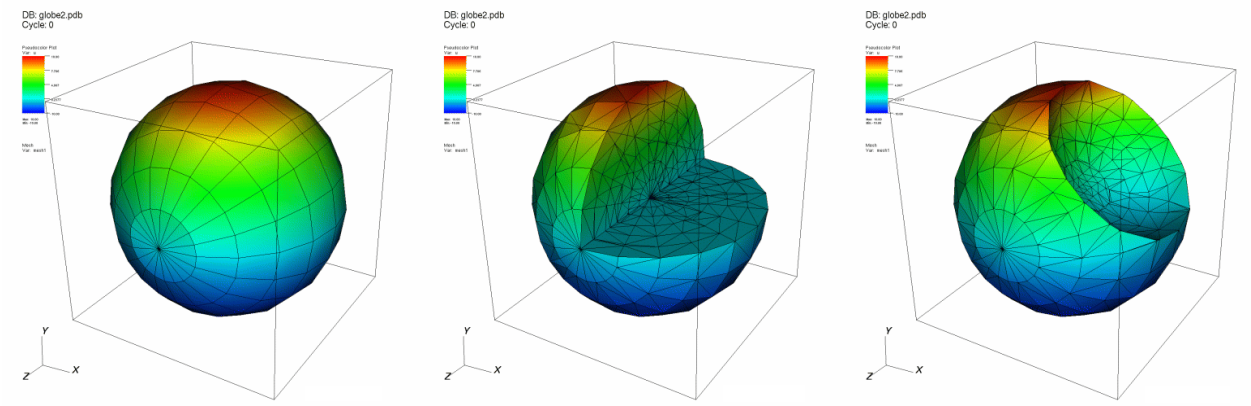


Fig. 4.92: Clip operator example: original plot; clipped with planes; clipped with sphere

1 then click the **Plane 1** check box for plane 1 (see Figure 4.93). When the **Apply** button is clicked, half of the plot should be removed. You can rotate the clipping plane by entering a new normal vector into the **Normal** text field. The normal is specified by three floating point values separated by spaces.

The **Accurate** option can be used when multiple planes are specified, to ensure accuracy when planes intersect a zone but do not clip the vertices. It can be up to 6x slower than the **Fast** option.

Removing one quarter of a plot

To remove a quarter of a plot, you need two clipping planes. To remove one of the plot, first remove one half of the plot. Now, enable the second clipping plane and make sure that it has the same origin as the first clipping plane but a different normal. To remove exactly one quarter of the plot, make sure that the normal is perpendicular to plane 1's normal. Also make sure that plane 2's new normal points into the region that was clipped away by plane 1. The two planes, when considered together, remove one quarter of the plot. For an illustration of this, see Figure 4.94. In general, the Clip operator removes regions defined by the intersection of the regions removed by each clipping plane. Follow the same procedure with the third clipping plane to remove only one eighth of the plot.

Spherical clipping

The Clip operator not only uses sets of planes to clip databases, it can also use a sphere. To make the Clip operator use a clipping sphere, click on the **Sphere** tab. To specify the location and size of the sphere, enter a new center location into the **Center** text field on the Sphere tab of the Clip attributes window and then enter a new sphere radius.

Inverting the clipped region

Once the Clip operator has been applied to plots and a region has been clipped away, clicking the **Inverse** check box brings back the clipped region and clips away the region that was previously unclipped. Using the **Inverse** check box is an easy way to get only the clipped region back so it can be used for other operations.

A common trick when creating animations is to have two identical plots with identical Clip operators applied and then switch one Clip operator to have an inverted clipping region. This will make the plot appear whole. The plot with the inverted clipping region can then be transformed independently of the first plot so it appears to slide out of the first plot. Then it is common to fade out the second plot and zoom in on the first plot's clipped region.

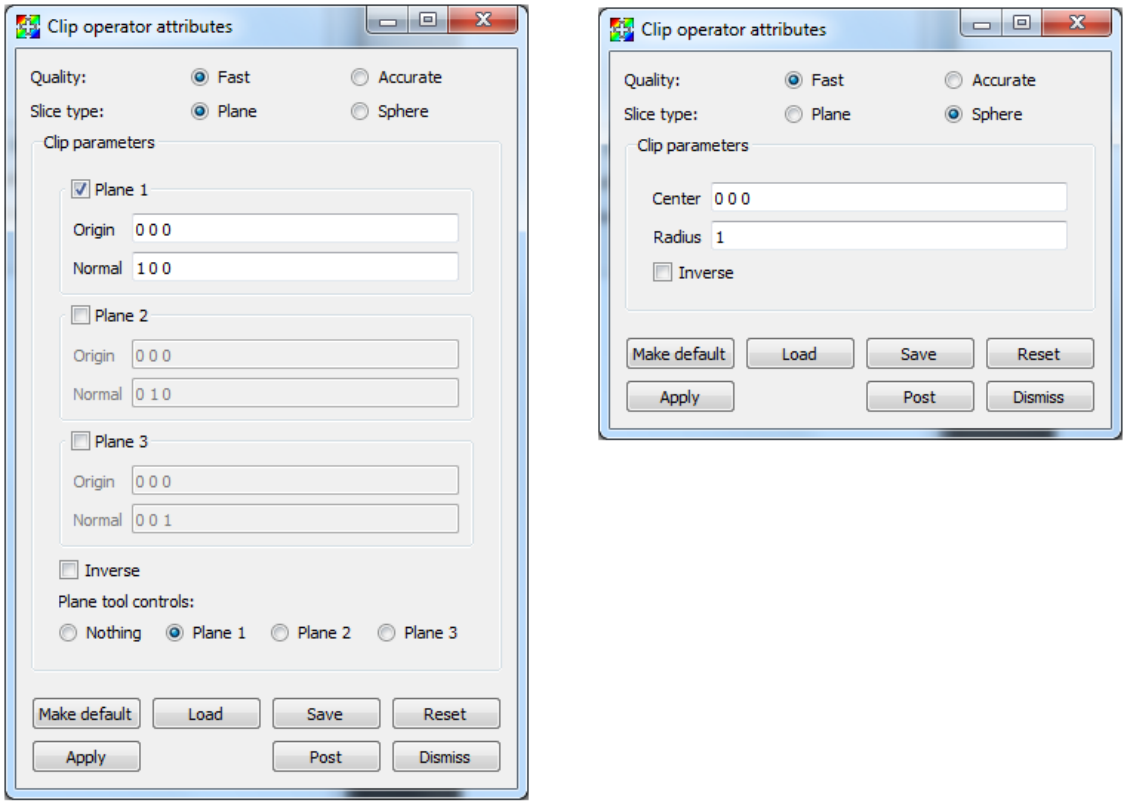


Fig. 4.93: Clip attributes window

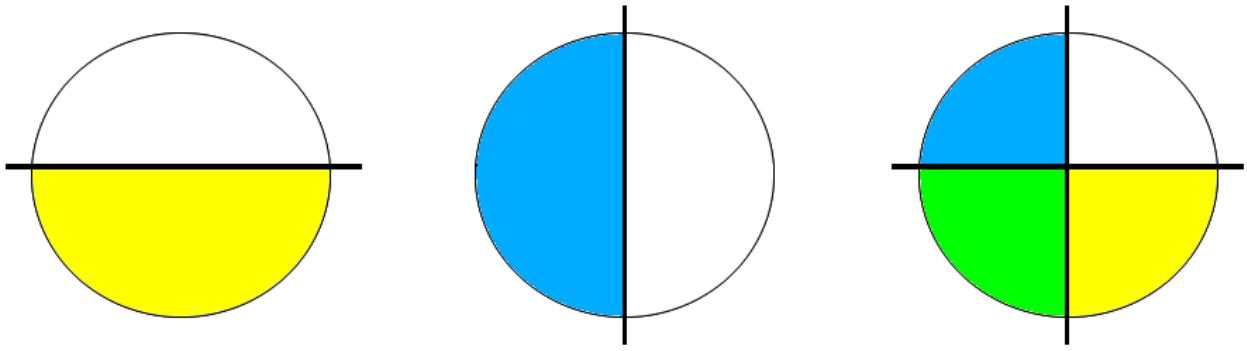


Fig. 4.94: Removing one quarter of a plot using two clip planes: Plane1 clipped region + Plane2 clipped region = One quarter removed

Using the crinkle clip

Generally, when using the Clip operator, the clipped surface will be smooth, but this often isn't representative of the natural surfaces of the cells along the clipped boundary. The often jagged edges of those cells are cut away and replaced with new faces to create this smooth result. There are times, though, when it may be desirable to retain the original cell faces that lay along this boundary. This can be accomplished by enabling the **Crinkle clip** option, shown in Figure 4.95.

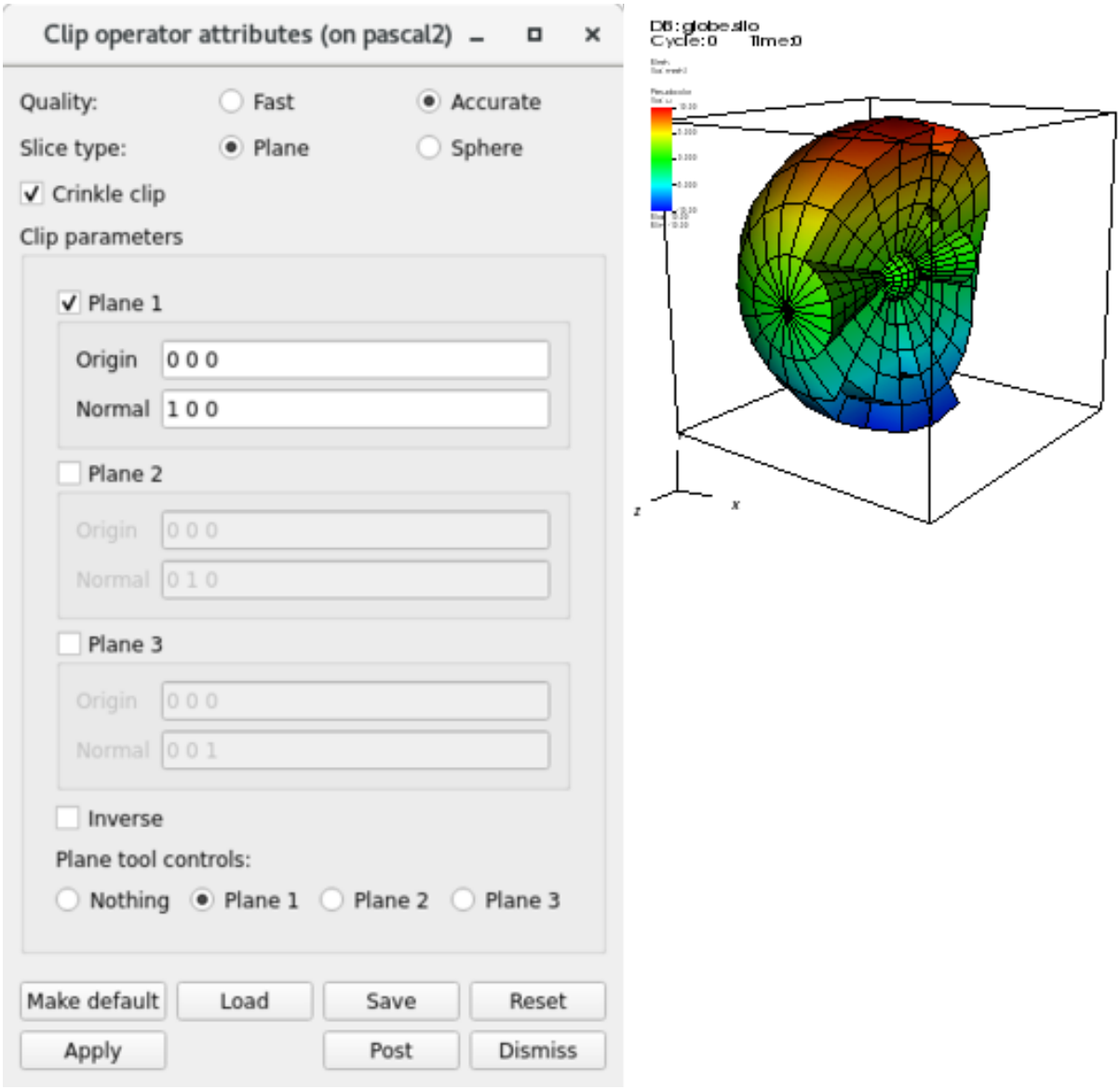


Fig. 4.95: Enabling the Crinkle clip option; Crinkle clip example

Cone operator

Like the Slice operator, the Cone operator is also a slice operator. The Cone operator slices a 3D database with a cone, creating a surface that can be left in 3D or be projected to 2D. Plots to which the Cone operator has been applied become surfaces that exist on the surface of the specified cone. The resulting plot can be left in 3D space or it can be

projected to 2D space where other operations can be done to it. A Pseudocolor plot to which a Cone operator has been applied is shown in Figure 4.96.

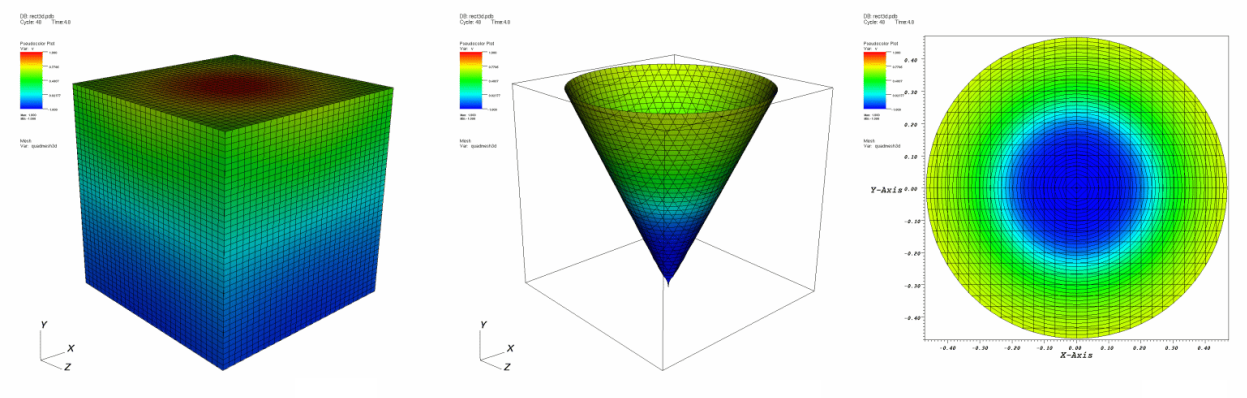


Fig. 4.96: Cone operator example: original plot; sliced with cone; sliced with cone and projected to 2D

Specifying the slice cone

You can specify the slice cone by setting various fields in the **Cone attributes window**, shown in Figure 4.97. To specify how pointy the cone should be, type a new angle (in degrees) into the **Angle** text field. The cone is defined relative to its origin, which is the point at the tip of the cone. To move the cone, type in a new origin vector into the **Origin** text field. The origin is represented by three floating point numbers separated by spaces. Once the cone is positioned, you can set its direction (where the cone points) by entering a new direction vector into the **Direction** text field.

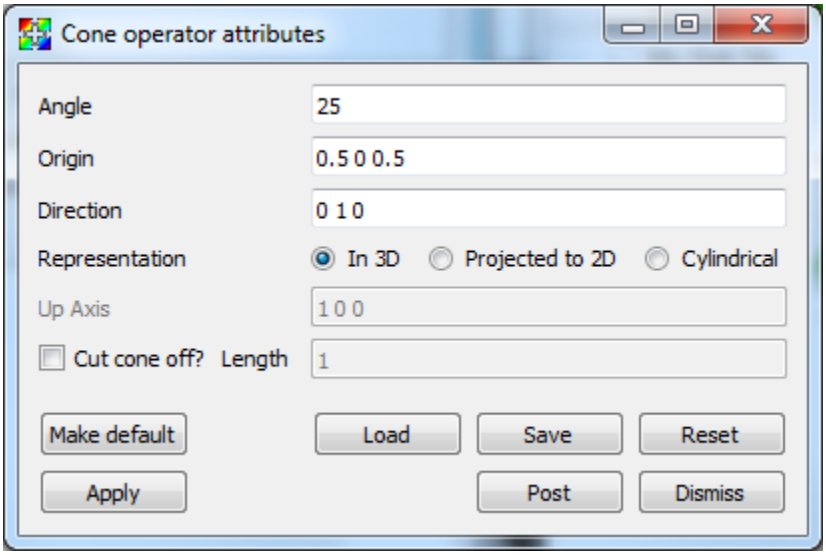


Fig. 4.97: Cone operator window.

The cone can extend forever or it can be clipped at some distance along its length. To clip the cone at a certain length, check the **Cut cone off** check box and enter a new length value into the **Length** text field.

Projecting the slice to 2D

The Cone operator usually flattens sliced plots to 2D along the cone's direction vector. This results in circular 2D plots in the visualization window. The Cone operator can also unfold sliced plots into a cylinder and then into rectangular 2D plots. Alternatively, the Cone operator can leave the sliced plots in 3D space where their cone shape is obvious. To set the cone projection mode, click on one of the following radio buttons: **In 3D**, **Project to 2D**, or **Cylindrical**.

Connected Components operator

The Connected Components operator is in a special class of operators, one that creates a new variable. In this case, the operator accepts as an input variable the name of a mesh, and constructs a scalar variable as output.

The operator creates unique labels for each connected mesh sub-component and tags each zone in the mesh with the label of the connected component it belongs to. Figure 4.98,

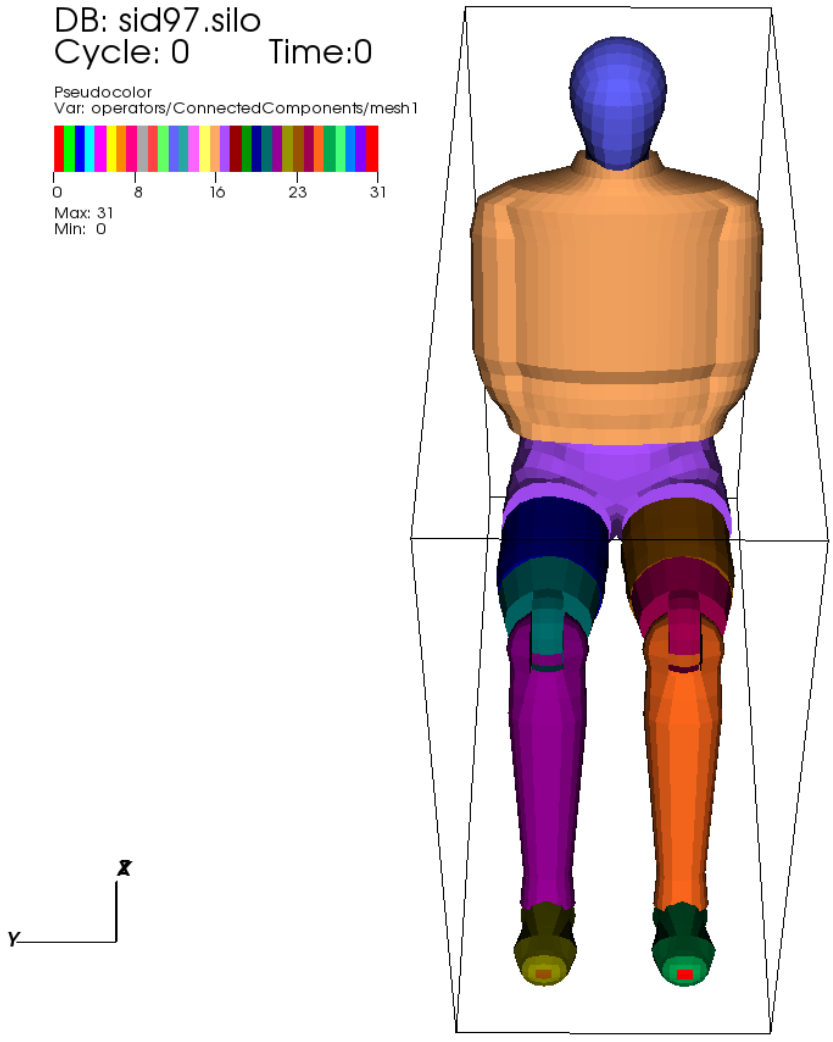


Fig. 4.98: Connected Components operator shown with Pseudocolor Plot.

The operator has one option which controls the use of Ghost Zone Neighbors for connectivity between domains. This option is turned on (set to true) by default. Figure 4.99

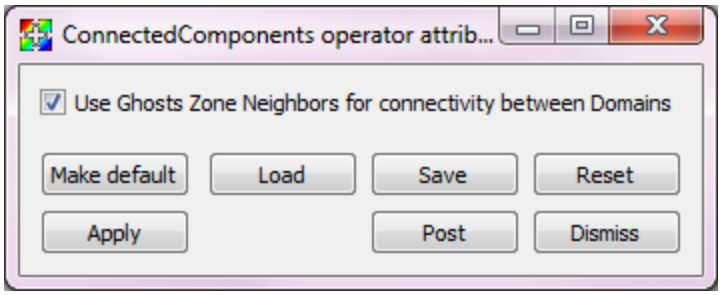


Fig. 4.99: Connected Components operator window.

Create Bonds operator

The CreateBonds operator is used to specify ranges of distances for various types of atoms and use those ranges to create bonds. The default behavior of this operator is to create a bonds between a Hydrogen and any other species if the atoms are separated by a distance between 0.4 and 1.2 units (e.g. Angstroms) and between a pair of atoms (not including H) if they are between 0.4 and 1.9 units apart. This works well for organic molecules. However, in Figure 4.100, the default distances were not useful. In this case, the values were changed to create a bond including H for distances between 0.4 and 1.5 units and for other species between 0.4 and 2.5 units. Figure 4.101.

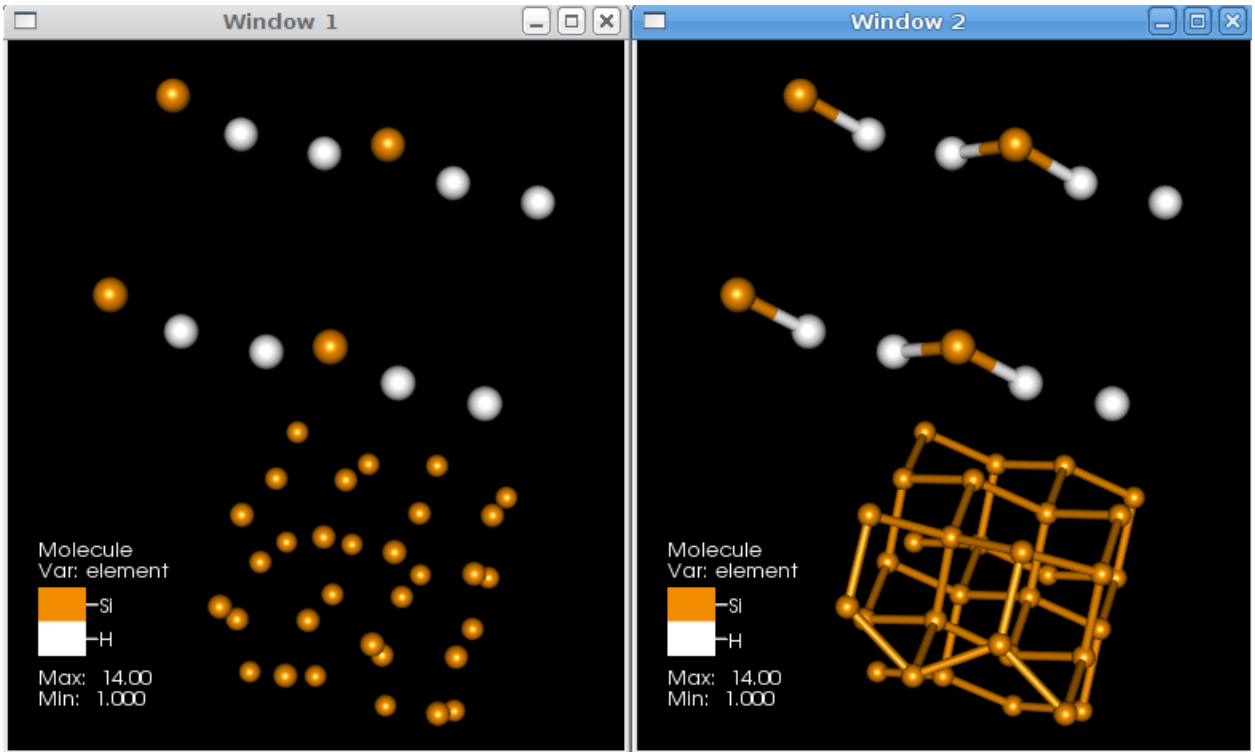


Fig. 4.100: Bonds created with different bonding distances

Setting the Bonding parameters

The Bonds list contains the bonding pair specifications to algorithm. Each row contains the species **1st** and **2nd**, and the **Min** and **Max** distance which could be considered a bond.

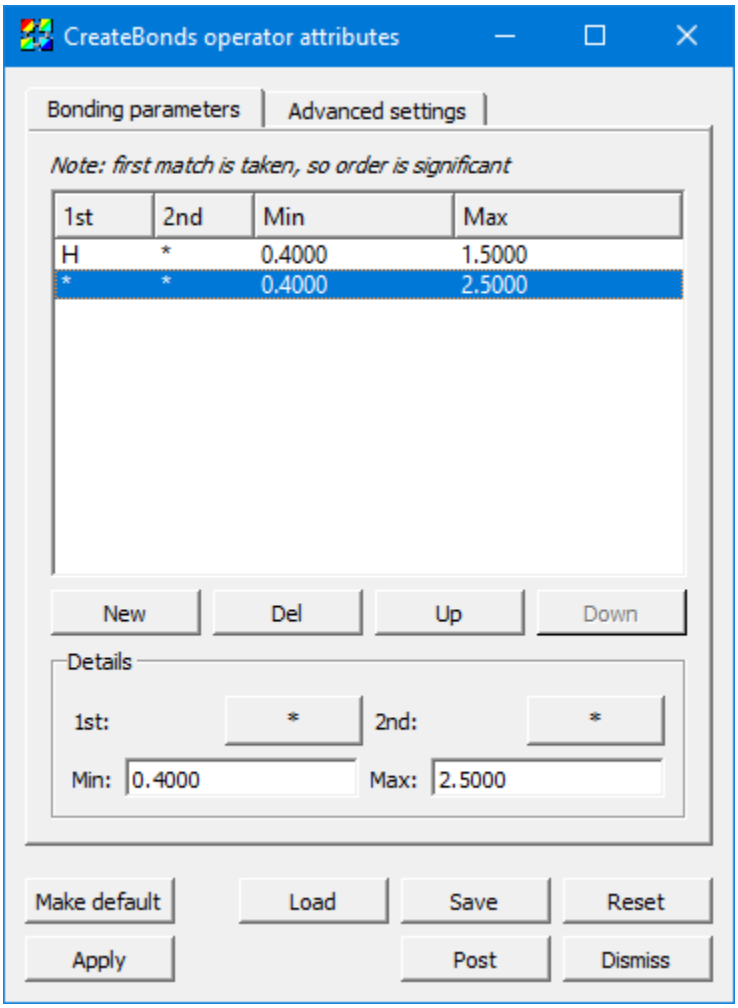


Fig. 4.101: CreateBonds bonding parameters

Note:

1. A "*" matches any species.
2. It does not matter which species is *1st* and which is *2nd*. The bonds are not unidirectional.
3. The first match in the list is taken, even if later lines also match, which allows you to specify more specific rules above less-specific rules.

For example, if the first line is "H", "*", "0.4", "1.2", this specifies that the algorithm should create a bond between two atoms if either one is Hydrogen and the distance between them is between 0.4 and 1.2 spatial units.

As a follow-on to this example, suppose the second line is "*", "*", "0.4", and "1.9". Now suppose two atoms exist, H and O, and they are separated by a distance of 1.5 units. Because one is H, it will match the first line, determine the distance is too great (since it's greater than 1.2), and so it will not create a bond between this atom pair. Since this atom pair matched the first line, the second line is not considered, even though the atom pair matches its criteria.

Just below the actual bonding rules list are several buttons: The **New** button creates a new rule, and **Del** deletes it. **Up** moves the currently selected rule up in the list, and **Down** moves the currently selected rule later in the list. Recall that the order of rules matters because only the first match is considered.

The **Details** section contains controls to set the values for a rule. The **1st** and **2nd** controls pop up the species selection widget shown in Figure 4.102.

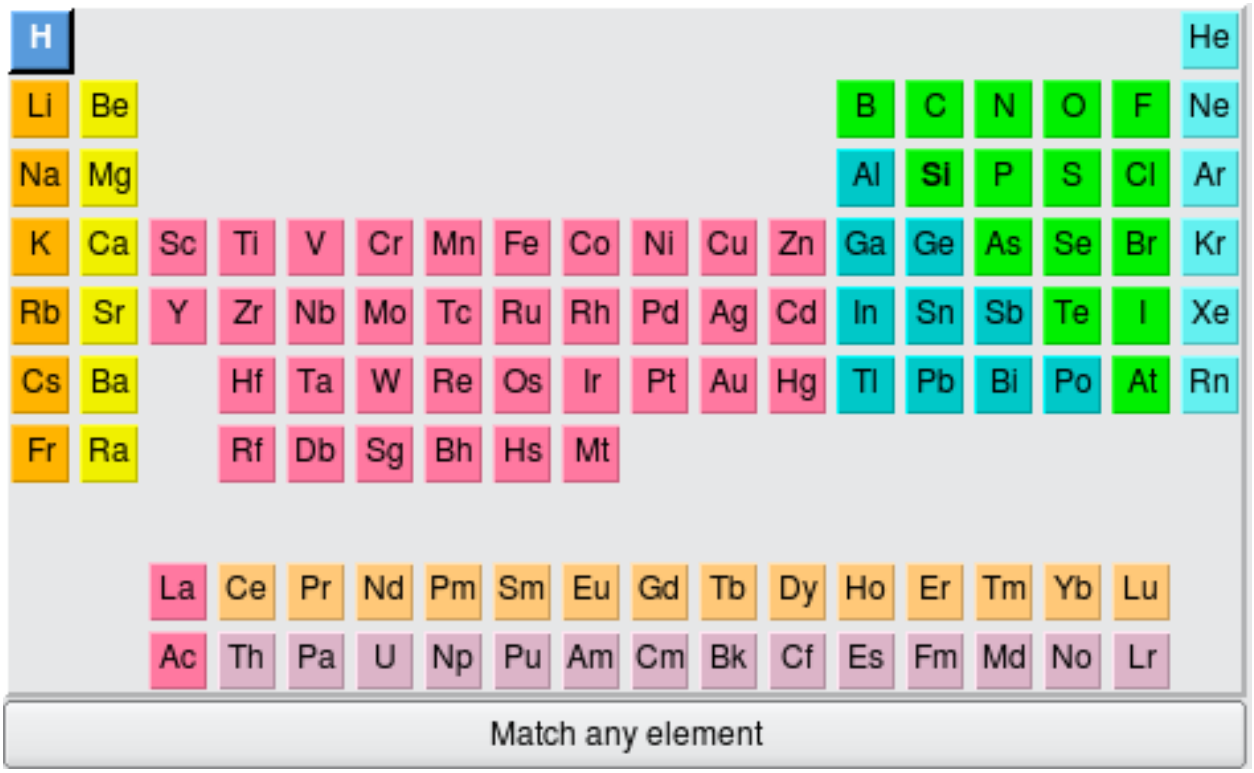


Fig. 4.102: The element selector

To get a wildcard which will match any type of atom, choose **Match any element** at the bottom; it is selectable just as any individual species in the periodic table.

Also, note that there is the possibility for some *hinting* to help guide your selection to the viable types of atoms. (This depends on conditions like file format support.) For example, in this screenshot, the **H** and **Si** elements are in boldface, since the file contains only those types of atoms. The **Min** and **Max** fields are standard text widgets.

Advanced settings

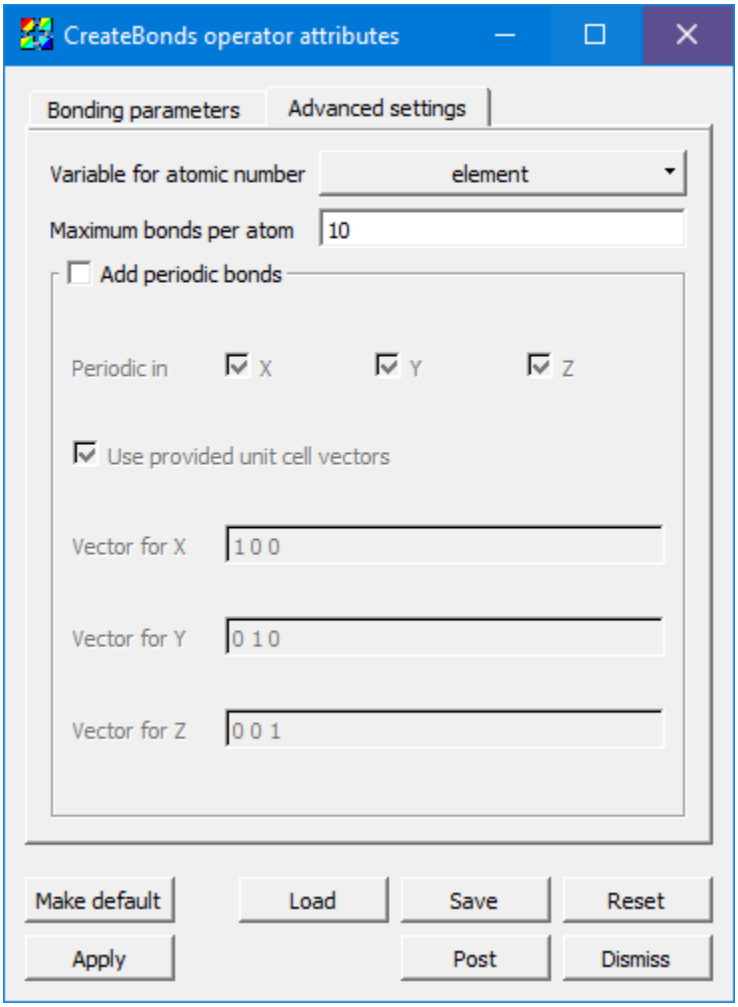


Fig. 4.103: The advanced settings tab

Variable for atomic number defaults to *element* as per the convention, but it can be set to any integral variable corresponding to the atomic number of each atom.

If you specify the wrong distance, each atom might try to bond to many other atoms. To keep an error like this from causing a severe hit to memory and performance, **Maximum bonds per atom** will stop the process before it gets out of hand. The default value is *10*, and it could safely be set lower in many cases, but it is user-settable for unusual cases where >10 bonds are needed on some atoms.

When **Add periodic bonds** is checked, this will make the algorithm see if an atom would bond with an atom past a periodic boundary edge, and add a dangling bond in that case. Checking this setting will enable the **Periodic in X,Y,Z** controls as well as the controls for **Unit cell vectors**.

Periodicity in X, Y, Z can be selected independently (or none).

Some file formats specify the vectors for the unit cell (sometimes called "direct lattice" vectors) containing the molecular data in the file. If they are present and **Use provided unit cell vectors** is checked, then it will use those values instead of the ones specified in this window.

Vector for X, Y, and Z controls the actual vectors describing the amount to displace in each of the three axes.

Examples in use

See *Molecular data features* for examples of the CreateBonds operator in use with the *Molecule Plot*.

Cylinder operator

The Cylinder operator, shown in Figure 4.104, slices a dataset with a cylinder whose size and orientation are specified by the user. The result is a cylindrical surface.

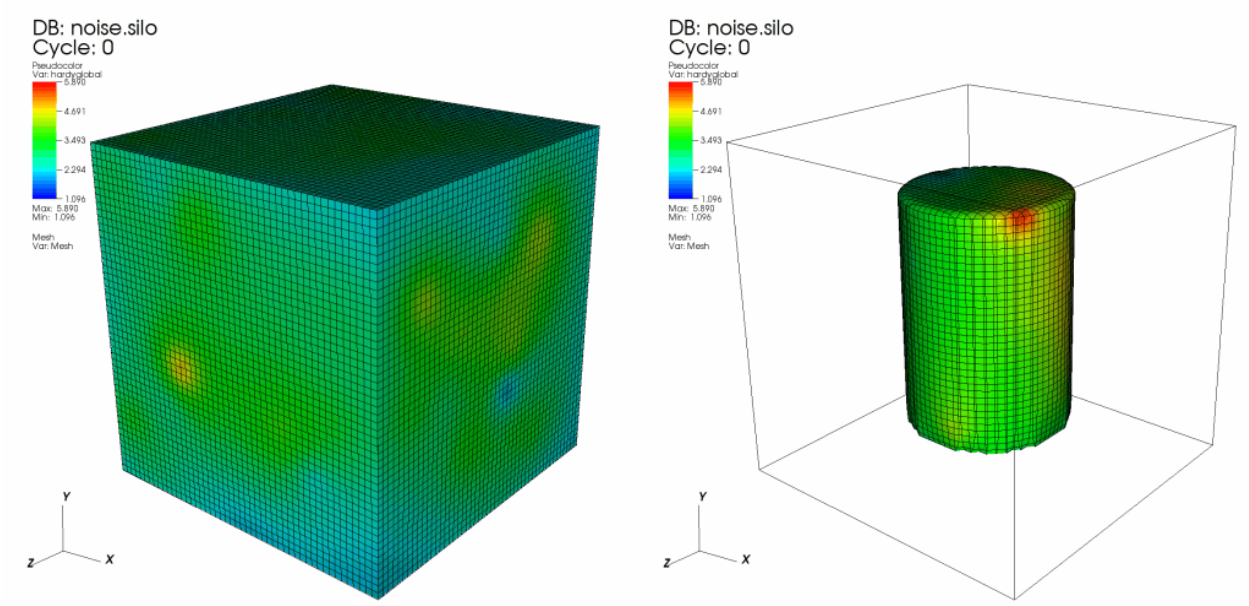


Fig. 4.104: Cylinder operator example: original plot; plot clipped by cylinder

Setting the cylinder's endpoints

There are two ways to set the endpoints for the Cylinder operator. First of all, you can open the **Cylinder operator window** (see Figure 4.105) and type new 3D points into the **Endpoint 1** and **Endpoint 2** text fields. The second, and more interactive way to set the endpoints for the Cylinder operator is to use VisIt's interactive Line tool, which is discussed in the *Interactive Tools* chapter. The Line tool lets you interactively place the Cylinder operator's endpoints anywhere in the visualization. The Line tool's endpoints correspond to the centers of the cylinder's top and bottom circular faces.

Setting the radius

To set the radius used for the Cylinder operator's clipping cylinder, type a new radius into the **Radius** text field in the **Cylinder attributes window** .

Inverting the cylinder region

Once the Cylinder operator has been applied to plots and a cylindrical region has been clipped away, clicking the **Inverse** check box brings back the cylindrical region and removes the region that was previously shown.

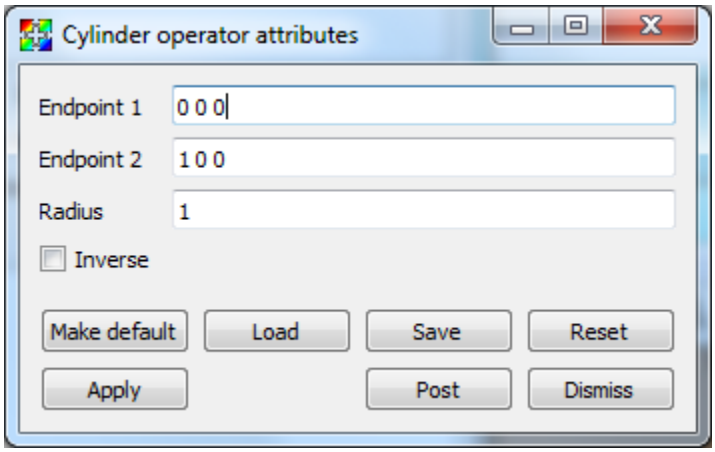


Fig. 4.105: Cylinder operator window.

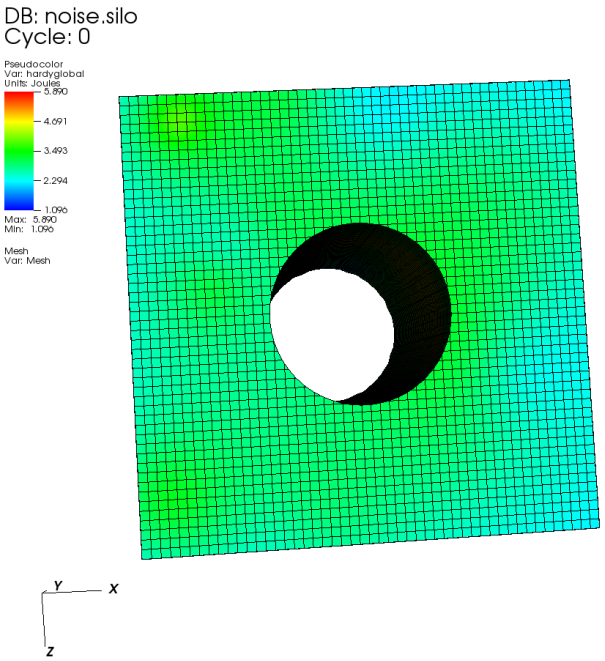


Fig. 4.106: Cylinder with inverse applied

Decimate operator

The Decimate operator, shown in Figure 4.107, removes nodes and cells from an input mesh, reducing the cell count while trying to maintain the overall shape of the original mesh. The Decimate operator can currently operate only on the external surfaces of the input geometry. This means that in order to apply the Decimate operator, you must first apply the *ExternalSurface operator*, which will be covered later in this chapter. The Decimate operator is not enabled by default but it can be turned on in the **Plugin Manager Window**.

Using the Decimate operator

The Decimate operator simplifies mesh geometry. This can be useful for producing models that have lower polygon counts than the model before the Decimate operator was applied. Models with lower polygon count can be useful for speeding up operations such as rendering. The Decimate operator has a single knob that influences how many cells are removed from the input mesh. The **Target Reduction** value is a floating point number in the range (0,1) and it can be set in the **Decimate attributes window** (see Figure 4.108). The number specified is the proportion of number of polygonal cells in the output dataset "over" the number of polygonal cells in the original dataset. As shown in Figure 4.107, higher values for **Target Reduction** value cause VisIt to simplify the mesh even more.

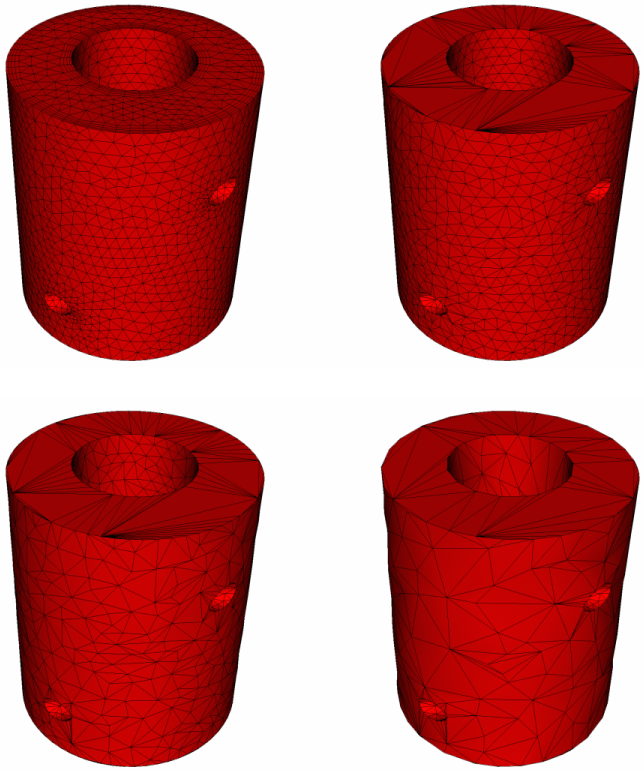


Fig. 4.107: Decimate operator applied to reduce the number of cells in the mesh. (Left-to-right, top-to-bottom): Original Mesh, Reduction = 0.1, Reduction = 0.5, Reduction = 0.75

DeferExpression operator

The DeferExpression operator is a special-purpose operator that defers expression execution until later in VisIt's pipeline execution cycle. This means that instead of expression evaluation taking place before any operators are applied, expression evaluation can instead take place after operators have been applied, at whatever point in the pipeline

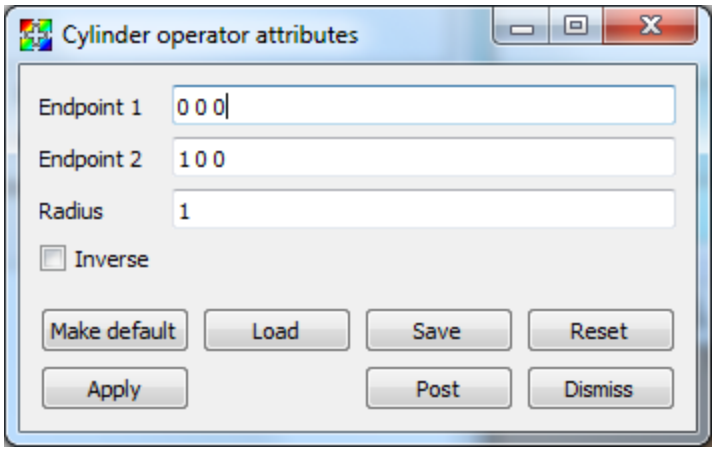


Fig. 4.108: Decimate attributes window

the DeferOperator exists. This may be necessary in cases where an expression involves the *output* of an operator, or the operator behaves in such a way as to change the outcome of an expression.

Plotting surface normals

VisIt can use the DeferExpression operator in conjunction with the *ExternalSurface operator* and the surface_normal expression to plot surface normals for your plot geometry. To plot surface normals, first create a vector expression using the surface_normal expression , which takes the name of your plot's mesh as an input argument. Once you have done that, you can create a Vector plot of the new expression. Be sure to apply the *ExternalSurface operator* first to convert the plot's 2D cells or 3D cells into polygonal geometry that can be used in the surface_normal expression. Finally, apply the DeferExpression operator and set its variable to your new vector expression. This will ensure that the surface_normal expression is not evaluated until after the ExternalSurface operator has been applied.

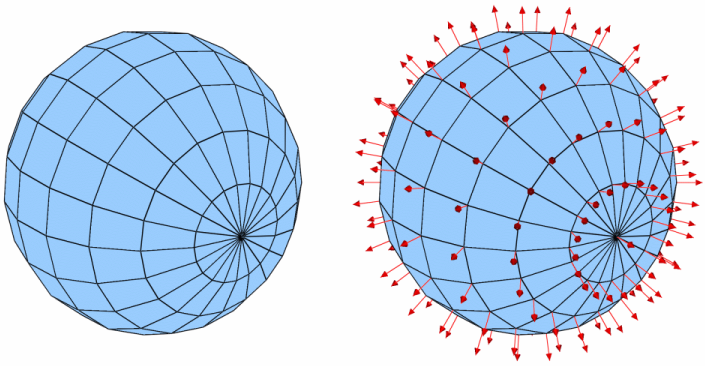


Fig. 4.109: DeferExpression operator example

Deferring multiple expressions

What if you want to color a surface by a new variable equal to `(1.0 - X)^2` where `X` is the x-component of the surface's normal? Starting with the previous example, and supposing the surface normal expression was defined as `surfn=point_surface_normal("mesh")`. You would create a new expression to grab the x-component of the normal: `X=(1.0-surfn[0])^2`. Add a *Pseudocolor plot* of `X`. Apply the *ExternalSurface operator*. Apply the DeferExpression operator and add both `surfn` and `X` to the list of variables being deferred.

Displace operator

The Displace operator deforms a mesh variable using a vector field that is defined on the nodes of that mesh. Many engineering simulation codes write a mesh for the first time state of the simulation and then write vector displacements for the mesh for subsequent time states. The Displace operator makes it possible to use the mesh and the time-varying vector field to observe the behavior of the mesh over time. The Displace operator provides a multiplier that can amplify the effects of the vector field on the mesh so slight changes in the vector field can be exaggerated. An example showing a mesh and a vector field, along with the results of the mesh displaced by the vector field is shown in Figure 4.110.

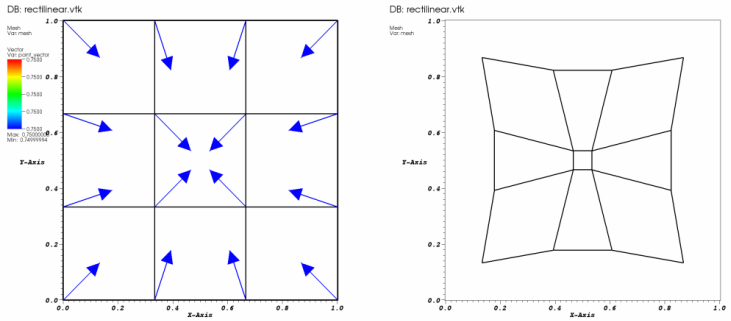


Fig. 4.110: Mesh and Vector plots and a Mesh plot that uses the Displace operator to deform the mesh using a vector field.

Using the Displace operator

The Displace operator takes as inputs a mesh variable and a vector variable and a displacement multiplier value. For each node in the mesh, the Displace operator adds the vector field defined at that node to the node's coordinates. Before adding the vector to the mesh, VisIt multiplies the vector by the displacement multiplier so the effects of the vector field can be exaggerated. To set a new value for the displacement multiplier, type a new value into the **Displacement multiplier** text field in the **Displace attributes window** (see Figure 4.111). To set the name of the vector variable that VisIt uses to displace the mesh, select a new vector variable from the **Displacement variable** variable button.

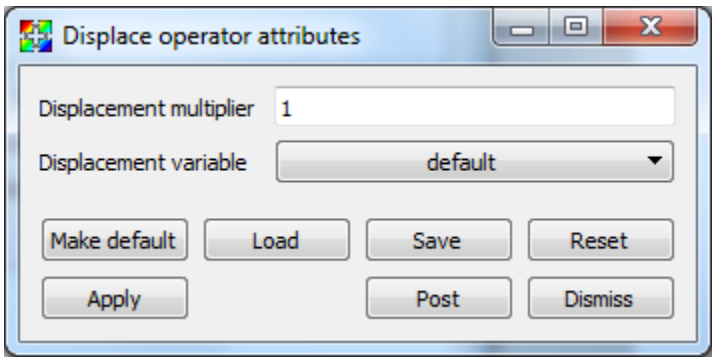


Fig. 4.111: Displace attributes window

Elevate operator

The Elevate operator uses a scalar field on a 2D mesh to elevate each node in the input mesh, resulting in a topologically 2D surface in 3D. The Elevate operator allows you to perform much of the same functionality as a Surface plot and

it allows you to do additional things like elevate plots that do not accept scalar variables. The Elevate operator can also elevate plots whose input data was produced from higher dimensional data that has been sliced. Furthermore, the Elevate operator allows you to display multiple scalar fields in a single plot such as when a Pseudocolor plot of scalar variable A is elevated by scalar variable B (see: Figure 4.112).

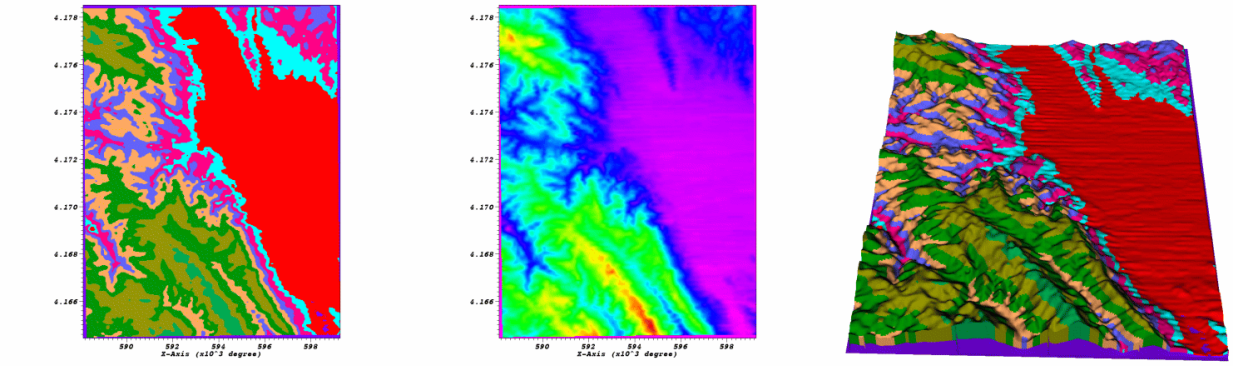


Fig. 4.112: Elevate operator example: 2D plot of rainfall; 2D plot of elevation; Plot of rainfall elevated by elevation

Using the Elevate operator

The Elevate operator can be used to create plots that look much like a Surface plot if you simply apply the Elevate operator to a plot that accepts scalar values. The Elevate operator is more flexible than a Surface plot because whereas the Surface plot limits you to elevating by one variable and coloring by the same variable, the Elevate operator can be used with any plot and still achieve the Surface plot's elevated effect. You could use the Elevate operator to elevate a Pseudocolor plot of rainfall by elevation. You could also take Vector or FilledBoundary plots (among others) and elevate them by a scalar variable.

Since the Elevate operator uses a scalar variable to elevate all of the points in the mesh, the Elevate operator has a number of controls related to scaling scalar data. For example, the Elevate operator allows you to artificially set minimum and maximum values for the scalar variable so you can eliminate data that might otherwise cause your elevated plot to be stretched undesirably in the Z direction. To set minimum and maximum values for the Elevate operator, click on the **Min** or **Max** check boxes in the **Elevate attributes window** (see Figure 4.113) and type new values into the adjacent text fields. The options for scaling the plots created using the Elevate operator are the same as those for scaling Surface plots. For more information on scaling, see the Surface plot documentation.

The most useful feature of the Elevate operator is its ability to elevate plots using an arbitrary scalar variable. By default, the Elevate operator uses the plotted variable in order to elevate the plot's mesh. This only works when the plotted variable is a scalar variable. When you apply the Elevate operator to plots that do not accept scalar variables, the Elevate operator will fail unless you choose a specific scalar variable using the **Elevate by Variable** variable menu in the **Elevate attributes window**.

Changing elevation height

The Elevate operator uses a scalar variable's data values as the Z component when converting a mesh's 2D coordinates into 3D coordinates. When the scalar variable's data extents are small relative to the mesh's X and Y extents then you often get what appears to be a flat 2D version of the data floating in 3D space. It is sometimes necessary to scale the scalar variable's data extents relative to the spatial extents in order to produce a visualization where the Z value differs noticeably. If you want to exaggerate the Z values that the scalar variable contributes to make differences more obvious, you can click on the **Elevation height relative to XY limits** check box in the **Elevate attributes window**.

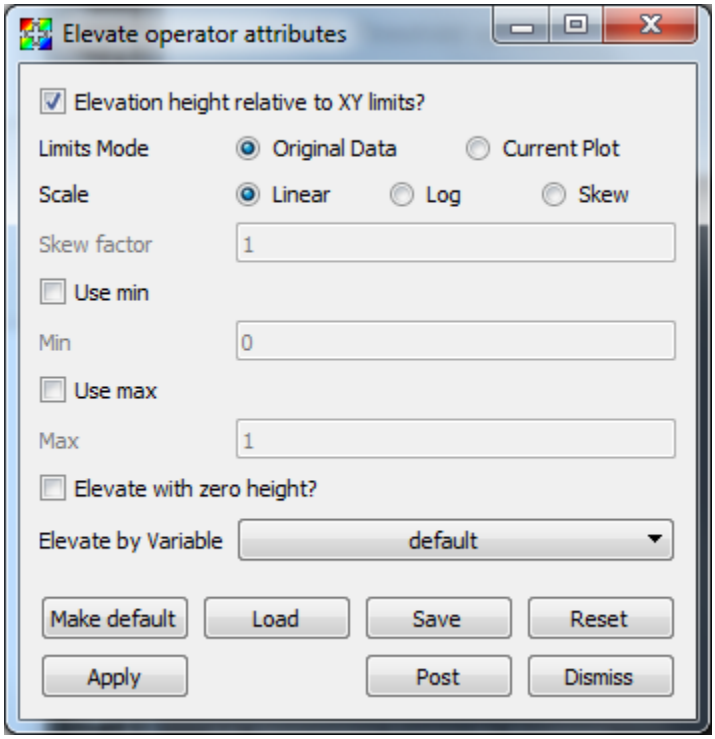


Fig. 4.113: Elevate window

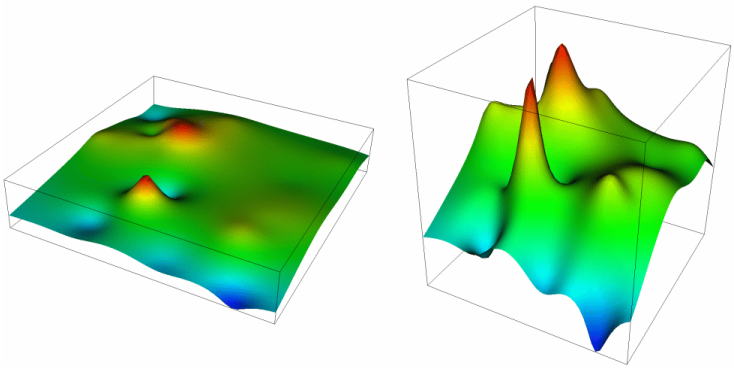


Fig. 4.114: Effect of scaling relative to XY limits

The Elevate operator can be used to simply place a 2D plot in 3D space by use of the **Elevate with zero height** option. This will assign a value of zero to all of the z coordinates when converting into 3D.

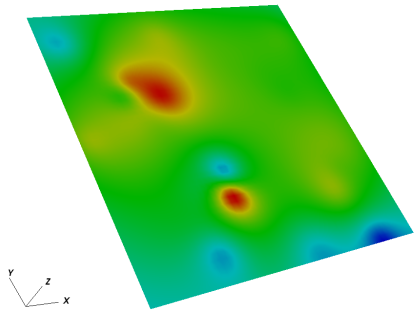


Fig. 4.115: Effect of elevating with zero height

Explode operator

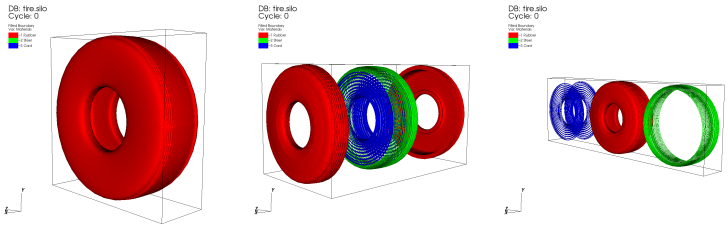


Fig. 4.116: Explode operator example: original plot; exploding cells of a material; exploding materials.

The Explode operator has three primary targets, which are **materials, domains, and cells**. There are three different origins of explosion—**point, plane, and cylinder**—all of which have unique results and can be applied to any of the above mentioned targets. While this operator is primarily meant to be used on datasets containing materials or domains, the capability of exploding all cells remains available for datasets that lack either.

Using the Explode operator

The Explode operator has three areas for user definition. These are the **Origin** of explosion, **Material Explosion** settings, and **Cell Explosion** settings. You can add as many explosions as you'd like to a single instance of the operator, and you have the ability to **Add**, **Remove**, or **Update** explosions through the **Explode attributes window** shown below.

Explode origin

As mentioned earlier, there are three different choices for an explode **Origin**. To explode from a **Point**, click the tab labeled Point in the **Origin** section of the **Explode attributes window**. You will then have the opportunity to enter a 3D coordinate defining your point. Similarly, to explode from a **Plane**, you must click on the Plane tab. You will then have the option to define a plane by a point located on that plane and the plane's normal. Lastly, to explode from

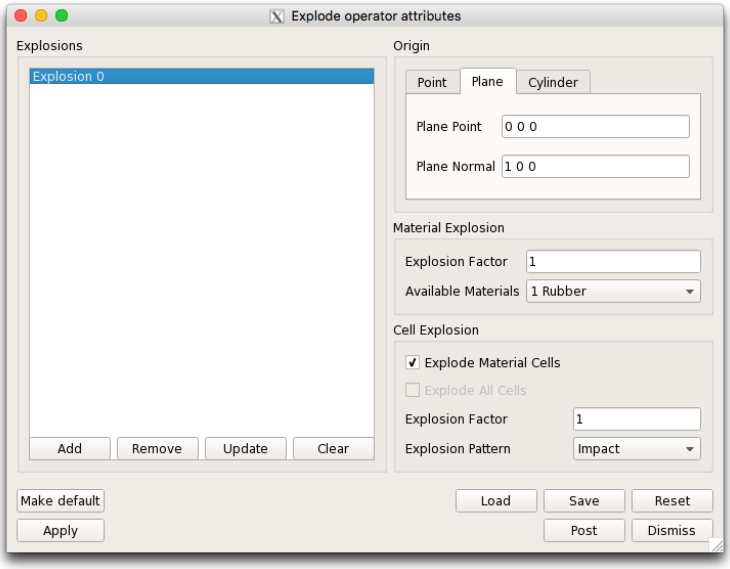


Fig. 4.117: Explode attributes window

a **Cylinder**, first click on the Cylinder tab, and then enter two points that lie on a line traveling through the center (lengthwise) of your cylinder. By default, the cylinder has a radius of zero and is treated as a *line* to explode from. If you do define a positive radius, any data that is located within that radius will *not* be exploded when executing this explosion.

Exploding materials

Exploding a material results in an individual material within a dataset being displaced by a specified **Factor** from a specified origin. Both the factor with which the material is displaced and the actual material to be acted upon are set within the **Material Explosion** section of the attributes window. If you refer to the far right image in Figure 4.116, you will find an example of two material explosions. In this example, we see the materials Cord and Steel, shown in blue and green, being exploded from the Tire dataset.

Exploding domains

To explode the domains of a dataset, you must first make sure that your dataset has domains that can be plotted using the Subset plot. If this condition is met, all you need to do is apply the Explode operator to a Subset plot of your domains. The domains will then be substituted in for materials and treated as such. You can then refer to the section on exploding materials for usage tips.

Exploding cells

Exploding cells results in the separation and displacement of the cells within your dataset. This can either be applied to an individual material or the entire dataset. If you refer to the middle image in Figure 4.116, you will see the cells of the material Rubber, shown in red, being exploded by a plane. As a result, the material is split open and separated to allow us to see the inner contents. As before, you also have control over the explosion **Factor** that is applied to the cells. Additionally, you have two options for the **Explosion Pattern**. The first option is to explode through **Impact**, which results in cells that are *closest* to the origin being displaced furthest from the origin. The second option is to explode through **Scatter**, which results in cells *furthest* from the origin being displaced furthest from the origin.

ExternalSurface operator

The ExternalSurface operator takes the input mesh and calculates its external faces and outputs polygonal data. The ExternalSurface operator is not enabled by default but it can be turned on in the **Plugin Manager Window**. The ExternalSurface operator can be useful when creating plots that only involve the external geometry of a plot - such as when you create a Vector plot of surface normals.

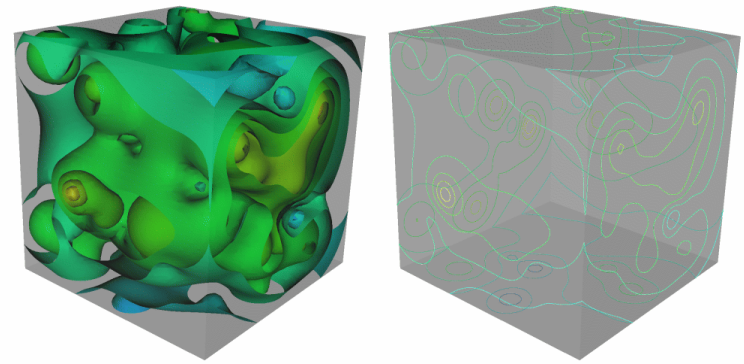


Fig. 4.118: ExternalSurface operator example

ExtrudeStacked operator

The ExtrudeStacked operator uses one or more scalar fields on a 2D mesh to extrude each node in the input mesh, resulting in a 3D height field. The ExtrudeStacked operator can also extrude higher dimensional meshes that have been sliced producing a 2D mesh. The resulting extruded mesh can be displayed by the extrusion height or an index (see: Figure 4.119).

Using the ExtrudeStacked operator

The ExtrudeStacked operator can be used to create 3D height field plots by applying the operator to a plot that accepts scalar values. The ExtrudeStacked operator is quite flexible as it can extrude by more than one variable and be displayed different ways.

The most useful feature of the ExtrudeStacked operator is its ability to extrude a mesh using multiple arbitrary scalar variables along an arbitrary **Extrusion axis**. By **default**, the ExtrudeStacked operator uses the plot's current variable to extrude the mesh. This only works when the plotted variable is a scalar variable. When one applies the ExtrudeStacked operator to plots that do not accept scalar variables, the ExtrudeStacked operator will fail unless a specific scalar variable is chosen in the **Add Variable** variable menu in the **ExtrudeStacked attributes window** (see Figure 4.120).

When a variable is added, it is added to the top of the stack and given the next largest index. That is the bottom of the stack is the variable with index 0. It is possible to reorder and delete variables via the **Move up** or **Move down** and **Delete** buttons in the **ExtrudeStacked attributes window** (see Figure 4.121).

As the ExtrudeStacked operator uses a scalar variable to extrude all of the points in the mesh, one can set the minimum and maximum values as well as scale for each scalar variable. This allows one to eliminate data that might otherwise cause the extruded mesh to be stretched undesirably. To set the minimum and maximum values and scale for the ExtrudeStacked operator, one first selects the variable, then sets the **Min** or **Max** or **Scale** in the **ExtrudeStacked attributes window** (see Figure 4.122). The values will automatically be updated in the variable table and then applied to the mesh (see Figure 4.123).

It is possible to **Display by** the resulting field via the **Index** of the variable in the stack or by the scalar values used to extrude the mesh. If cell based scalar values are used then the **Node Height** and the **Cell Height** will be same. If node

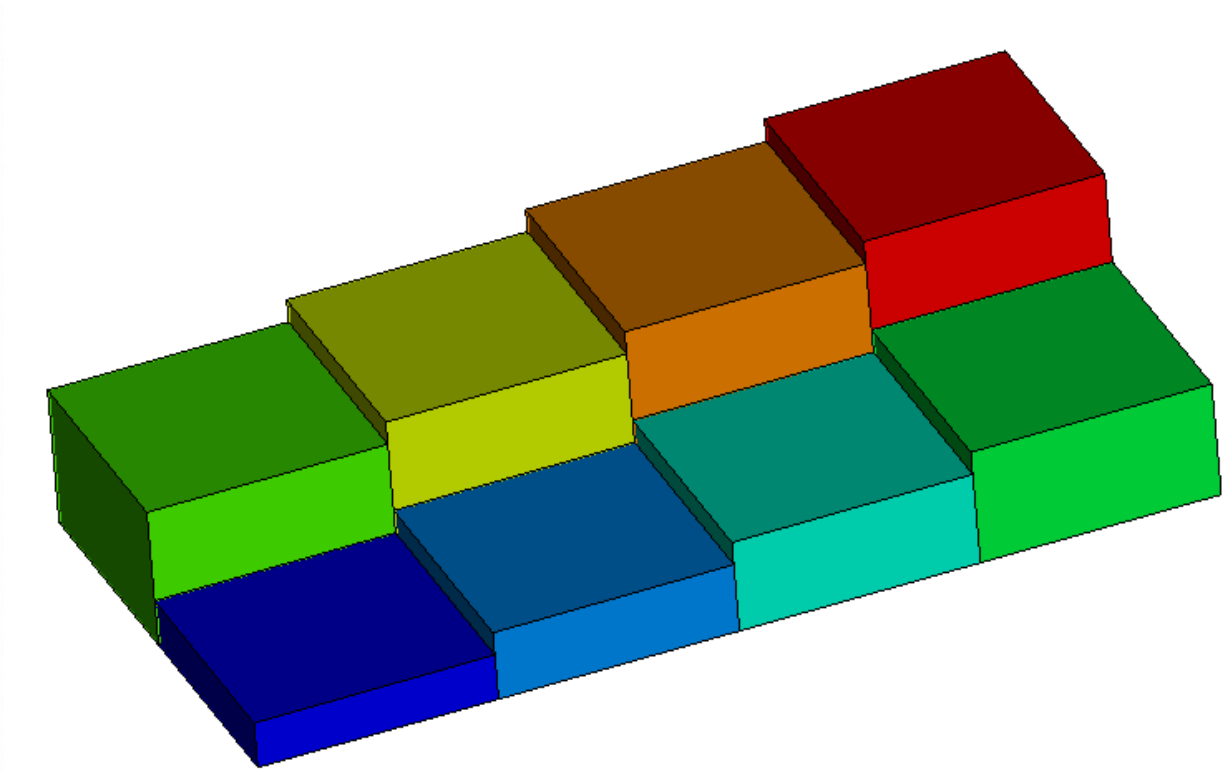


Fig. 4.119: ExtrudeStacked operator example: A simple structured grid extruded by a single scalar cell value.

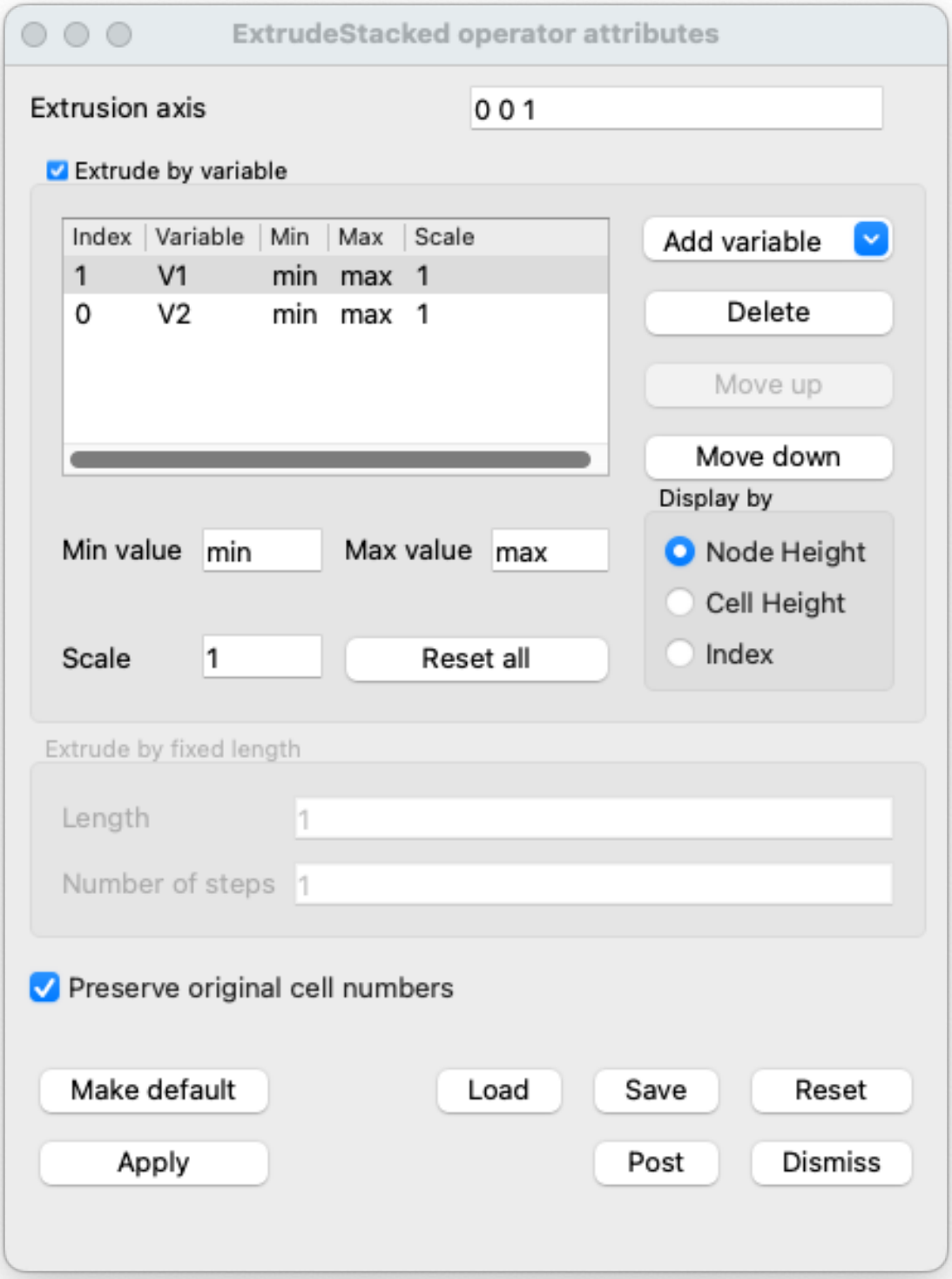


Fig. 4.120: ExtrudeStacked operator window.

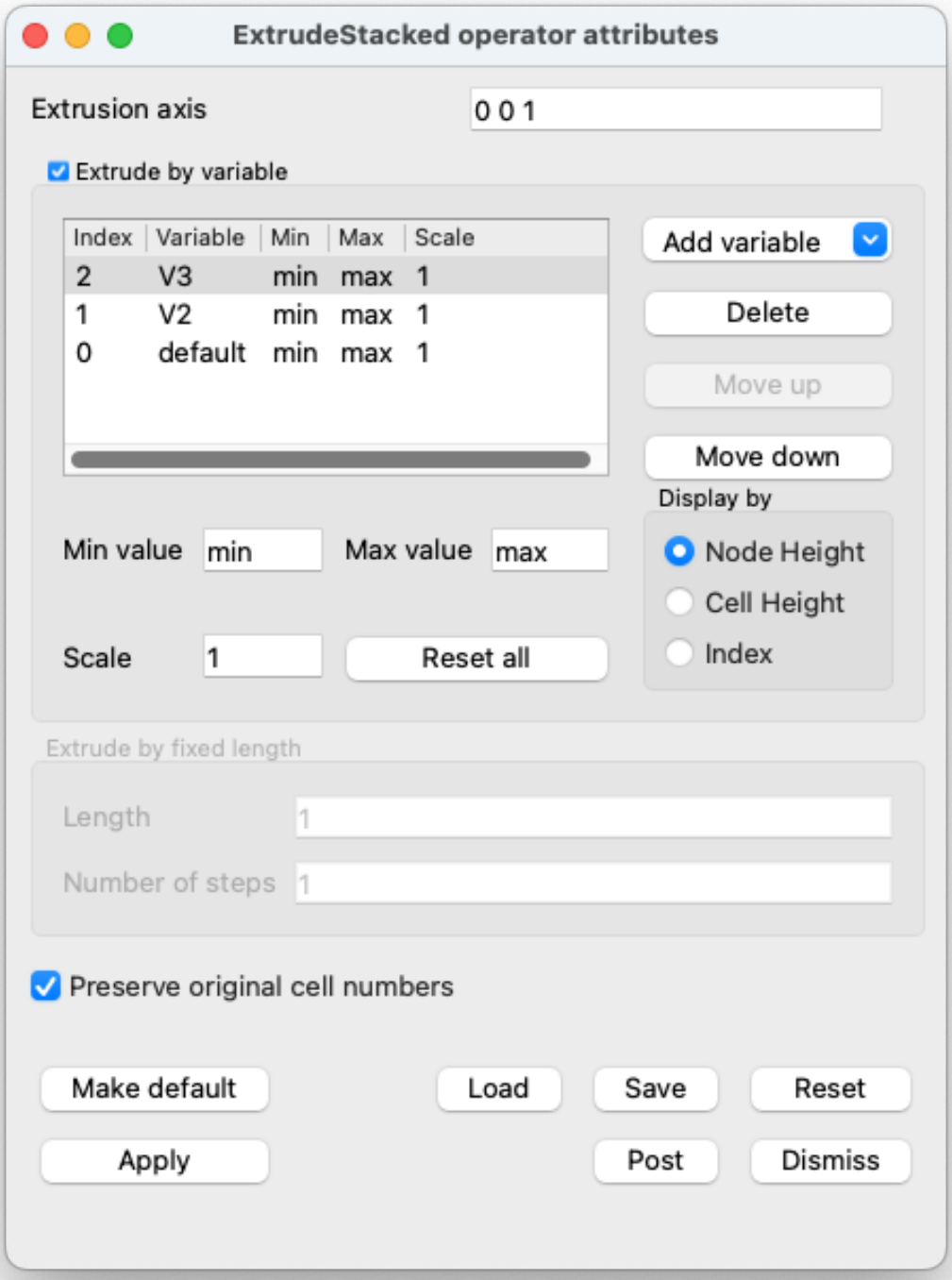


Fig. 4.121: ExtrudeStacked operator window multiple variables.

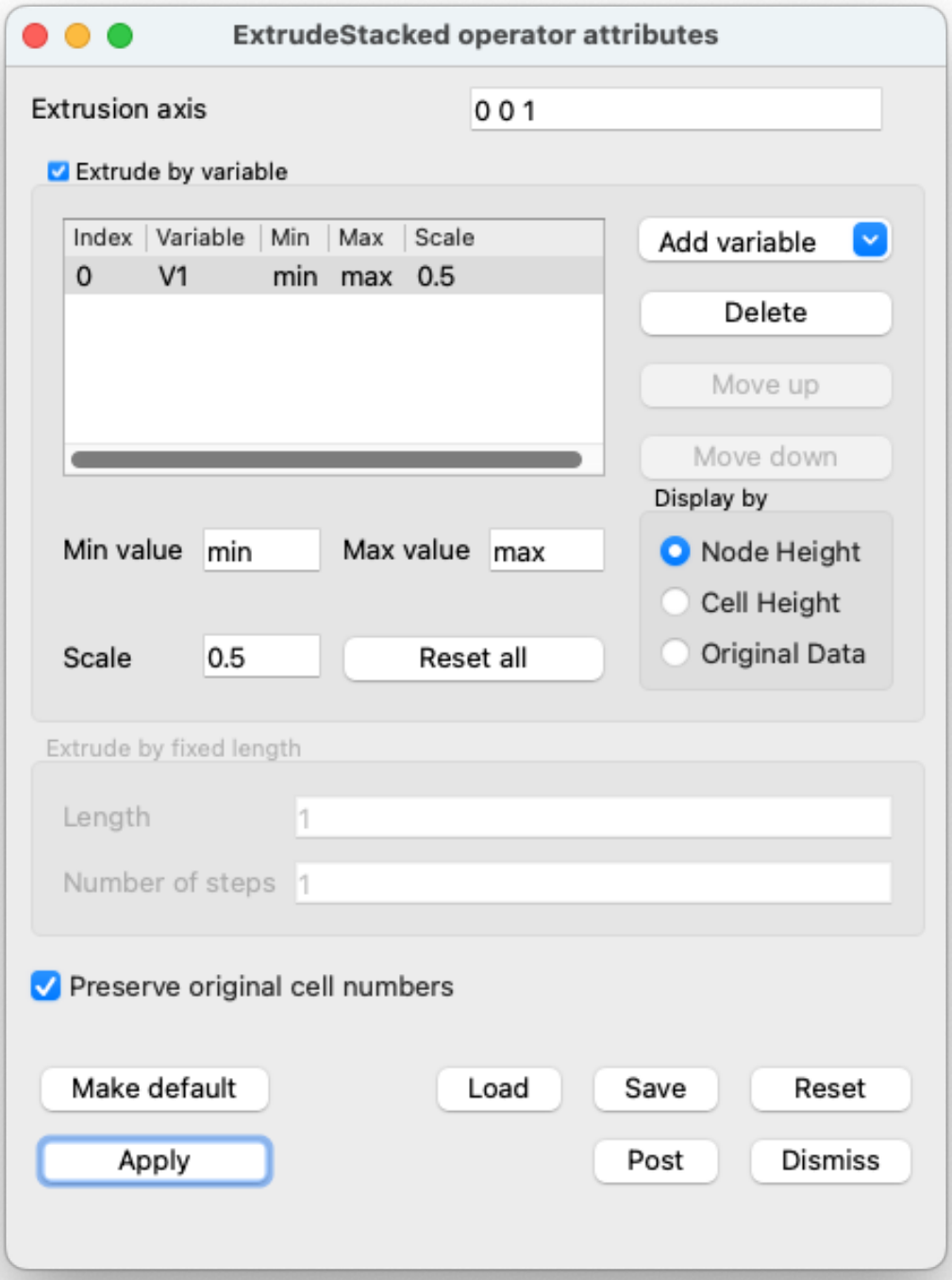


Fig. 4.122: ExtrudeStacked operator window with the values scaled.

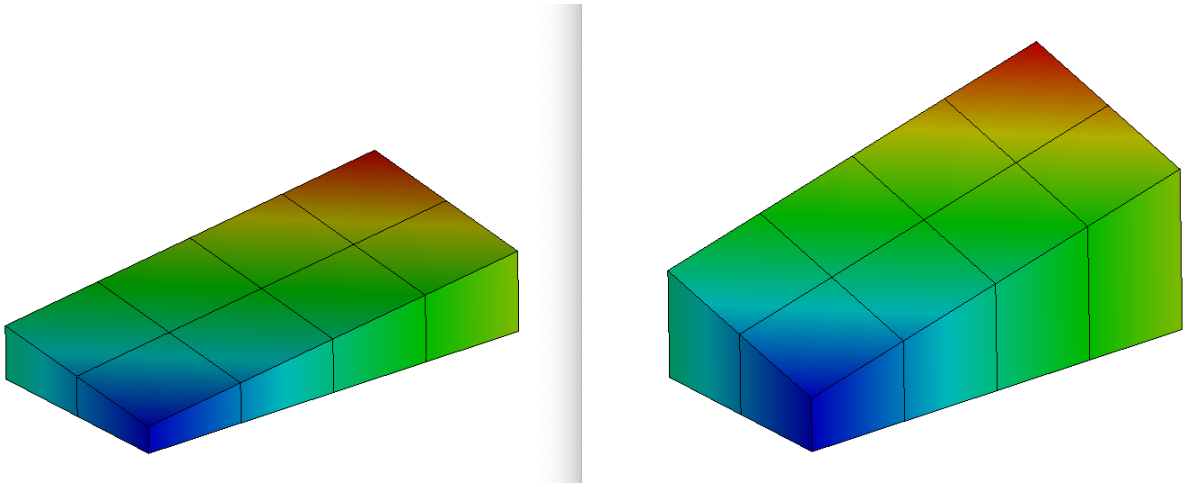


Fig. 4.123: ExtrudeStacked operator used to create a scaled plot using one variable. The left image used node based scalar values to extrude the mesh, the right image scaled the scalar values by two. Both are displayed by the node based scalar values (see the next section).

based scalar values are used then the **Node Height** will relfect the scalar value while the **Cell Height** will be average scalar value for that cell. (see: Figure 4.124). Note: when extruding by a single variable it is possible to **Display by** the resulting field via the **Original Data**. Thus making it possible to extrude by one variable and **Display by** a different variable

It is also possible to extrude by a fixed **Length** while dividing the resulting mesh into a **Number of Steps**. When extruding by a fixed **Length** the **Extrusion axis** will be used.

Integral Curve System

Within the VisIt infrastructure is the ability to generate integral curves. An integral curve is a curve that begins at a seed location and is tangent at every point in a vector field. It is computed by numerical integration of the seed location through the vector field. For example, the image below shows integral curves through the magnetic field of a core-collapse supernova simulation from the GenASiS code.

The generation of integral curves forms the basis of VisIt's Integral Curve System (ICS), made up of the *Integral Curve operator*, the *Lagrangian Coherent Structure (LCS) operator*, the *Limit Cycle operator*, and the *Poincaré operator*. Much of the underlying infrastructure and interface is the same for each operator: the user selects a series of seed locations where curves are generated, which are then visualized and analyzed.

The ICS allows for the computation of Lagrangian Coherent Structures (LCS) using a variety of techniques developed by George Haller and his group at ETH Zürich. For more information on LCS, see `K. Onu, F. Huhn, & G. Haller, LCS Tool: A Computational platform for Lagrangian coherent structures, J. of Computational Science, 7 (2015) 26-36.`

Many of the terms used in the ICS are familiar to experts in dynamical systems but may be new to many users. Users can refer to a `glossary` sepcific to dynamical systems and can reference VisIt's *Glossary* for some terms that are specific to VisIt's ICS. Any additional terms can be defined through a simple online search.

Integral Curve operator

The Integral Curve Operator allows the user to compute an integral curve from a seed point through a vector field without any analysis of its structure.

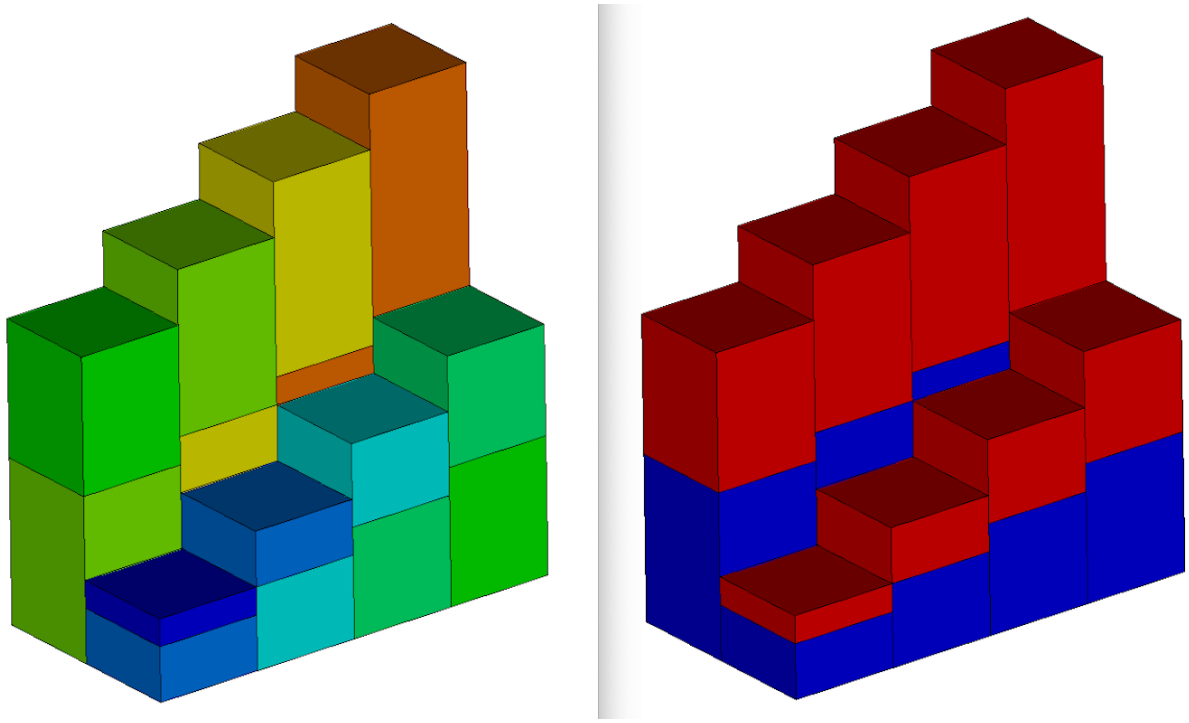


Fig. 4.124: ExtrudeStacked operator used to create a stacked plot using two variables. The left image is displayed using cell based scalar values used to extrude the mesh, the right image is displayed by using the variable index.

Source

The set of points that seed the integral curves. In addition to the *Source* attributes common to all ICS operators, the Integral Curve operator supports the following attributes:

Source type

The source type controls how the seeds for the curves are created. There are various options, the names of which are self-descriptive such as creating them along a *line* or around a *sphere*. Only those options that require further clarification are described further here.

Point List Seed from a list of points. In addition to *Add Point*, *Delete Point*, and *Delete All Points*, the user can *Read Text File* that is formatted with one point per each line either as "X Y Z" or "X, Y, Z".

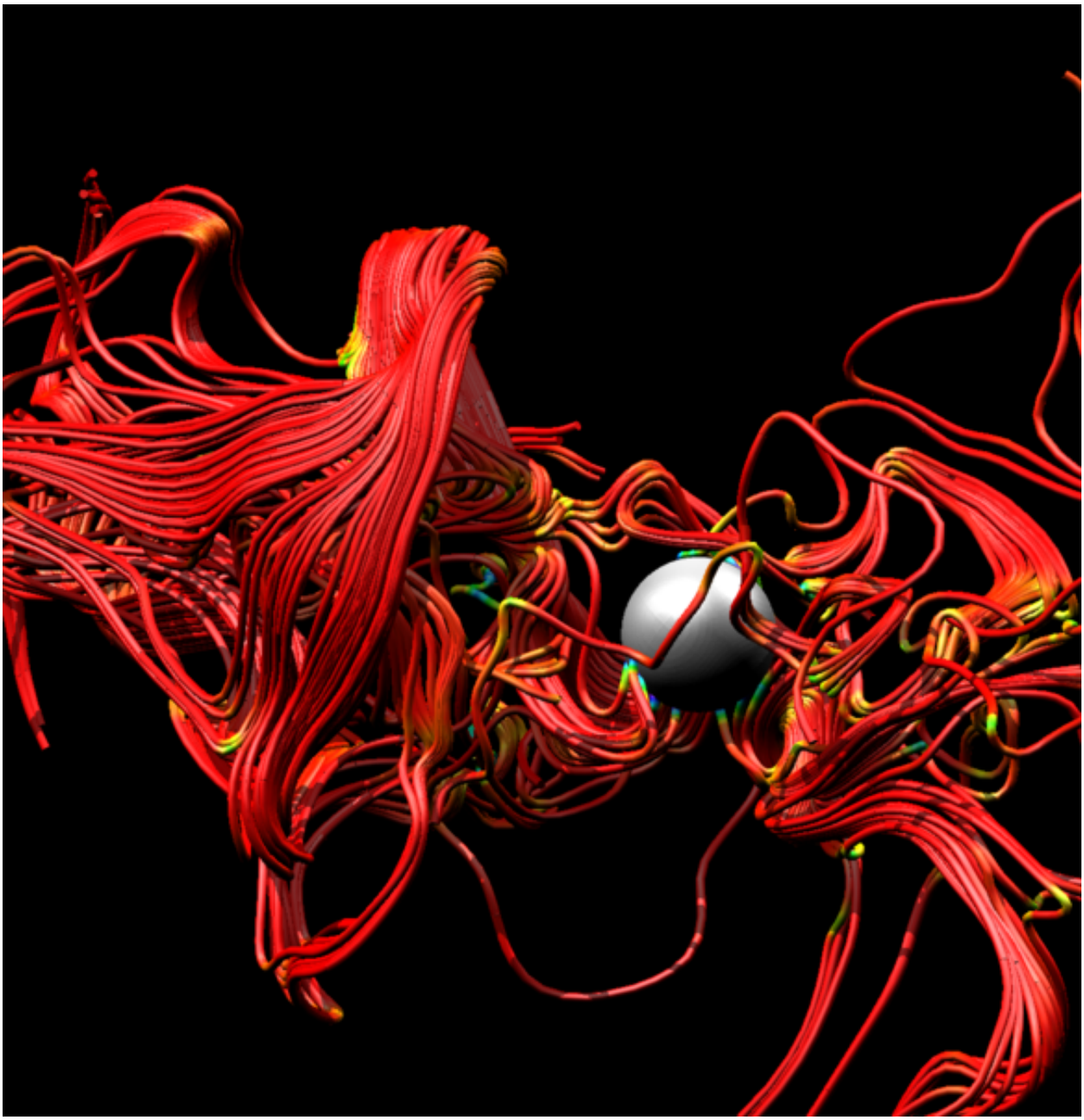
Selection Seed with a named selection.

Field Data The seed points are defined by another operator and passed to the Integral Curve operator. The name of the array containing the seed points must begin with the string "Seed Points".

Up Axis The "up axis" serves as the "Y" axis embedded in the plane or circle.

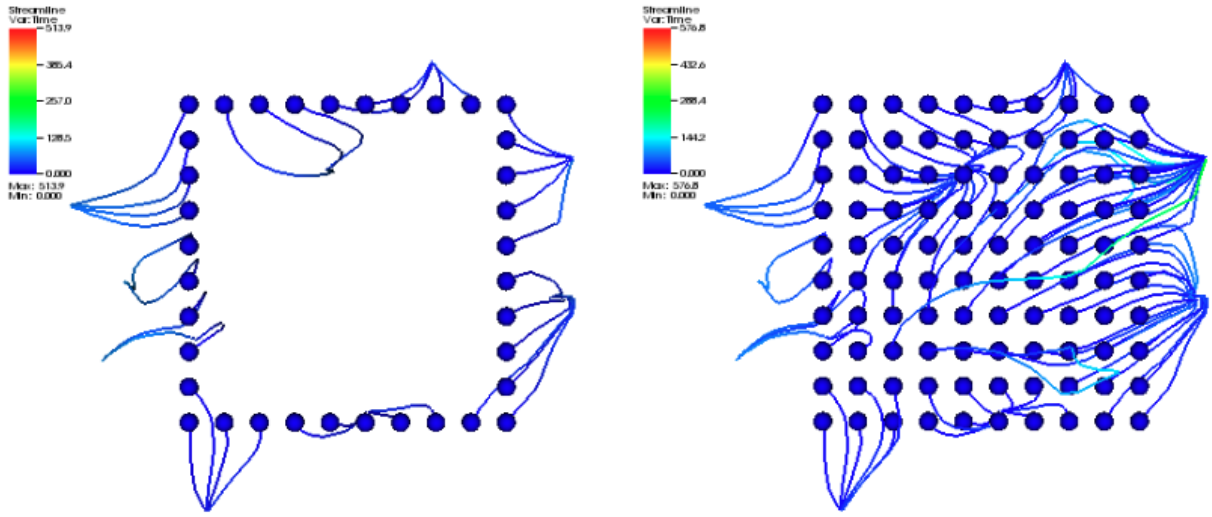
Sampling type

For samples taken from a geometric object, there is an option to generate uniform or random samples from the specified region. Random samples can be reproduced by supplying a random number seed.



Boundary vs Interior Samples

Samples from a geometric object can be taken either from the boundary or the interior. For example, when sampling a plane, the samples can either lie along the edges of the planar region or within the bounded rectangle, as shown below.



Integration

Specify settings for numerical integrators. In addition to the *Integration* attributes common to all ICS operators, the Integral Curve operator supports the following attributes.

Integration Direction

Sets the integration direction through time. The user can choose from a combination of forward, backward, and directionless. Eigen vectors are an example of a directionless vector field. In order to integrate using a directionless field, any orientation discontinuity must be corrected prior to linear interpolation. That is, all vectors must be rotated to match the orientation of the trajectory. The ICS code will do this processing for standard fields (e.g non-higher order elements).

Termination

Integral curve termination can be controlled in several different ways. The termination is based on the most conservative criteria, so only one criteria must be met for termination. The options are:

Maximum number of steps The maximum number of integration steps that will be allowed.

Appearance

The appearance tab specifies how the integral curve will be rendered. In addition to the *Appearance* attributes common to all ICS operators, the Integral Curve operator supports the following attributes:

Coloring

There are various coloring options, the names of which are self-descriptive such as coloring the curves with a *solid* color or according to a *seed*. Only those options that require further clarification are described further here.

Average Distance from seed Each curve is colored according to the average distance of all the points in the curve from the seed.

Variable Each curve's color varies by the value of a scalar variable.

Cleanup

Allows the user to remove points along the integral curve according to difference schemes. Options are self-descriptive, with additional information provided here as needed.

Delete points before Delete all points that come before a critical point defined by a velocity threshold. This cleaning will reveal when an integral curve may stop advecting because of some other reason than the critical point (i.e. the advection continues temporally but not spatially), so this cleaning will remove all duplicate points leaving the last temporal value. If the last point's temporal value is different than the value as dictated by the elapsed time or max steps, then the advection may have reached a critical point but terminated because of some other reason.

Delete points after Delete all points that come after a critical point defined by a velocity threshold. This cleaning will reveal when an integral curve reaches a critical point (i.e. the advection continues temporally but not spatially, so this cleaning will remove all duplicate points leaving the first temporal value).

Warning: Cleanup will always be called if the user displays integral curves using tubes or ribbon regardless of the settings here because they cannot contain duplicate points.

Crop the integral Curve (for animations)

Integral curves can be cropped so that they appear to grow over time. This option is useful for creating animations. Users can crop the curves based on several criteria and within a desired time range.

Advanced

In addition to the *Advanced* attributes common to all ICS operators, the Integral Curve operator supports the following attributes:

Warnings

Issue warning if the advection limit is not reached If the maximum time or distance is not reached, issue a warning.

Issue warning if the spatial boundary is reached If the integral curve reaches the spatial domain boundary, issue a warning.

Lagrangian Coherent Structure (LCS) operator

The LCS operator utilizes Lyapunov Exponents based on the Cauchy-Green Tensor to highlight Lagrangian Coherent Structures in vector fields. When performing a Finite Time Lyapunov Exponent (FTLE) calculation, the time can be specified as one would for a traditional FTLE, and the resulting value will be based on the maximal Eigen value.

However, when performing the calculation with Finite Space Lyapunov Exponents (FSLE), instead of assuming a uniform mesh discretization and specifying the dispersion distance, we specify a dispersion factor. In a traditional FTLE, this is the dispersion distance divided by the initial distance. In the equivalent definition, the dispersion distance is the maximal Eigen value. Thus when the maximal Eigen value is greater than the specified dispersion factor, then the exponent is calculated.

More details can be found `in this paper`.

Source

The set of points that seed the integral curves that reveal the Lagrangian Coherent Structures. In addition to the *Source* attributes common to all ICS operators, the LCS operator supports the following attributes:

Source types

The source type controls how the seeds for curves are created. The user can seed the integral curves using the native mesh or define a rectilinear grid. The nodes of the mesh are the seed points.

Auxilary Grid

When calculating the Jacobian for the Cauchy-Green tensor, one can use the neighboring points from the native mesh or one can specify an auxiliary grid that allows for the detection of finer features but at greater computational expense. Using an auxiliary grid is advantageous because it is independent of the native mesh, so it gives more accurate results for higher order elements. For simulation flows, using the auxiliary grid for eigenvalue calculations gives better results.

Integration

Specify settings for numerical integrators. In addition to the *Integration* attributes common to all ICS operators, the LCS operator supports the following attributes:

Integration Direction

Sets the integration direction through time: either forward or backward.

Appearance

The appearance tab specifies how the LCS's will be rendered. In addition to the *Appearance* attributes common to all ICS operators, the LCS operator supports the following attributes.

Seed Generation

Filter the number of seeds generated from the mesh (either native or rectilinear). There are various self-descriptive filtering options.

Advanced

In addition to the *Advanced* attributes common to all ICS operators, the LCS Operator supports the following attributes:

Warnings

Issue warning if the advection limit is not reached If the maximum time or distance is not reached, issue a warning.

Issue warning if the spatial boundary is reached If the integral curve reaches the spatial domain boundary, issue a warning.

Limit Cycle operator

The Limit Cycle Operator detects limit cycles within a vector field. Integral curves are seeded at a Poincaré section and integrated through the vector field. Curves that return to the Poincaré section indicate a limit cycle, and the integration of the curve will stop. Those integral curves that do not return to the Poincaré section are terminated according to separate termination criteria.

A signed return distance is calculated for the integral curves that return to the Poincaré section. Curves with a return distance below the cycle tolerance are considered to be limit cycles. If a curve does not satisfy the tolerance, then its return distance is compared to its neighboring integral curves. If a zero crossing is found, then a binary search is conducted. The binary search is also limited by a maximum number of iterations.

Source

The set of points that seed the integral curves that reveal the Limit Cycles. In addition to the *Source* attributes common to all ICS operators, the Limit Cycle operator supports the following attributes:

Source Type

The source type controls how the seeds for curves are created. The Limit Cycle operator only supports uniform samples on a line.

Integration

Specify settings for numerical integrators. In addition to the *Integration* attributes common to all ICS operators, the Limit Cycle operator supports the following attributes.

Integration Direction

Sets the integration direction through time. The user can choose from a combination of forward, backward, and directionless. Eigen vectors are an example of a directionless vector field. In order to integrate using a directionless field, any orientation discontinuity must be corrected prior to linear interpolation. That is, all vectors must be rotated

to match the orientation of the trajectory. The ICS code will do this processing for standard fields (e.g non-higher order elements).

Termination

Integral curve termination can be controlled in several different ways. The termination is based on the most conservative criteria, so only one criteria must be met for termination. The options are:

Maximum number of steps The maximum number of integration steps that will be allowed.

Appearance

The appearance tab specifies how the integral curve will be rendered. In addition to the *Appearance* attributes common to all ICS operators, the Integral Curve operator supports the following attributes.

Cycle tolerance

The smallest return distance for classifying an integral curve as a limit cycle.

Maximum iterations

The maximum numbers of iterations when performing the bi-section method.

Show partial results

If the maximum number of bi-section iterations has been reached without finding a limit cycle, show the integral curves still in the queue.

Show the signed return distances for the first iteration

Instead of plotting the limit cycles, plot the return distances along the Poincaré section.

Coloring

There are various coloring options, the names of which are self-descriptive such as coloring the curves with a *solid* color or according to a *seed*. Only those options that require further clarification are described further here.

Average Distance from seed Each curve is colored according to the average distance of all the points in the curve from the seed.

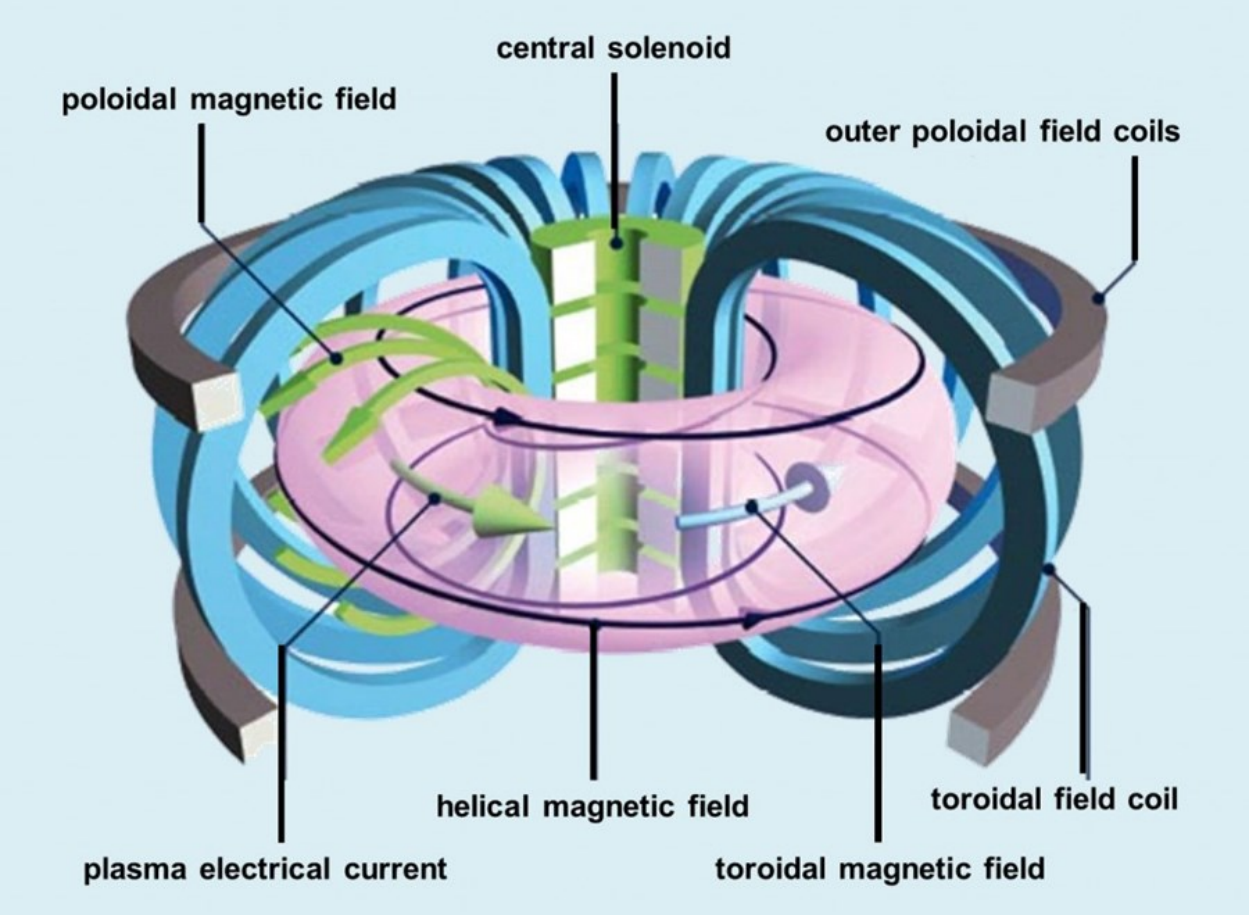
Variable Each curve's color varies by the value of a scalar variable.

Advanced

See *Advanced* tab attributes that are common to all ICS operators.

Poincaré operator

The Poincaré operator constructs a Poincaré section for toroidal geometry. The basis of constructing a connected plot is to accurately determine the number of toroidal and poloidal windings (i.e. the winding pair). The image below is helpful for visually understanding what is meant by toroidal and poloidal:



This process is iterative, starting with a minimum number of puncture points through a Poincaré section and continuing until the toroidal and poloidal windings are known or the maximum number of punctures is reached. If an accurate winding pair is determined, then the puncture points are connected based on it. For more information, refer to the following resources:

A.R. Sanderson, G. Chen, X. Tricoche, E. Cohen. "Understanding Quasi-Periodic Fieldlines and Their Topology in Toroidal Magnetic Fields," In Topological Methods in Data Analysis and Visualization II, Edited by R. Peikert and H. Carr and H. Hauser and R. Fuchs, Springer, pp. 125--140. 2012.

A.R. Sanderson, G. Chen, X. Tricoche, D. Pugmire, S. Kruger, J. Breslau. "Analysis of Recurrent Patterns in Toroidal Magnetic Fields," In Proceedings Visualization / Information Visualization 2010, IEEE Transactions on Visualization and Computer Graphics, Vol. 16, No. 4, pp. 1431-1440. 2010.

Source

The set of points that seed the integral curves that reveal the Poincaré section. In addition to the *Source* attributes common to all ICS operators, the Poincaré operator supports the following attributes:

Source Type

The source type controls how the seeds for curves are created. There are various options, the names of which are self-descriptive such as creating them along a *line*. Only those options that require further clarification are described further here.

Point List Seed from a list of points. In addition to *Add Point*, *Delete Point*, and *Delete All Points*, the user can *Read Text File* that is formatted with one point per each line either as "X Y Z" or "X, Y, Z".

Warning: If the Field is set to M3D-C1 integrator the point locations will be converted from Cartesian to Cylindrical coordinates. In the 2D case, phi will be set to 0.

Integration

Specify settings for numerical integrators. In addition to the *Integration* attributes common to all ICS operators, the Poincaré operator supports the following attributes.

Punctures

While integrating the integral curve to be used the for Poincaré plot, the user has the option to require a minimum number of initial punctures through the Poincaré section for the analysis. The user may limit the integration in case of run-a-way integral curve that cannot be fully analyzed.

Puncture plot type The type of the puncture plot. Options are:

- Single - the analysis is based on the standard double periodic system (toroidal-poloidal periodicity)
- Double - the analysis is based on the double Poincaré plot. In addition to the toroidal-poloidal periodicity a third periodicity exists that is based on the integration time.

When selecting double, Poincaré plot puncture points are accepted if and only if the period is within the tolerance of the period (the period is set as part of the Poincaré Pathline Options).

- Period tolerance - when an integral curve punctures the plane, the period must be within the tolerance value.

Warning: When selecting "Toroidal" the "Analysis" must also be set to "Punctures only" as there is currently no analysis in the toroidal plane.

Analysis

The user may adjust settings for how Poincaré analysis is to be done. Some options include:

None - Puncture only This will result in constructing a traditional Poincaré plot using only points.

Full This will analyze each curves' geometry and attempt to reconstruct the cross sectional profile of the surface which the curve lies on. Further, the analysis attempts to identify the topology of the surface.

Maximum toroidal winding Limit the search of the toroidal winding to lower order values. Zero indicates no limit.

Override toroidal winding In some cases, such as debugging, it may be informative to force the toroidal winding to have a set value. Zero indicates no override.

Override poloidal winding In some cases such as debugging, it may be informative to force the poloidal winding to have set value. Zero indicates no override.

Winding pair confidence (Range 0-1, Default 0.9) Sets the limit for the number of mismatches in the consistency in the winding pairs.

Detect Rational Surface Allows for the construction of rational surfaces via an iterative process. Typically, they can be constructed with 5-10 iterations.

> **Danger:** The rational surface construction is experimental code and does not always work.

Detect O Points Allows for the detection of O points in "island chains" via an iterative process. Typically, they can be detected with 5 iterations.

> **Danger:** The critical point detection is experimental code and does not always work.

Perform O-Line Analysis Calculate the poloidal winding relative the O-Line (central axis) which provides a more accurate winding value.

- O-Line toroidal windings (Default 1) – sets the toroidal winding value, i.e. the period (for the central axis the period is 1).
- O-Line Axis Point File - allows the user to select a text file containing the points along the axis from 0 to 360 degrees (note there is no overlap P(0) != P(n)).

Show chaotic fieldlines as points Because chaotic curves cannot be classified, they are not displayed unless this is checked.

Show islands only Culls the results so that only island chains are displayed.

Show ridgelines Displays the 1D plots of the distance and ridgeline samples.

Verbose Dumps information regarding the analysis to the terminal. The final summary may be useful. For example,

```
Surface id = 0 < 2.35019 0 0.664124 > 121:11 121:11 (11) flux surface with 4 nodes␣
↪(Complete)
Surface id = 0
seed location < 2.35019 0 0.664124 >
the winding pair 121:11
the toroidal:poloidal periods (as a winding pair) 121:11
the multiplication faction (11) i.e. diving by this number will give the base winding␣
↪values, in this case 11:1.
surface type: flux surface
number of nodes in each winding group: with 4 nodes
analysis state: complete.
```

Appearance

The appearance tab specifies how the integral curve will be rendered. In addition to the *Appearance* attributes common to all ICS operators, the Poincaré operator supports the following attributes.

Coloring

The various coloring options are:

None Solid color from the single color

Safety Factor Q Use the safety factor

Safety Factor P Use the safety factor as defined when there are two possible choices for the magnetic axis

Safety Factor Q == P Render the surfaces on if the safety factor Q is equal to the safety factor P

Safety Factor Q != P Render the surfaces on if the safety factor Q is not equal to the safety factor P

Toroidal Windings Q Use the toroidal winding value used in the calculation of Q

Toroidal Windings P Use the toroidal winding value used in the calculation of P

Poloidal Windings Use the poloidal winding value

Fieldline Order Use input order of the seeds used to generate the integral curves.

Point Order Use the puncture point index

Plane Use the plane value (integer from 0 to N where N is the number of planes)

Winding Group Order Use the winding group order (integer from 0 to T where T is the toroidal winding)

Winding Point Order Use the index of the puncture points within each winding group

Winding Point Order Modulo Order Use the order of the punctures within each winding group modulo the toroidal windings (useful for islands in islands)

Display

Allows the users to display the results in a single plane or multiple planes. Further, one can reconstruct the 3D surface that the curves lies on.

Overlapping Curve Sections

When displaying the data in a connected manner the raw data will often overlap itself. As such, for visually pleasing results it may be preferable to remove the overlaps.

Raw Display all of the punctures points in a connected fashion.

Remove Display all of the punctures points in a connected fashion, removing the overlapping sections.

Merge Display all of the punctures points in a connected fashion, merging the overlapping sections. Experimental.

Smooth Display all of the punctures points in a connected fashion, removing the overlapping sections while smoothing between points.

Danger: Smooth is experimental and does not always work.

Advanced

See *Advanced* attributes that are common to all ICS operators.

Parameters

Common to all ICS operators is a four tab GUI: Source, Integration, Appearance, and Advanced (the Poincaré operator also has an Analysis tab). These tabs contain many functions that are common across all four operators. The following is a description of those common features.

Source

The set of points that seed the integral curves. See each operator for varied settings.

Field

Sets the field type so that the native elements are used when interpolating the vector fields. Each operator provides the following options:

Warning: Each option below besides "Default" requires the respective third party library.

Default Use VisIt's native VTK mesh structure to perform linear interpolation on the vector field.

Flash Evaluates the velocity field via the Lorentz force. Parameters are:

- Constant - A constant multiple applied to the velocity.
- Velocity - When combined with Leap-Frog integration, this sets the initial velocity used in the integration.

M3D-C1 2D Evaluates the 3D magnetic field via a 2D poloidal 6th order polynomial. Parameters are:

- Constant - A constant multiple applied to the perturbed part of the field.

M3D-C1 3D Evaluates the 3D magnetic field via a 2D poloidal 6th order polynomial and 1D toroidal 4th order Bezier spline.

Nek5000 Evaluates the 3D vector field using Nek5000 spectral elements.

Nektar++ Evaluates the 3D vector field using Nektar++ spectral elements.

Integration

Specify settings for the numerical integrator. See each operator for varied settings.

Integrator

Sets the integration scheme. There are various options common among numerical integration packages, such as *Leap Frog* and *Runge-Kutta*. More details on the different schemes can be found through a simple online search.

Step Length

Most integrators use a fixed step length. Runge-Kutta-Dormand-Prince (RKDP) uses adaptive step size, which can be clipped by the step length.

Tolerances

RKDP, Adams-Bashforth, and MD3-C1 make use of the tolerance options.

RKDP The step size adapts to ensure that the maximum error at each step is less than the maximum between the absolute tolerance and the relative tolerance times the value of the vector field at the current point. The absolute tolerance can be truly absolute or relative to the bounding box.

Termination

The criteria for terminating the integration. See specific operator for details.

Appearance

Specify appearance settings for the curves. See each operator for varied settings.

Streamlines vs Pathlines

The user may select the integral curve to be based on an instantaneous or time-varying vector field producing streamlines or pathlines, respectively. A streamline is a path rendered by an integrator that uses the same vector field for the entire integration. A pathline uses the vector field that is in-step with the integrator, so that as the integrator steps through time, it uses data from the vector field at each new time step. Pathline options are:

Override starting time Instead of starting with the current time step, utilize another time for the start time.

Interpolation over time Interpolate the integral curve with a static mesh for all time or with a varying mesh at each time step. The mesh is typically static, but this cannot always be assumed and should be verified for each dataset before use.

Advanced

Parallel integration

The user may select one of four different parallelization options when integrating curves in parallel:

Parallelize over curves Distribute the curves between the processors. Parameters are:

- Domain cache size - number of blocks to hold in memory for level of details.

Parallelize over domains Distribute the domains between the processors. Parameters are:

- Communication threshold - number of integral curve to process before communication occurs.

Parallelize over curves and domains Distribute both the curves and domains between the processors.

Have VisIt select the best algorithm VisIt automagically selects the best parallelization algorithm.

Warnings

Alerts for various conditions that may occur during the integration or analysis.

Issue warning when the maximum number of steps is reached The maximum number of steps limits run-a-way integration.

Issue warning when a step size underflow is detected If the step size goes to zero, issue a warning.

Issue warning when stiffness is detected Stiffness refers to one vector component being so much larger than another that tolerances can't be met.

Issue warning when a curve doesn't terminate at a critical point For example, the curve may circle around a critical point without converging.

Index Select operator

The Index Select operator selects a subset of a 2D or 3D structured mesh based on ranges of cell indices. Structured meshes have an implied connectivity that allows each cell in the mesh to be specified by an i,j or i,j,k index depending on the dimension of the mesh. The Index Select operator allows you to specify different ranges for each mesh dimension. The ranges are used to select a brick of cells from the mesh. In addition to indices, the Index Select operator uses stride to select cells from the mesh. Stride is a value that allows the operator to count by 2's, 3's, etc. when iterating through the range indices. Stride is set to 1 by default. When higher values are used, the resulting mesh is more coarse since it contains fewer cells in each dimension. The Index Select operator attempts to preserve the size of the mesh when non-unity stride values are used. An example of the Index Select operator appears in Figure 4.125.

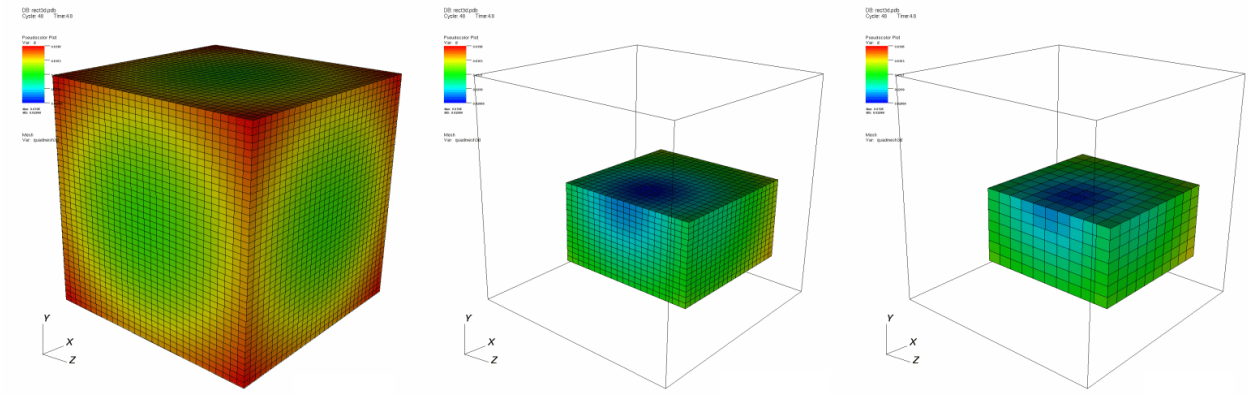


Fig. 4.125: Index Select operator example: original plot; index selected (stride=1); index selected (stride=2)

Setting a selection range

The **Index Select attributes window**, shown in Figure 4.126, contains nine spin boxes that allow you to enter minimum and maximum ranges for i,j,k. To select all cells in the **X** dimension whose index is greater than 10, you would enter 10 into the spin box in the **I** row and **Min** column. Then you would enter max into the spin box in the **Max** column in the **I** row. Finally, you would enter a stride of 1 into the spin box in the **Incr** column in the **I** row. If you wanted to sub-select cell ranges for the Y dimension, you could follow a similar procedure using the spin boxes in the **J** row and so forth. To set a range, first select the maximum number of dimensions to which the Index Select operator will apply. To set the dimension, click on the **1D** , **2D** , **3D** radio buttons. Note that if the chosen number of dimensions is larger than the number of dimensions in the database, the extra dimension ranges are ignored. It is generally best to select the same number of dimensions as the database. The three range text fields are listed in i,j,k order from top to bottom. To restrict the number of cells in the X-dimension, use spin boxes in the **I** row. To restrict the number of cells

in the Y-dimension, use the spin boxes in the **J** row. To restrict the number of cells in the Z-dimension, use the spin boxes in the **K** row.

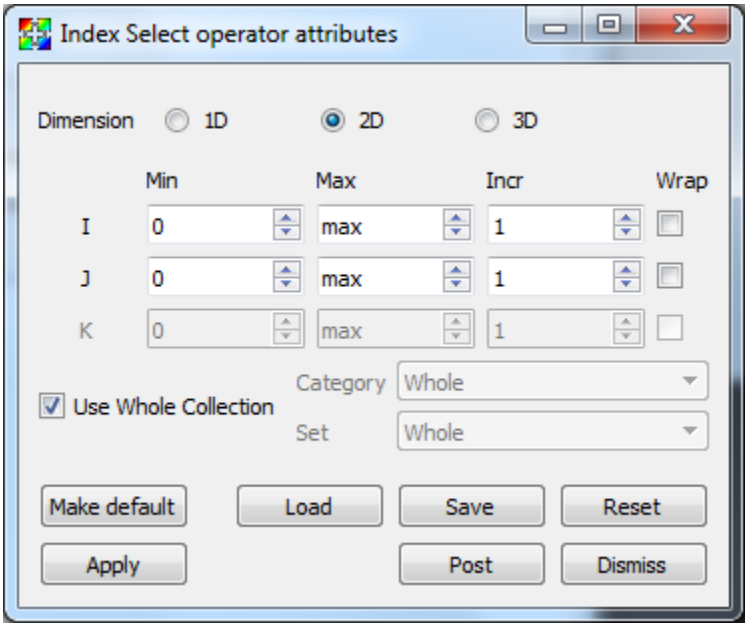


Fig. 4.126: Index Select attributes window

Restricting to a subset of the whole database

Some databases are composed of multiple groups of meshes, which are often called groups or blocks. Some databases are composed of multiple meshes, often called blocks or domains. Some are composed of both groups and domains. When examining a database, you might want to look at only one block or group at a time. By default, the Index Select operator is applied to all blocks in the database. This means that each index range is applied to each block in the database and will probably result in an image featuring several small chunks of cells. When the Index select operator is set to apply to just one block or group, the index ranges are relative to the specified block or group.

To make the Index Select operator apply to just one block or group, uncheck the **Use Whole Collection** check box. The **Category** and **Set** combo boxes will be filled according to how the database has named the groups or sub-meshes. Choose the correct category from the **Category** combo box, and the desired set from the **Set** combo box. Figure 4.127 shows a single mesh selection for a multiple mesh database whose sub-meshes are called domains.

InverseGhostZone operator

The InverseGhostZone operator makes ghost cells visible and removes real cells from the dataset so plots to which the InverseGhostZone operator have been applied show only the mesh's ghost cells. Ghost cells are a layer of cells around the mesh that usually correspond to real cells in an adjacent mesh when the whole mesh has been decomposed into smaller domains. Ghost cells are frequently used to ensure continuity between domains for operations like contouring. The InverseGhostZone operator is useful for debugging ghost cell placement in simulation data and for database reader plugins under development.

The InverseGhostZone operator's attributes window (Figure 4.129) has various **Show** options allowing you to select which types of ghost cells are returned. By default all options are turned on.

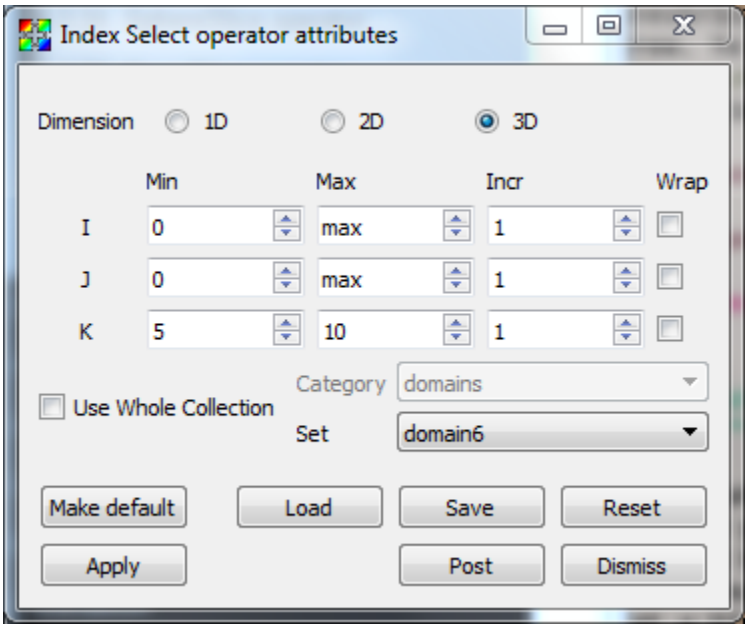


Fig. 4.127: Setting the category for index selection

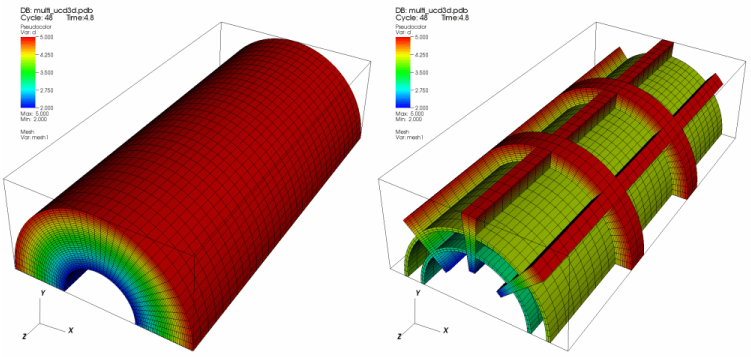


Fig. 4.128: InversetGhostZone example

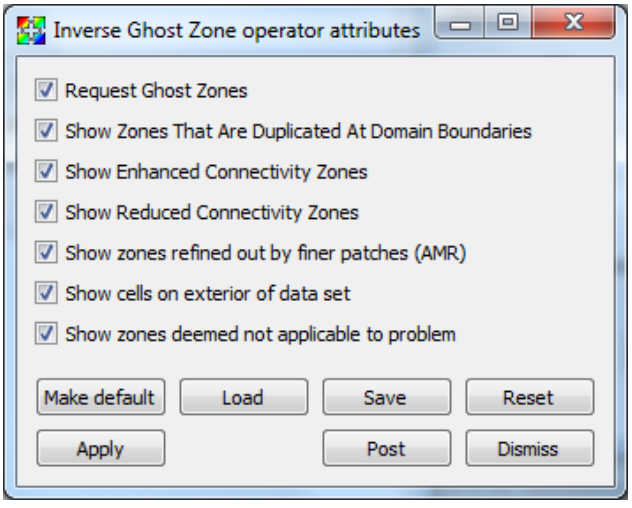


Fig. 4.129: InversetGhostZone window

Isosurface operator

The Isosurface operator extracts surfaces from 2D or 3D databases and allows them to be plotted. The Isosurface operator takes as input a database and a list of values and creates a set of isosurfaces through the database. An isosurface is a surface where every point on the surface has the same data value. You can use an isosurface to see a surface through cells that contain a certain value. The Isosurface operator performs essentially the same visualization operation as the Contour plot, but it allows the resulting data to be used in VisIt's other plots. For example, an Isosurface operator can be applied to a Pseudocolor plot where the Isosurface variable is different from the Pseudocolor variable. In that case, not only are the isosurfaces shown, but they are colored by another variable. An example of the Isosurface operator is shown in Figure 4.130.

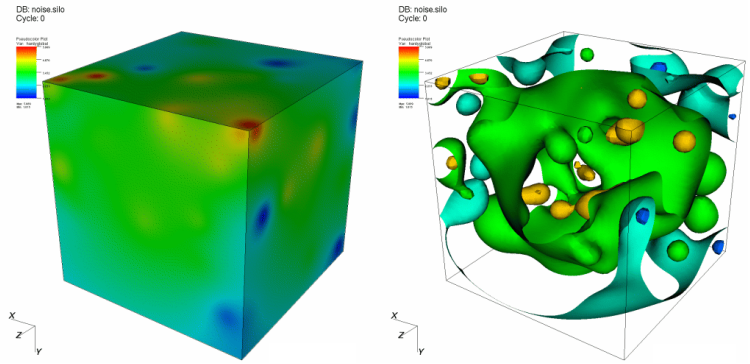


Fig. 4.130: Isosurface operator example

Setting isosurface levels

By default, VisIt constructs 10 levels into which the data fall. These levels are linearly interpolated values between the data minimum and data maximum. However, you can set your own number of levels, specify the levels you want to see or indicate the percentages for the levels.

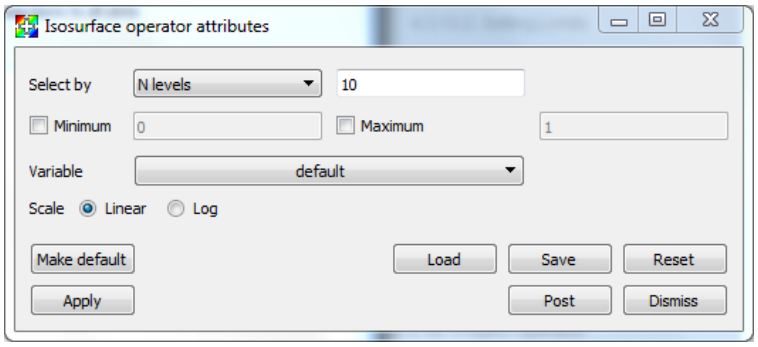


Fig. 4.131: Isosurface attributes

To choose how levels are specified, make a selection from the **Select by** menu. The available options are: **N levels**, **Levels**, and **Percent**. **N levels**, the default method, allows you to specify the number of levels that will be generated, with 10 being the default. **Levels** requires you to specify real numbers for the levels you want to see. **Percent** takes a list of percentages like 50.5 60 40. Using the numbers just mentioned, the first isosurface would be placed at the value which is 50.5% of the way between the minimum and maximum data values. The next isosurface would be placed at the value that is 60% of the way between the minimum and maximum data values, and so forth. You specify all values for setting the number of isosurfaces by typing into the text field to the right of the **Select by** menu.

Setting Limits

The **Isosurface attributes window**, shown in Figure 4.131, provides controls that allow you to specify artificial minima and maxima for the data in the plot. You might set limits when you have a small range of values that you are interested in and you only want the isosurfaces to be generated through that range. To set the minimum value, click the **Minimum** check box to enable the **Minimum** text field and then type a new minimum value into the text field. To set the maximum value, click the **Maximum** check box to enable the **Maximum** text field and then type a new maximum value into the text field. Note that either the minimum, maximum or both can be specified. If neither minimum nor maximum values are specified, VisIt uses the minimum and maximum values in the dataset.

Scaling

The Isosurface operator typically creates isosurfaces through a range of values by linearly interpolating to the next value. You can also change scales so a logarithmic function is used to get the list of isosurface values through the specified range. To change the scale, click either the **Linear** or **Log** radio buttons in the **Isosurface attributes window**.

Setting the isosurfacing variable

The Isosurface operator database variable can differ from the plotted variable. This enables plots to combine information from two variables by having isosurfaces of one variable and then coloring the resulting surfaces by another variable. You can change the isosurfacing variable, by selecting a new variable name from the **Variable** variable button.

Sometimes it is useful to set the isosurfacing variable when the plotted variable is not a scalar. For example, you might want to apply the Isosurface operator to a Mesh plot but the Mesh plot's plotted variable is not a scalar so the Isosurface operator does not know what to do. To avoid this situation, you can set the isosurfacing variable to one you know to be scalar and the operator will succeed.

Isovolume operator

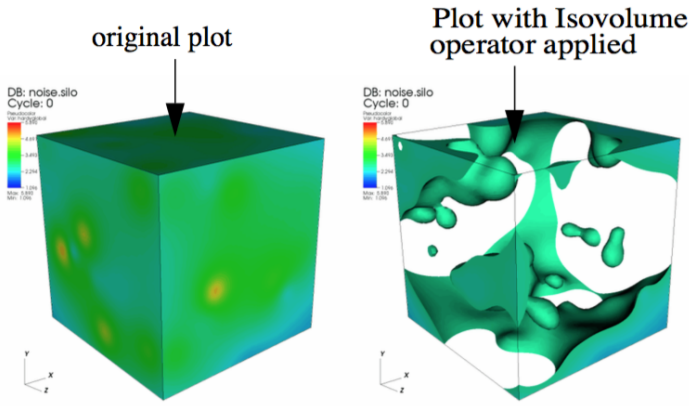


Fig. 4.132: Isovolume Operator Example

The Isovolume operator creates a new unstructured mesh using only cells and parts of cells from the original mesh that are within the specified data range for a variable. The resulting mesh can be used in other VisIt plots. You might use this operator when searching for cells that have certain values. The Isovolume operator can either use the plotted variable or a variable other than the plotted variable. For instance, you might want to see a Pseudocolor plot of pressure

while using the Isovolume operator to remove all cells and parts of cells below a certain density. An example of a plot to which an Isovolume operator has been applied is shown in .

Using the Isovolume operator

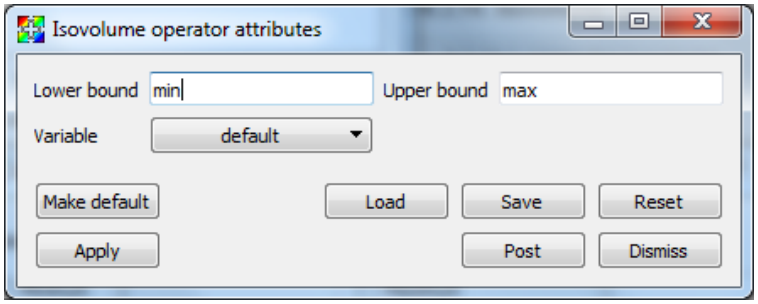


Fig. 4.133: Isovolume Attributes Window

The Isovolume operator iterates over every cell in a mesh and determines which parts of the cell, if any, contain a value that falls within a specified data range. If any parts of the cell are within the specified data range, they are kept as part of the operator's output. The Isovolume operator uses an isosurfacing algorithm to determine the interfaces where cells should be split so the interfaces for neighboring cells are all continuous and fairly smooth. To specify a data range, type new upper and lower bounds into the **Lower bound** and **Upper bound** text fields in the **Isovolume Attributes Window**, which is shown in Figure 4.133.

The variable that the Isovolume operator uses does not necessarily have to match the plotted variable. If the plotted variable is to be used, the **Variable** text field must contain the word: default. If you want to make the Isovolume operator use a different variable so you can, for example, plot temperature but only look at regions that have a density greater than 2g/mL, you can set the Isovolume's variable to temperature. To make the Isovolume operator use a different variable, select a new variable from the **Variable** variable button in the **Isovolume Attributes Window**.

If you apply this operator to a plot that does not operator on scalar variables such as the Mesh or Subset plots, be sure to set the variable because the default variables for those plots is never a scalar variable. Without a scalar variable, the Isovolume operator will not work.

Lineout operator

The Lineout operator samples data values along a line, producing a 1D dataset from datasets of greater dimension. This operator is used implicitly by VisIt's Lineout capability and cannot be added to plots. For more information on Lineout, see the *Lineout* section in the *Quantitative Analysis* chapter.

LineSampler operator

The Line Sampler operator is used for sampling a 3D/2D dataset in much the same way as the Lineout tool except in a much more defined manner. That is, the user can define a series of "arrays" (e.g. planes) that consists of one or more "channels" over which the data is sampled. For each array the orientation of the plane can be defined. Whereas for each channel, its orientation within the plane and the sampling type and spacing can be defined. For instance, the sampling can be a series of lines through the data, or a series of points integrated over time.

For more information on LineSampler, see the *LineSampler* section in the *Quantitative Analysis* chapter.

Merge operator

VisIt's Merge operator merges all geometry that may exist on separate processors into a single geometry dataset on a single processor. The Merge operator can be useful when applying other operators like the Decimate operator or when creating Streamline plots. The Merge operator is not enabled by default.

OnionPeel operator

The OnionPeel operator creates a new unstructured mesh by taking a seed cell or node from a mesh and progressively adds more layers made up of the initial cell's neighboring cells. The resulting mesh is then plotted using any of VisIt's standard plots. The OnionPeel operator is often useful for debugging problems with scientific simulation codes, which often indicate error conditions for certain cells in the simulated model. Armed with the cell number that caused the simulation to develop problems, the user can visualize the simulation output in VisIt and examine the bad cell using the OnionPeel operator. The OnionPeel operator takes a cell index or a node index as a seed from which to start growing layers. Only the seed is shown initially but as you increase the number of layers, more of the cells around the seed are added to the visualization. An example of the OnionPeel operator is shown in Figure 4.134.

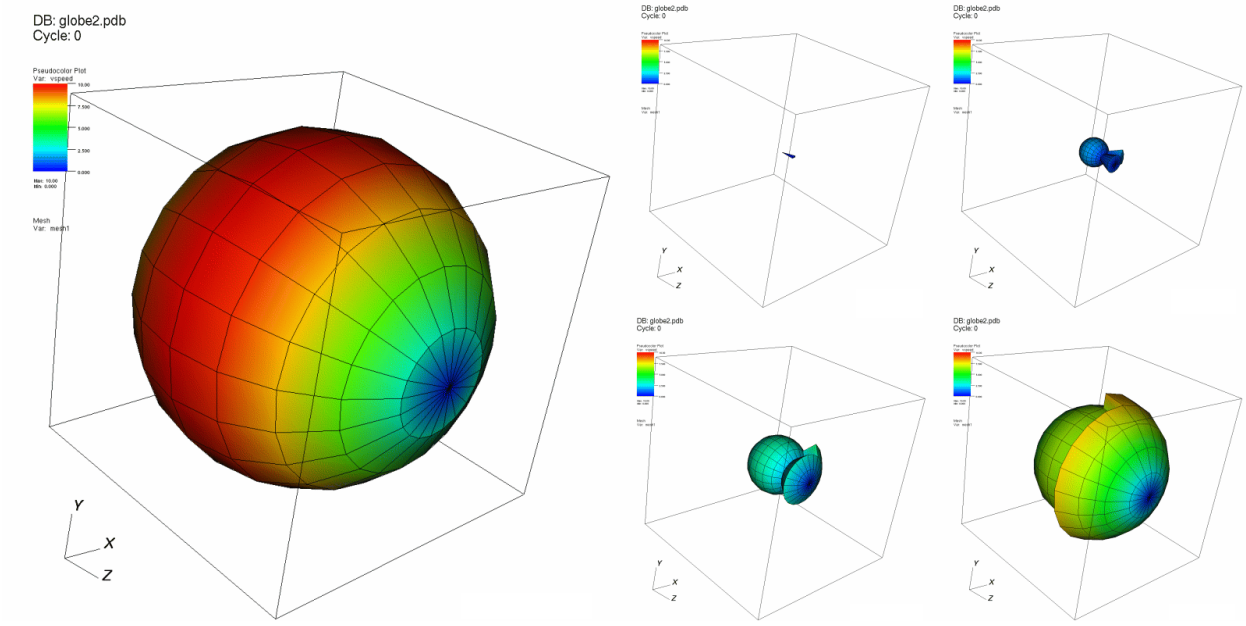


Fig. 4.134: Onion peel operator example

Setting the seed

The OnionPeel operator uses a seed cell or a seed node as the seed to which all cells from other layers are added. When a layer is added around the seed, the new cells are those immediately connected to the seed. You specify the seed as a cell index or a node index by typing a new seed value into the **Seed# or ij[k]** text field. VisIt interprets the seed as a cell index by default. If you want to start growing cell layers around a given node, click on the **Node** radio button before entering a new seed value. The form of the seed index depends on how the underlying mesh is organized. Unstructured meshes, which are a collection of independent cells, require only a single integer value for the seed while structured meshes are indexed with i,j or i,j,k indices depending on the dimension of the mesh. To set the seed using i,j,k indices, type the i and j and k indices, separated by spaces, into the **Seed# or ij[k]** text field.

Some meshes that have been decomposed into multiple smaller meshes known blocks or domains have an auxiliary set of cell indices and node indices that allow cells and nodes from any of the domains to be addressed as though each

domain was part of a single, larger whole. If you have such a mesh and want to specify seed indices in terms of global cell indices or global node indices, be sure to turn on the **Seed# is Global** check box.

The OnionPeel operator can only operate on one domain at a time and when the operator grows layers, they do not cross domain boundaries. The seed cell index is always relative to the active domain. To make a cell in a different domain the new seed cell, change the domain number by selecting a new domain from the **Set** combo box.

Growing layers

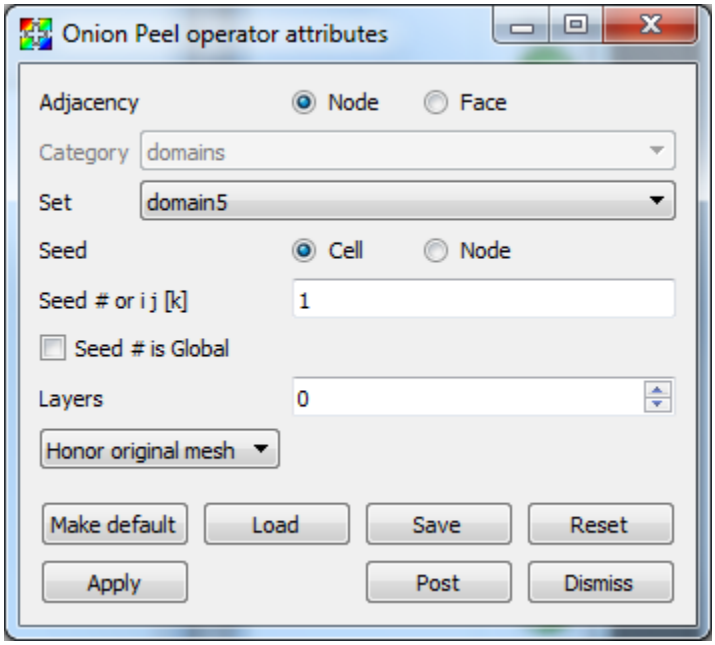


Fig. 4.135: Onion peel attributes

The OnionPeel operator starts with a seed and adds layers of new cells around that seed. The added cells are determined by the layer number and the adjacency information. The cell adjacency rule determines the connectivity between cells. Cells are next to each other if they share a cell face or a cell node. The visualization will differ slightly depending on which adjacency rule is used. To change the adjacency rule, click the **Node** or the **Face** radio buttons in the **OnionPeel attributes window**, shown in Figure 4.135.

The OnionPeel operator initially shows zero layers out from the seed, so only the seed is shown in the visualization when the OnionPeel operator is first applied. Consequently, the visualization might appear to be empty since some seed cells are very small. To add more layers around the seed, enter a larger layer number into the **Layer Number** text field. Clicking the up or down buttons next to the **Layer Number** text field also increments or decrements the layer number.

By default, Onion Peel will honor the structure of the original mesh. In some cases, as with arbitrary polyhedral data, you may want to see how VisIt split the original mesh. In this case, use the combo box to change to **Honor actual mesh**.

Project operator

The Project operator sets all of the Z values in the coordinates of a 3D mesh to zero and reduces the topological dimension of the mesh by 1. The Project operator is, in essence, an operator to make 2D meshes out of 3D meshes. An example of the Project operator is shown in Figure 4.136.

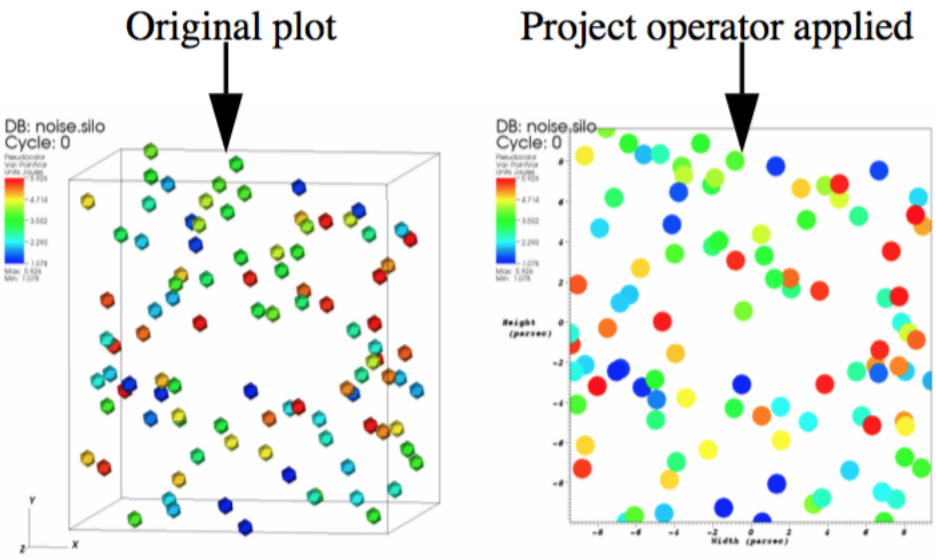


Fig. 4.136: Project Operator Example

Setting the projection type

The Project operator can project 3D down to 2D using either Cartesian or Cylindrical transforms, which can be performed along the X, Y or Z axis, as shown in (see Figure 4.137). To specify which of these transforms you want to use when using the Project operator, choose the appropriate option from the **Projection type** combo box. **Z-Axis Cartesian** is the default option.

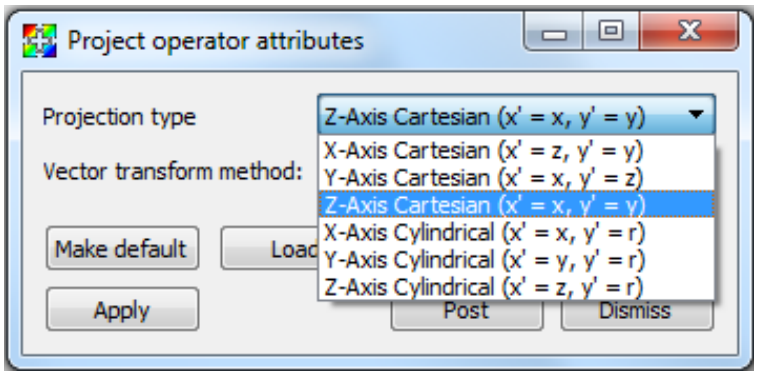


Fig. 4.137: Project Attributes Window showing available projection types

Choosing how vectors are treated

The Project operator can treat vectors as instantaneous directions, as coordinate displacements or as point coordinates. The Project operator can also ignore the vectors and not transform them at all. To specify how you wish vectors to be treated during the projection transform, choose the appropriate option from the **Vector transform method** combo box. (see Figure 4.138) The default is **Treat as instantaneous directions**.

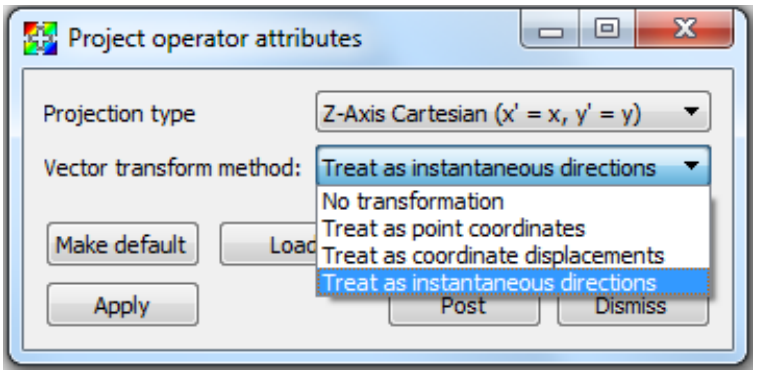


Fig. 4.138: Project Attributes Window showing available vector treatments

Reflect operator

Use the Reflect operator to reflect database geometry across one or more axes. Scientific simulations often rely on symmetry so they only need to simulate part of the problem. When creating a visualization, most users want to see the entire object that was simulated. This often involves reflecting the database geometry to create the full geometry of the simulated object. VisIt's Reflect operator can be applied to both 2D and 3D databases and can reflect them across one or more plot axes. An example of the Reflect operator is shown in Figure 4.139.

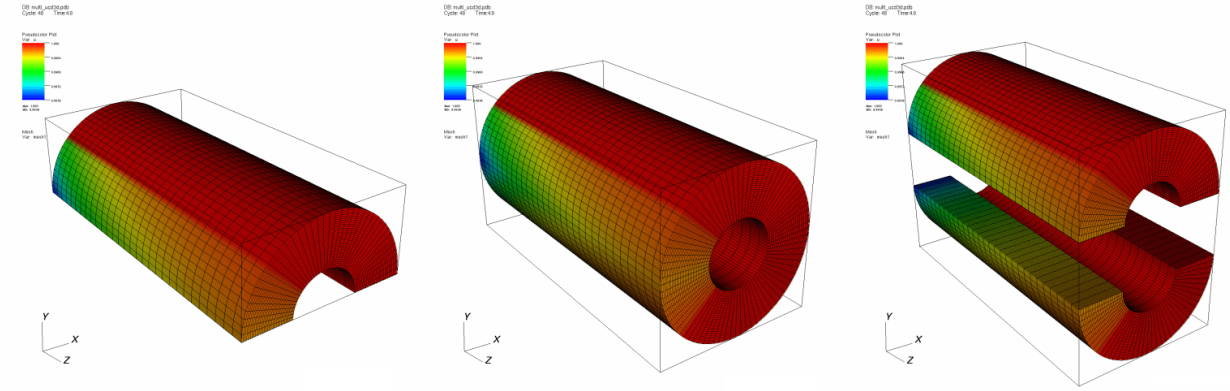


Fig. 4.139: Reflect operator example

Setting the Reflect attribute window's input mode

The **Reflect attributes window**, shown in Figure 4.140, has two input modes. One input mode is for 2D data, in which only reflection quadrants are shown, and the second input mode is for 3D data for which the window shows 3D octants. In either input mode, clicking on the brightly colored shapes turns on different reflections and in the 3D input mode, clicking on the cyan arrow rotates the view so you can more easily get to reflections in the back. To set the input mode, click either the **2D** or **3D** radio buttons.

Setting the data octant

The Reflect operator assumes that the database being reflected resides in the +X+Y+Z octant when performing its reflections. Sometimes, due to the orientation of the database geometry, it is convenient to assume the geometry exists in another octant. To change the data octant, make a new selection from the **Original data octant** menu in the **Reflect**

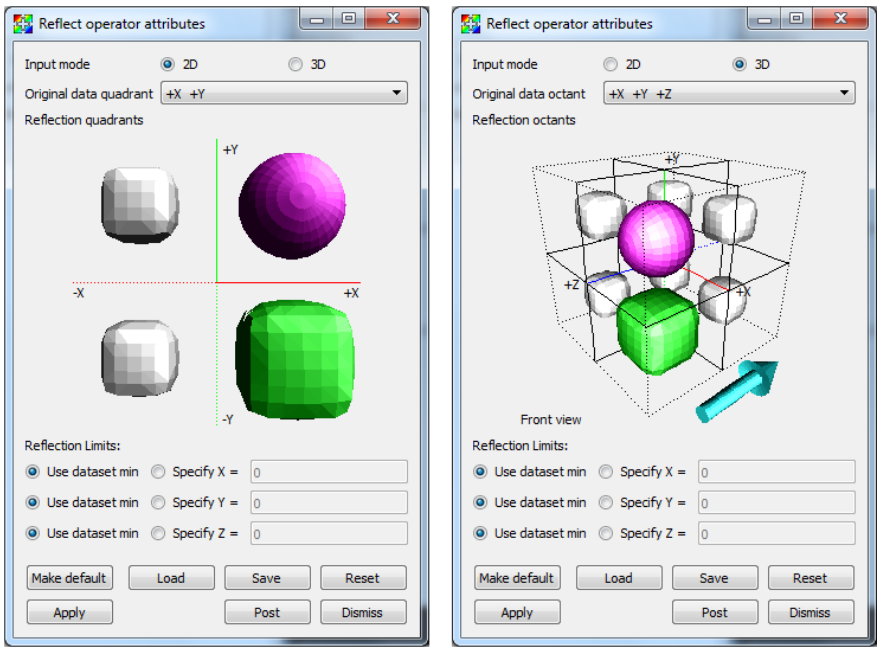


Fig. 4.140: Reflect attributes window

attributes window. The **Reflect attributes window** graphically depicts the original data octant as the octant that contains a sphere instead of a cube, which correspond only to reflections.

Reflecting plots

Once the Reflect operator has been applied to plots, you must usually specify the direction in which the plots should be reflected. To set the plot direction, click on the glyphs below the **Original data octant** menu. The possible reflections are shown by cube and sphere glyphs. When a reflection is set to be on, the glyph in the octant or quadrant will be green or magenta. When a reflection is not on, its glyph is smaller and silver. To turn a reflection on or off, just click on its glyph. If the window is in its 3D input mode and you need to access octants in the back that are obscured by other octants, clicking on the cyan arrow will rotate the glyphs so the octants in the back will be more accessible.

Reflection limits

Reflection limits determine the axes about which the database geometry is reflected. The Reflect attributes window has three reflection limits controls; one for each dimension. You will usually want to reflect plots using the dataset min value, which you set by clicking the **Use dataset min** radio button. When using the dataset min value to reflect plots, the reflected plots will touch along the reflected edge. You can also specify another axis of reflection. When using a custom axis of reflection, the reflected plots will not necessarily touch. This option, though not normally needed, can produce interesting effects in animations. To specify a custom axis of reflection, click the **Specify X, Specify Y, or Specify Z** radio buttons and enter a new X, Y, or Z value into the appropriate text field.

Replicate operator

The Replicate operator is most often used in atomic and molecular visualization in VisIt, and can be combined with the *Molecule Plot* and *Create Bonds operator*. See *Molecular data features* for examples of the Replicate operator in use with the *Molecule Plot*.

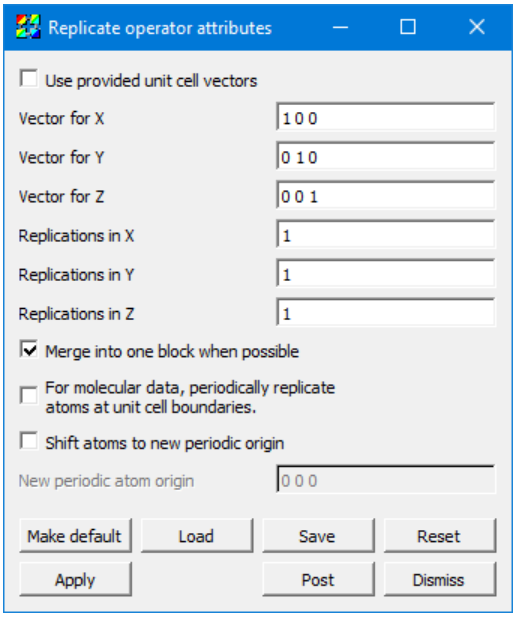


Fig. 4.141: Replicate attributes window

Some file formats specify the vectors for the unit cell (sometimes called "direct lattice" vectors) containing the molecular data in the file. If they are present and **Use provided unit cell vectors** is checked, then it will use those values instead of the ones specified in this window.

Vector for X, Y, and Z are controls for specifying the actual vectors describing the amount to displace for a replication in each of the three axes. (The X, Y, and Z labels are only for disambiguation; there is no requirement that the actual vectors specified be related to their name.)

Replications in X, Y, and Z specifies the total number of instances of the data set to create. E.g. 1,1,1 specifies the original data set with no replications. 2,1,1 specifies a total of two instances – one is the original, and the other is a new one created at a displacement of 1x along the "X" vector.

The **Merge into one block when possible** flag specifies that the output of this operator should be created in a single "chunk", and helps with correct operation of the *Create Bonds operator*. It is recommended to leave this enabled.

When there are periodic boundary conditions, atoms at the boundaries of the unit cell are, by definition, logically present at the matching opposite boundaries as well. By checking **For molecular data, periodically replicate atoms at unit cell boundaries**, it creates those atoms which, after replication, would still fall in the unit cell's inclusive boundaries.

For example, in a periodic unit cell with origin [0,0,0] and dimensions [1,1,1], suppose there is an atom centered on the minimum-Z face, i.e. located at [0.5, 0.5, 0]. Due to the periodic boundary conditions, this means that there should be another instance of this atom at the maximum-Z face, i.e. at [0.5, 0.5, 1]. If you set the number of Z replications to at least 2, then it will create this other instance of the atom as desired. However, it will also create any atoms which lie in the replicated cell between z=1 and z=2. Sometimes you want to replicate just those atoms which are still within the original unit cell after replication (within epsilon). By checking this flag, but leaving the number of replications at 1,1,1, this operator will create the instance of the atom at [0.5, 0.5, 1] without adding the other atoms at z>1.

Shift atoms to new periodic origin enables the ability to set an origin (using **New periodic atom origin**) for periodic atom creation.

Resample operator

The Resample operator extracts data from any input dataset in a uniform fashion, forming a new 2D or 3D rectilinear grid onto which the original dataset has been mapped. The Resample operator is useful in a variety of contexts such as downsampling a high resolution dataset (shown in Figure 4.142), rendering Constructive Solid Geometry (CSG) meshes, or mapping multiple datasets into a common grid for comparison purposes.

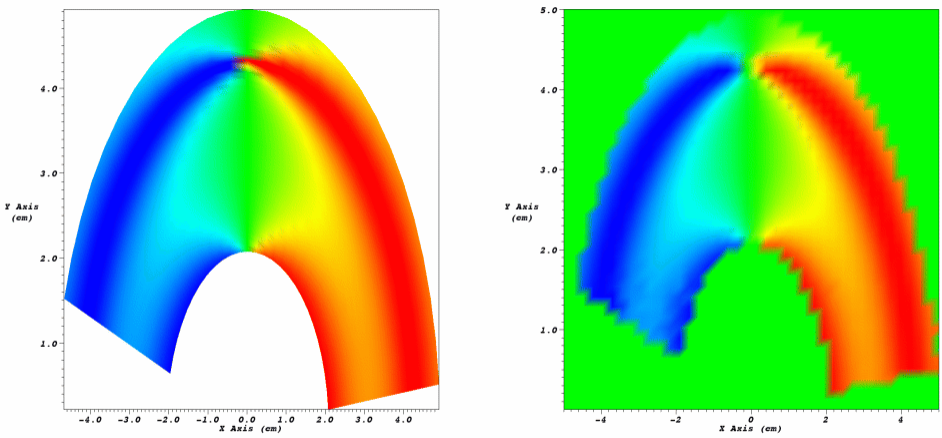


Fig. 4.142: Resample operator example

Resampling onto a rectilinear grid

Resampling a high resolution dataset onto a rectilinear grid is the most common use case for the Resample operator. When a Resample operator is applied to a plot, the Resample operator clips out any data values that are not within the operator's bounding box. For the data that remains inside the bounding box, the operator samples it using the user-specified numbers of samples for the X, Y, and Z dimensions. The default for the Resample operator is to use the entire extents of the dataset. If you want to choose a smaller region, unselect the **Resample Entire Extents** checkbox and enter new bounding box information. The bounding box is specified by entering new start and end values for each dimension. For example, if you want to change the locations sampled in the X dimension then you could type new floating point values into the **Start X** and **End X** text fields. The same pattern applies to changing the locations sampled in the Y and Z dimensions. One difference between resampling 2D and 3D datasets is that 3D datasets must have the **3D resampling** check box enabled to ensure that VisIt uses the user-specified Z-extents and number of samples in Z.

Samples for which there was no data in the original input dataset are provided with a default value that you can change by typing a new floating point number into the **Value for uncovered regions** text field.

Using Resample with CSG meshes

Constructive Solid Geometry (CSG) modeling is a method whereby complex models are built by adding and subtracting primitive objects such as spheres, cubes, cones, etc. When you plot a CSG mesh in VisIt, VisIt resamples the CSG mesh into discrete cells that can be processed as an unstructured mesh and plotted. The Resample operator can be used to tell VisIt the granularity at which the CSG mesh should be sampled, overriding the CSG mesh's default sampling. Naturally, higher numbers of samples in the Resample operator produce a more faithful representation of the original CSG mesh. Figure 4.144 depicts a CSG model that contains a disc within a smooth ring. Note that as the number of samples in the Resample operator increases, the model becomes smoother and jagged edges start to disappear.

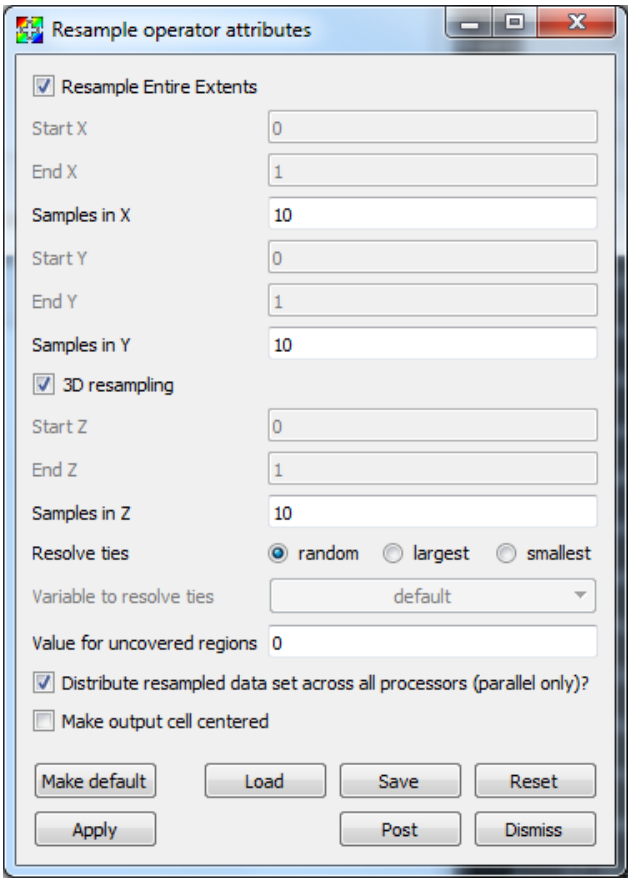


Fig. 4.143: Resample attributes window

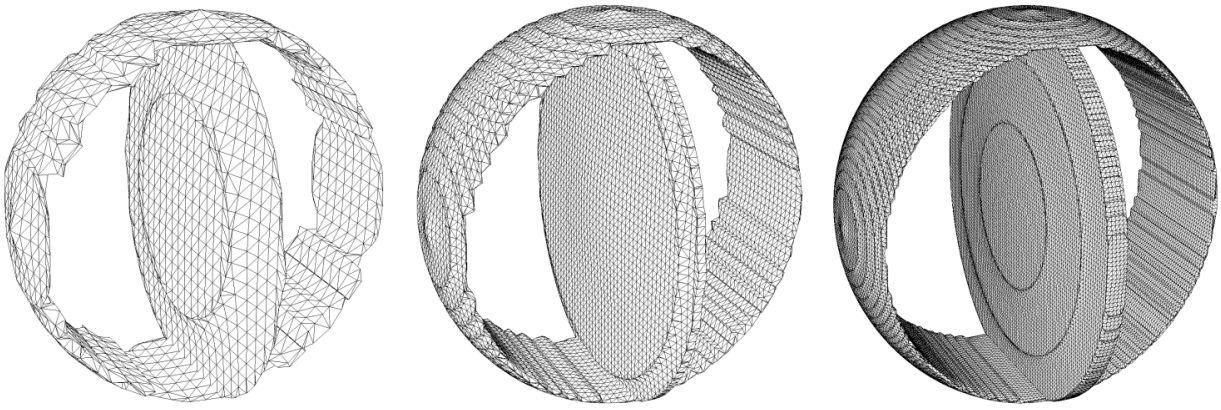


Fig. 4.144: The Resample operator can be used to control the resolution of CSG meshes. Resolution is increased from left to right.

Resampling surfaces projected to 2D

Sometimes is is useful to project complex surfaces into 2D and resample them onto a 2D mesh so queries and other analysis can be performed.

When you project a complex surface to 2D using the Project operator, all of a plot's geometry remains and its Z coordinates are set to zero. This results in some areas where the plot is essentially crushed on top of itself, as shown in Figure 4.145. When resampling the plot onto a new 2D grid, these overlapping areas can be treated in three different ways. You can ensure that the top value is taken if you choose the random option by clicking on the **random** button in the **Resolve ties** button group. You can use a mask variable to decide ties by clicking on the **largest** or **smallest** buttons and by selecting an appropriate variable using the **Variable to resolve ties** menu.

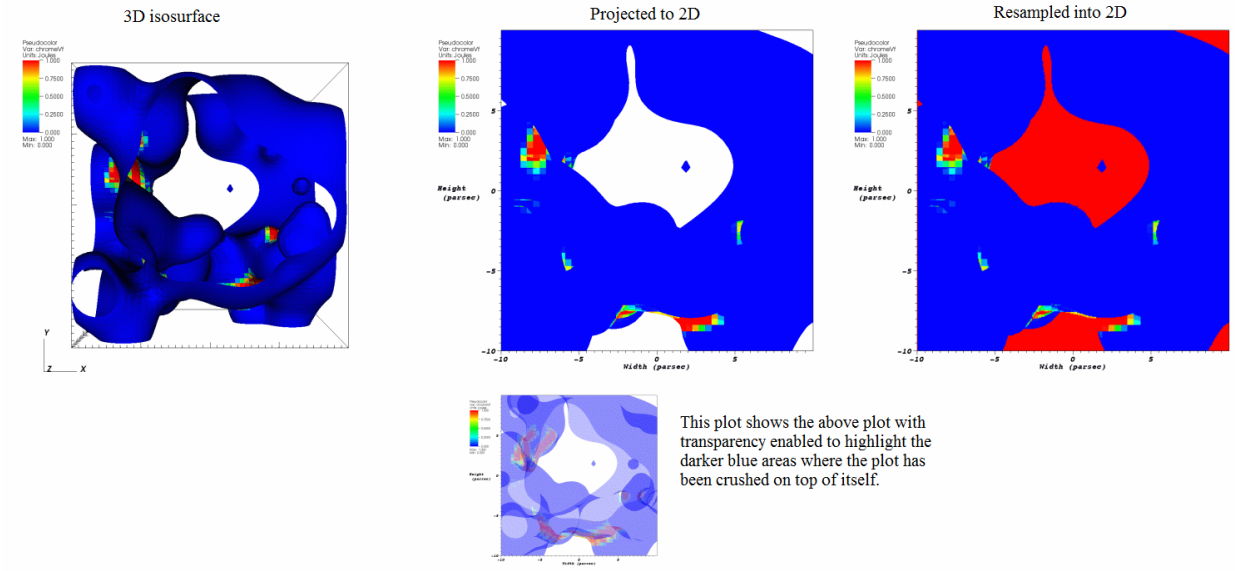


Fig. 4.145: Using the Resample operator to create a 2D projection

When used in parallel, the resampled data is distributed across all processors. This can be changed by unselecting the checkbox.

You can also force the output data to be cell centered by selecting the **Make output cell centered** checkbox.

Revolve operator

The Revolve operator is for creating 3D geometry from 2D geometry by revolving the 2D about an axis. The Revolve operator is useful for incorporating 2D simulation data into a visualization along with existing 3D data. An example of the Revolve operator is shown in Figure 4.146.

Using the Revolve operator

To use the Revolve operator, the first thing to do is pick an axis of revolution. The axis of revolution is specified as a 3D vector in the **Axis of revolution** text field (see Figure 4.147) and serves as the axis about which your 2D geometry is revolved. If you want to revolve 2D geometry into 3D geometry without any holes in the middle, be sure to pick an axis of revolution that is incident with an edge of your 2D geometry. If you want 3D geometry where the initial 2D faces do not meet, be sure to specify start and stop angles in degrees in the **Start angle** and **Stop angle** text fields.

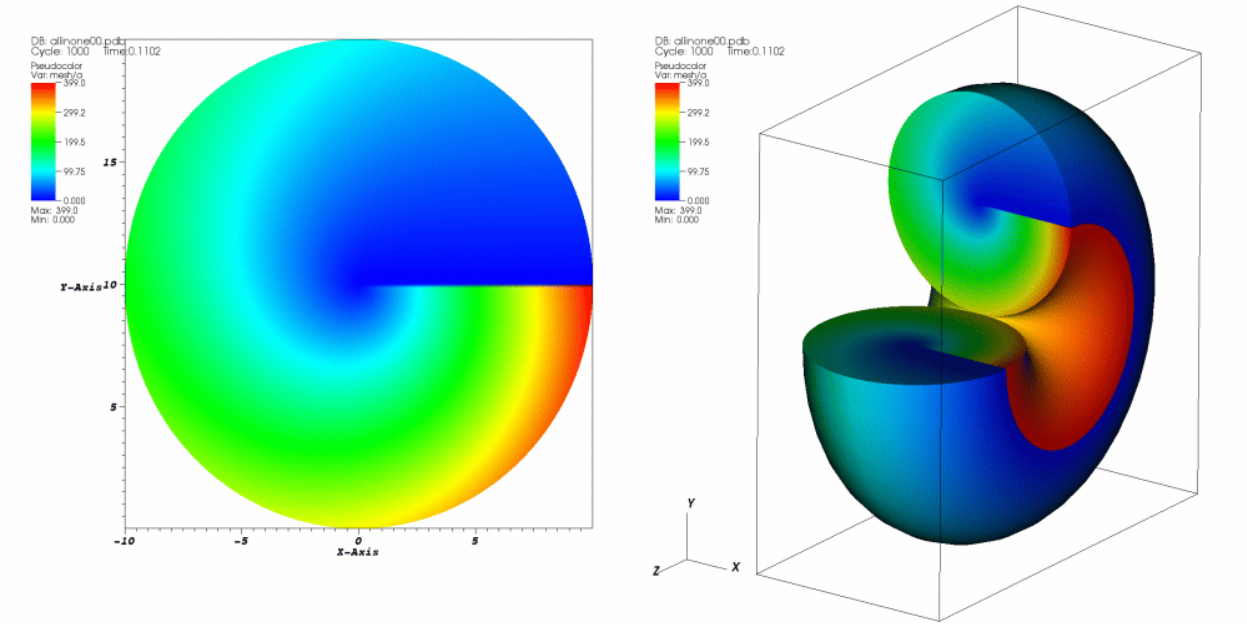


Fig. 4.146: Revolve operator example

Finally, the number of steps determines how many times the initial 2D geometry is revolved along the way from the start angle to the stop angle. You can specify the number of steps by entering a new value into the **Number of steps** text field.

By default, VisIt will choose the axis of revolution based on mesh type, which is also determined automatically. You can specify the mesh type manually by selecting a radio button other than **Auto**. To specify the axis of revolution manually, uncheck the **Choose axis based on mesh type** checkbox.

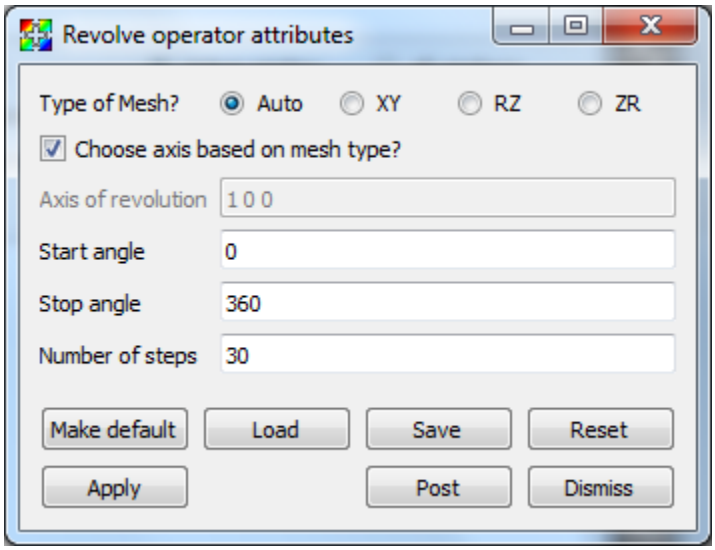


Fig. 4.147: Revolve attributes window

Slice operator

This operator slices a 3D database with a plane that can have an arbitrary orientation. Plots to which the Slice operator has been applied are turned into 2D planar surfaces that are coplanar with the slice plane. The resulting plot can be left as a 2D slice in 3D space or it can be projected to 2D space where other operations can be done to it. A Pseudocolor plot to which a Slice operator has been applied is shown in Figure 4.148.

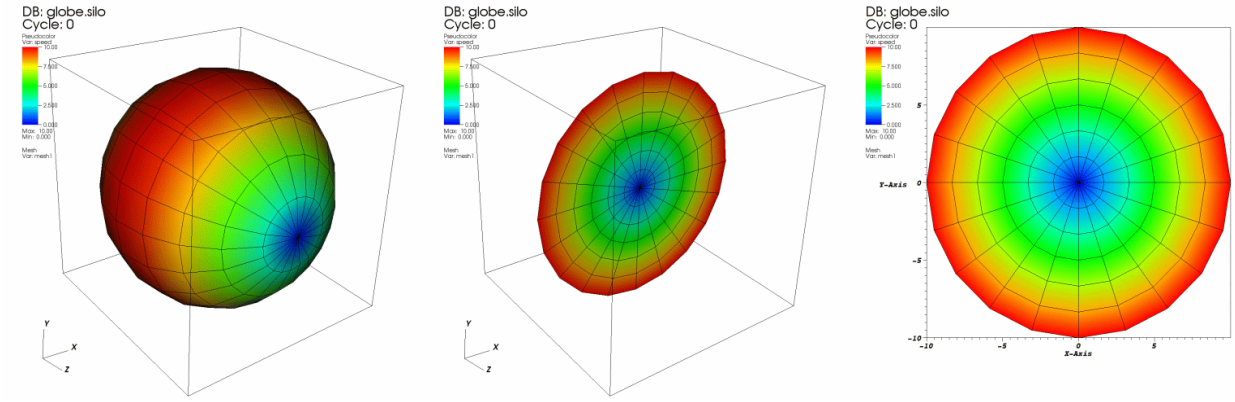


Fig. 4.148: Slice operator example

Positioning the slice plane

You can position the slice plane by setting the origin, normal, and up-axis vectors in the **Slice operator attributes window,** shown in Figure 4.149 . The slice plane is specified using the origin-normal form of a plane where all that is needed to specify the plane are two vectors; the origin and the normal. The origin of the plane is a point in the slice plane. The normal vector is a vector that is perpendicular to the slice plane.

VisIt allows the slice plane normal to be aligned to a specific axis or it can be set to any arbitrary vector. If you want the slice plane to be along any of the three axes, click the **X-Axis**, **Y-Axis**, or **Z-Axis** radio button. If you want to make a slice plane that does not align with the principle axes, click the **Arbitrary** or **Theta-Phi** radio button and then type a direction vector into the text field to the right of the radio button. The vector need not be normalized since VisIt will normalize the vector before using it.

The slice plane's origin, which specifies the location of the slice plane, can be set five different ways. The middle of the **Slice attributes window**, or **Origin area** (see the Figures below), provides the necessary controls required to set the slice plane origin. The **Origin area** provides five radio buttons: **Point**, **Intercept**, **Percent**, **Zone**, and **Node**. Clicking on one of these radio buttons causes the **Origin area** to display the appropriate controls for setting the slice plane origin. To set the slice plane origin to a specific point, click the **Point** radio button in the **Origin area** and then type a new 3D point into the **Point** text field. To set the slice plane origin to a specific value along the principle slice axis (usually an orthogonal slice), click the **Intercept** radio button and then type a new value into the **Intercept** text field.

If you don't know a good value to use for the intercept, consider using the percent slice mode. Percent slice mode, which is most often used for an orthogonal slice, allows you to slice along a particular axis using some percentage of the distance along that axis. For example, this allows you to see what the slice plane looks like if its origin is 50% of the distance along the X-Axis. To set the origin using a percentage of the distance along an axis, click the **Percent** radio button and then type a new percentage value into the **Percent** text field or use the **Percent** slider.

Sometimes it is useful to slice through a particular zone or node. The Slice operator allows you to pick an origin for the slice plane so a specific zone or node lies in the slice plane. To make sure that a particular zone is sliced by the Slice operator, click on the **Zone** radio button and then enter the zone to be sliced into the **Zone** text field. Be sure to also enter the domain that contains the zone into the **Domain** text field if you are slicing a multi-domain database. If you

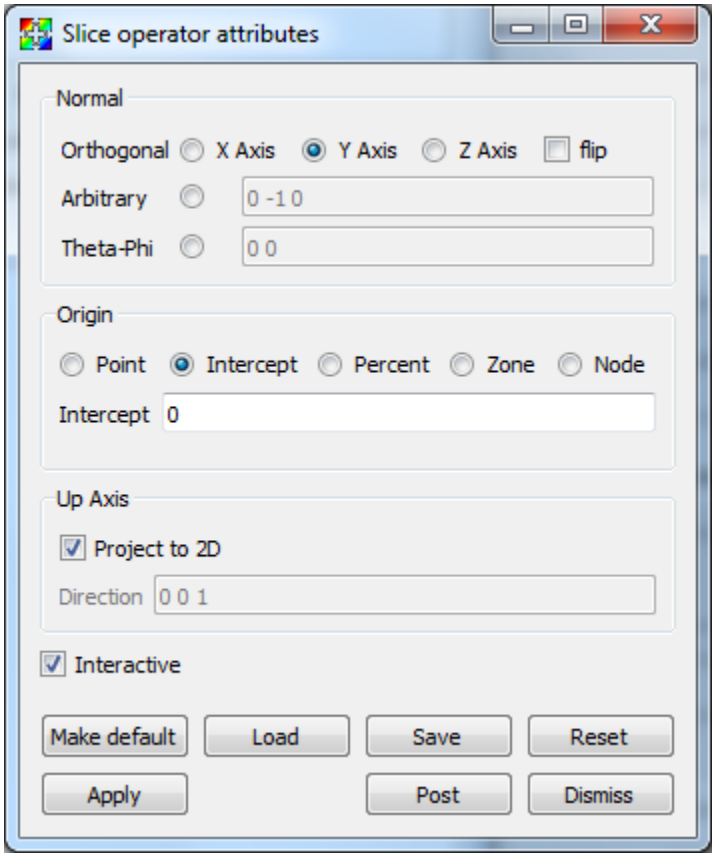


Fig. 4.149: Slice attributes window

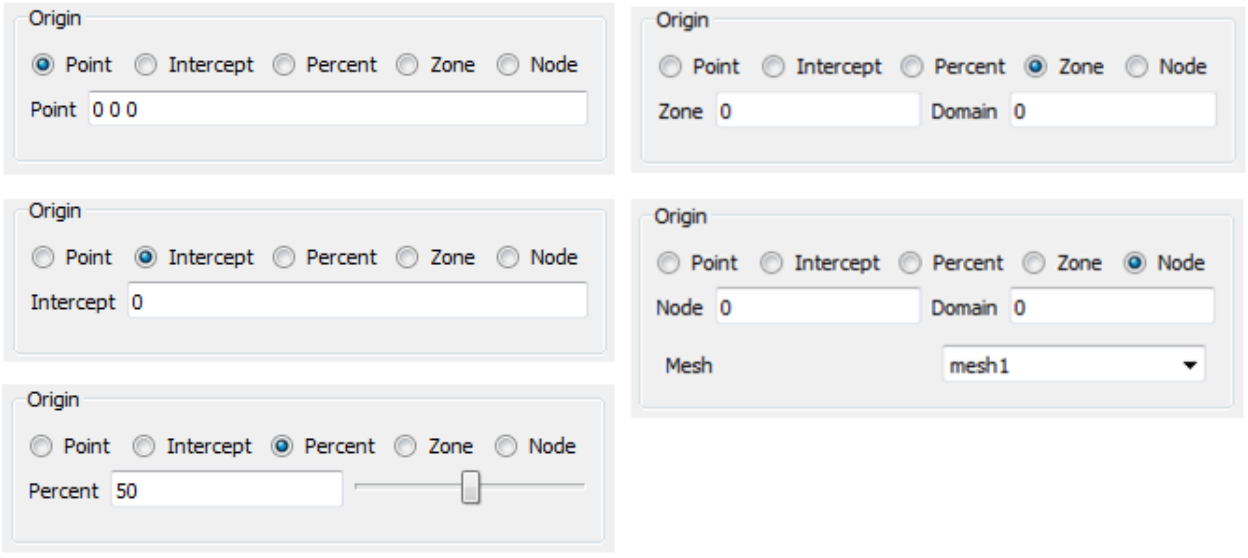


Fig. 4.150: Origin area appearance

want to make sure that the slice plane's origin is at a specific node in a mesh, click the **Node** radio button and enter a new node number into the **Node** text field. Note that you must also specify a domain if you are slicing a multi-domain database. If the database contains multiple meshes, their will also be **Mesh** combo box option from which to choose the mesh to use, as seen in the **Node** example in Figure 4.150.

Use the up-axis vector when you want the slice plane to be projected to 2D. The up-axis vector is a vector that lies in the slice plane and defines a 2D coordinate system within the plane where the up-axis vector corresponds to the Y-axis. To change the up-axis vector, type a new 3D vector into the **Direction** text field in the **Up Axis** area of the window.

Positioning the slice plane using the Plane Tool

You can also position the slice plane using VisIt's interactive plane tool. The plane tool, which is available in the visualization window's popup menu, allows you to position a slice plane interactively using the mouse. The plane tool is an object in the visualization window that can be moved and rotated. When the plane tool is changed, it gives its new slice plane to the Slice operator if the operator is set to accept information interactively. To make sure that the Slice operator can accept a new slice plane from the plane tool, check the **Interactive** check box in the **Slice attributes window**. For more information about the plane tool, read the *Interactive Tools* chapter.

Projecting the slice to 2D

The Slice operator usually leaves sliced plots in 3D so you can position the slice with the plane tool. However, you might want the plot projected to 2D. When a sliced plot is projected to 2D, any 2D operation, like **Lineout** , can be applied to the plot. To project a plot to 2D, check the **Project 2D** check box in the **Slice attributes window** .

Smooth operator

The Smooth Operator applies Laplacian Smoothing to a mesh, making the cells better shaped and the nodes more evenly distributed, which results in smoother edges and peaks throughout the mesh. Each node is moved towards a coordinate that is the average of its connected nodes. The **relaxation factor** determines how far along that path to move the node. A sweep over all nodes is a single iteration.

Using the Smooth operator

The Smooth operator has a number of controls that can be used to tune mesh smoothness.

Maximum number of iterations Controls the maximum number of times the mesh relaxation algorithm is applied to the input mesh. Larger numbers of iterations will produce smoother meshes but will also take more time to compute. Values must be integers.

Relaxation Factor Controls how much the mesh is relaxed. Values near 1 produce a mesh that is very smooth relative to the input mesh. Values must be floating point numbers between 0 and 1.

Convergence Limit the maximum point motion. The smoothing process will terminate if the maximum point motion during an iteration is less than this value. Smaller numbers result in more iterations, and if 0 is supplied then the Smooth operator will perform **Maximum number of iterations**. Values must be floating point numbers between 0 and 1.

Maintain Features When the surface angle between two zones sharing an edge is greater than **Feature Angle**, then the edge is classified as a *feature edge*. The nodes in the mesh are then classified as *simple* (not used by a feature edge), *interior edge* (used by exactly two feature edges), or *fixed* (all other nodes). *Simple* nodes are smoothed as before, *fixed* nodes are not smoothed at all, and *interior edge* nodes are only smoothed along the two connected *feature edges*, and only if the angle between the edges is less than **Max Angle Edge**. This distinction allows the smoothing operation to preserve sharp peaks in the mesh while still smoothing out most of the mesh.

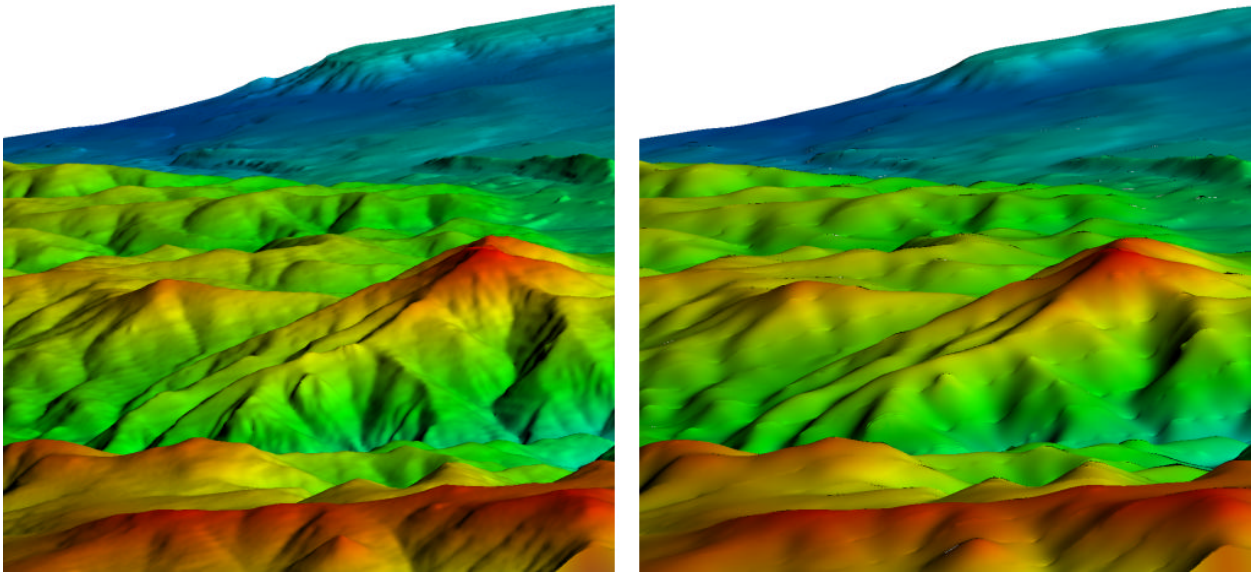


Fig. 4.151: Smooth operator example

Feature Angle Used to determine the number of feature edges for each node. Values must be floating point numbers between 0 and 90.

Max Angle Edge Used to determine if *interior edge* nodes will be smoothed. Values must be floating point numbers between 0 and 90.

Smooth Along Boundaries Enable the smoothing operation on nodes that are along the boundaries of the mesh.

SphereSlice operator

The SphereSlice operator slices a 2D or 3D database with an arbitrary sphere. Plots to which the SphereSlice operator have been applied become 2D surfaces that are coincident with the surface of the slicing sphere. The resulting plots remain in 3D space. You can use the SphereSlice operator to slice objects to judge their deviation from being perfectly spherical. An example of the SphereSlice operator is shown in Figure 4.153.

Positioning and resizing the slice sphere

You can position the slice sphere by setting its origin in the **SphereSlice attributes window** shown in Figure 4.154 . The slice sphere is specified by a center point and a radius. To change the slice sphere's center, enter a new point into the **Origin** text field. The origin is a 3D coordinate that is represented by three space-separated floating point numbers. To resize the sphere, enter a new radius number into the **Radius** text field.

Positioning the slice sphere using the Sphere tool

You can also position the slice sphere using VisIt's interactive sphere tool. The sphere tool, available in the visualization window's popup menu, allows you to position and resize a slice sphere interactively using the mouse. The sphere tool is an object in the visualization window that can be moved and resized. When the sphere tool is changed, it gives its new slice sphere to the SphereSlice operator. For more information about the sphere tool, read the *Interactive Tools* chapter.

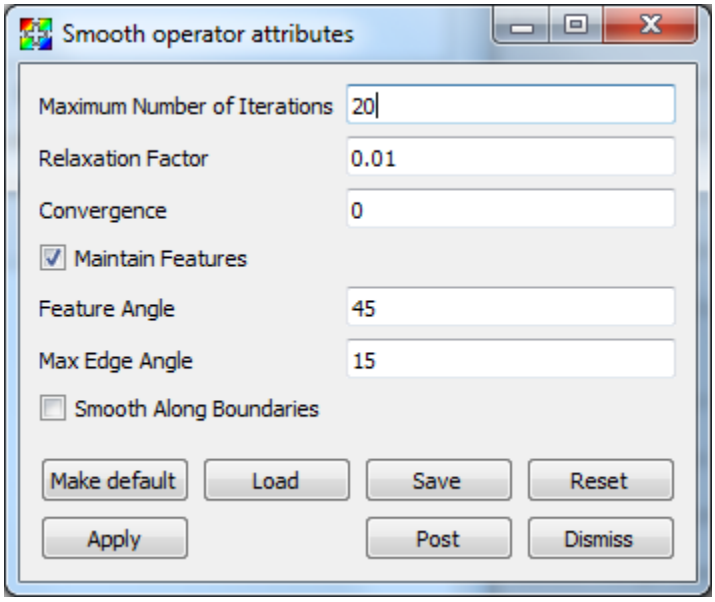


Fig. 4.152: Smooth attributes

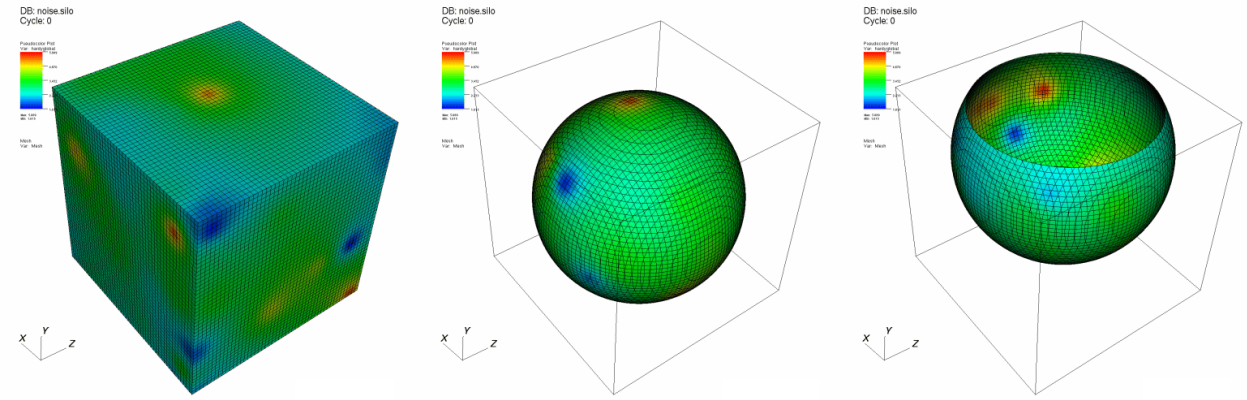


Fig. 4.153: SphereSlice operator example

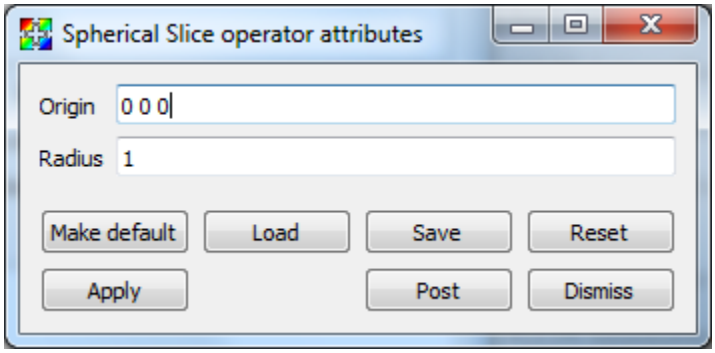


Fig. 4.154: SphereSlice attributes window

Tessellate operator

The Tessellate operator is an operator that tessellates high order elements so that they appear curved.

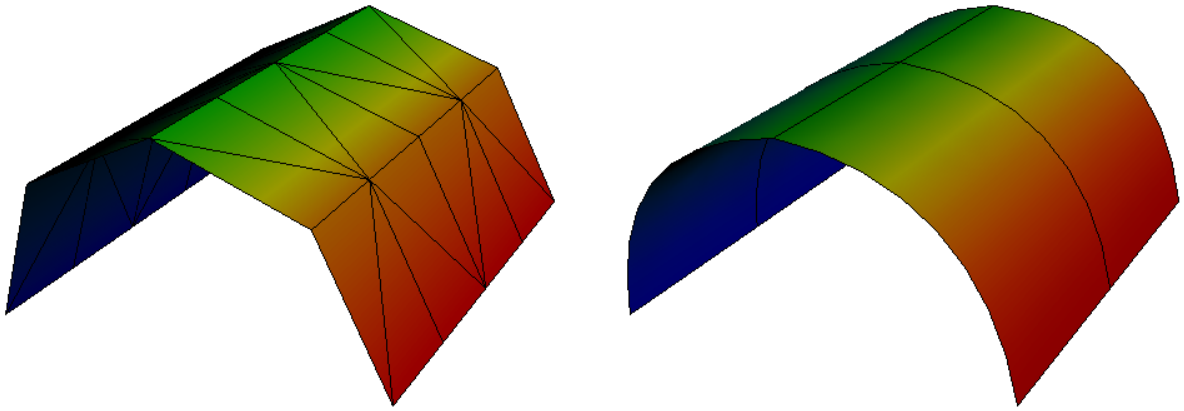


Fig. 4.155: Tessellate operator example

The Tessellate operator supports the following high order element types.

- QUADRATIC_EDGE
- CUBIC_LINE
- LAGRANGE_TRIANGLE
- QUADRATIC_TRIANGLE
- BIQUADRATIC_TRIANGLE
- LAGRANGE_QUADRILATERAL
- BIQUADRATIC_QUAD
- QUADRATIC_QUAD
- LAGRANGE_TETRAHEDRON
- QUATRADIC_TETRA
- LAGRANGE_HEXAHEDRON
- QUADRATIC_HEXAHEDRON

If the Tessellate operator encounters an unsupported element type it will remove the element from the mesh.

Changing the tessellation accuracy

The tessalation accuracy is controlled by the **Chord error** and **Field criterion**. The **Chord error** is with respect to the curvature of the element and is ratio of a chord to the distance from the curve and is independent of the scale of the object. The default **Chord error** is 0.035, which will typically do a good job. The **Field criterion** is with respect

to the error in the field within the element. The default **Field criterion** is also 0.035, which will also typically do a good job. Reducing the **Chord error** and **Field criterion** will both improve the discretization. They should only be decreased if necessary, since reducing them will increase the number of elements a single high order element is tessellated into. This in turn increases the memory usage and the time to perform operations. The number of elements a single high order element gets tessellated into may easily get into the hundreds.

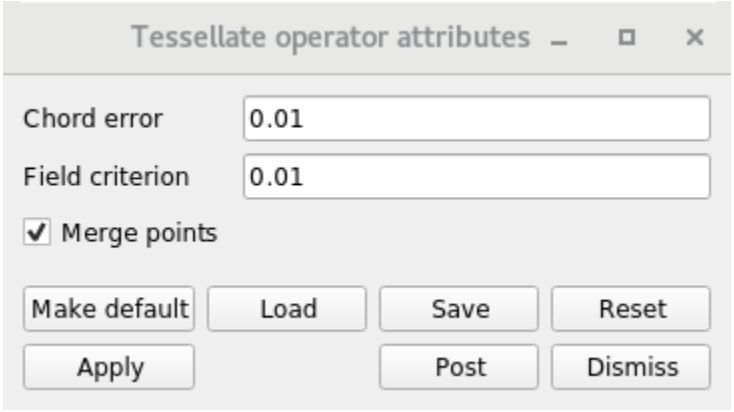


Fig. 4.156: Tessellate attributes window

Merging the points

The points from the cells generated by the tessellation can either be shared or not shared by cells. The default **Merge points** setting will merge the points. Point merging typically only affects the appearance of the Mesh plot. When points are merged, the mesh lines of individual cells of the tessellation will be visible. When points are not merged, the mesh lines of the high order element will typically only be visible.

ThreeSlice operator

The ThreeSlice operator slices 3D databases using three axis-aligned slice planes and leaves the resulting planes in 3D where they can all be viewed at the same time. The ThreeSlice operator is meant primarily for quick visual exploration of 3D data where the internal features cannot be readily observed from the outside.

Moving the ThreeSlice operator

The ThreeSlice operator is controlled by moving its origin, which is the 3D point where all axis-aligned slice planes intersect. There are two ways to move the ThreeSlice operator's origin. First, you can directly set the point that you want to use for the origin by entering new x, y, z values into the respective **X** , **Y** , **Z** text fields in the **ThreeSlice operator attributes window** , shown in Figure 4.158. You can also make sure that the **Interactive** toggle is turned on so you can use VisIt's interactive Point tool to set the ThreeSlice operator's origin. When you use the Point tool to set the origin for the ThreeSlice operator, the act of moving the Point tool sets the ThreeSlice operator's origin and causes plots that use the ThreeSlice operator to be recalculated with the new origin. For more information about the point tool, read the *Interactive Tools* chapter.

Threshold operator

The Threshold operator extracts cells from 2D and 3D databases where the plotted variable falls into a specified range. The resulting database can be used in other VisIt plots. You might use this operator when searching for cells with

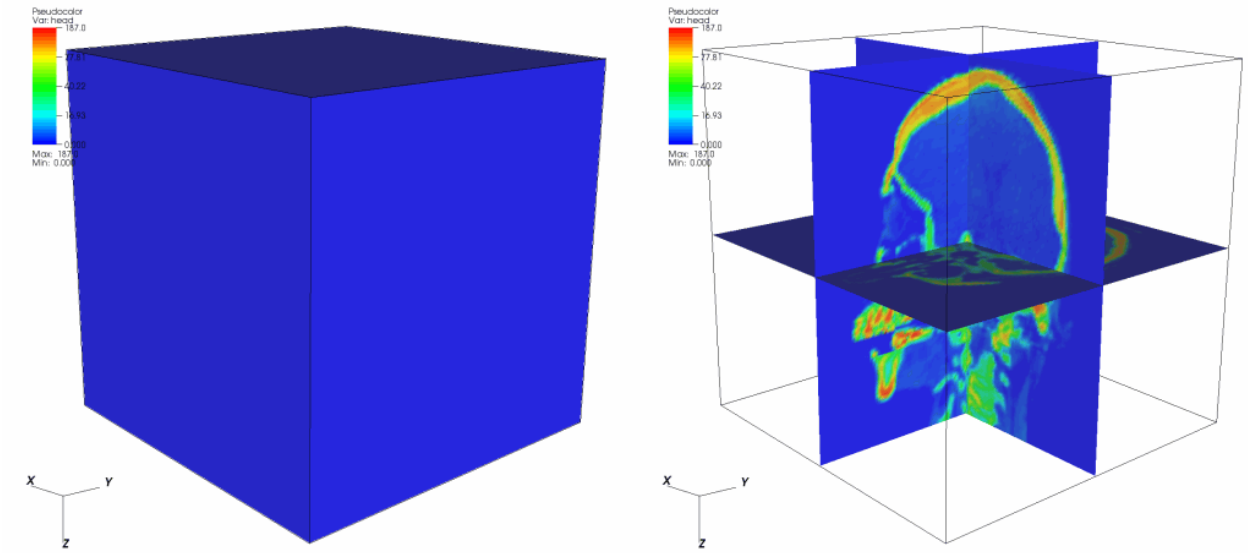


Fig. 4.157: ThreeSlice operator example

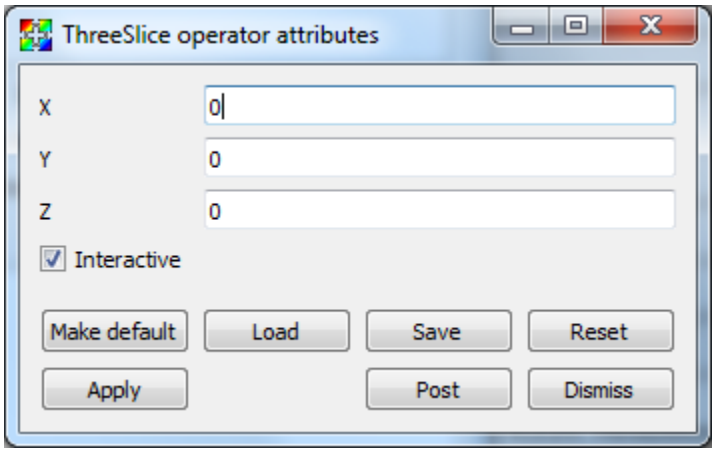


Fig. 4.158: ThreeSlice attributes window

certain values. One such example is searching for the cell with the minimum or maximum value for the plotted variable. The Threshold operator removes all cells that do not have values in the specified range, making it easy to spot cells with the desired values. The Threshold operator can also use variables other than the plotted variable, for instance, you might want to see a Pseudocolor plot of pressure while using the Threshold operator to remove all cells below a certain density. By specifying a different threshold variable, it is possible to visualize different quantities over the subset of cells specified by the threshold variable and range. An example of the Threshold operator is shown in Figure 4.159.

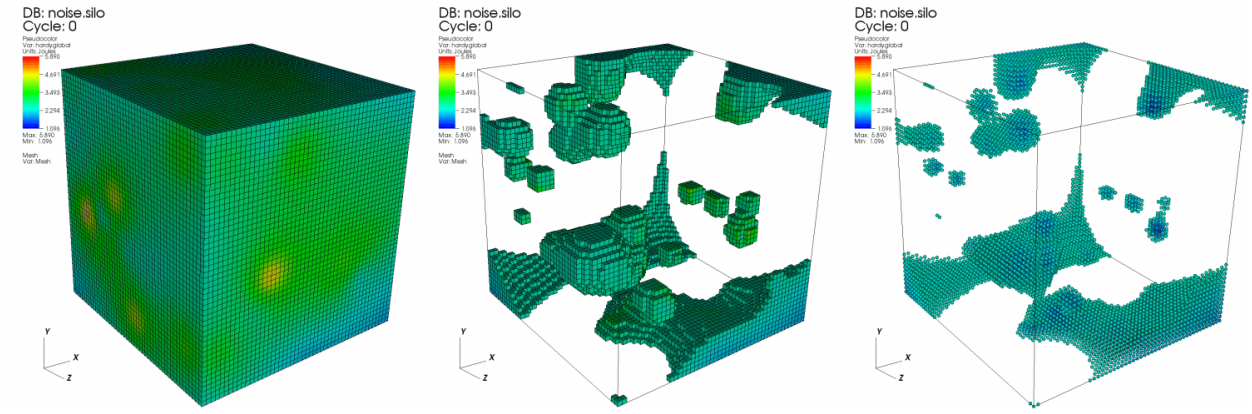


Fig. 4.159: Threshold operator example

Setting the variable range

The Threshold operator uses a range of values to determine which cells from the database should be kept in the visualization. For the **Default** bounds input, you specify the range of values by lower and upper bounds on the threshold variable. Cells with values below the lower bound or with values above the upper bound are removed from the visualization. To specify a new lower bound, type a new number or the special keyword: min into the **Threshold attributes window's** (Figure 4.160) **Lower bound** text field. To specify a new upper bound, type a new number or the special keyword: max into the **Upper bound** text field.

For the **Custom** bounds input, you can specify a list of ranges in the **Range** text field. A colon - ':' defines a range and a comma - ',' defines a logical OR. The range shown in Figure 4.161 has the following meaning:

```
1 <= default <= 10 OR default = 17 OR 23 <= default <= max
```

Numbers, commas, and colons are the only valid symbols that can be used in specifying a range list.

When the threshold variable is a nodal quantity, the cell being considered by the Threshold operator has values at each node in the cell. In this case, the Threshold operator provides a control that determines whether or not to keep the cell if some nodes have values in the threshold range or if all nodes have values in the threshold range. More cells are usually removed from the visualization when all nodes must be in the threshold range. Select **Part in range** from the **Show zone if** combo box to allow cells where at least one value is in the threshold range into the visualization. Select **All in range** from the **Show zone if** combo box to require that all nodal values exist in the threshold range.

Setting the threshold variable

The Threshold operator uses the threshold variable to determine whether cells remain in the visualization. The threshold variable is usually the plotted variable in which case the **Variable** column displays: default. To specify a threshold variable other than the plotted variable, click on the **Add variable** variable button and select a new scalar variable from the list of available variables.

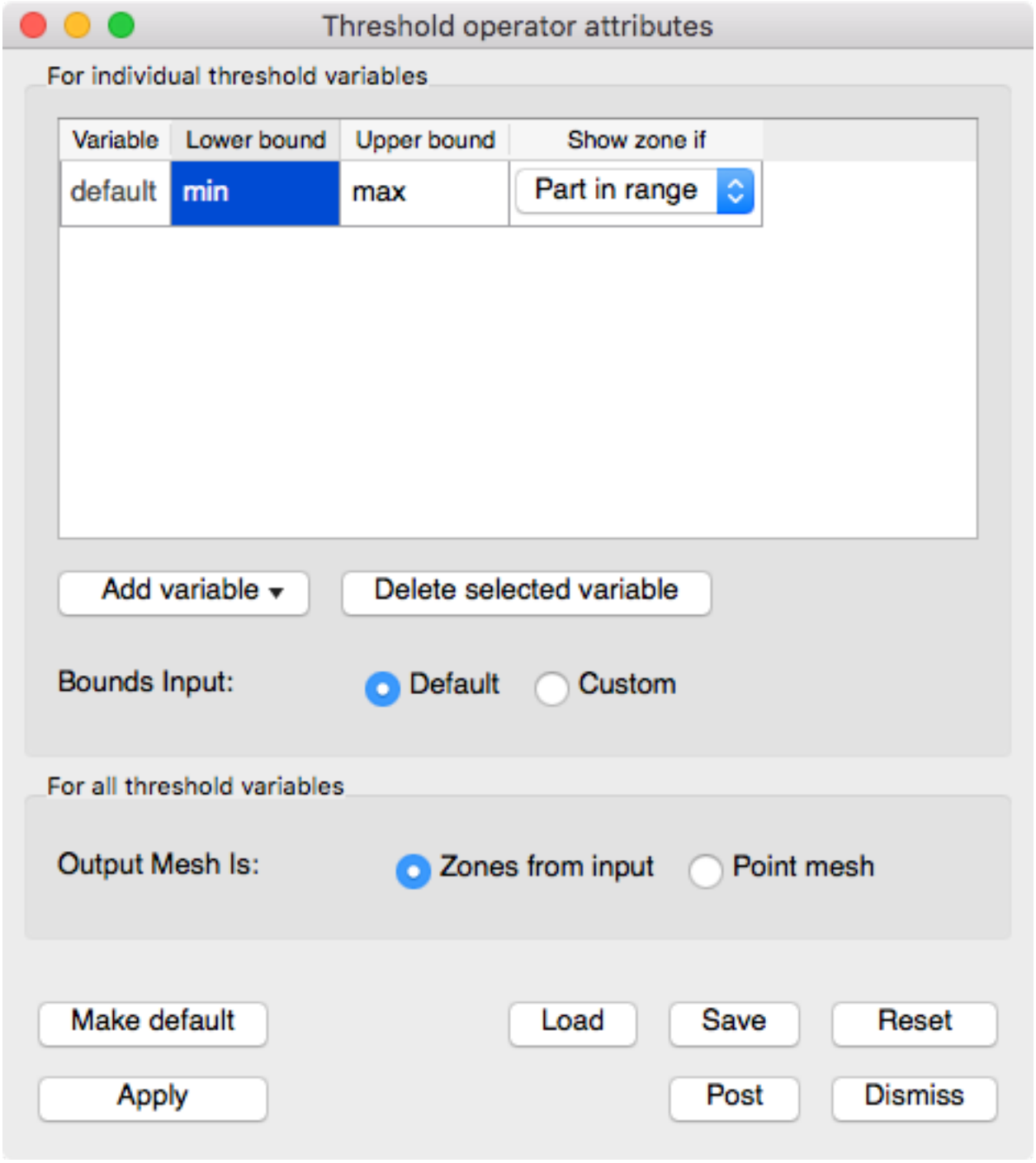


Fig. 4.160: Threshold attributes window - Default

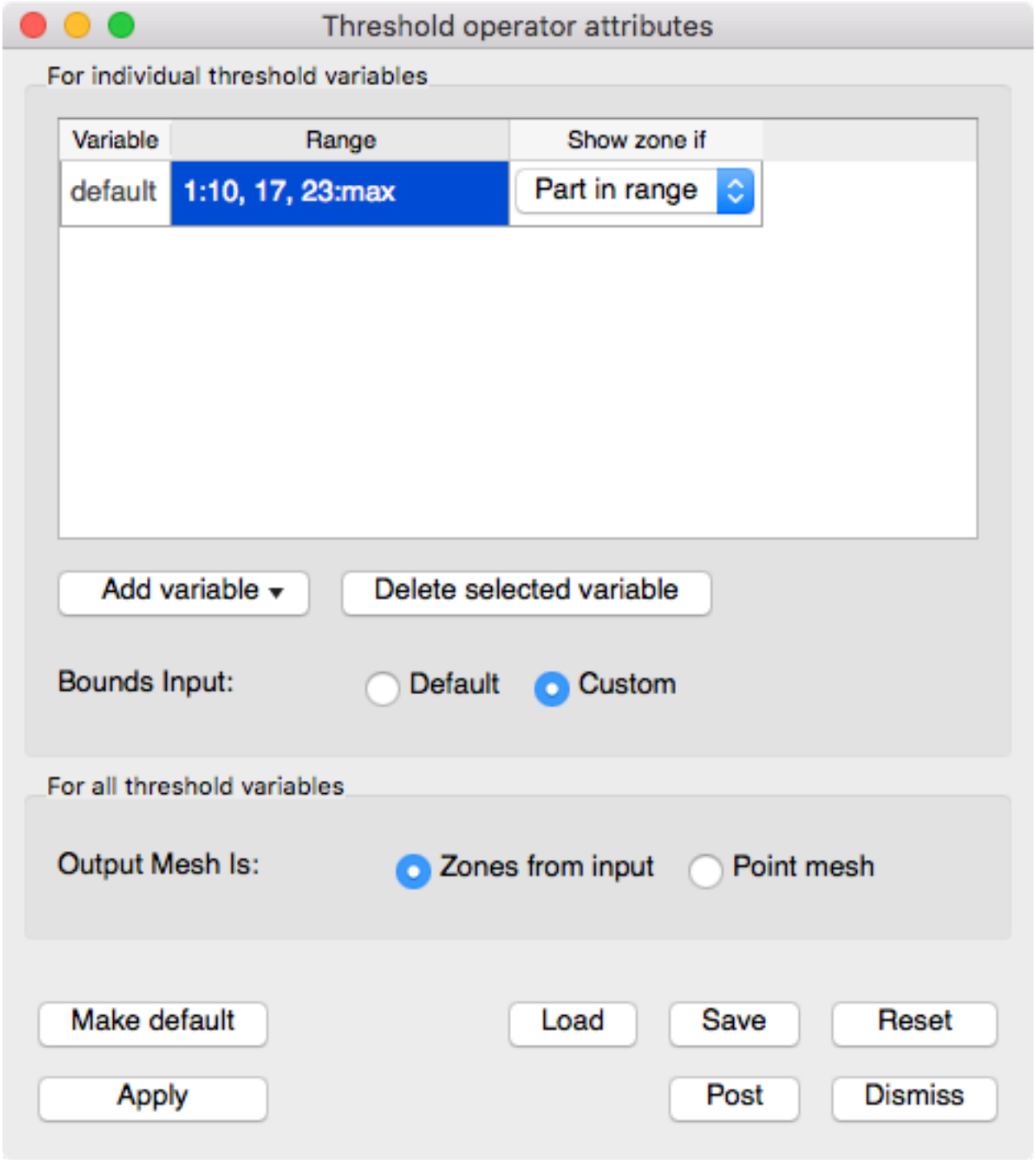


Fig. 4.161: Threshold attributes window - Custom

You might set the threshold variable when you apply the Threshold operator to plots which do not take scalar variables as input. An example of this is the Mesh plot. When you apply the Threshold operator to a Mesh plot, you must set the threshold variable to a valid scalar variable for cells to be removed from the plot. You can also use the threshold variable to remove cells based on one variable while viewing the plotted variable.

Setting the output mesh type

The Threshold operator removes all cells that do not meet the threshold criterion, leaving behind a set of cells that are gathered into an unstructured mesh. Sometimes, it can be useful to transform the remaining cells into a point mesh. You can specify the desired output mesh type using the **Cells from input** and **Point mesh** radio buttons in the **Threshold attributes window** .

Transform operator

The Transform operator manipulates a 2D or 3D database's coordinate field by applying rotation, scaling, and translation transformations. The operator's transformations are applied in the following order: rotation, scaling, translation. The Transform operator is applied to databases before they are plotted. You might use the Transform operator to rotate database geometry to a more convenient orientation or to scale database geometry to make better use of the visualization window. You can also use the Transform operator to make objects rotate and move around the visualization window during animations. This works well when only one part of the visualization should move while other parts and the view remain fixed. An example of the Transform operator is shown in Figure 4.162.

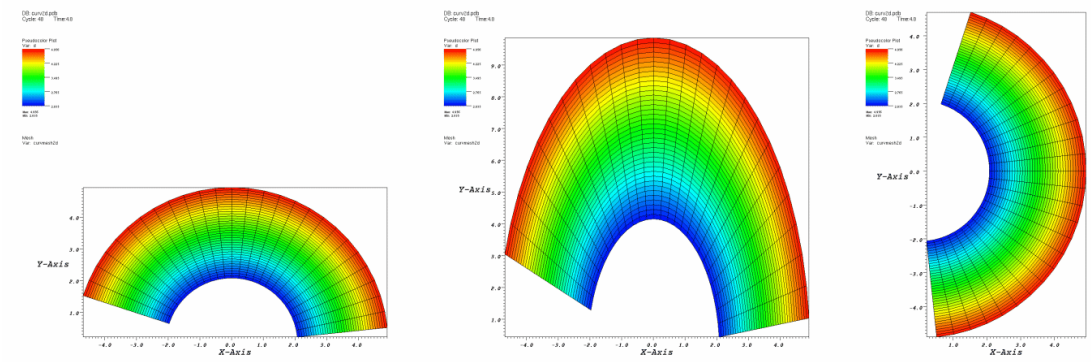


Fig. 4.162: Transform operator example

Rotation

You can use the Transform operator to rotate plots around an arbitrary axis in 3D and around the Z-axis in 2D. To apply the rotation component of the Transform operator, be sure to check the **Rotate** check box in the **Transform attributes window** (Figure 4.163). An origin and normal are needed to specify the axis of rotation. The origin serves as a reference point for the object being rotated. The axis of rotation is a 3D vector that, along with the origin, determines the 3D axis that will serve as the axis of rotation. You must supply an origin and an axis vector to specify an axis of rotation. To change the origin, type a new 3D vector into the top **Origin** text field. To change the 3D axis, type a new 3D vector into the **Axis** text field. Both the origin and the axis are represented by three space-separated floating point numbers.

When applying the Transform operator to plots, you probably want to make the origin the same as the center of the plot extents which can be found by looking at the axis annotations. When the Transform operator is applied to 3D

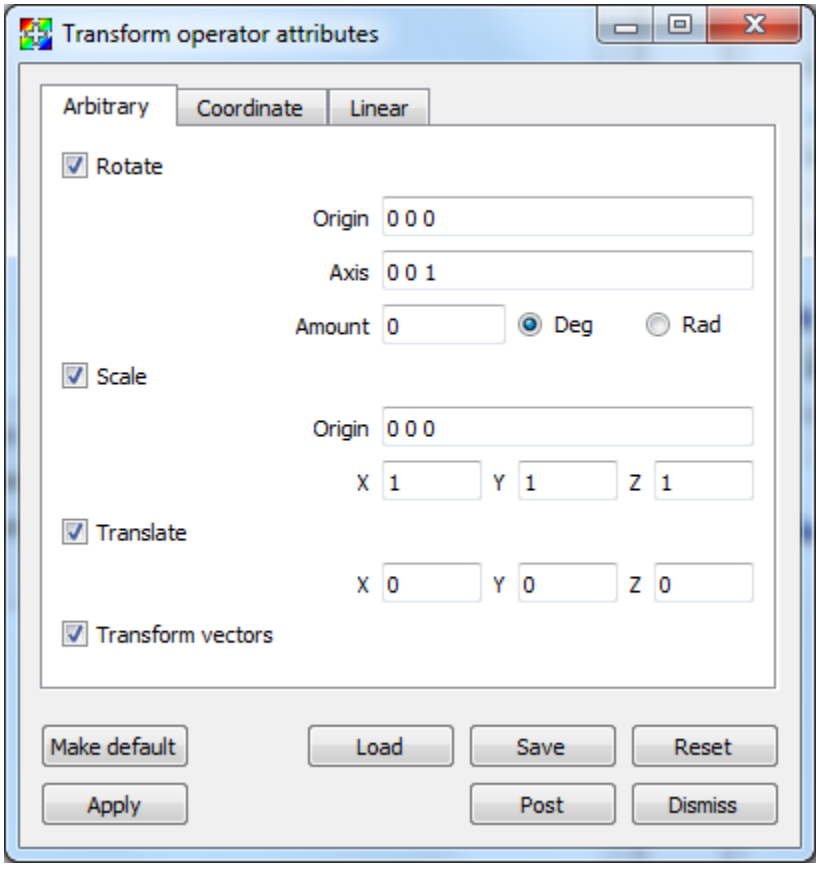


Fig. 4.163: Transform attributes window

plots, the axis of rotation can be set to any unit vector. When the Transform operator is applied to 2D plots, the axis of rotation should always be set to the Z-axis (0 0 1).

Once you specify the axis of rotation, you must supply the angle of rotation. The default angle of rotation is zero degrees, which gives no rotation. To change the angle of rotation, enter a number in degrees or radians into the **Amount** text field and click the **Deg** radio button for degrees or the **Rad** radio button if the angle is measured in radians.

Scale

You can use the Transform operator to scale plots. Each dimension can be scaled independently by entering a new scale factor into the **X, Y, Z** text fields. Each scale factor is a multiplier so that a value of 1 scales plots to their original size while a value of 2 scales plots to twice their original size. To apply the scale component of the Transform operator, be sure to check the **Scale** check box in the **Transform attributes window** . All dimensions are scaled relative to a scaling origin which can be changed by typing a new origin into the middle lower **Origin** text field.

Translation

You can use the Transform operator to translate plots. To apply the translation component of the Transform operator, be sure to check the **Translate** check box in the **Transform attributes window** . To translate plots in the X dimension, replace the default value of zero in the **X** translation text field. Translations in the Y and Z dimensions are handled in the same manner.

Coordinate system conversion

In addition to being able to rotate, scale, and translate plots, the Transform operator can also perform coordinate system conversions. A plot's coordinates can be specified in terms of Cartesian, Cylindrical, or Spherical coordinates (illustrated in Figure 4.164). Ultimately, when a plot is rendered in the visualization window, its coordinates must be specified in terms of Cartesian coordinates due to the implementation of graphics hardware. If you have a database where the coordinates are not specified in terms of Cartesian coordinates, you can apply the Transform operator to perform a coordinate system transformation so the plot is rendered correctly in the visualization window.

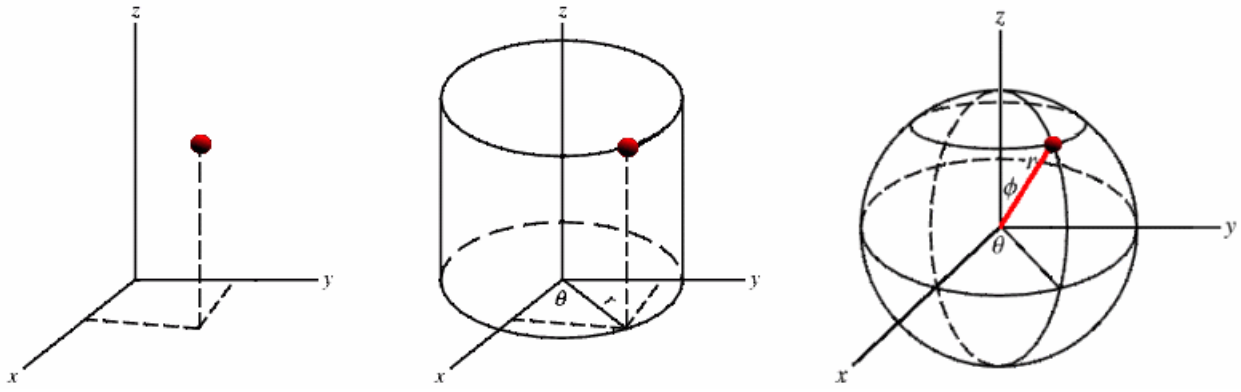


Fig. 4.164: Cartesian, Cylindrical, Spherical coordinate systems

Figure 4.165 shows a model of an airplane that is specified in terms of spherical coordinates. When it is rendered initially, VisIt assumes that the coordinates are Cartesian, which leads to the plot getting stretched and tangled. The Transform operator was then applied to convert the plot's spherical coordinates into Cartesian coordinates, which allows VisIt to draw the plot as it is intended to look.

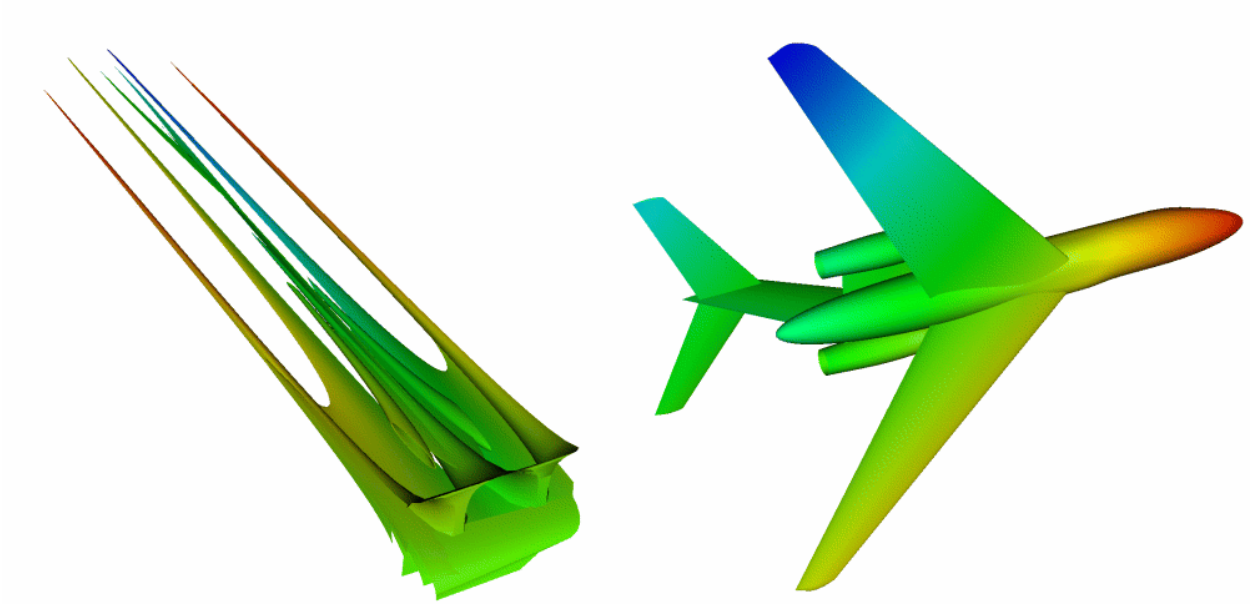


Fig. 4.165: Coordinate system conversion using the Transform operator

The Transform operator allows coordinate system transformations between any of the three supported coordinate systems, shown in Figure 4.166 . To pick a coordinate system transformation, you must first pick the coordinate system used for the input geometry. Next, you must pick the desired output coordinate system. In the example shown in Figure 4.165, the input coordinate system was Spherical and the output coordinate system was Cartesian. Note that if you use the Transform operator to perform a coordinate system transformation then you cannot also perform rotation, scaling, or translation. If you must perform any of those operations, add a second Transform operator to your plots.

Linear (Affine) transforms

Linear, or Affine, transforms can be specified via a 4x4 matrix as shown in Figure 4.167. This represents a class of operations often used for coordinate system transformations known as spatial transformation matrices. Why is the matrix 4x4 instead of 3x3? Typically, the 4th row of the matrix is left unchanged but the 4th column permits the inclusion of *translation* in the same linear matrix operator used to support other things like rotation and scaling.

Vectors will be transformed by default, uncheck the **transform vectors** checkbox if this is not desired. The inverse transform can be applied by selecting **Invert linear transform**.

Tube operator

The Tube operator is an operator that turns line geometry into tubes, making the lines appear fatter and shaded.

Changing tube appearance

The Tube operator provides a few knobs that control the appearance of the generated tubes. First of all, the tube radius can be set by typing a new radius into the **Radius** text field in the **Tube attributes window** (Figure 4.169). The specified radius can either be a **Fraction of Bounding Box** (default) or **Absolute** by changing the combo box option next to the **Radius** text box. If you want the radius scaled by a variable instead, check the **Scale width by variable?** checkbox, and choose a variable from the **Variable** menu.

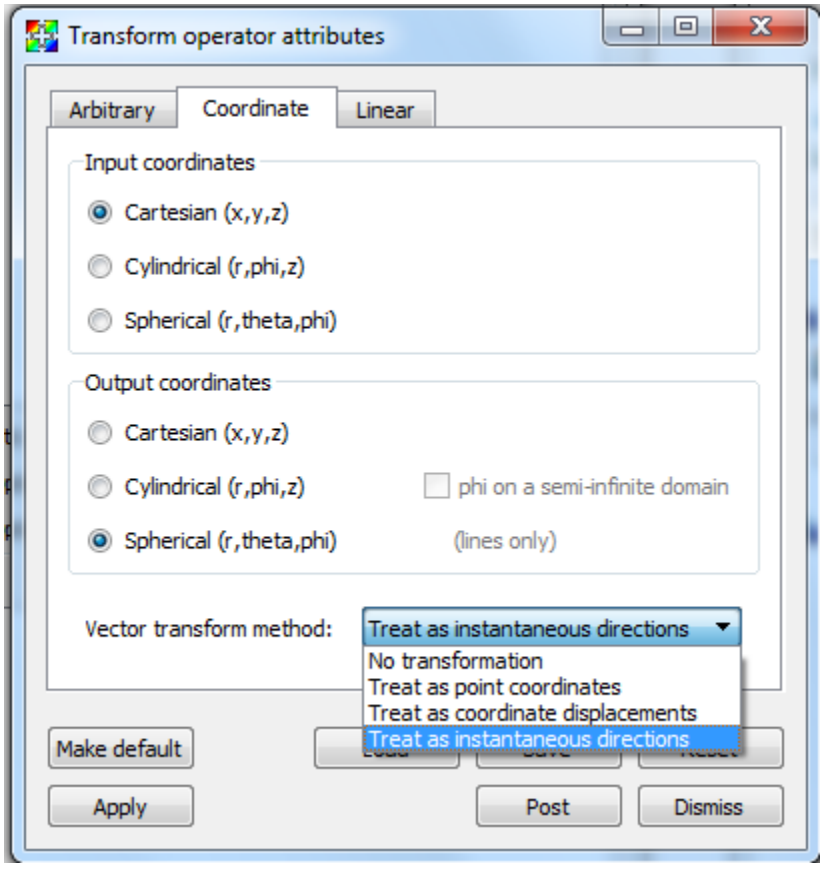


Fig. 4.166: Supported coordinate systems

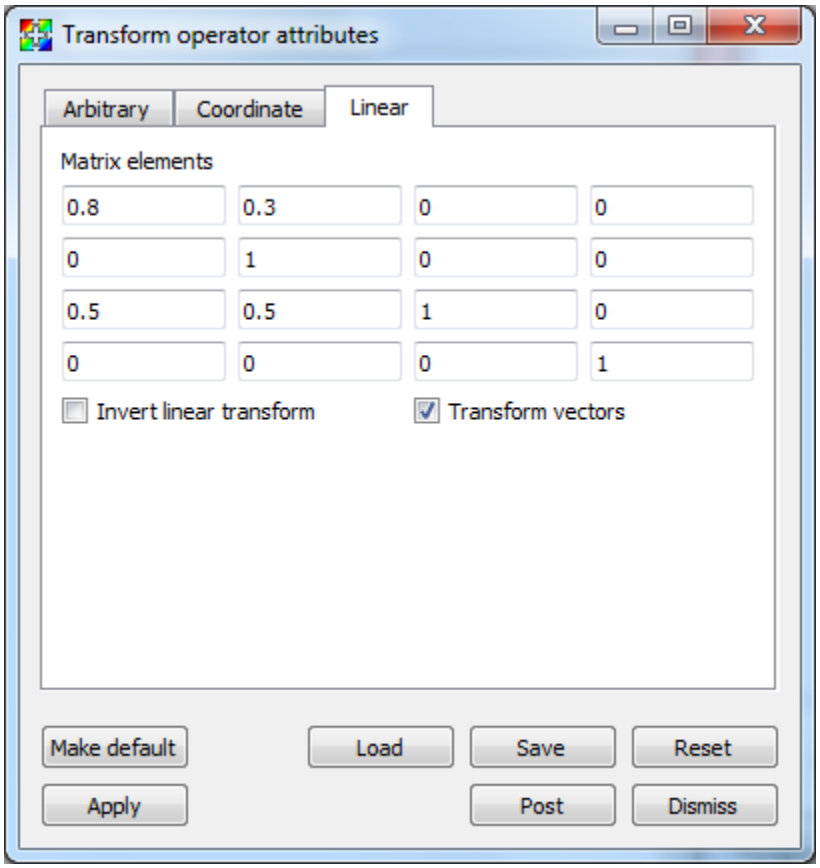


Fig. 4.167: Linear transformation options

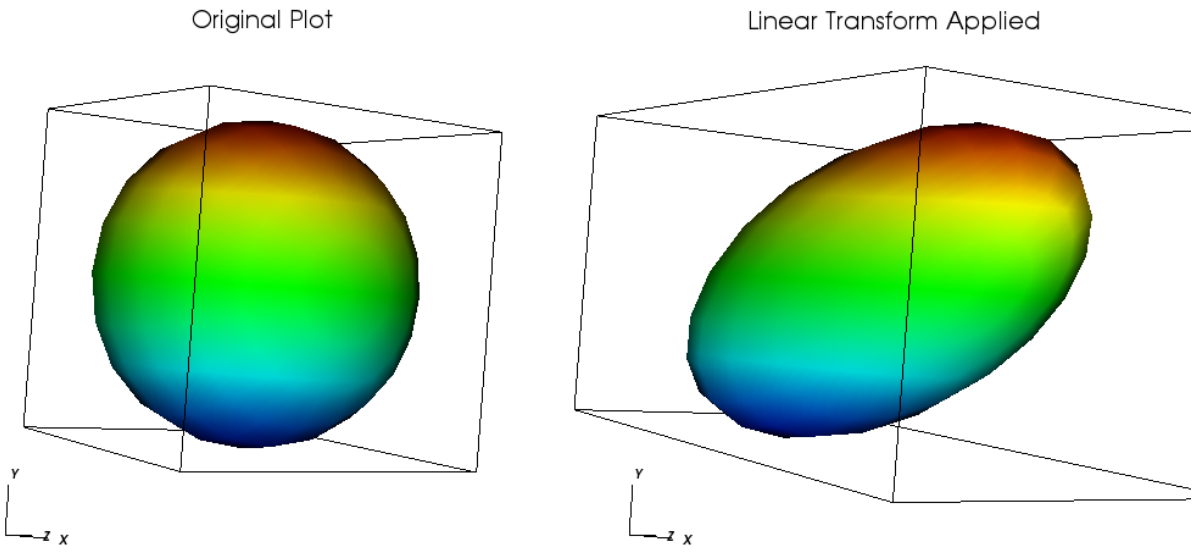


Fig. 4.168: Linear transformation example

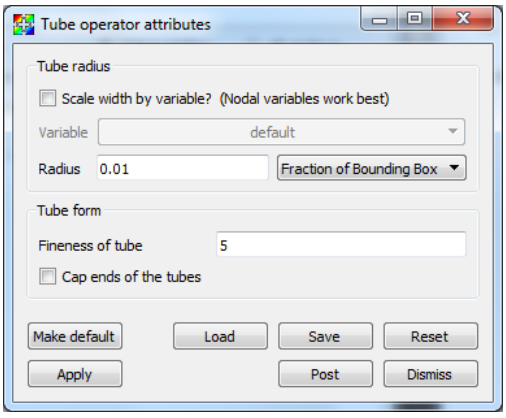


Fig. 4.169: Tube attributes window

The number of polygons used to make up the circumference of the tube can be altered by typing a new number of sides into the **Fineness of tube** text field. Finally, the ends of tubes can be capped instead of remaining open by turning on the **Cap Tubes** check box. See Figure 4.170 for result of capping.

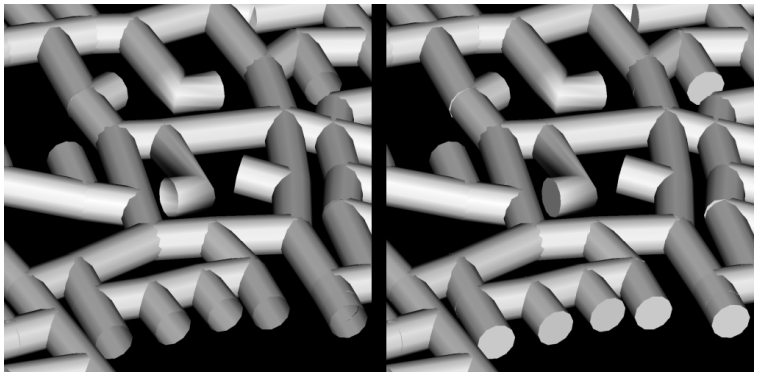


Fig. 4.170: Uncapped and capped tubes

4.5 Saving and Printing

In this chapter, we discuss how to save and print files from within VisIt. The section on saving files is further broken down into four main areas: saving session files, saving images, saving movies, saving Cinema databases, and exporting databases. We first learn about saving session files using the **Save Session** window. We then learn about saving images of visualizations using the **Save Window** and then we move on to saving movies and sets of image files using the **Save movie wizard**. In addition to movies, VisIt provides the **Save to Cinema** wizard to create Cinema image databases, which surpass movies and allow the user to explore data from different viewpoints. After learning to save images, movies, and Cinema databases, this chapter concentrates on exporting VisIt plots as databases using the **Export Database window**. Finally, we learn to print images of visualizations using the **Printer Window**.

4.5.1 Session files

A session file is an XML file that contains all of the necessary information to recreate the plots and visualization windows used in a VisIt session. You can set up complex visualizations, save a session file, and then run a new VisIt session later and be able to pick up exactly where you left off when you saved the session file. If you often look at the same types of plots with the same complex setup then you should save a session file for your visualization once it is set up so you don't have to do any manual setup in the future.

Saving session

Once you have set up your plots, you can select **Save session** option in the **Main Window's File** menu to open up a **Save file** dialog. Once the **Save file** dialog is opened, select the location and filename that you want to use to store the session file. By default, VisIt stores all session files in your .visit directory on UNIX and MacOS X computers and in the directory where VisIt was installed on Windows computers. Once you select the location and filename to use when saving the session file, VisIt writes an XML description of the complete state of all vis windows, plots, and GUI windows into the session file so the next time you come into VisIt, you can completely restore your VisIt session.

Restoring session

Restoring a VisIt session file deletes all plots, closes all databases, etc before VisIt reads the session file to get back to the state described in the session file. After restoring a session file, VisIt will look exactly like it did when the session

file was saved. To restore a session file, click the **Restore session** option from the **Main Window's File** menu to open an **Open file** dialog. Choose a session file to open using the **Open file** dialog. Once a file is chosen, VisIt restores the session using the selected session file. If you are on the Windows platform, you can double-click session files (.vses files) stored on your computer in order to directly open them with VisIt.

4.5.2 Saving the Visualization Window

VisIt allows you to save the contents of any open visualization window to a variety of file formats. You can save visualizations as images so they can be imported into presentations. Alternatively, you can save the geometry of the plots in the visualization window so it can be imported into other computer modeling and visualization programs.

VisIt currently supports the image files formats: *BMP*, *JPEG*, *PNG*, *PPM*, *Raster Postscript*, *RGB*, and *TIFF*

VisIt currently supports the geometry file formats: *Curve*, *Alias WaveFront Obj*, *PLY*, *POV*, *STL*, *ULTRA*, and *VTK*

The Curve and ULTRA file formats are specially designed to store the data created from curve plots and can be used with other Lawrence Livermore National Laboratory visualization software. The Alias Wavefront Obj file format is supported so visualizations produced with VisIt can be imported into rendering programs such as Maya. VisIt can save visualizations into STL files, which are used with stereolithographic printers to fabricate three-dimensional parts. Finally, VisIt can save visualizations into the VTK (Visualization Toolkit) format so they can be read back into VisIt and used in other VTK-based applications.

When saving the geometry of plots in the visualization window into any of the afore-mentioned formats, you are performing a type of database export operation. However, saving geometry in this manner differs from exporting databases using the **Export Database Window**. Only the external faces of the plots are saved out when saving plot geometry whereas during a database export, 3D cells are preserved in the final exported database. The topic of exporting databases is covered later in this chapter.

The Save Window

You can set the **Save window** options before saving by selecting **Set Save options...** from the **Main Window's File menu**. The **Set save options** window contains the controls that allow you to set the options that govern how visualizations are saved.

The **Set Save options** window, shown in Figure 4.171, contains four basic groups of controls. The first group, *Filename*, allows you to set the file information. Use the file information controls to set the name and destination. If the *Family* checkbox is selected, then each time an image is saved with the same name, a number will be appended to the filename that is one more than the current file with the same name. The second group, *Format options*, allows you to set the file type, compression type, and any optional quality parameters that may exist for the selected file type. Use the third group of controls, *Aspect ratio and resolution*, to specify the dimensions of the saved image. If *Screen capture* is checked, the aspect ratio and width/height will be ignored and the current screen image will be saved. The last group, *Multi-window save*, allows you to set options for each window being saved by clicking on the **Window** drop-down and selecting the appropriate window. When the save options are set and applied by clicking the **Apply** button, the active visualization can be saved either through the **Save Window** option in the **Main Window's File menu**, by the keyboard shortcut *Ctrl+S* , or by clicking the **Save** button in the **Set Save options** window.

Selecting the output directory for saved files

On most platforms, VisIt's default behavior is to save output files to the current directory, which is the directory where VisIt was started. On the Windows platform, VisIt saves images to the *location* `VUSER_HOME/My images`. If you want to specify a special output directory for your output files, you can turn off the **Output files to current directory** check box and type in the path to the directory where you want VisIt to save your files in the **Output directory** text field. If you want to browse the file system to find a suitable directory in which to save your images, click on the *"..."* button to the right of the **Output directory** text field to bring up a **Directory chooser** dialog. Once you select

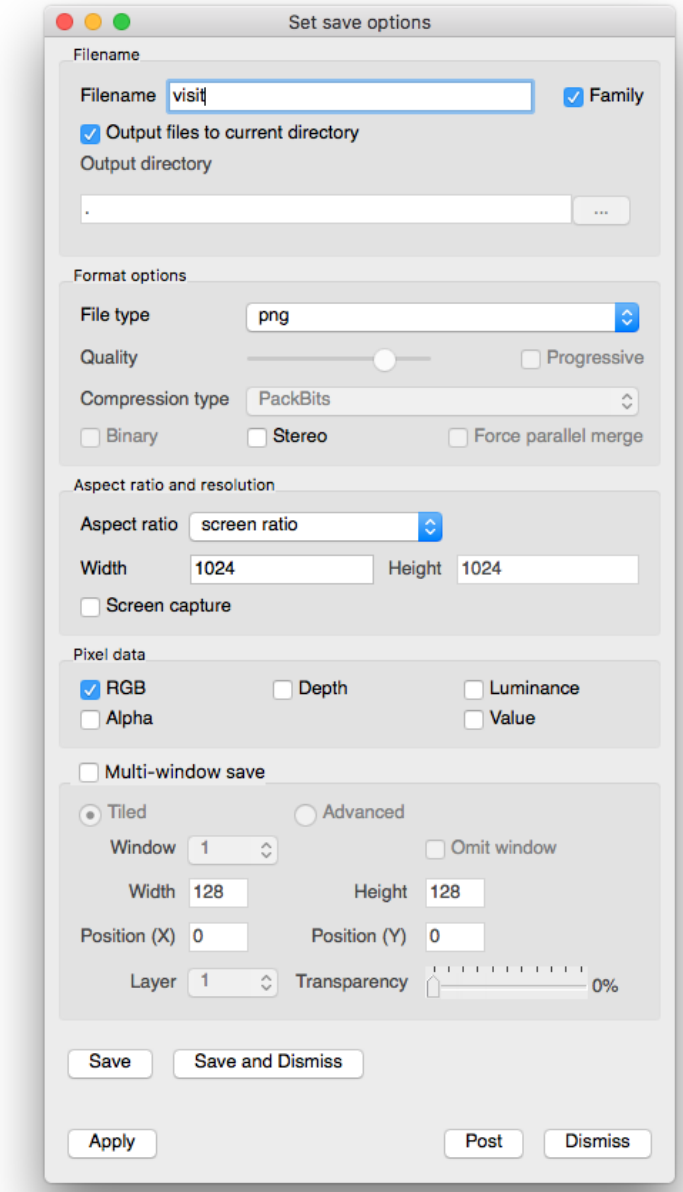


Fig. 4.171: Save Window

a suitable directory using the **Directory chooser** dialog, the path that you chose is inserted into the **Output directory** text field.

Setting the save file name

To set the file name that will be used to save files, type a file name into the **Filename** text field. The file name that you use may contain a path to a directory where you want to write the saved files. If no path is specified, the saved files are written to the directory from which VisIt was launched. A file extension appropriate for the type of file being generated is automatically appended to the file name. For example, a *BMP* file will have a *".bmp"* extension, while a *JPEG* file will have a *".jpeg"* extension, and so on.

The file name that VisIt uses to save visualizations is based on the specified file name, the file format, and also the family toggle setting. The family toggle setting is set by checking the **Family** check box towards the top right part of the **Save Window**.

The family toggle setting allows you to save series of files that all have essentially the same name except for a number that is appended to the file name. The number increases by one each time an image is saved. If the family toggle setting is on then a file named *"visit"* of type *TIFF* will save out as *"visit0000.tiff"*. If the family toggle setting is off, the file will save as *"visit.tiff"*.

Setting the file type

You set the file type by making a selection from the **File type** menu. You can choose from image file types or geometry file types. Note that some areas of the **Save Window** become enabled or disabled for certain file types.

Choosing *JPEG* format files enables the **Quality** slider and the **Progressive** check box. These controls allow you to specify the desired degree of quality in the resulting JPEG images. A lower quality setting results in blockier images that fit into smaller files. The progressive setting stores the *JPEG* images in such a way that they progressively refine as they are downloaded and displayed by Web browsers.

Choosing *TIFF* format files enables the **Compression type** combo box. The available compression types are: *None*, *PackBits*, *JPEG*, and *Deflate*. When compression is enabled for *TIFF* files, they are smaller than they would be without compression.

Choosing *STL* or *VTK* file formats saves visualizations as geometry files instead of images and also enables the **Binary** check box. The **Binary** check box tells these formats to write their geometry data as binary data files instead of human-readable ASCII text files. In general, files written with the binary option are smaller and faster to load than their non-binary counterparts.

Saving images with screen capture

The **Screen capture** check box tells VisIt to grab the image directly off of the computer screen. This means that the saved image will be exactly the same size as the image on the screen. There are advantages and disadvantages to using screen capture. An advantage is that capturing the image from the screen does not require VisIt to redraw the image to an internal buffer before saving, which usually results in a faster save. A disadvantage of screen capture is that any other windows on top of VisIt's visualization window occlude portions of the image. Screen capture can also be very slow over a sluggish network connection. Finally, using screen capture might not provide images that have enough resolution. Weigh the advantages and disadvantages of using screen capture for your own situation. Screen capture is on by default.

Setting image resolution

You set image resolution using the controls in the **Aspect ratio and resolution** group. These controls are disabled unless the file being saved is an image format and screen capture is not being used. You specify the image height and

width by typing new values into the **Height** and **Width** text fields. If the **Maintain 1:1 aspect** check box is on, VisIt forces the image's height and width to be the same, yielding a square image. Turn off this setting if you want to save rectangular images. The image resolution is ignored unless you turn off the **Screen capture** check box.

Saving stereo images

When the **Stereo** check box is turned on and you save an image, VisIt will save a separate image for the left eye and for the right eye. The cameras used to generate each image are offset such that when the images are played together at high rates, they appear to have more depth. To enable saving of stereo images, click the **Stereo** check box in the **Save Window** before you try to save an image.

When **Family** mode is not enabled, VisIt will prepend *left_* and *right_* designators to the saved filenames. However, when **Family** mode is enabled, VisIt saves the two images in sequence without any left/right designation. The left image is saved first followed by the right image. If next available number in the **Family** is odd, the left will be odd and right will be even. On the other hand, if next available number in the **Family** is even, the left will be even and right will be odd. However, the notification messages VisIt produces about the saved images may only mention the first (left) saved image filename.

Saving binary geometry files

Some geometry file formats such as *STL* and *VTK* have both ASCII and binary versions of the file format. The ASCII file formats are human-readable and are larger and slower for programs to process than binary formats, which are not human-readable but are smaller and quicker for programs to read. When geometry file formats support both ASCII and binary formats, the **Binary** check box is enabled. By default VisIt writes ASCII geometry files but you can click the **Binary** check box to make VisIt write binary geometry files.

Selecting pixel data

Normally when saving an image, VisIt will simply save the RGB pixel data into the specified image format. It is possible to request that VisIt saves additional pixel data when saving an image. This may result in additional files being saved alongside the normal image file. These additional images will share the same filename root as the image file but will have suffixes such as "value", "depth", or "lum", depending on their contents. Special file formats such as OpenEXR can contain all of these additional image channels. When OpenEXR is the selected file format, a single ".exr" file will be written containing all pixel data.

The **Save options** window contains a **Pixel data** group that lets you request additional image channels. The **RGB** check box selects RGB pixel data. The **Alpha** check box tells VisIt to also request transparency information and to not render with a background when saving an image. This lets VisIt save images with a transparent background, which makes compositing such an image in front of other backgrounds far easier (see Figure 4.172). The **Depth** check box tells VisIt to export the depth buffer (Z-buffer) to a ZLib-compressed binary file containing 32-bit floating point numbers. The **Luminance** check box tells VisIt to save a luminance image, which shows how much lighting is used in various parts of the scene. The luminance image is saved to the selected image format. The **Value** check box tells VisIt to produce a rendering of the actual scalar values in the scene in the form of a ZLib-compressed 32-bit floating point buffer (same format as the depth image).

Saving tiled images

A tiled image is a large image that contains the images from all visualization windows that have plots. If you want to save tiled images, make sure to check the **Save tiled** check box in the **Set Save options** window. To get an idea of how VisIt saves your visualization windows into a tiled image, see Figure 4.173 and Figure 4.174.

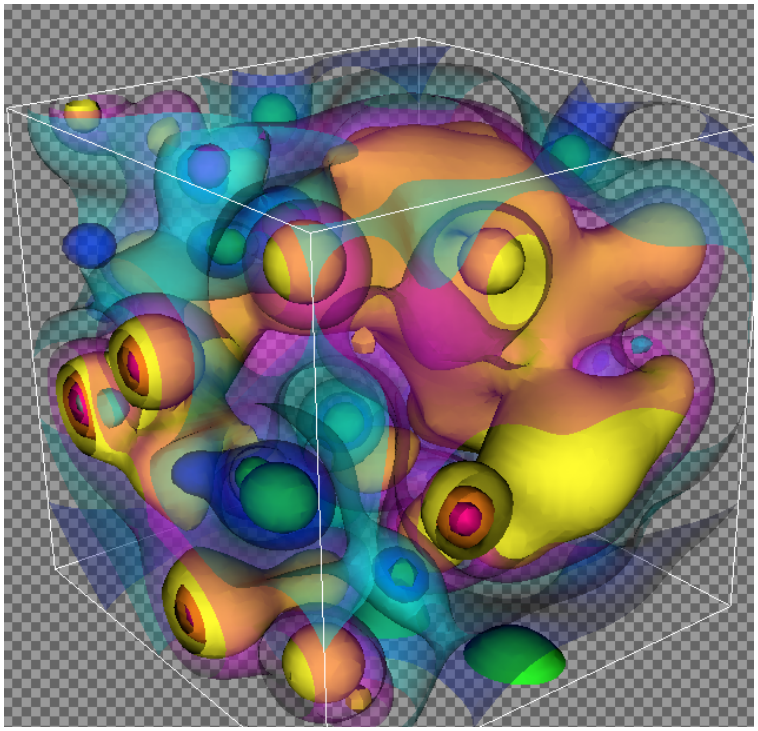


Fig. 4.172: Partially transparent plot saved to PNG with alpha channel

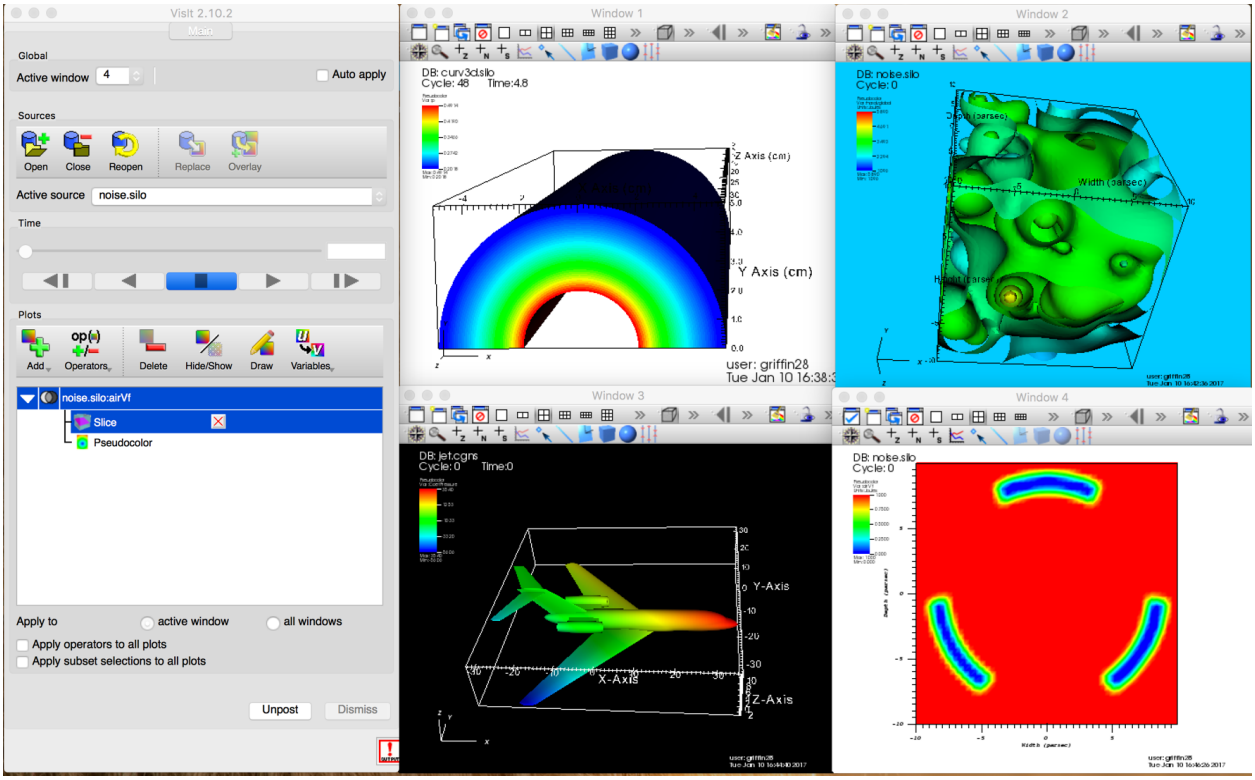


Fig. 4.173: Saving tiled images example (before)

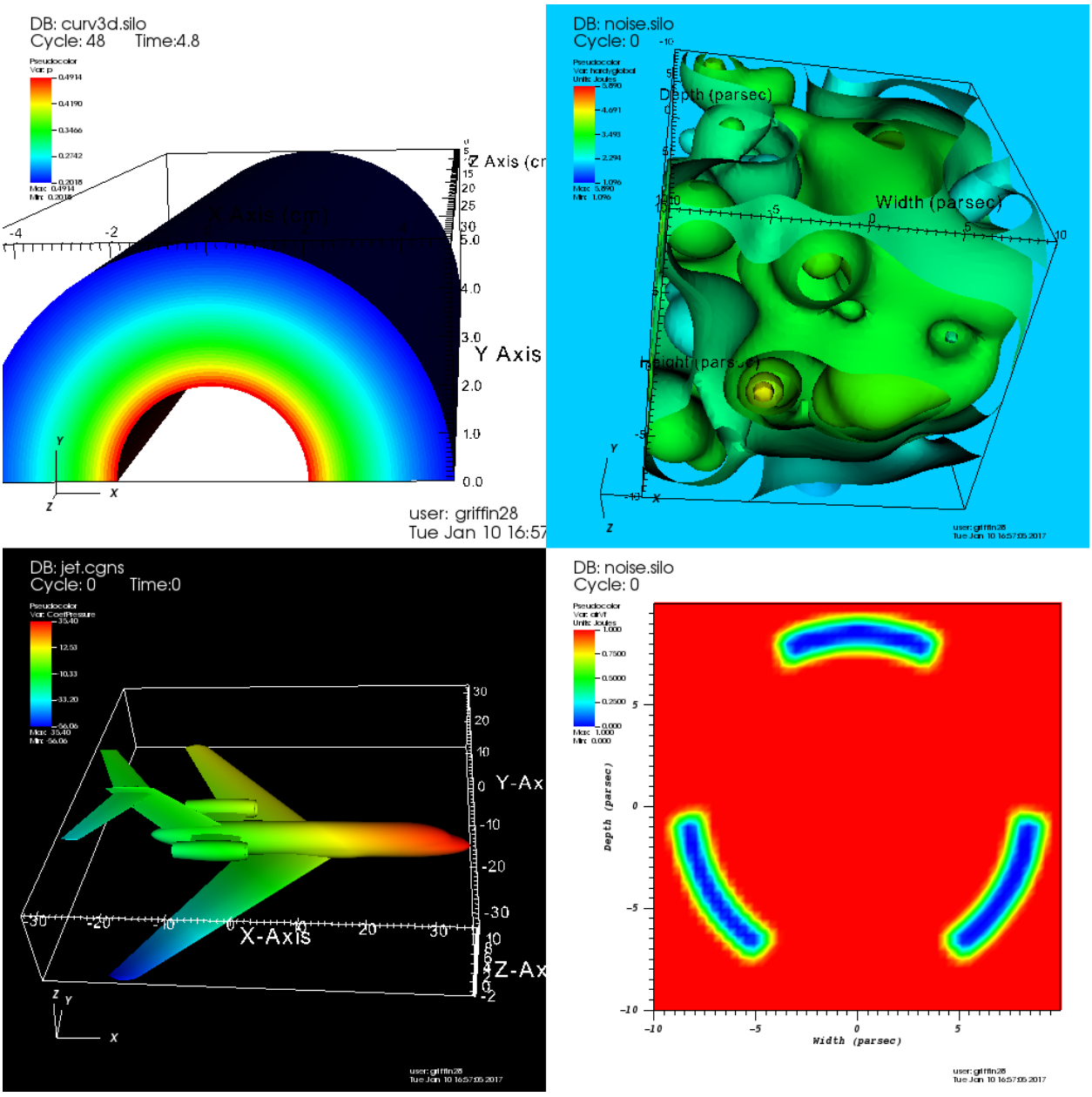


Fig. 4.174: Saving tiled images example (after)

4.5.3 Saving movies

In addition to allowing you to save images of your visualization window for the current time state, VisIt also allows you to save movies and sets of images for your visualizations that vary over time. There are multiple methods for saving movies with VisIt. This section introduces the Save movie wizard and explains how to use it to create movies from within VisIt's GUI. The *Animation* chapter explains some auxiliary methods that can be used to create movies.

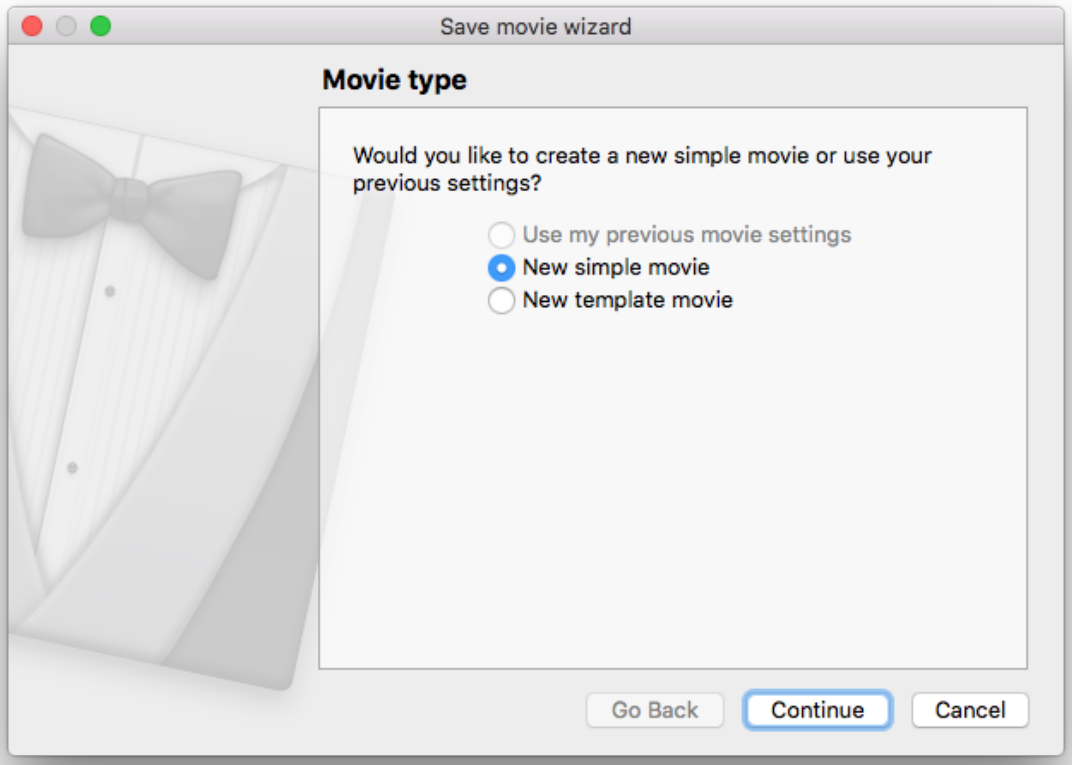


Fig. 4.175: Save movie wizard (screen 1)

The **Save movie wizard** (see Figure 4.175) is available in the **Main Window's Files** menu. The **Save movie wizard's** purpose is to lead you through a set of simple questions that allow VisIt to gather the information required to create movies of your visualizations. For example, the **Save movie wizard** asks which image and movie formats you want to generate, where you want to store the movies, what you want to call the movies, etc. Each of these questions appears on a separate screen in the **Save movie wizard** and once you answer the question on the current screen, clicking the **Next (Continue for macOS)** button advances you to the next screen. You can cancel saving a movie at any time by clicking on the **Cancel** button. If you advance to the last screen in the **Save movie wizard** then you have successfully provided all of the required information that VisIt needs to make your movie. Clicking the **Finish** button at that point invokes VisIt's movie-making script to make the movie. If you want to make subsequent movies, you can choose to use the settings for the movies that you just made or you can choose to create a new movie and provide new information.

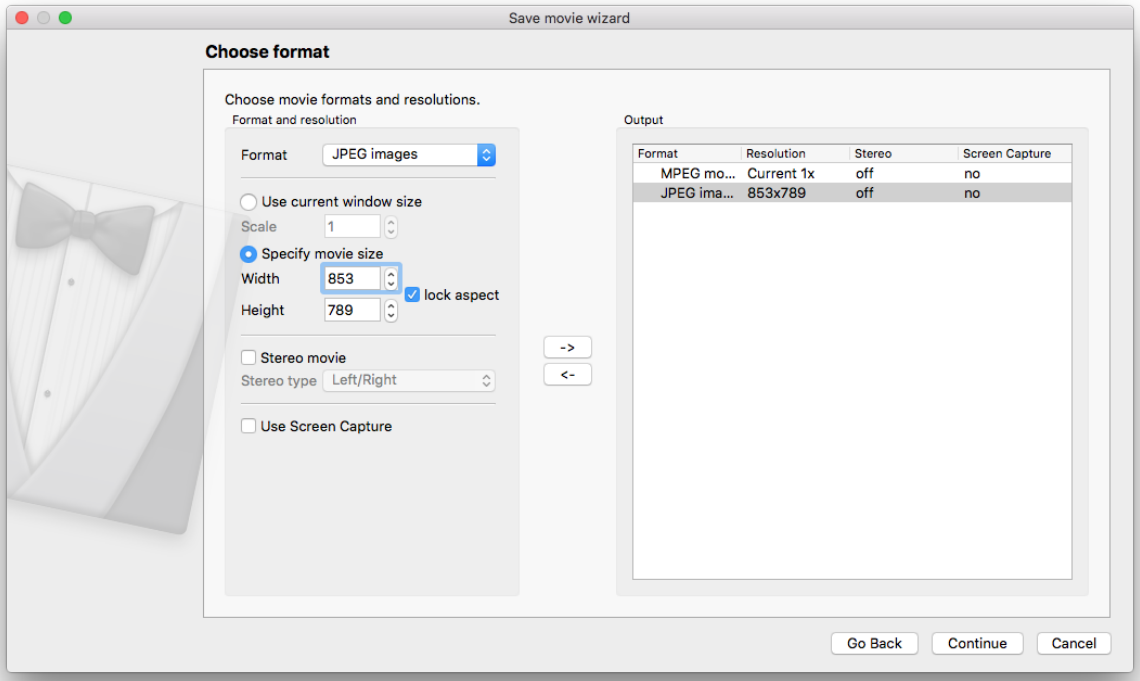


Fig. 4.176: Save movie wizard (screen 2)

Choosing movie formats

The **Save movie wizard's** second screen, shown in Figure 4.176, allows you to pick the types of movies that you want to create. You can select as many image and movie formats as you want and you can even specify multiple resolutions of the same movie. VisIt allows you to order multiple versions of your movie because it is often easier to create different versions of the movie all at once as opposed to doing it later once it is discovered that you need a new version to play on a laptop computer or a tiled display wall.

The **Save movie wizard's** second screen is divided vertically into two main areas. On the left you will find the **Format and resolution** area, which displays the format and resolution for the current movie. On the right, you will find the **Output** area, which lists the formats and resolutions for all of the movies that you have ordered. By default no movie formats are present in the **Output** area's list of movies. You cannot proceed to the next screen until you add at least one movie format to the list of movies in the **Output** area.

To add a movie format to the list of movies in the **Output** area, first choose the desired movie format from the **Format** combo box in the **Format and resolution** area. Next, choose the movie resolution. The movie resolution can be specified in terms of the visualization window's current size or it can be specified in absolute pixels. The default movie resolution uses the visualization window's current size with a scale of 1. You can change the scale to shrink or grow the movie while keeping the visualization window's current aspect ratio. If you want to specify an absolute pixel size for the movie, click on the **Specify movie size** radio button and type the desired movie width and height into the **Width** and**Height** text fields. Note that if you specify a width and height that causes the movie's shape to differ from the visualization window's shape, you might want to double-check that the view used for the visualization window's plots does not change appreciably.

The **Save movie wizard** allows you to create stereo movies if you check the **Stereo movie** box and select a stereo type from the **Stereo type** drop-down menu. The default is to create non-stereo movies because stereo movies are not widely supported.

Note: "Streaming movie" format is an LLNL format

The only movie format that VisIt produces that is compatible with stereo movies is the "Streaming movie" format, which is an LLNL format commonly used for tiled displays. The "Streaming movie" format can support stereo movies where the image will flicker between left and right eye versions of the movie, causing a stereo effect if you view the movie using suitable liquid-crystal goggles. The stereo option has no effect when used with other movie formats. However, if you choose to save a stereo movie in any of VisIt's supported image formats, VisIt will save images for the left eye and images for the right eye. You can then take the left and right images into your favorite stereo movie creation software to create your own stereo movie.

Once you have selected the desired movie format, width, and height, click on the right-arrow button that separates the **Format and resolution** area from the **Output** area. Clicking the right-arrow button adds your movie to the list of movies that you want to make. Once you have at least one movie in the **Output** area, the screen's Next button will become active. Click the **Next** button to go to the next screen in the **Save movie wizard**

Choosing movie length

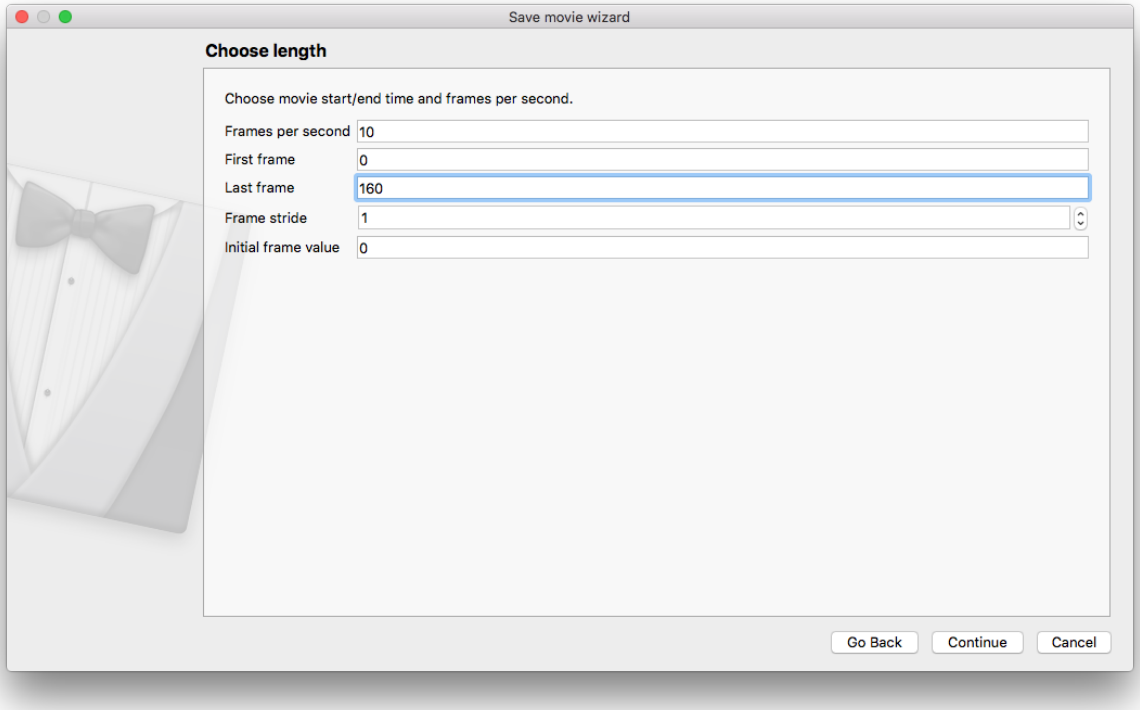


Fig. 4.177: Save movie wizard (screen 3)

It is possible to specify the range of time states to use for the movie, as well as specify a stride if you have too many time states saved (see Figure 4.177). The wizard will automatically set the range of time states.

Choosing the movie name

Once you have specified options that tell VisIt what kinds of movies that you want to make, you must provide the base name and location for your movies. By default, movies are saved to the directory in which you started VisIt. If you

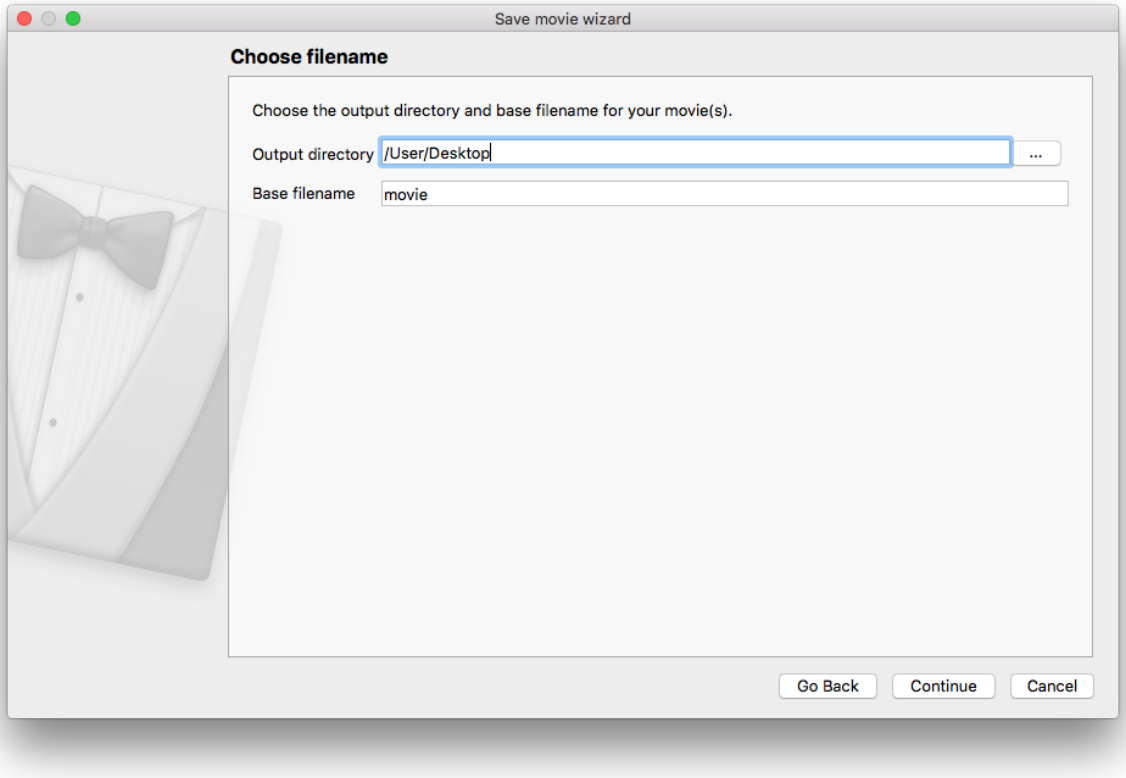


Fig. 4.178: Save movie wizard (screen 4)

want to specify an alternate directory, you can either type in a new directory path into the **Output directory** text field (see Figure 4.178) or you can select a directory from the **Choose directory** dialog box activated by clicking on the "..." button.

The base filename for the movie is the name that is prepended to all of the movies that you generate. When generating multiple movies with differing resolutions, the movie resolution is often encoded into the filename. VisIt may generate many different movies with different names but they will all share the same base filename that you provided by typing into the **Base filename** text field.

Choosing e-mail notification

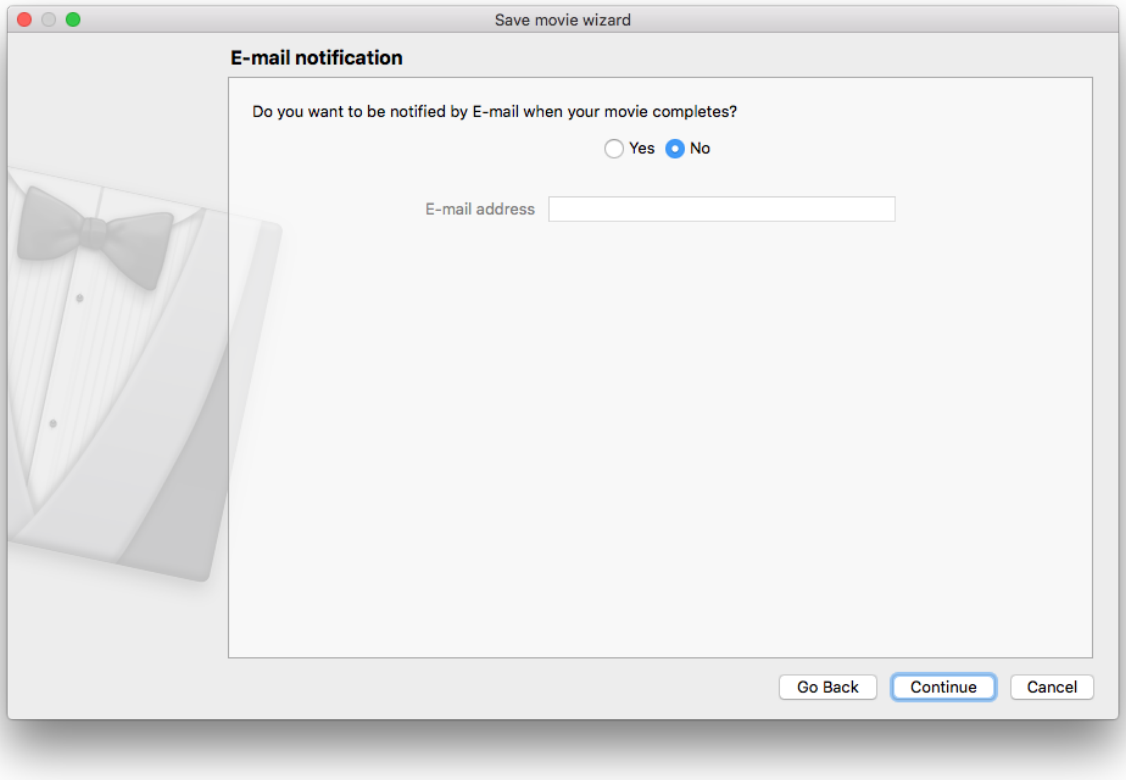


Fig. 4.179: Save movie wizard (screen 5)

If you want to be notified by e-mail when the movie creation is complete, then select the **Yes** option and enter the appropriate e-mail address (see Figure 4.179). By default, no e-mail notification is sent once the movie creation is complete.

Choosing movie generation method

After all movie options are specified, VisIt prompts you how you would like your movie made. At this point, you can click the **Finish/Done** button to make VisI_t start generating your movie. You can change how VisIt creates your movie by clicking a different movie generation method on the **Save movie wizard's** sixth screen, shown in Figure Figure 4.180.

The default option for movie creation allows VisIt to use your current VisIt session to make your movies. This has the advantage that it uses your current compute engine and allocated processors, which makes movie generation start

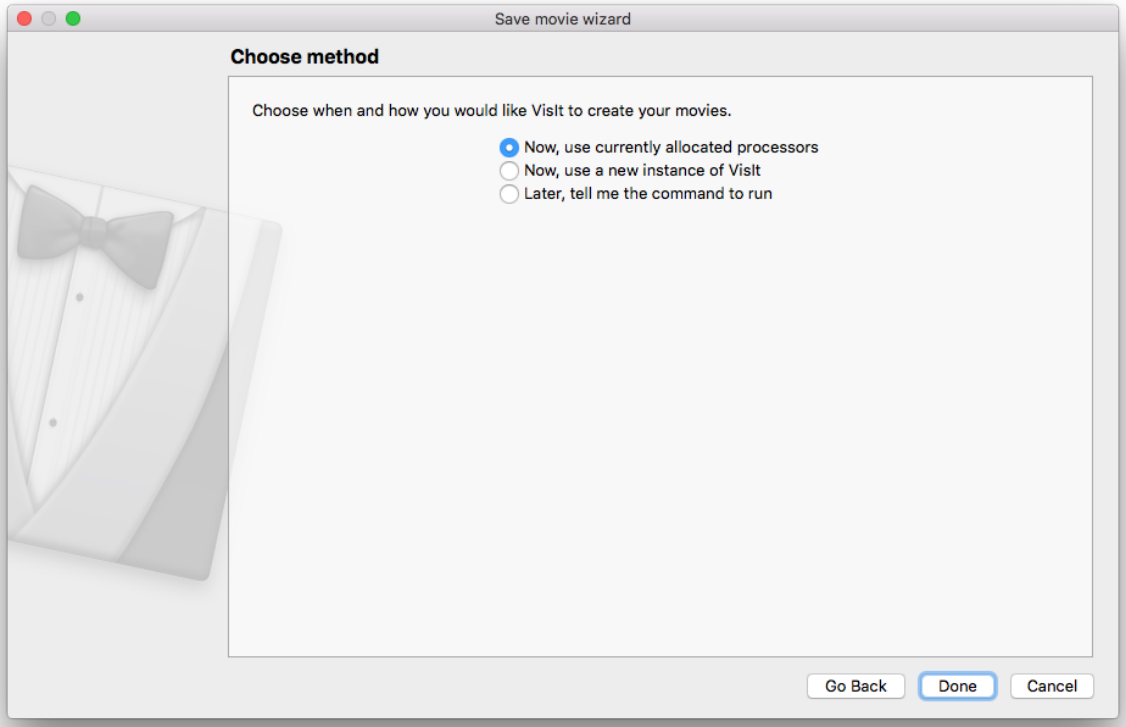


Fig. 4.180: Save movie wizard (screen 6)

immediately. When you use this movie generation method, VisIt will launch its command line interface (CLI) and execute Python movie-making scripts in order to generate your movie. This means that you have both the VisIt GUI and CLI controlling the viewer. If you use this movie generation method, you will be able to watch your movie as it is generated. You can track the movie's progress using the **Movie progress dialog** , shown in Figure 4.181. The downside to using your currently allocated processors is that movie generation takes over your VisIt session until the movie is complete. If you want to regain control over your VisIt session, effectively cancelling the movie generation process, you can click the **Movie progress dialog's Cancel** button.

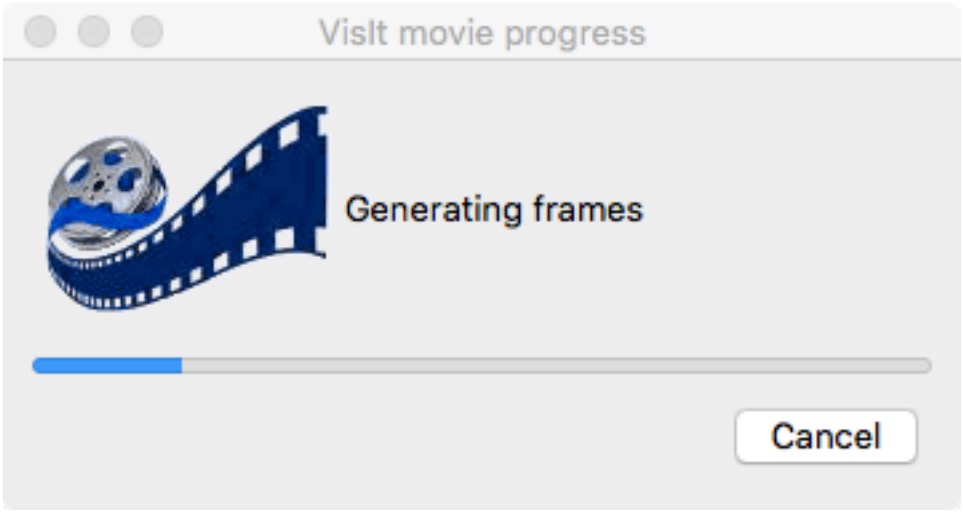


Fig. 4.181: Movie progress dialog

The second movie generation method will cause VisIt to save out a session file containing every detail about your visualization so it can be recreated by a new instance of VisIt. This method works well if you want to create a movie without sacrificing your current VisIt session but you cannot watch the movie as it is generated and you may have to wait for the second instance's compute engine to be scheduled to run. The last movie generation option simply makes VisIt display the command that you would have to type at a command prompt in order to make VisIt generate a movie of your current visualizations.

Templated movies

Movie templates enable you to create a set of viewports and map the contents of various visualization windows to those viewports, also incorporating simple movie transitions.

To create a movie template, you can open the Save movie wizard as you normally would. Instead of choosing New simple movie, choose New template movie. VisIt will load custom user interface pages for the chosen movie template.

You can choose to use an existing template, edit an existing template, or create a new template.

Existing Movie Templates

Curve Overlay Movie Template

The **Curve Overlay** template lets you create regular plots and in a separate viewport, create a curve plot that completes as the movie advances in time.

You will need: A 2D or 3D time-varying dataset and a Curve file containing the curve that will be animated.

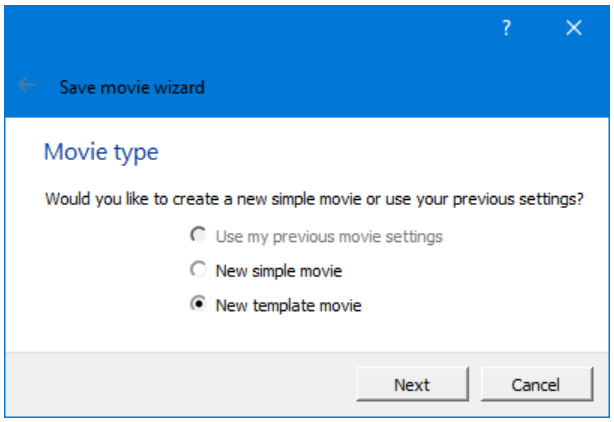


Fig. 4.182: Movie type selection page

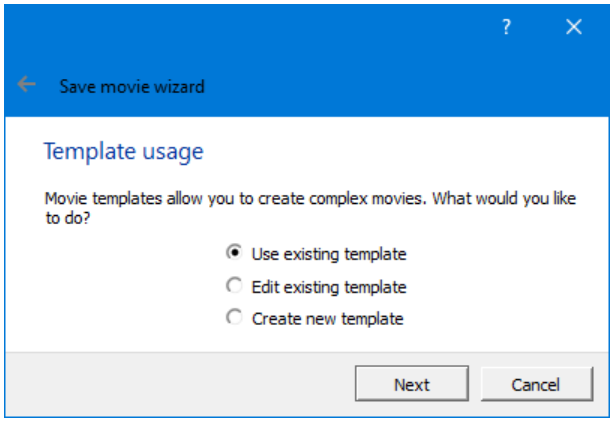


Fig. 4.183: Template choice.

If you use a 3D dataset, you will want to make sure that you set the view to something appropriate and that you position the 3D plot in the upper portion of the vis window. The 2D curve plot will be composited into the bottom part of the window.

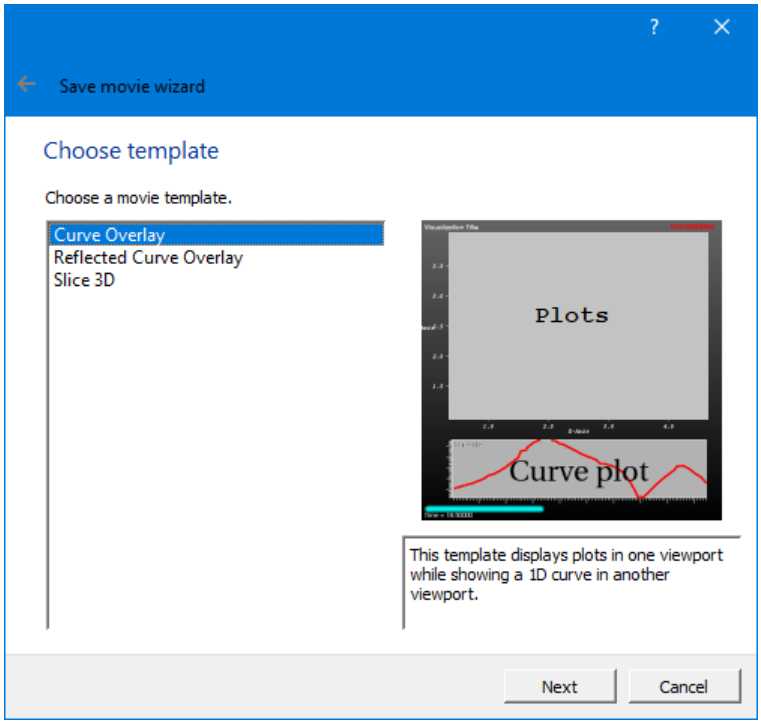


Fig. 4.184: Curve Overlay preview.

Figure 4.185 shows the options for the 2D/3D plots:

- Select your time varying database.
- Select the plot and variable that you want to plot.
- Adjust title and colors.

Select the curve file that contains the curve you want to animate as shown in Figure 4.186.

The rest of the **Save Movie wizard** pages for the **Curve Overlay** template are the same as for all movies described previously.

Figure 4.187 shows a frame from a **Curve Overlay** movie.

Reflected Curve Overlay Movie Template

The **Reflected Curve Overlay** template shows two plots in one viewport. One of the plots is reflected from the other's position. Similar to **Curve Overlay**, a Curve plot is shown in another viewport.

Figure 4.189 shows the options for the 2D/3D plots:

- Select the first time varying database.
- Select the plot and variable that you want to plot.
- Select the second time varying database (it can be the same as the first).
- Select the plot and variable that you want to plot.

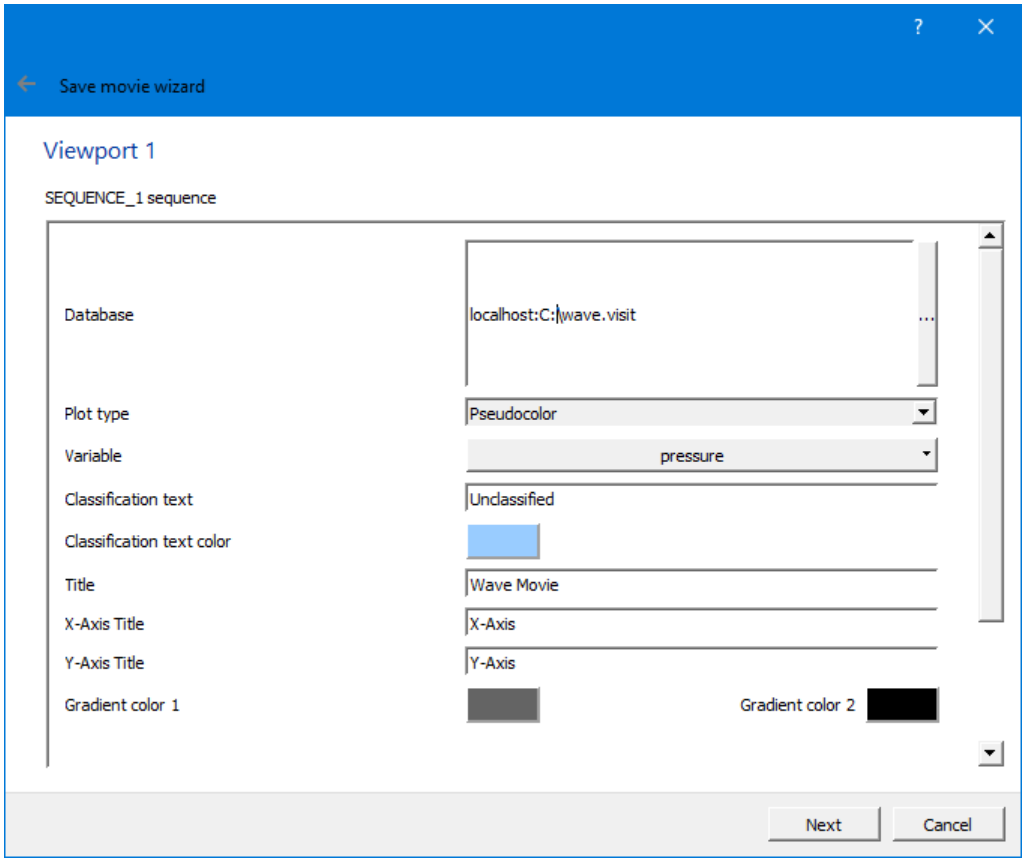


Fig. 4.185: Curve Overlay 2D/3D plot options

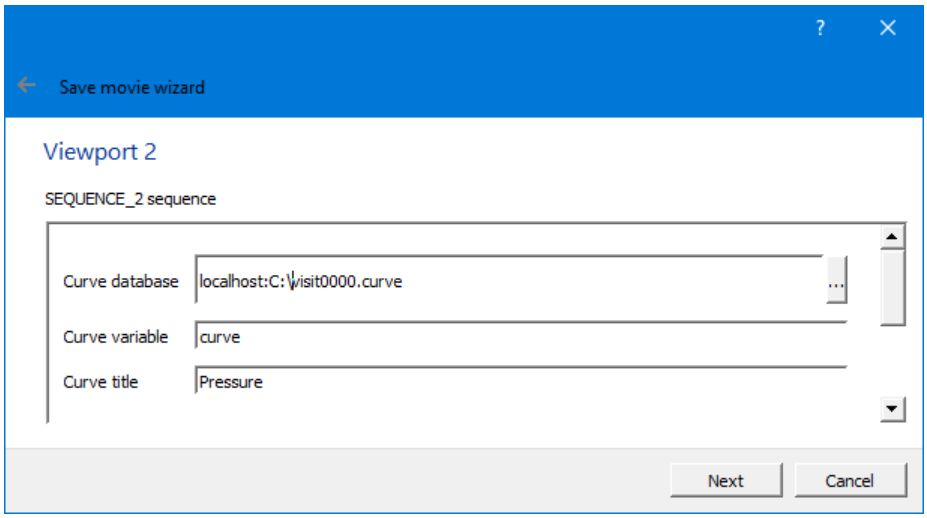


Fig. 4.186: Curve Overlay Curve plot options

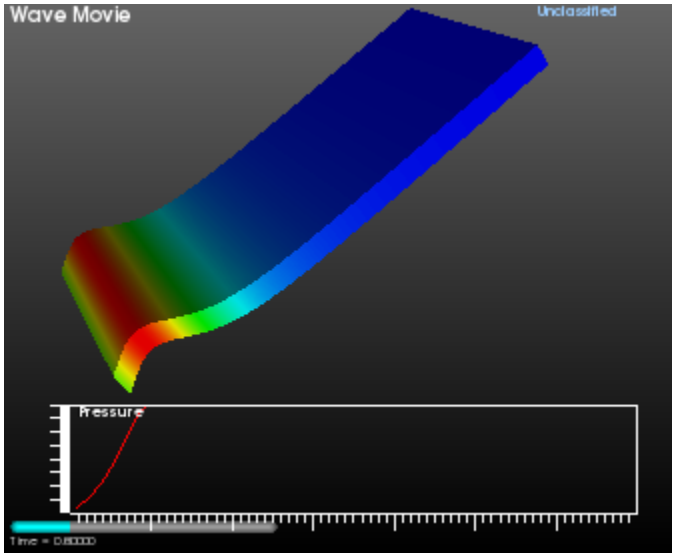


Fig. 4.187: Frame from a Curve Overlay template movie

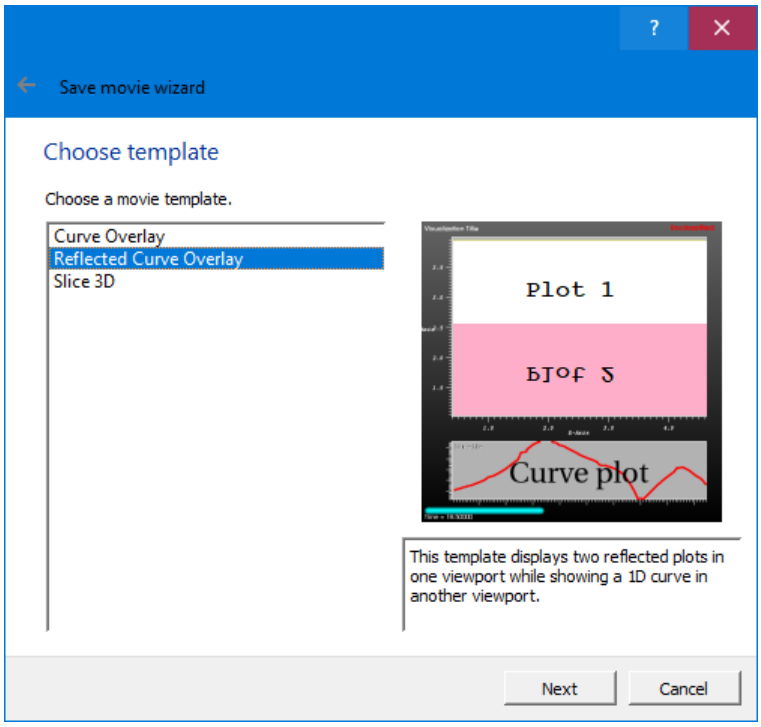


Fig. 4.188: Curve Overlay preview.

- Adjust title and colors.

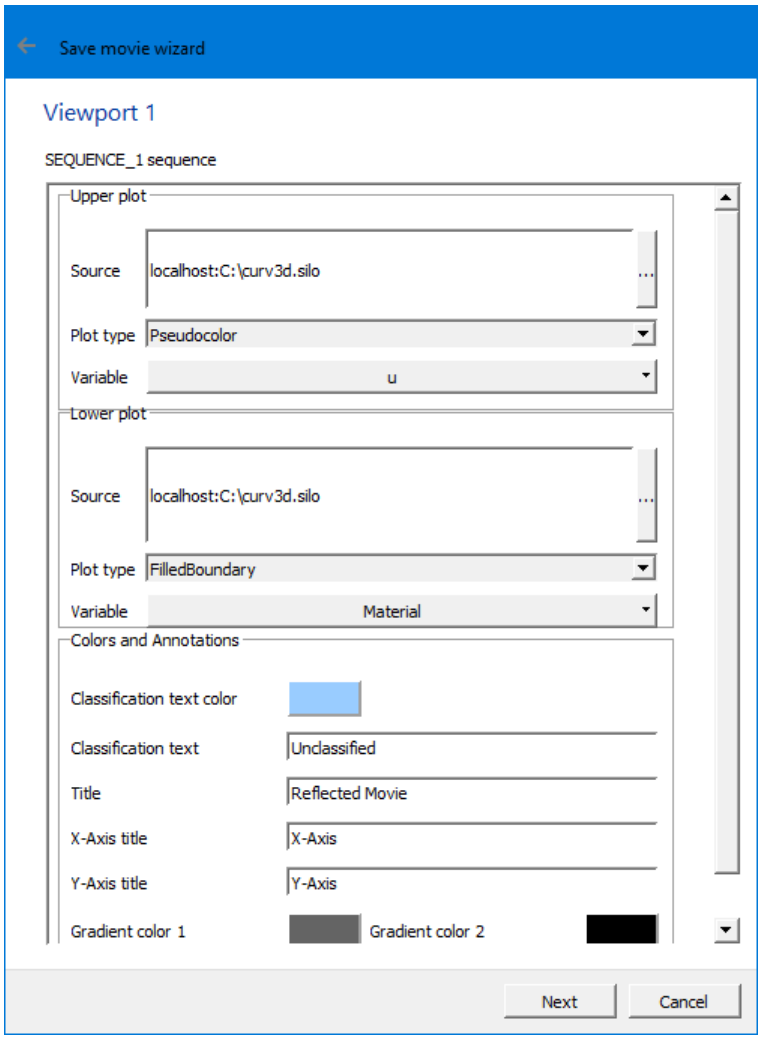


Fig. 4.189: Reflected Curve Overlay 2D/3D plot options

The options for the Curve plot portion are the same as used by the **Curve Overlay** template.

The rest of the Save Movie wizard pages for the **Reflected Curve Overlay** template are the same as for all movies described previously.

Slice 3D Movie Template

The **Slice 3D** movie template animates a slice plane through all three dimensions of a 3D dataset before fading to black.

The next page (Figure 4.192) lets you fill in details such as which dataset will be sliced. You can also set visualization title and colors.

This movie template ends with a fade to black sequence. You can set the duration of this sequence as well as the colors used as shown in Figure 4.193.

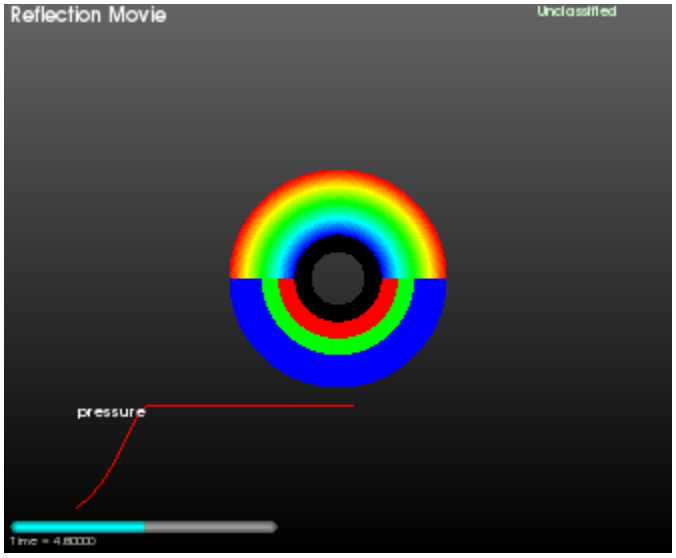


Fig. 4.190: Frame from a Reflected Curve Overlay template movie

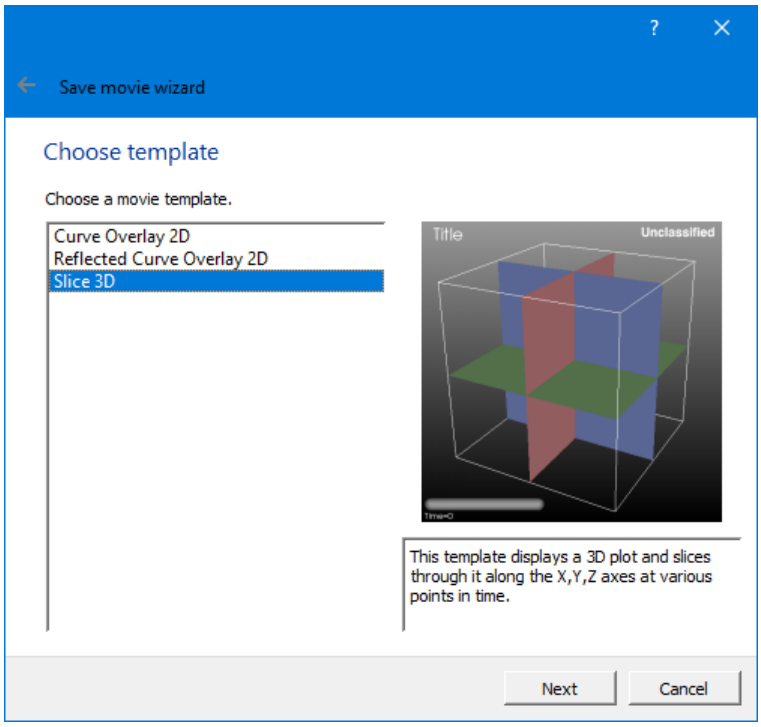


Fig. 4.191: Slice 3D template preview.

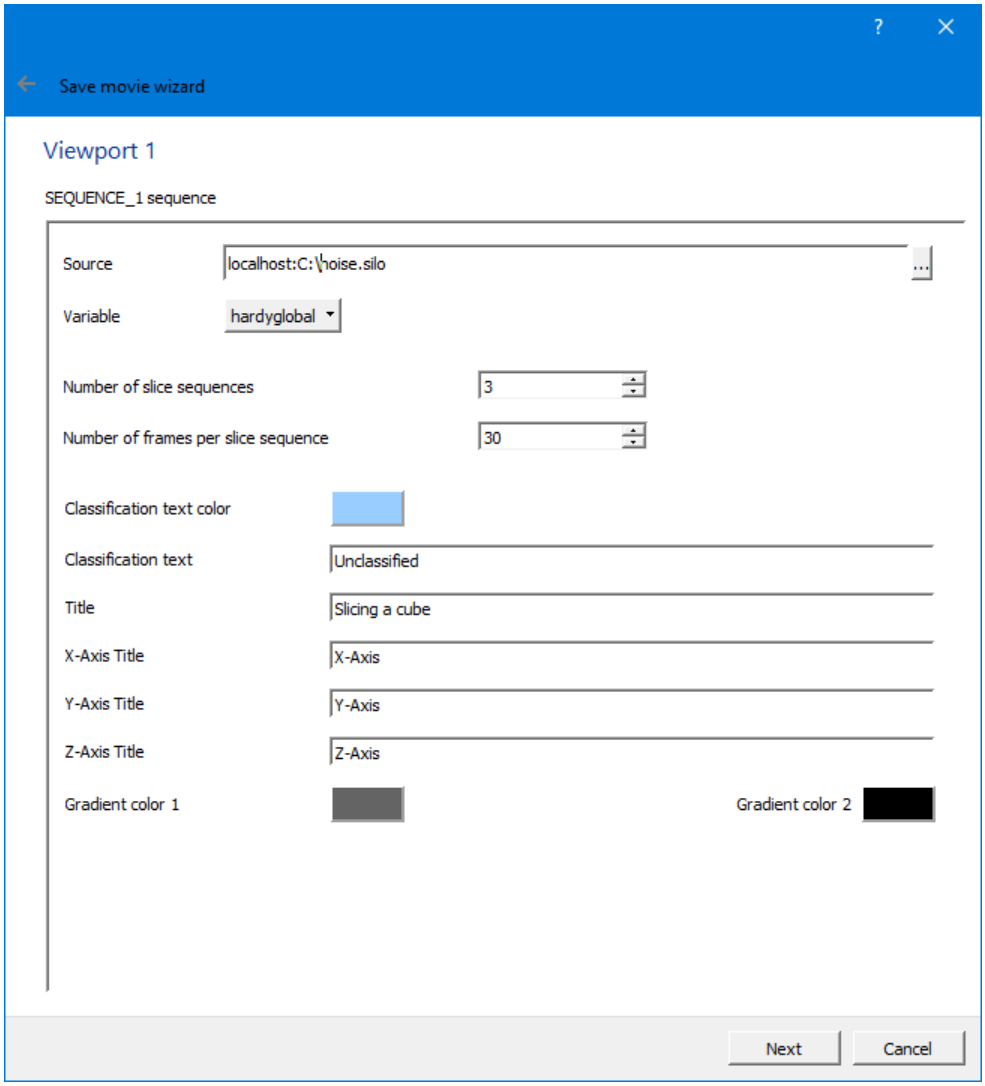


Fig. 4.192: Slice 3D parameters.

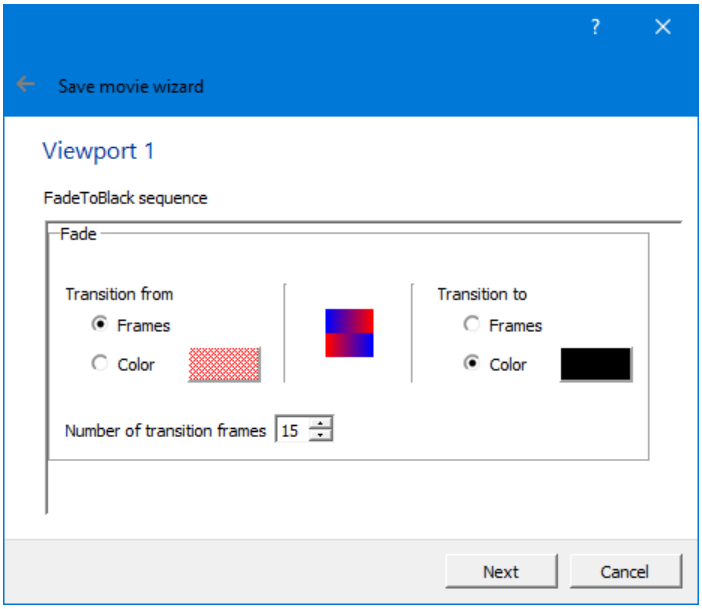


Fig. 4.193: Slice 3D fade sequence options.

Creating new Templates

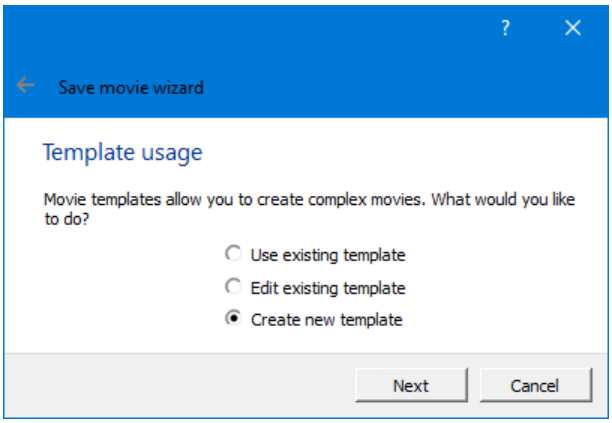


Fig. 4.194: Creating a new template

Side by Side Movie

To make a side by side movie you can use the **Create template** option. Before saving the movie with this template, you must set up the plots that you want to make in multiple vis windows, as shown in Figure 4.195.

Since you are creating a new movie template, you first must set up the movie viewports. You can create many viewports and the viewports can be adjacent or can overlap (e.g. picture in picture). When viewports overlap, you can make the top viewports semi-transparent, make a background color transparent, or even add drop shadows. There are some existing layouts if you don't want to get fancy. Select the **Side by Side** layout and VisIt will create 2 viewports next to one another.

After you create movie viewports, you must map sequences of images to the viewports. These can be produced from

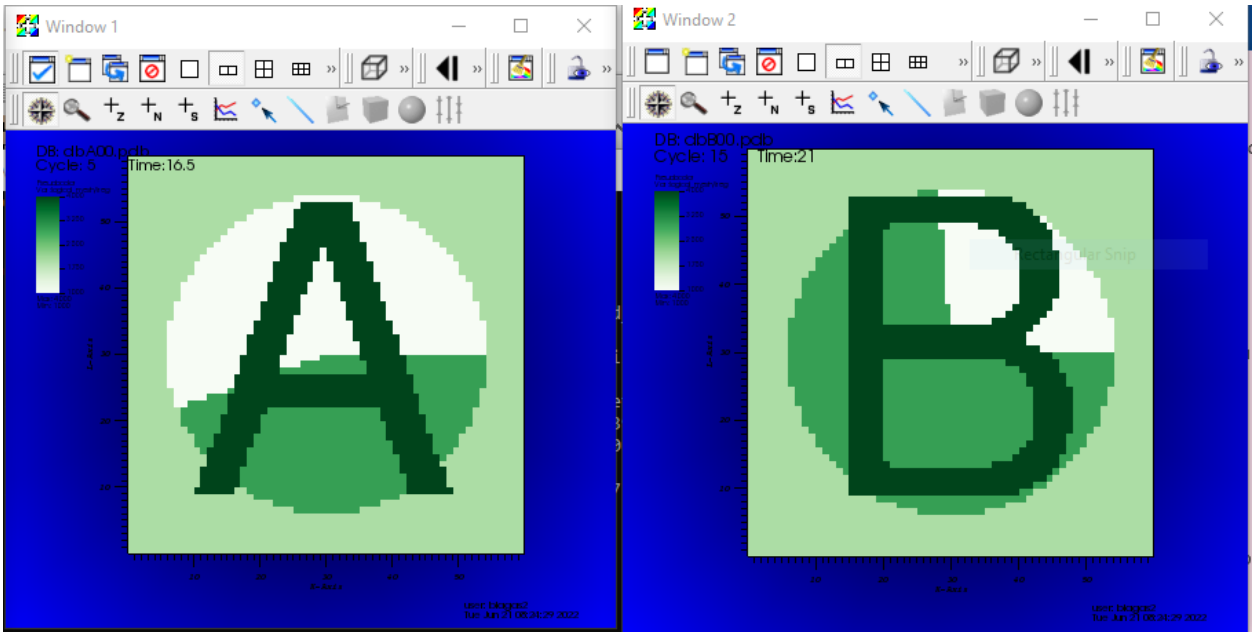


Fig. 4.195: Setting up plots for Side by Side movie generation.

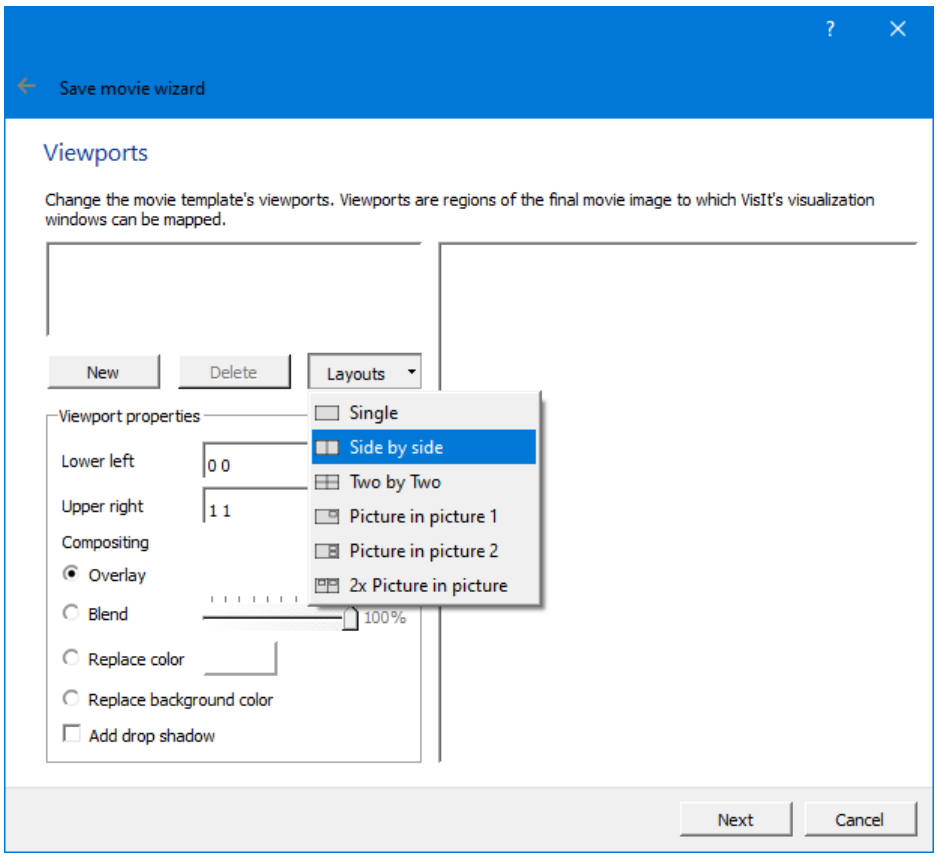


Fig. 4.196: Selecting Side by Side layout.

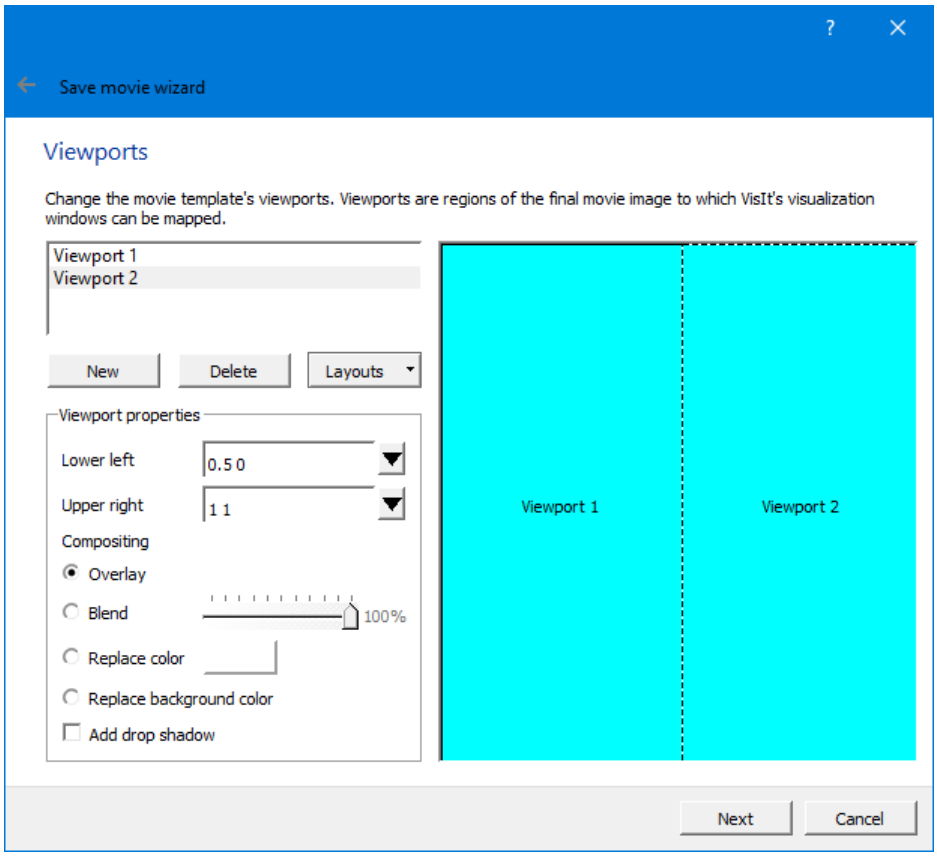


Fig. 4.197: The Side by Side viewports.

the plots in your vis windows or transitions such as fades, etc. For a side by side movie, you will want to map the image sequence produced from vis window 1 to movie viewport 1. You will also map the image sequence from vis window 2 to movie viewport 2. If you had more vis windows, you could map more than one vis window to a movie viewport. When more than one vis window is mapped to a single viewport, the movie will have the animation form the first vis window followed by the animation from the second vis window and so on for all the vis windows mapped to that viewport. It is possible to add transition effects in between sequences or after sequences so that you can include some simple animation effects in your movies.

Tip: If you wanted to show side by side plots that advance through time and then switch to another plot and do the animation over again, you could set up 4 vis windows and then map 2 vis windows to each movie viewport.

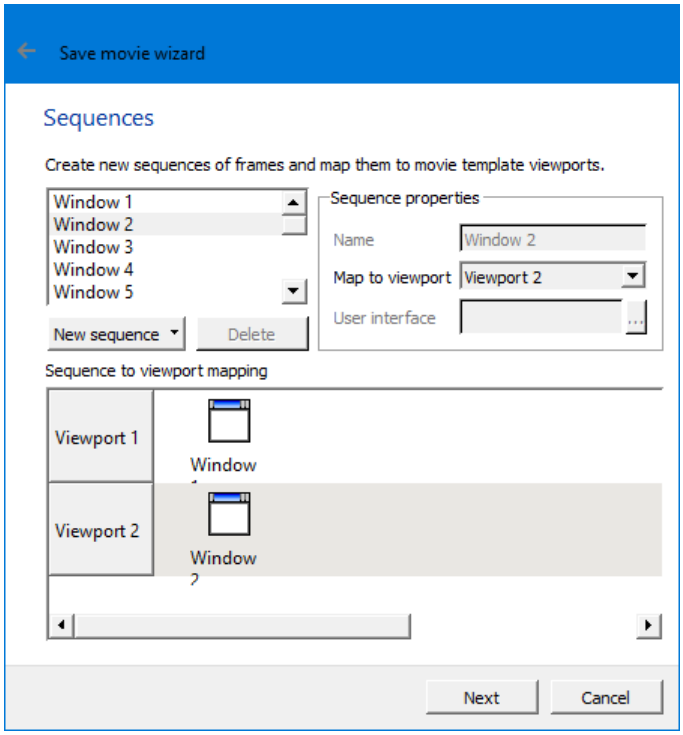


Fig. 4.198: Mapping Windows to viewports.

For the time being, you will not save the movie template but simply use it.

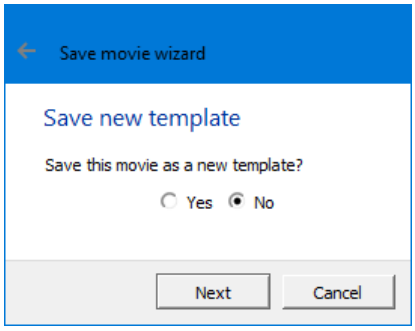


Fig. 4.199: Save template page.

Now that you have set up the movie template, the rest of the screens in the Save movie wizard are the normal screens that gather input about which movie formats you would like to save. Note that for a side by side movie, you will want to override the image size so that you do not get a square movie but rather 2 square images next to one another. Use

an image size where the width is 2x the height: 1000x500 or some other image size.

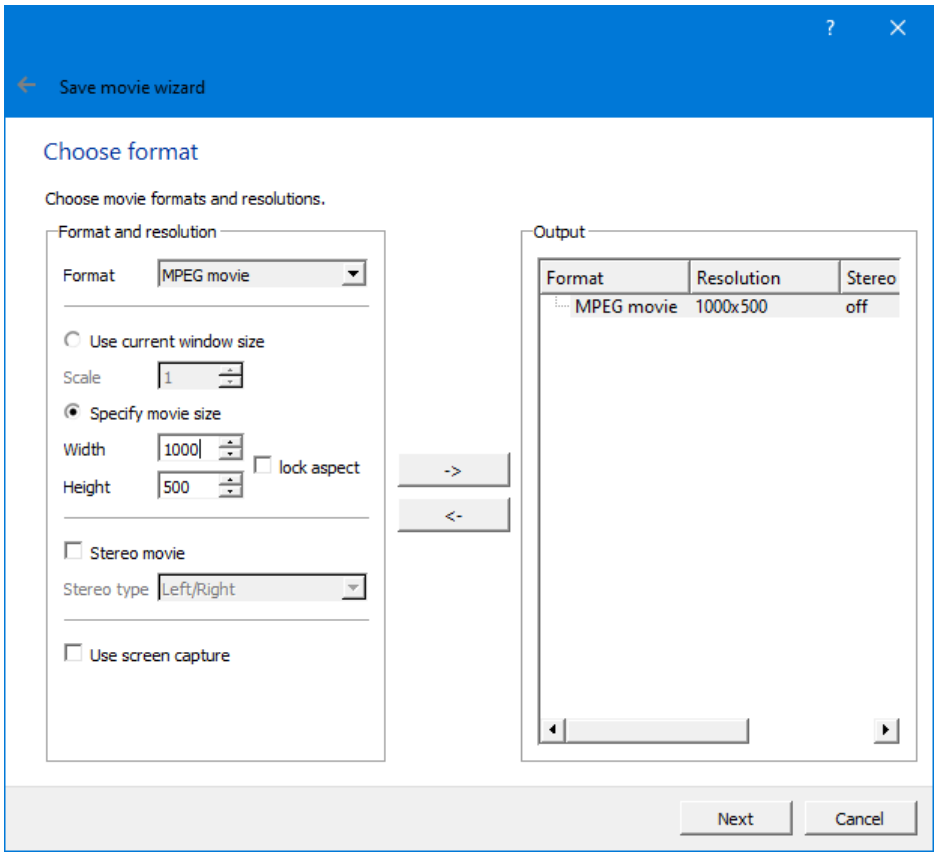


Fig. 4.200: Choose image size where width is 2x height.

The result from this movie template will be a side by side MPEG movie that contains the plots from vis window 1 and vis window 2.

4.5.4 Saving Cinema

VisIt lets you save Cinema databases in addition to saving images and movies of your plots. A Cinema database is an image-based proxy for large scale data that lets you explore the data using far fewer computational resources. Where post-processing full data might take a supercomputer, exploring a Cinema database can be done on a tablet. Cinema databases consist of images that are indexed by a JSON file or CSV file. The index file is used by the Cinema viewer (available at www.cinemascience.org) to determine a set of parameters that can be changed by the user. These parameters are used to look up corresponding image files for display in the Cinema viewer. For example, Cinema databases typically allow the user to navigate through time using a time parameter. Cinema databases also can be saved using a spherical camera that is described by phi and theta parameters to let the user see the plots from various camera angles. It is possible to create Cinema databases in situ using Libsim so Cinema databases can be created incrementally as a simulation runs. This section introduces the **Save Cinema** wizard and explains how to create Cinema databases from within VisIt's GUI.

The **Save Cinema** wizard (see Figure 4.201) is available in the **Main Window's Files** menu. The **Save Cinema** wizard's purpose is to let you set the options that are used to take the current visualizations and produce a Cinema database. Progress through the screens using the **Next** button until the last screen is reached. Clicking **Cancel** at any time will close the wizard. Clicking the **Finish** button will tell VisIt to produce a Cinema database with the current settings.

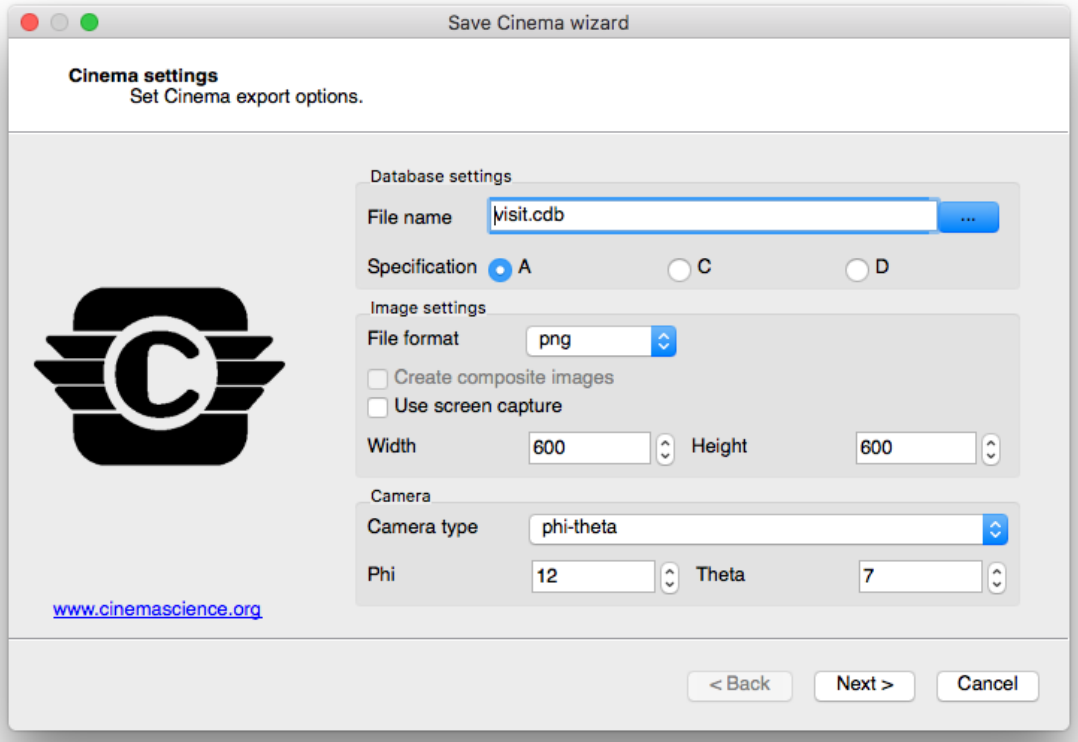


Fig. 4.201: Save Cinema wizard (screen 1)

Choosing filename

Cinema databases are stored as a directory structure containing various nested directories with image files and an index file. When saving a Cinema database, you must pick the name of the top level directory under which all other files will be saved. The **Save Cinema** wizard contains a **File name** selection control that lets you select the name of the Cinema ".cdb" directory. The control can accept file names that are typed in and clicking the **...** button opens a filename selection window that permits a new filename to be selected.

Choosing specification

Cinema databases are described by specifications that dictate the format and allowable contents for the files that they contain. There are currently 3 Cinema specifications in use: A, C, D.

Specification A describes a Cinema database format that contains image files (PNG, TIFF, etc.) that are associated with various user-defined parameters such as time or camera angles in the case of a phi-theta camera. This specification is compatible with any of the VisIt plots since images of the currently set up visualizations are saved. Specification C describes a Cinema database format that adheres to a different directory structure over specification A and can contain composite images. Composite images are comprised of 3 separate files: a PNG file containing a luminance image, a ZLib-compressed file containing the Z-buffer, and a ZLib-compressed file containing a rendering of actual scalar values for the plot. Specification D is similar to specification A except that it uses a CSV file to associate image files with a set of parameters, enabling sparse sets of images.

The **Save Cinema** wizard contains a set of A, C, D radio buttons to let you choose the most appropriate specification for the type of Cinema database to be created.

Image settings

The **Save Cinema** wizard contains controls for image settings such as the file format, image width/height, and whether to use screen capture. The **File format** control lets you select the image file format to be used. Several pixel-based image file formats are available such as BMP, PNG, TIFF, and when available EXR. OpenEXR is a format from ILM that can store various image channels and data in multiple layers that can be composited later. Support for OpenEXR is optionally compiled into VisIt. The **Width** and **Height** controls allow the output image width and height to be specified when screen capture is not in use by setting the **Use screen capture** controls. This permits VisIt to save images in a custom size as opposed to saving images based on the current visualization window's size. Note that using screen capture is faster for normal images since it does not require VisIt to re-render the visualizations.

Composite images

Specification C Cinema databases support saving composite images which consist of a luminance image, a Z image, and a scalar image. The luminance image is a gray scale image that indicates the lighting used in the scene and it is saved as a PNG image or other pixel format image. The Z image is contains the Z-buffer for the luminance image, stored as a buffer of 32-bit floating point values that have been ZLib-compressed and written to a raw binary file. The scalar image is stored the same as the Z buffer image but it contains float values that correspond to the actual scalars that were rendered in the visualization. The scalar values are used in the Cinema viewer to dynamically recolor the scene at render time. Composite images are most appropriate for surface-based VisIt plots that employ a continuous color table, such as the Pseudocolor plot. Composite images can be enabled by turning on the **Create composite images** check box in the **Save Cinema** wizard when specification C is used. When this setting is in effect, each VisIt plot will be saved to a separate "layer" in the Cinema database so it can be composited into the scene at will. Figure 4.203 shows multiple VisIt plots that have been saved as separate layers to a composite image specification C Cinema database that enables layers to be turned on and off at view time.

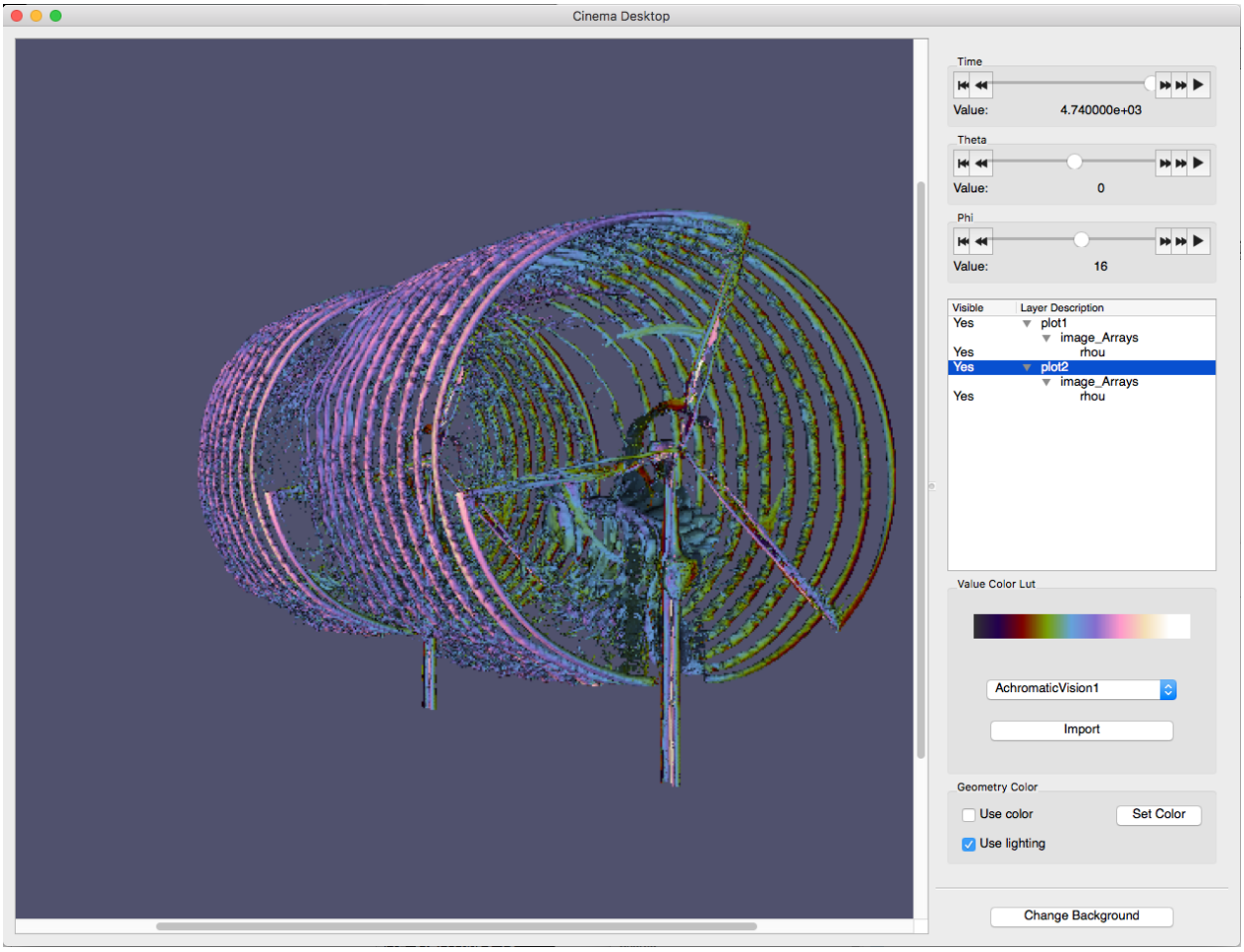


Fig. 4.202: Cinema viewer with composite layers

Choosing Camera type

Cinema databases support multiple camera types. VisIt's Cinema export supports static cameras and phi-theta cameras. A static camera corresponds to the view that is currently in effect in the visualization and when it is used, all time states in the Cinema database will be viewed from that camera orientation. A phi-theta camera defines 2 angles, phi and theta, that define the view direction as in a spherical coordinate system. When a phi-theta camera is used, the Cinema export will save the visualization from a multitude of different camera orientations. This allows the user later in the Cinema viewer to interactively rotate around the object much as though the object was live instead of just a collection of image frames. The camera type can be selected using the **Camera type** control in the **Save Cinema** wizard and either static or phi-theta cameras can be selected. When a phi-theta camera is selected, the number of camera angles in the phi and theta dimensions can be set using the **Phi** and **Theta** controls.

Frame settings

The second tab in the **Save Cinema** wizard (see Figure 4.203) contains controls that select the range and stride of time states that will be included in the Cinema database. Use the **Frame start** controls to select the beginning time state for the Cinema database. A value of zero corresponds to the first time state. Use the **Frame end** controls to set the last time state that will be included in the Cinema database. Finally, use the **Frame stride** controls to set the stride that will be used between the start and end time states, which is useful when making shorter preview databases that vary over time but do not include all time states.

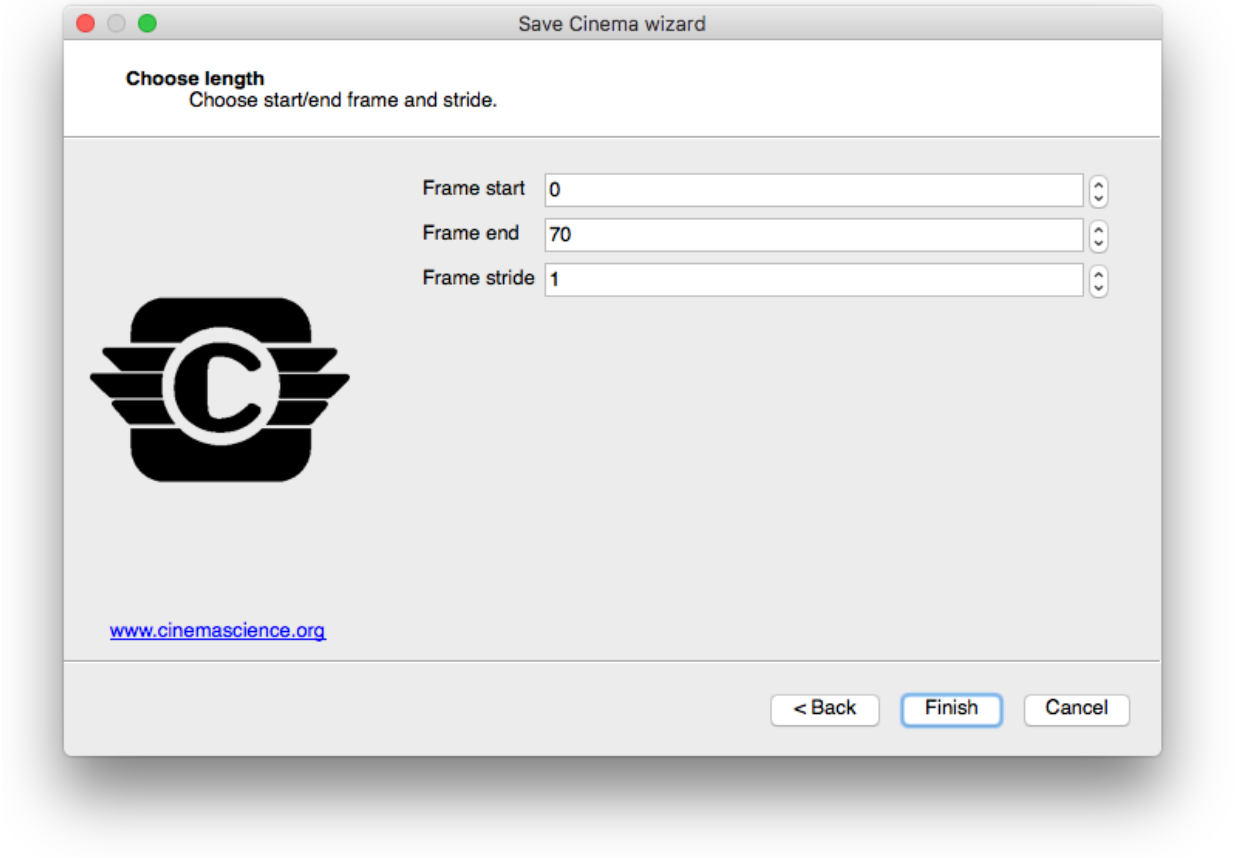


Fig. 4.203: Save Cinema wizard (screen 2)

Saving Cinema from Libsim

It is possible to use VisIt's Libsim to directly save Cinema databases in situ from an instrumented simulation. This means that the Cinema database can be generated incrementally as the simulation runs, making it possible to periodically check in on the simulation by viewing the Cinema database. To add Cinema support to a simulation instrumented with Libsim, there are 3 calls that need to be made. First, the simulation must call `VisItBeginCinema`, which passes the parameters that describe the Cinema database format and returns a handle to a Cinema object. Next, the simulation must call `VisItSaveCinema` to make Libsim generate and add the appropriate images to the Cinema database, taking into account the type of camera being used. The `VisItSaveCinema` function can be called repeatedly to add new time states to the Cinema database. It is the simulation's responsibility to make Libsim calls that set up VisIt plots or restore a session so there are plots when `VisItSaveCinema` is called. Finally, the simulation must call `VisItEndCinema` to close out the Cinema database context and free associated memory. A working example can be found in the batch simulation example in VisIt's simulation directory. The overall call structure for creating a Cinema database looks something like this:

```
visit_handle h = VISIT_INVALID_HANDLE;
visit_handle hvar = VISIT_INVALID_HANDLE;
double time_value = 0.;
VisItBeginCinema(&h, "visit.cdb", VISIT_CINEMA_SPEC_A, 0,
                 VISIT_IMAGEFORMAT_PNG, 800, 800,
                 VISIT_CINEMA_CAMERA_PHI_THETA, 12, 7,
                 hvar);

while(1) /* Simulation main loop */
{
    /* Compute... */

    VisItSaveCinema(h, time_value);

}

VisItEndCinema(h);
```

The above code example will generate a Cinema database using the plots that have been set up elsewhere using Libsim. Since Cinema output may sometimes serve as the only simulation data product, it can be useful to save out additional variables. The last argument to `VisItBeginCinema` is a handle to a name list object. When the handle is set to `VISIT_INVALID_HANDLE`, there is no name list and the argument does nothing. If instead, the name list is created and filled with a list of variable names from the simulation, the VisIt plots will have their variables changed to the variables in the name list and Libsim will generate a Cinema database with images for each variable. The variable becomes a parameter in the Cinema viewer. A name list object is created and populated like this:

```
visit_handle hvar;
VisIt_NameList_alloc(&hvar);
VisIt_NameList_addName(hvar, "pressure");
VisIt_NameList_addName(hvar, "rho");
VisIt_NameList_addName(hvar, "energy");
```

4.5.5 Exporting databases

Plot geometry can be saved to a handful of geometric formats by saving the plots in the window to a format such as VTK. Often saving the plot geometry, which only consists of the visible faces required to draw the plot, is not enough. When interfacing VisIt to other tools you may want to save out the database in a different file format. For instance, you might plot a 3D database and want to export actual 3D cells for the entire database instead of just the externally visible geometry. You might also want to save out additional variables that you did not plot. VisIt allows this kind of data export via the **Export Database Window** , shown in Figure 4.204 .

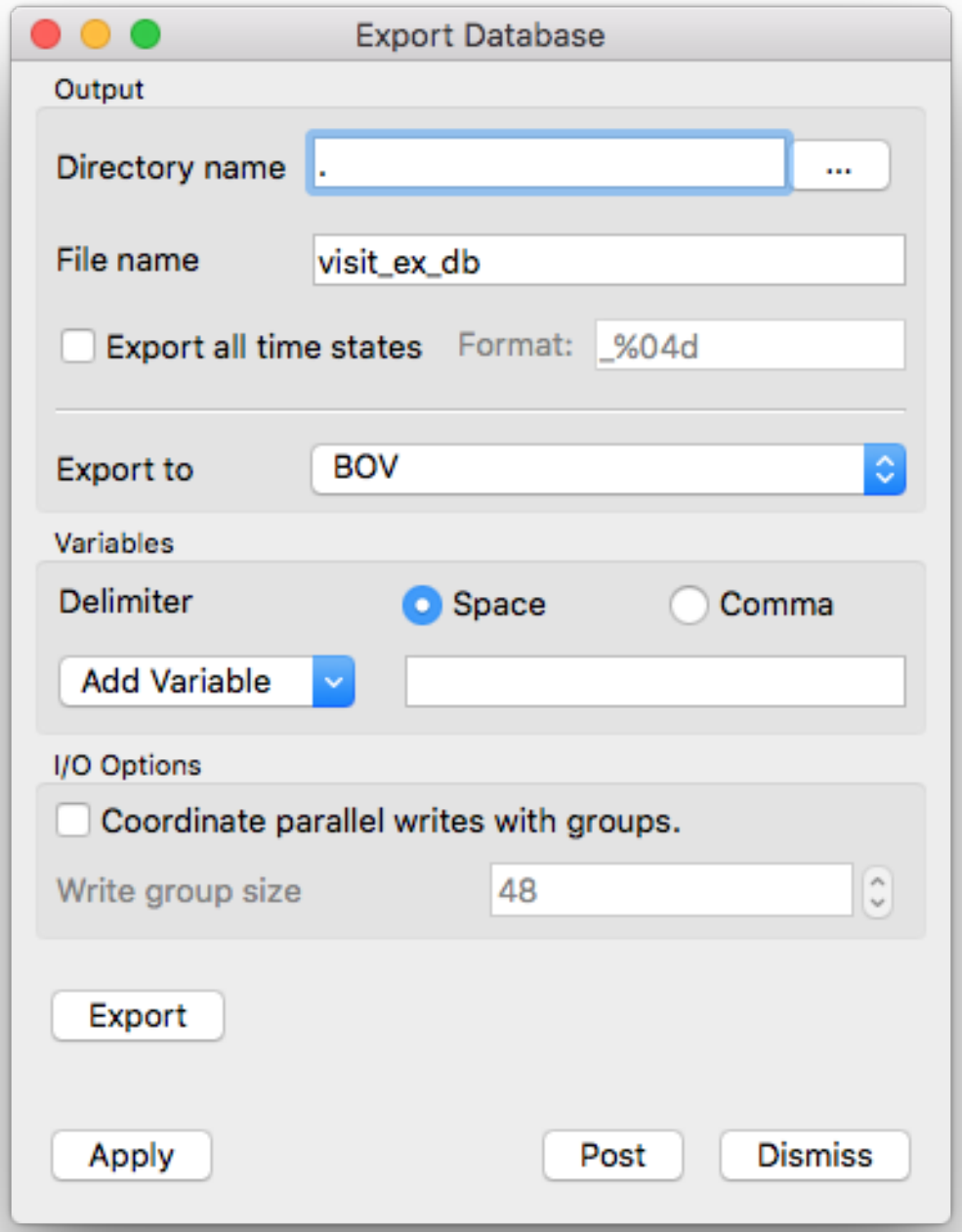


Fig. 4.204: Export Database Window

You can find the **Export Database Window** in the **Main Window's File** menu. To save a database, you must first have opened a database and created a plot. Note that the data transformations applied by plots or operators will affect the data that you export. This allows you to alter the data using sophisticated chains of operators before you export it for use in another tool.

Exporting variables

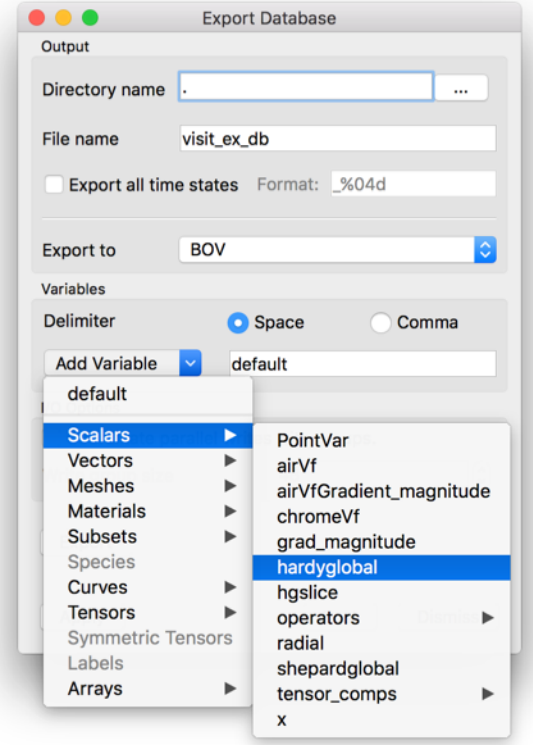


Fig. 4.205: Variables menu

The **Export Database Window** allows you to export a subset of the variables for your active plot's database by letting you specify which variables are to be exported. To choose which variables should be exported, you can type the names of the variables to export into the **Variables** text field or you can select from the available variables in the **Variables** menu depicted in Figure 4.205 . You can select as many variables as you want from the menu. Each time you select a variable from the **Variables** menu, VisIt will append it to the list of variables to be exported.

Choosing an export file format

The **Export Database Window** lists the names of the database reader plugins that can also write data back into their native file formats. A small handful of the total number of database plugins currently support this feature but in the future most formats will support this capability more fully, making VisIt not only a powerful visualization tool but a powerful database conversion tool.

You can try to use any of the supported export formats to export your data but some of the file formats may not be able to accept certain types of data. The Silo file format can safely export any type of data that you may want to export. If you want to export data to other applications and the data must be stored in an ASCII file that contains columns of data, you might want to choose the Xmdv file format. If you want to choose a specific database plugin to export your data files, make a selection from the **Export to** menu shown in Figure 4.206 .

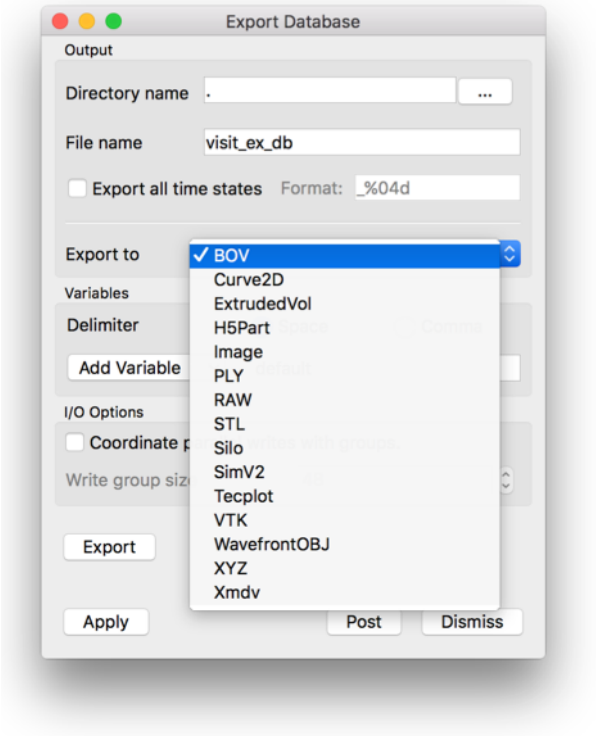


Fig. 4.206: Export file types

Export Options

Some export formats support various options. Those options will be presented in a dialog box when the **Export** button is pressed in the **Export Database Window**. For example, shown below are some options for exporting to the Silo database.

If VisIt has been compiled with HDF5 support, Silo's export options will include the ability to select either the `PDB` or `HDF5` driver. The `Checksums` check-box indicates where the Silo library should compute checksums on the exported data. In addition, the **DBSetCompression()** option text box is for specifying a compression string to be used in Silo's `DBSetCompression()` method before exporting data.

When the meaning of an export option is not clear, try also pressing the **Help** button in **Export options for XXX writer** window to get more information.

4.5.6 Printing

VisIt allows you to print the contents of any visualization window to a network printer or to a *PostScript* file.

The Printer Window

Open the **Printer Window** by selecting **Print window** from the **Main Window's File** menu. The **Printer Window's** appearance is influenced by the platform on which you are running VisIt so you may find that it looks somewhat different when you use the Windows, Unix, or MacOS X versions of VisIt. The MacOS X version of the **Printer Window** is shown in Figure 4.208 .

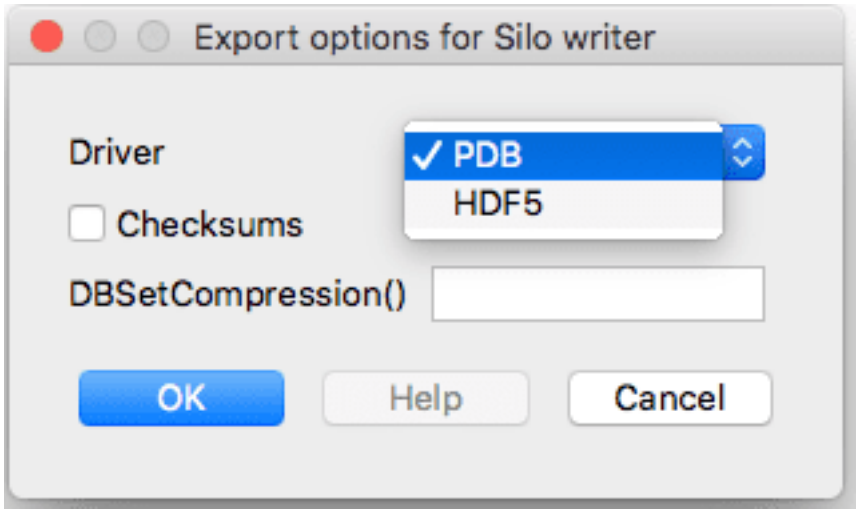


Fig. 4.207: Export options example (for Silo)

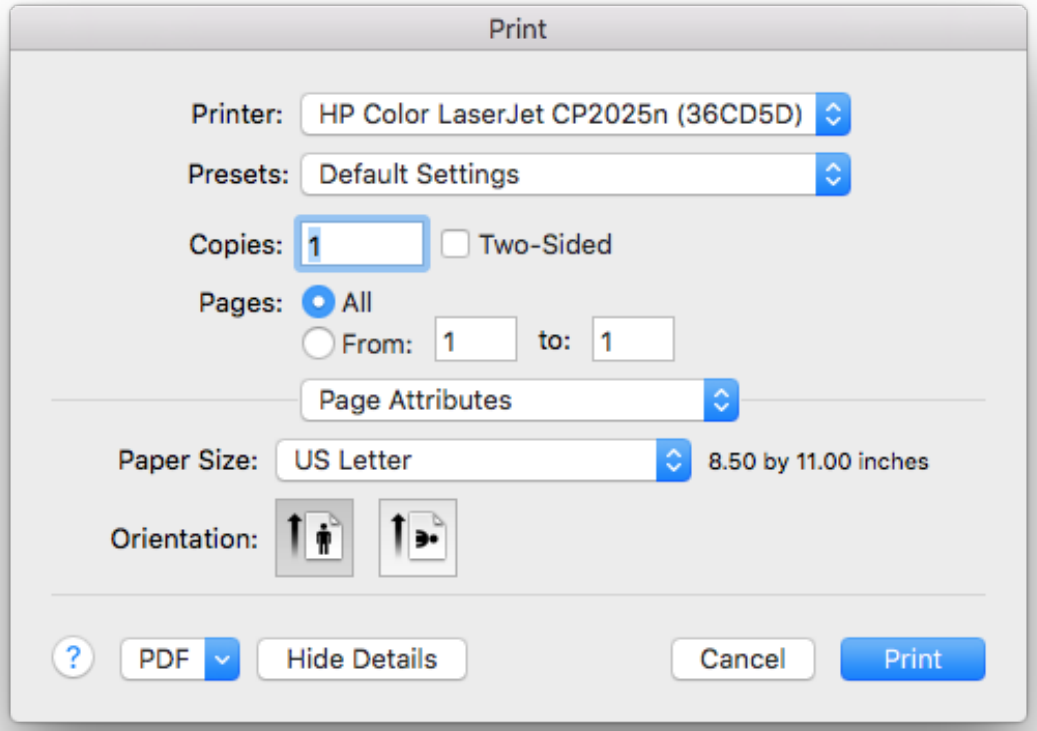


Fig. 4.208: Printer window

4.6 Visualization Windows

A visualization window, also known as a vis window, is a window that displays plots and allows you to interact with them using the mouse. The vis window not only allows for direct manipulation of plots but it also provides a popup menu and toolbar that allow you to switch window modes, activate interactive tools, and perform commonly used operations. This chapter explains how to manage and use vis windows.

4.6.1 Managing vis windows

VisIt allows you to create up to 16 vis windows and to manage those vis windows, VisIt provides controls to add vis windows, remove vis windows or alter their layout. The controls for managing vis windows are located in the **Main Window's Windows** menu (see Figure 4.209), as well as in the vis window's **Toolbars** and **Popup menu**.

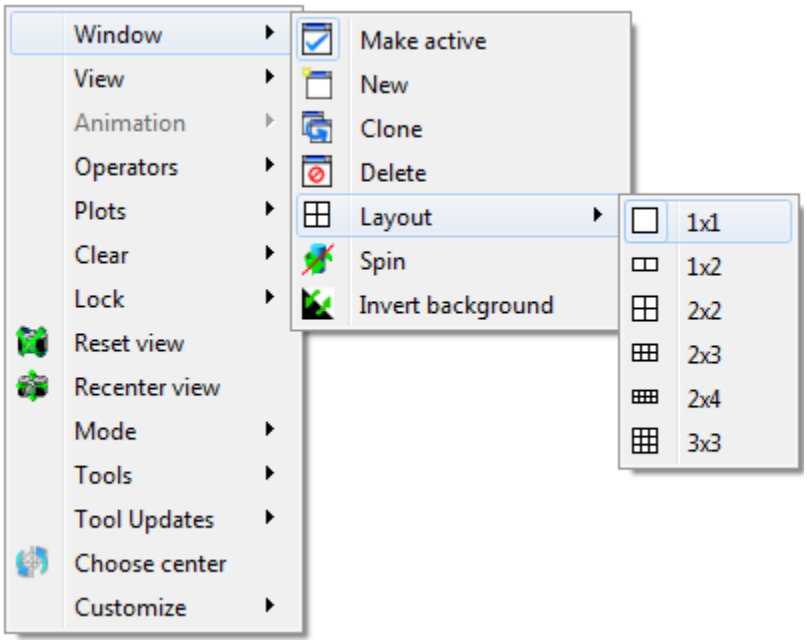


Fig. 4.209: Window menu

Adding a new vis window

You can add a new vis window in a few different ways, the first of which is by selecting the **New** option from the **Main Window's Windows** menu. You can also click on the **New window icon** in the vis window's **Toolbar** or you can select the **New window** option from the **Windows** submenu in the vis window's **Popup menu** to add a new vis window. When you add a new window, it will be sized according to the window layout so if you have only a single, large vis window, the new vis window will also be large. You can change the window layout to shrink the vis windows so that they both fit on the screen. Vis windows are numbered 1 to 16 so the new window will have the first available number for which there is not already a window. If you have windows 1, 2, and 4, vis window 3 would be created by adding a new window. Adding a new window also makes the new window the active window.

A new vis window can also be added by cloning the active window. You can clone the active window by selecting the Clone option from the **Main Window's Windows** menu or you can click the **Clone window icon** in the vis window's **Toolbar**. When you clone the active window, VisIt creates a new window as if you had clicked the Add option but it also copies the plots, annotations, and lighting from the active window so that the new window is identical in appearance to the active window. When plots are copied to the new cloned window, they have not yet been generated

so their plot list entries in the **Plot list** are green. You can force the plots to be generated by clicking the **Draw** button in the **Main Window**.

Deleting a vis window

There are four ways to delete a vis window. The first way is to select the **Delete** option from the **Main Window's Windows** menu. When you delete a window in this manner, the active window gets deleted and VisIt makes the window with the smallest number the new active window. The second way to delete a window is to click on the **close window button** in the window decorations provided by the windowing system. The window decorations' appearance varies based on the platform and windowing system used to run VisIt, but the button used to close windows is commonly a button with an X in it. An example of a **close window button** is shown in Figure 4.210.

Fig. 4.210: Window decorations with close button

The third way to delete a vis window is to click on the **Delete window icon** in the vis window's **Toolbar**. The fourth way to delete a vis window is to use the **Delete** option in the vis window's **Popup menu**. When you use the **Toolbar** or the **Popup menu** to delete a window, the window does not need to be the active window as when other controls are used.

Clearing plots from vis windows

The **Main Window's Windows** menu provides a **Clear all** option that you can use to clear the plots from all vis windows. Selecting this option does not delete the plots from a vis window's plot list but it does clear the plots so they have to be regenerated by VisIt's compute engine. You can also clear the plots for just the active window by selecting the **Plots** option from the **Clear** submenu in the **Main Window's Windows** menu (see Figure 4.211). You might find clearing plots useful when you want to make several changes to plot attributes because, unlike plots that are already generated, setting attributes of cleared plots does not force them to regenerate when you change their attributes.

In addition to clearing plots, you can also clear pick points and reference lines from a vis window. A pick point is a marker that VisIt adds to a vis window when you click on a plot in pick mode. The marker indicates the location of the pick point. A reference line is a line that you draw in a vis window when it is in lineout mode. You can clear a vis window's pick points or reference lines, by selecting the **Pick points** or **Reference lines** options from the **Clear** submenu in the **Main Window's Windows** menu.

Changing window layouts

VisIt uses different window layouts to organize vis windows so they all fit on the screen. Changing the window layout typically resizes all of the vis windows and moves them into a tiled formation. If there are not enough vis windows to complete the desired layout, VisIt creates new vis windows until the layout is complete. You can change the layout selecting a new layout from the **Layouts** menu located in the **Main Window's Windows** menu or you can click on a layout icon in the vis window's **Toolbar**.

Setting the active window

VisIt has the concept of an active window that is the window to which new plots are added. You can change the active window by selecting a window number from the **Active window** menu located near the top of the **Main Window**. Setting the active window updates the GUI so that it displays the state for the new active window. The **Active window** menu is shown in Figure 4.212. You can also set the active window using the **Active window** submenu in the **Main Window's Windows** menu or you can click on the **Active window icon** in the vis window's **Toolbar**.

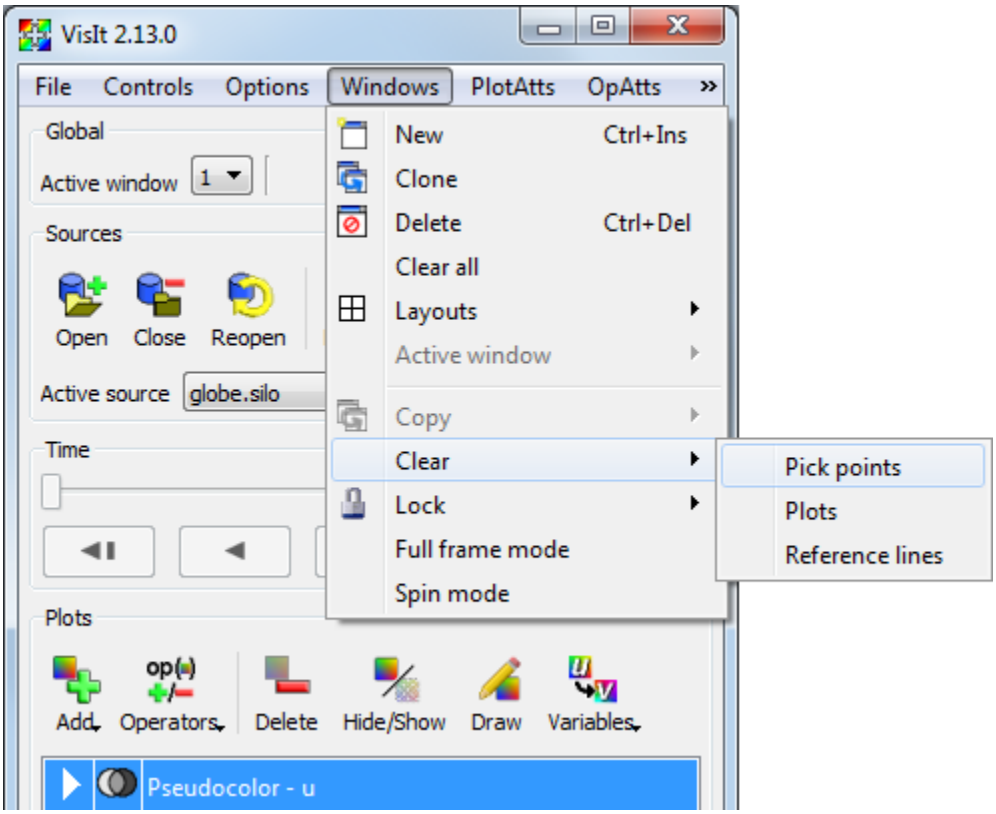


Fig. 4.211: Clear menu

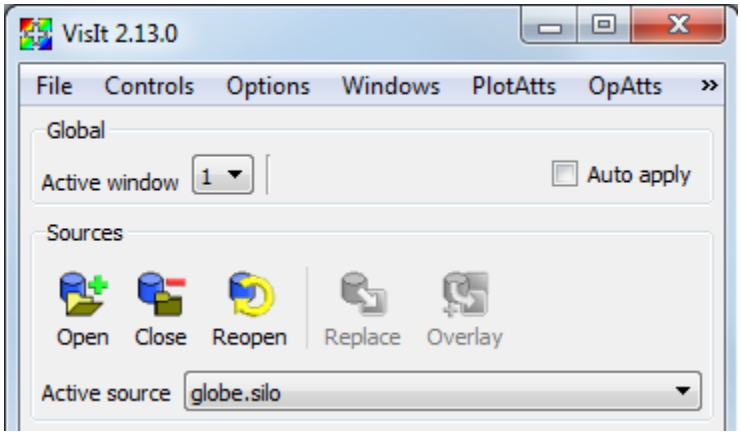


Fig. 4.212: Active window menu

Copying window attributes

VisIt allows you to copy window attributes and plots from one window to another when you have more than one window. This can be useful when you are comparing plots generated from similar databases. The **Copy** menu, shown in Figure 4.213, contains options to copy the view, lighting, annotations, plots, or everything from other from other vis windows. Under each option, the **Copy** menu provides a list of available vis windows from which attributes can be copied so, for example, if you have two windows and you want to copy the view from vis window 1 into vis window 2, you can select the **Window 2** option from the **View from** submenu. The list of available windows depends on the vis windows that you have created. You can copy the lighting from one window to another window by using the **Lighting from** submenu or you can use the **Annotations from** or **Plots from** to copy the annotations or plots, respectively. If you make a selection from the **Everything from** submenu, all attributes and plots are copied into the active vis window.

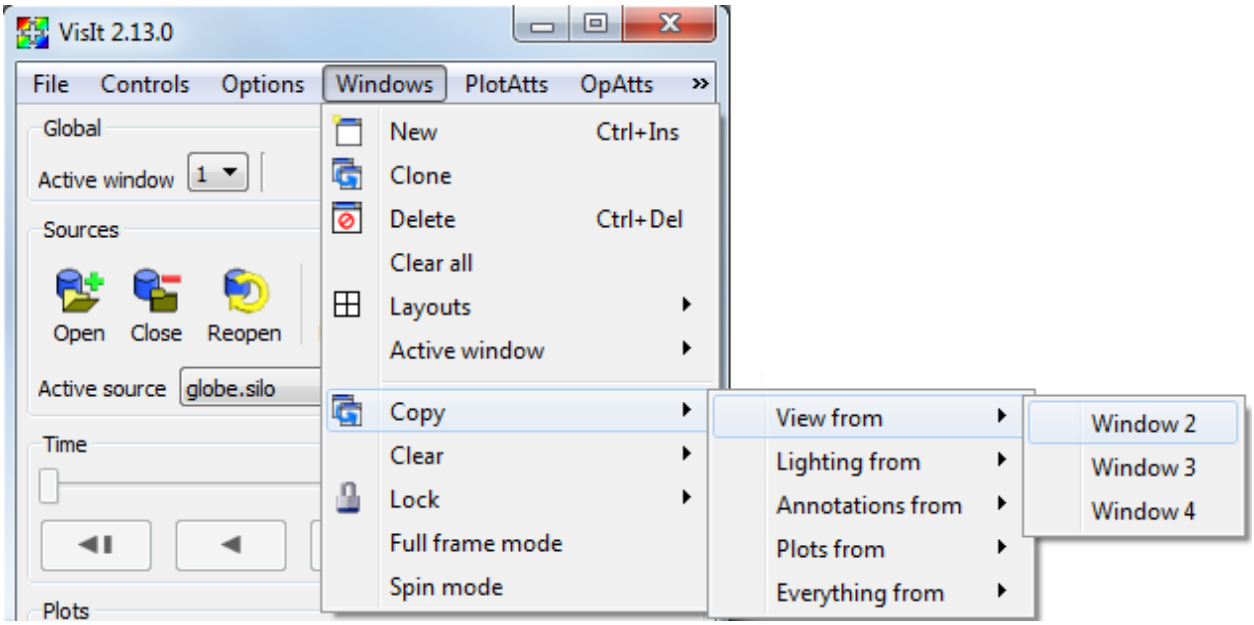


Fig. 4.213: Copy menu

Locking vis windows together

When you use VisIt to do side by side comparisons of databases, you may find is useful to lock vis windows together. Vis windows can be locked together in time so that when you change the active database timestep in one database, as when viewing an animation, all vis windows that are locked in time switch to the same database timestep. You can lock vis windows together in time by selecting the **Time** option from the **Lock** menu (see Figure 4.214) in the **Main Window's Windows** menu. Any number of windows can be locked together in time and you can turn off time locking at any time.

You can also lock interactive tools together so that updating a tool in one window updates the tool in other windows that have enabled tool locking. This can be useful when you have sliced a database using the plane tool in more than one window and you want to be able to change the slice using plane tool in either window and have it affect the other vis windows. You can enable tool locking by selecting the **Tools** option from the **Lock** menu.

In addition to locking vis windows in time, or locking their tools together, you can also lock vis windows' views together so that when you change the view in one vis window, other vis windows get the same view. When you change the view in a vis window that has view locking enabled, the view only effects other vis windows that also have view locking enabled and have plots of the same dimension. That is, when you change the view of a vis window that

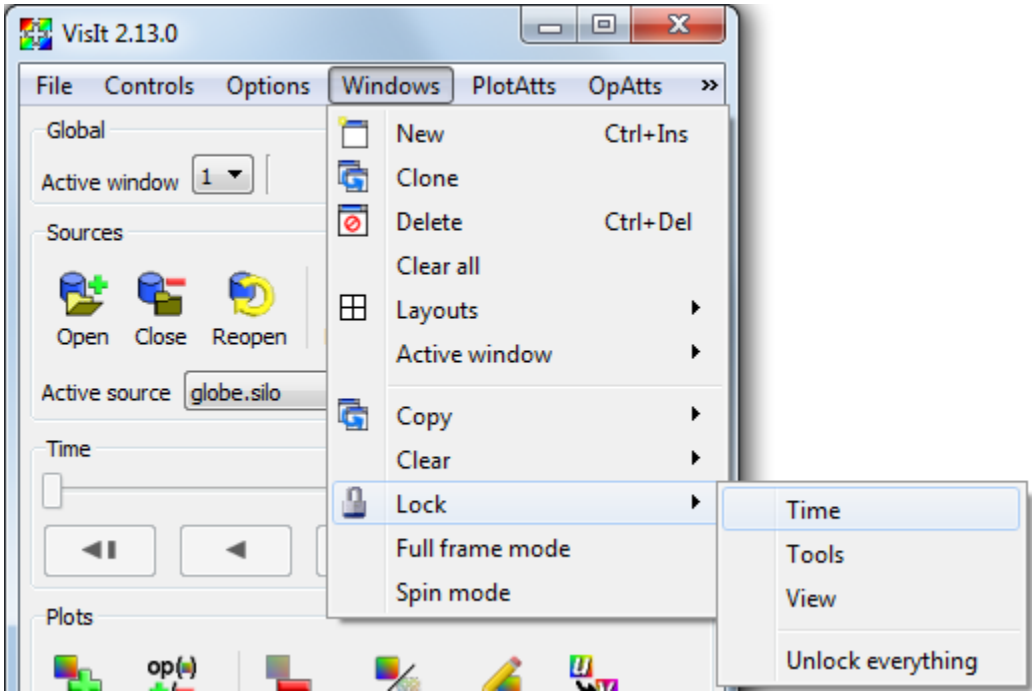


Fig. 4.214: Lock menu

contains 3D plots, it will only have an effect on other locked vis windows if they have 3D plots. Vis windows that contain 2D plots are not affected by changing the view of a vis window containing 3D plots and vice-versa. When you enable view locking, the vis window snaps to the view used by other vis windows with locked views or it stays the same if no other vis windows have locked views. To enable view locking, select the **View** option from the **Lock** menu or click on the **Lock view icon** in the vis window's **Toolbar**.

4.6.2 Using vis windows

The first thing to know about using a vis window is how to change window modes. A window mode is a state in which the vis window behaves in a specialized manner. There are four window modes: Navigate, Zoom, Lineout, and Pick. Vis windows are in navigate mode by default. This means that most mouse actions are used to move, rotate, or zoom-in on the plots that the vis window displays. Each vis window has a **Popup menu** that can be activated by clicking the right mouse button while the mouse is inside of the vis window. The **Popup menu** contains options that can put the vis window into other modes and perform other common operations. To put the vis window into another window mode, open the **Popup menu**, select **Mode** and then select one of the four window modes. You can also change the window mode using the vis window's **Toolbar**, which has buttons to set the window mode. You can find out more about the **Popup menu** and **Toolbar** later in this chapter.

Navigate mode

Navigate mode is VisIt lingo for moving and zooming-in on plots. When the vis window is in navigate mode, clicking the left mouse button and dragging with the mouse will perform an action that moves, rotates, or zooms the plot. The mouse motions used to rotate plots are shown in Figure 4.215. You can translate plots by holding down the *Shift* key before left-clicking and dragging the plot. You zoom in on plots by clicking the middle button and moving the mouse up or down. Sometimes the controls are modified based on the interactor settings. For more information, look at the section on *Interactor settings*.

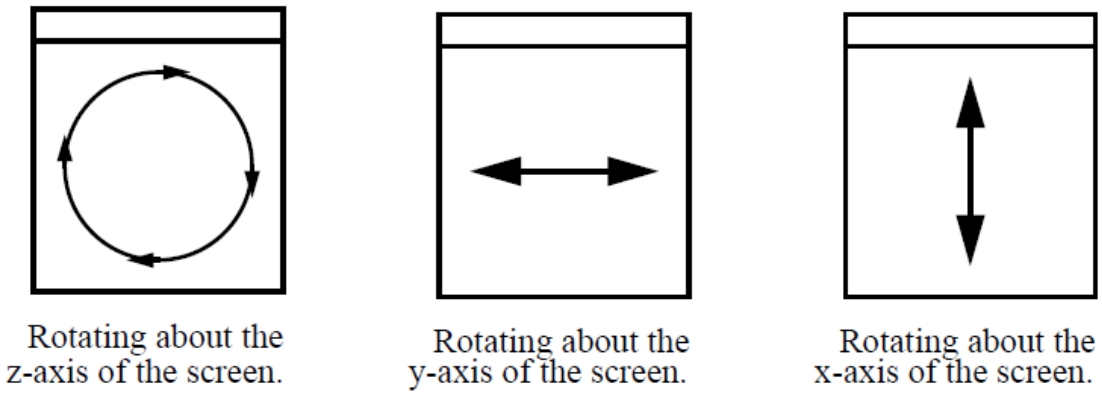


Fig. 4.215: Mouse motions used to rotate plots in navigate mode

Zoom mode

When the window is in zoom mode, you can draw a box around the area of the vis window that you want drawn larger. Press the left mouse button and move the mouse to sweep out a box that will define the area to be zoomed. Release the mouse button when the zoom box covers the desired area. If you start zooming and decide against it before releasing the left mouse button, clicking one of the other mouse buttons cancels the zoom operation. Changes to the view can be undone by selecting the **Undo view** option from the popup menu's **View** menu. Sometimes the zoom controls can change based on the interactor settings, which are described further on in Interactor settings.

Lineout mode

Lineout mode is only available when the vis window contains 2D plots. A lineout is essentially a slice of a two dimensional dataset that produces a one dimensional curve in another vis window. When a vis window is in lineout mode, pressing the left mouse button in the vis window creates the first endpoint of a line that will be used to create a curve. As you move the mouse around, the line to be created is drawn to indicate where the lineout will be applied. When you release the mouse button, VisIt adds a lineout to the vis window and a curve plot is created in another vis window.

Pick mode

When a vis window is in pick mode, any click with the left mouse button causes VisIt to calculate the value of the plot at the clicked point and place a pick point marker in the vis window to indicate where you clicked. The calculated value is printed to the **Output Window** and the **Pick Window**.

4.6.3 Interactor settings

Some window modes such as Zoom mode and Navigate mode have certain interactor properties that you can set. Interactor properties influence how user interaction is fed to the controls in the different window modes. For example, you can set zoom interactor settings that clamp a zoom rectangle to a square or fill the viewport when zooming. VisIt provides the **Interactors window** so you can set properties for window modes that have interactor properties. The **Interactors window** is shown in Figure 4.216.

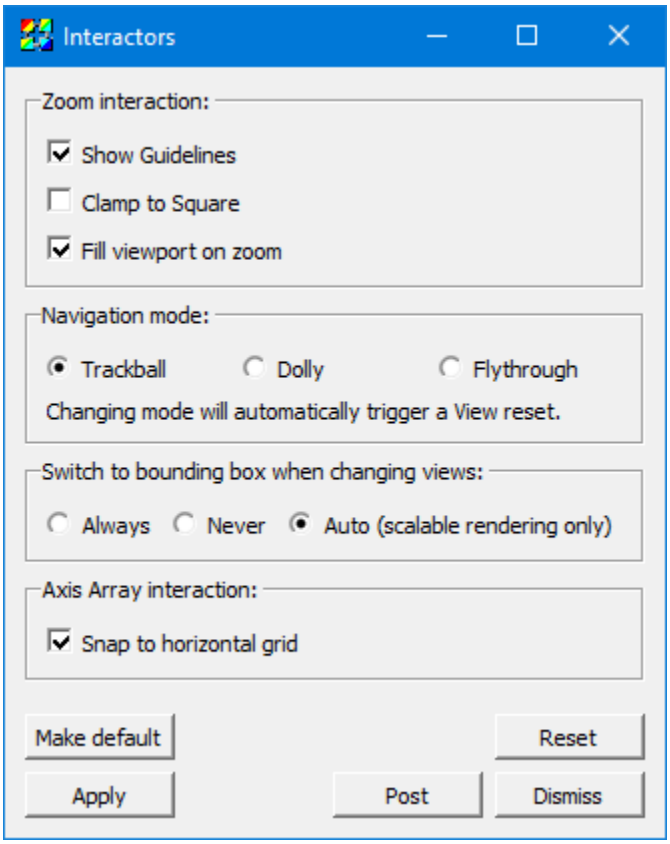


Fig. 4.216: Interactors window

Zoom interactor settings

The zoom interactor settings are mostly used when the vis window is in zoom mode. When the vis window is in zoom mode, clicking in the vis window will anchor a point that becomes one of the corners of a zoom rectangle. When you release the mouse, the point over which the mouse was released becomes the opposite corner of the zoom rectangle. VisIt's default behavior is to show guidelines that extend from the edges of the zoom rectangle to the edges of the plots' bounding box when the vis window is in 2D mode. If you want to turn off the guidelines, click off the **Show Guidelines** check box in the **Interactors window**.

When sweeping out a zoom rectangle in zoom mode, VisIt allows you to draw a rectangle of any proportion. The relative shape of the zoom rectangle, in turn, influences the shape of the viewport drawn in the vis window. If you hold down the *Shift* key while sweeping out the zoom rectangle, VisIt will constrain the shape of the zoom rectangle to a square. If you want VisIt to always force a square zoom rectangle so that you don't have to use the *Shift* key, you can click on the **Clamp to Square** check box, click **Apply** in the **Interactors window** and save your settings.

Using the **Clamp to Square** zoom mode is a good way to maximize the amount of the vis window that is used when you zoom in on plots and when the vis window is in zoom mode. When the vis window is in navigate mode, the middle mouse button also effects a zoom. By default, zooming with the middle mouse button zooms into the plots but keeps the same vis window viewport which may, depending on the aspect ratio of the plots, not make the best use of the vis window's pixels. Fortunately, you can turn on the **Fill viewport on zoom** check box to force middle mouse zooming to also enlarge the viewport to its largest possible size in order to make better use of the vis window's pixels.

Navigation styles

When VisIt displays 3D plots, there are a few navigation styles from which you can choose by clicking on the following radio buttons in the **Interactors window**: **Trackball**, **Dolly**, and **Flythrough**. The default navigation style for 3D plots is: Trackball and it allows you to interactively rotate plots and move around them but it keeps the camera at a fixed distance from the plots and while it can get infinitely close to plots when you zoom in, it can never touch them or go inside of them. The Dolly navigation style behaves like the trackball style except that the when the camera zooms, it is actually moved. The Flythrough navigation style moves the camera and allows you to fly into plots and out the other side. Changing navigation modes will automatically cause a view Reset, due to the differing ways the camera is handled in each mode.

4.6.4 The Popup menu and the Toolbar

Each vis window contains a **Popup menu** and a **Toolbar**, which can be used to perform several categories of operations such as window management, setting the window mode, activating tools, manipulating the view, or playing animations. Options in the **Popup menu** exist in the **Toolbar** and vice-versa. A group of actions that is represented in the **Popup menu** as a menu usually maps to a toolbar in the vis window's **Toolbar**. To perform an action using the **Toolbar**, you can just click on its buttons. Access the **Popup menu** by pressing the right mouse button in the vis window. Select the desired item, then release the mouse button.

Hiding toolbars

The **Popup menu** has a **Customize** menu that lets you customize the vis window's **Toolbar**. For instance, you can choose to hide all of the toolbars so that they do not take up any of your screen space if you use a small monitor. If you want to hide all toolbars, you can select the **Hide toolbars** option from the **Customize** menu. If you want to show the toolbars again, you can click the **Show toolbars** option in the **Customize** menu. Note that when you select the **Show toolbars** option, VisIt only shows the toolbars that were enabled before they were hidden. If you want to enable or disable individual toolbars, you can select from the **Toolbars** menu under the **Customize** menu so VisIt only shows the toolbars that you routinely need. Once you tell VisIt which toolbars you want to use, you can save your preferences using the **Save settings** option in the **Main Window's Options** menu so that the next time you run VisIt, it only shows the toolbars that you enabled.

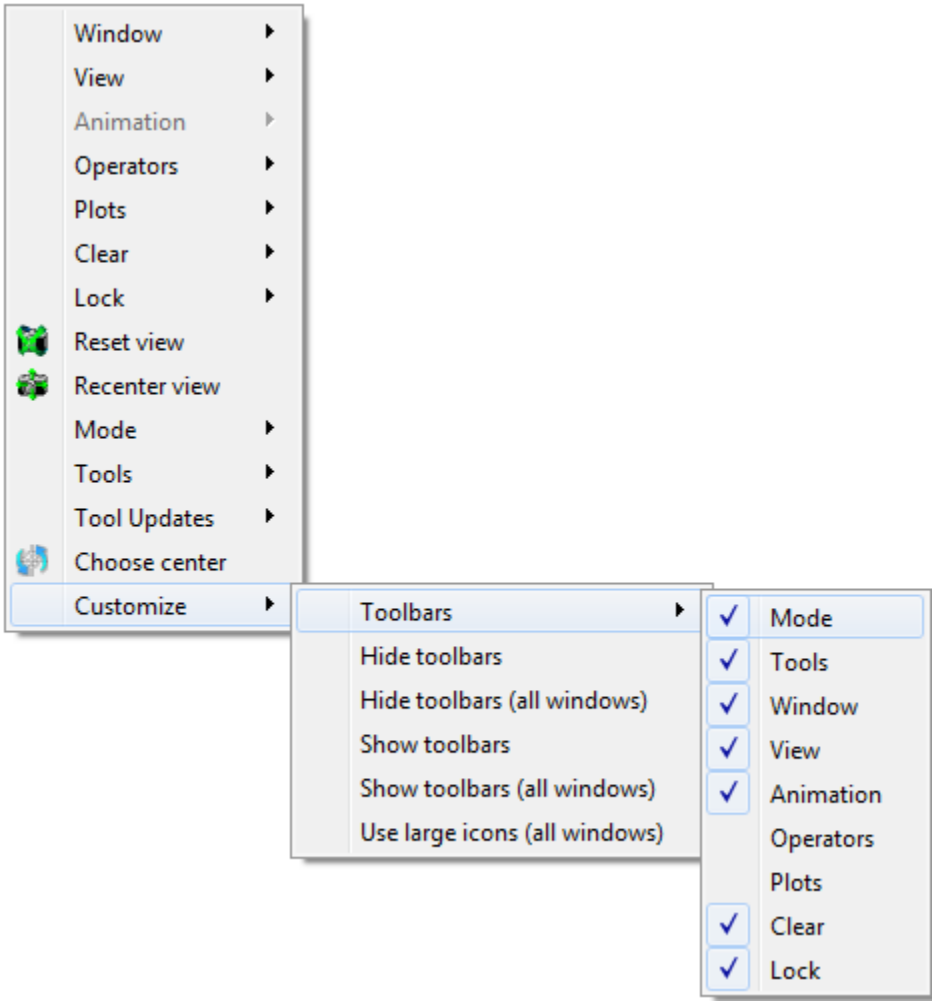


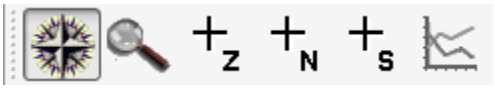
Fig. 4.217: Customize menu

Moving toolbars

Each of the vis window **Toolbar's** smaller toolbars can be moved to other edges of the vis window by clicking the small tab on the left or top side of the toolbar and dragging it to other edges of the vis window.

Switching window modes

The **Popup menu** contains a **Mode** menu (see Figure 4.218) that contains the 5 window modes. You can select a window mode from the **Mode** menu to change the vis window's mode. If you want to move or zoom the plot, choose navigate or zoom modes. If you want to extract data from the plots in the vis window, choose lineout mode or one of the pick modes. You can also use the **Mode toolbar** to change the vis window's window mode.



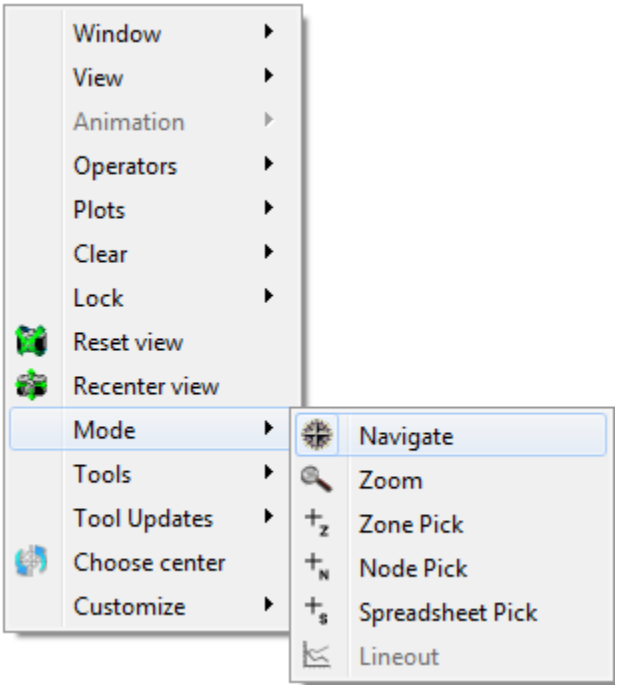


Fig. 4.218: Mode toolbar and menu

Activating tools

The **Popup menu** contains a **Tools** menu (see Figure 4.219) that lists of all of VisIt's interactive tools. Each tool shown in the menu has an associated icon that is used to indicate if the tool is enabled and if it is available in the vis window. Some tools are not available if the vis window does not contain plots or if the plots in the vis window are the wrong dimension to be used with the tool. In that event, the tool cannot be activated and the menu and toolbar entries for that tool are disabled. If a tool is available, its icon is bright blue; otherwise the icon is grayed out. If a tool is enabled, its icon has a selection rectangle around it. To activate a tool, choose an inactive tool from the **Tools** menu or click on its button in the **Toolbar**. To deactivate a tool, choose the tool that you want to deactivate from the **Tools** menu or click on its button in the **Toolbar**.

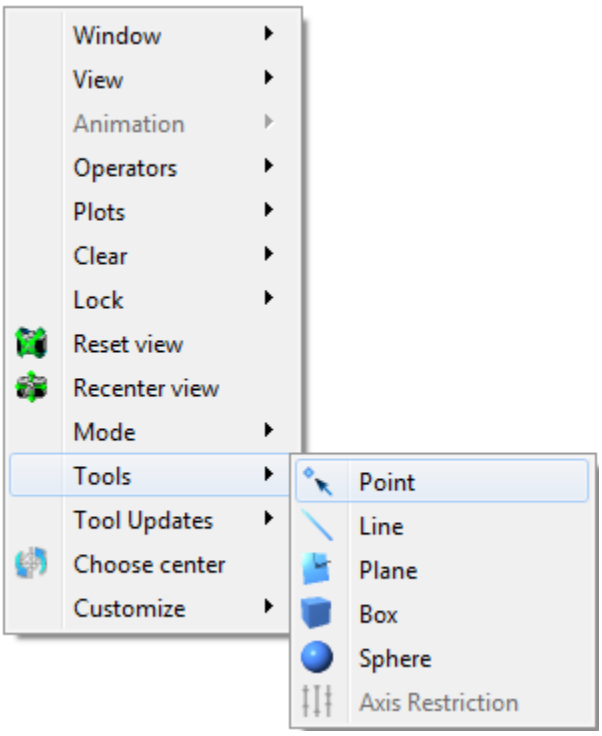


Fig. 4.219: Tool toolbar and menu

View options

VisIt's **Popup menu** and **Toolbar** (see Figure 4.220) have several options that are available for manipulating the view. You can reset the view, recenter the view, undo a view change, toggle perspective viewing, save and reuse useful views, or choose a new center of rotation.

Resetting the view

The **Popup menu** has a **Reset view** option (see Figure 4.220) that resets the view used to view the plots in the vis window. The view is typically reset to look down the -Z axis in a right-handed coordinate system. You can reset the view by selecting the Reset view option from the **Popup menu** or by clicking on the **Reset view icon** in the **Toolbar**.

Recentering the view

Sometimes adding a plot to a vis window that already contains plots can result in a lop-sided visualization. This happens when the spatial extents of the plots do not match. The **Popup menu** has a **Recenter view** option (see Figure 4.220) to calculate a new center of rotation for the plots so they are drawn in the center of the window. You can also recenter the view by clicking on the **Recenter view icon** in the **Toolbar**. To make sure that the view updates appropriately when new plots are added to the vis window, you may also want to check the **Auto center view** check box that is available in the **View Window** .

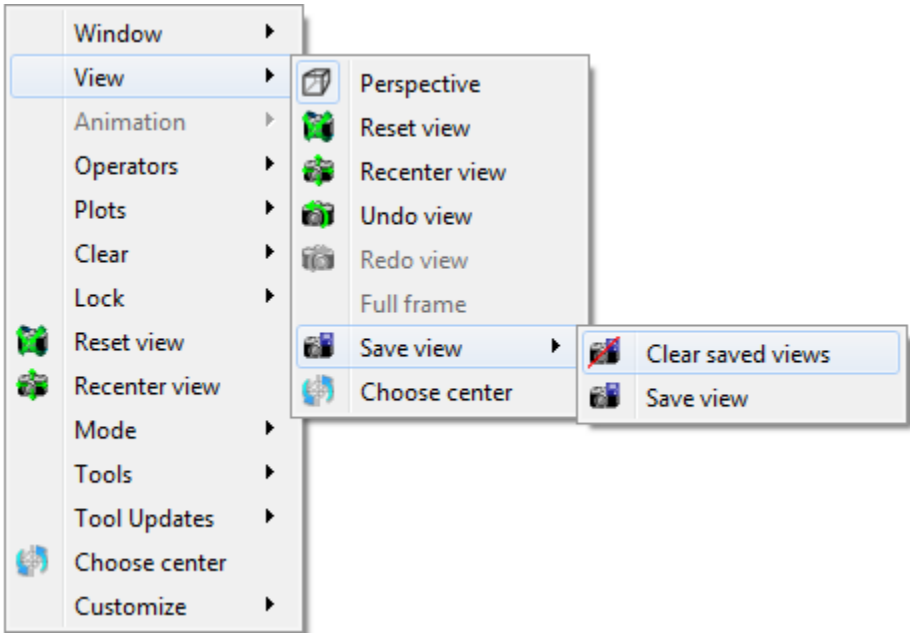


Fig. 4.220: View toolbar and menu

Undo view

The vis window saves the last ten views in a buffer so that you can restore them if you make an unintended change to the view. You can undo a view change, by selecting the **Undo view** option in the **Popup menu's View** menu or by clicking the **Undo view icon** in the **Toolbar** (see Figure 4.220).

Changing view perspective

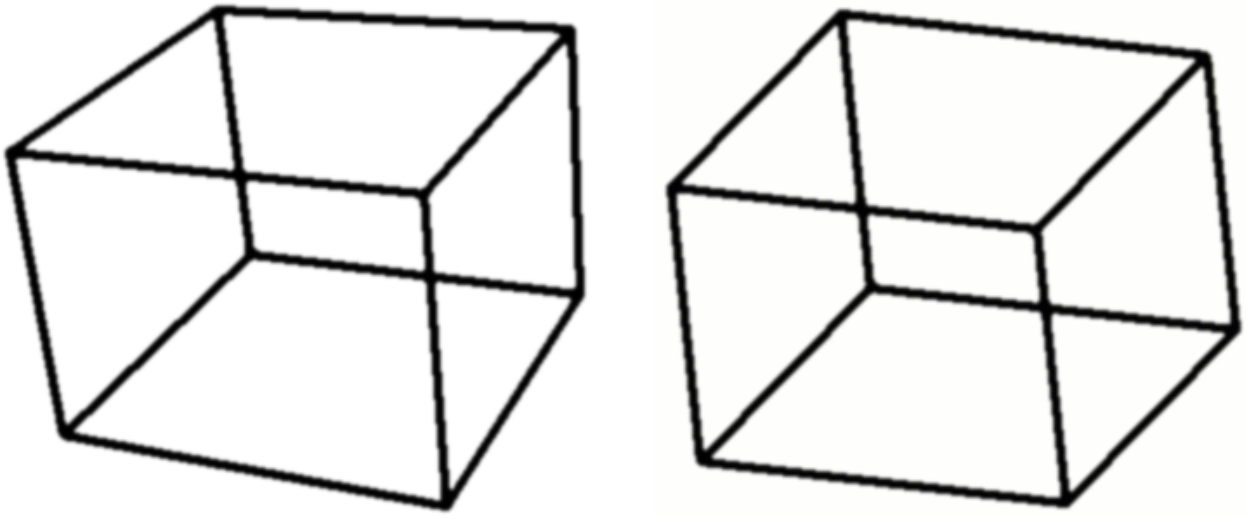


Fig. 4.221: Perspective examples

When the vis window contains 3D plots, the perspective setting can be used to enhance how 3D the plot looks. In a

perspective projection, graphics grow smaller as they recede into the distance which makes them look more realistic. To change the perspective setting, click on the **Perspective** option in the **Popup menu's View** menu (see Figure 4.220). When the vis window uses a perspective projection, the Popup menu's Perspective option will have a selection rectangle around its icon. You can also turn perspective on or off by clicking on the **Perspective icon** in the **Toolbar**. The difference in appearance having perspective and not having it is shown in Figure 4.221.

Locking views

The vis window can lock its view to other vis windows. When this toggle is set, making a change that affects the view in the active vis window will cause other vis windows that have the lock views toggle set to receive the same view as the active window. To lock the view, select the **Lock view** option from the **Popup menu's View** menu (see Figure 4.220) or click on the **Lock view icon** in the **Toolbar**. Note that you can lock 2D and 3D windows separately.

Saving and reusing views

Sometimes when analyzing a database, it is useful to be able to toggle between several different views. VisIt allows you to save up to 15 views that you can then use to look at different parts of your visualization. When you navigate to a view that you like, click the **Save view** icon in the **View** toolbar or click the **Save view** option in the **Popup menu's View** menu to save the view. When you save a view, VisIt adds a new numbered camera icon to the **View** toolbar and the **Popup menu** . Clicking on a view icon makes VisIt use the view that is associated with the clicked icon so you have one-click access to all of your saved views. You can preserve the saved views across VisIt sessions if you save your settings. If you want to delete the saved views so you can create different saved views, click the **Clear saved views** icon next to the **Save views** icon in the **View** toolbar.

Fullframe mode

Some databases yield plots that are so long and skinny that they leave most of the vis window blank when VisIt displays them. VisIt provides Fullframe mode to stretch the plots so they fill more of the vis window so it is easier to see them. It is worth noting that Fullframe mode does not preserve a 1:1 aspect ratio for the displayed plots because they are stretched in each dimension so they fit better in the vis window. To activate Fullframe mode, click on the **Fullframe** option in the **Popup menu's View** menu.

Choosing a new center of rotation

When you are working with a 3D database and you have created plots and zoomed in on them, you should set the center of rotation. The center of rotation is the point about which the plots are rotated when you rotate the plots in navigate mode. Normally, the center of rotation is set to the center of the plots being visualized. When you zoom way in on plots and attempt to rotate them, the default center of rotation often causes plots to whiz off of the screen when you rotate because the center of rotation is not close enough to the geometry that you are actually viewing. To set the center of rotation to something more suitable, VisIt provides the **Choose center** button, which can be accessed in the **Popup** menu or in the **View** toolbar. Once you click the **Choose center** button, VisIt temporarily switches to pick mode so you can click on the part of your visualization that you want to become the new center of rotation. Once you click on a plot, VisIt exits pick mode and uses the picked point as the new center of rotation. After setting the center of rotation, VisIt will make sure that the picked point is visible at all times.

Animation options

The animation controls in VisIt's **Main Window** are not the only controls that are provided for playing animations. Each vis window's **Popup menu** and **Toolbar** has options for playing and stepping through animations. To play an

animation, select the Play option from the **Popup menu's Animation** menu or click on the **Play icon** in the **Toolbar**, shown in Figure 4.222. To play the animation in reverse, select the **Reverse play** option or click on the **Reverse play icon** in the **Toolbar**. To stop the animation from playing, select the **Stop** option in the **Animation** menu or click on the **Stop icon** in the **Toolbar**. If you want to advance or reverse one frame at a time, use forward or reverse step.

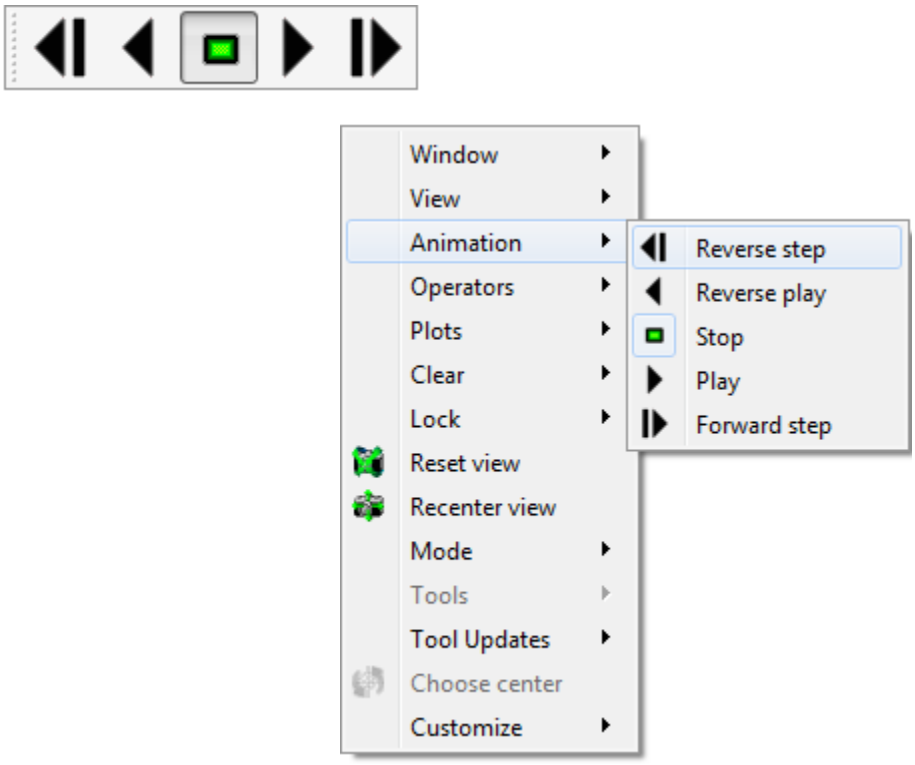


Fig. 4.222: Animation toolbar and menu

Window options

Many window options have previously been explained in this chapter so this section describes some addition options that were not covered. Many of the options in the **Main Window's Windows** menu are also present in the **Popup menu's Window** menu and toolbar (see Figure 4.223).

Changing bounding-box mode

The vis window allows a simple wireframe box to be substituted for complex plots when you want to rotate or move them. This is called bounding-box navigation and you can use it during navigate mode for complex plots so you can navigate faster when a vis window contains plots that take a long time to redraw. You can change the bounding-box mode by selecting the **Navigate bbox** option from the **Popup menu's Window** menu shown in Figure 4.223. You can also change the bounding-box mode by clicking on the **Bounding-box icon** in the **Toolbar**.

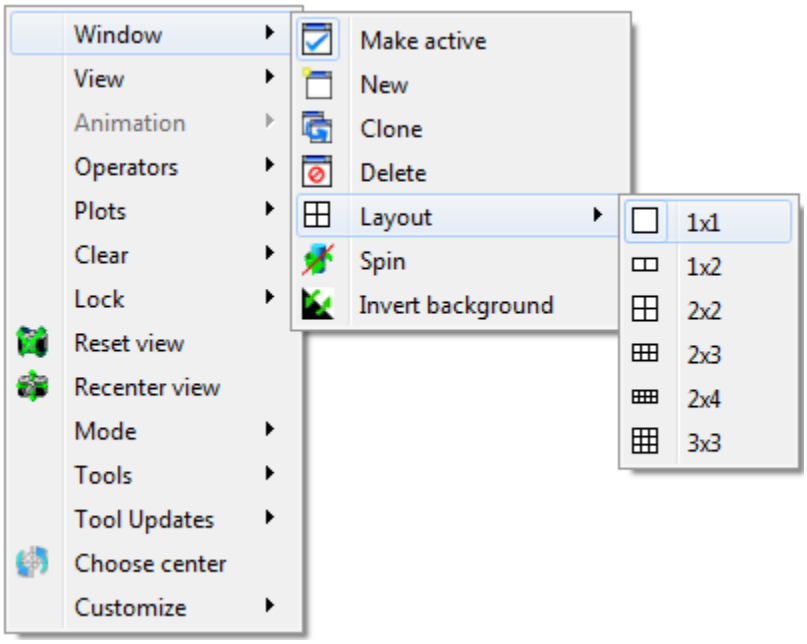


Fig. 4.223: Window toolbar and menu

Engaging spin

Spin is a setting that makes plots spin after the user stops rotating them and it provides a nice, easy way to see the entire plot without having to actively rotate it. To spin a 3D plot, turn on the **Spin** option in the **Popup menu's Windows** menu and then rotate the plot as you would in navigate mode. The plot will continue to spin after you release the mouse buttons. You can also engage spin using the **Spin** option in the **Main Window's Windows** menu or by clicking the **Spin icon** in the vis window's **Toolbar**. You can stop plots from spinning by turning off spin.

Inverting the foreground and background colors

Sometimes it is useful to swap the vis window's foreground and background colors. You can invert the background and foreground colors by clicking on the **Windows** menu's **Invert background** option. Note that this option is disabled when the vis window has a gradient background.

Clear options

The **Clear** menu (see Figure 4.224) in the **Popup menu** contains options that cause certain items such as: plots, pick points, and reference lines to be removed from a vis window. The **Clear** menu also appears in the **Main Window's Windows** menu.

Clearing plots from all windows

Sometimes it is useful to clear all plots from the vis window. Clearing plots from the vis window does not delete the plots but instead deletes their computed geometry and returns them to the new state so they appear green in the **Plot list**. An example of when you might want to clear plots is when you change material interface reconstruction options since changing them requires a plot to be regenerated. Rather than deleting plots that existed before changing the

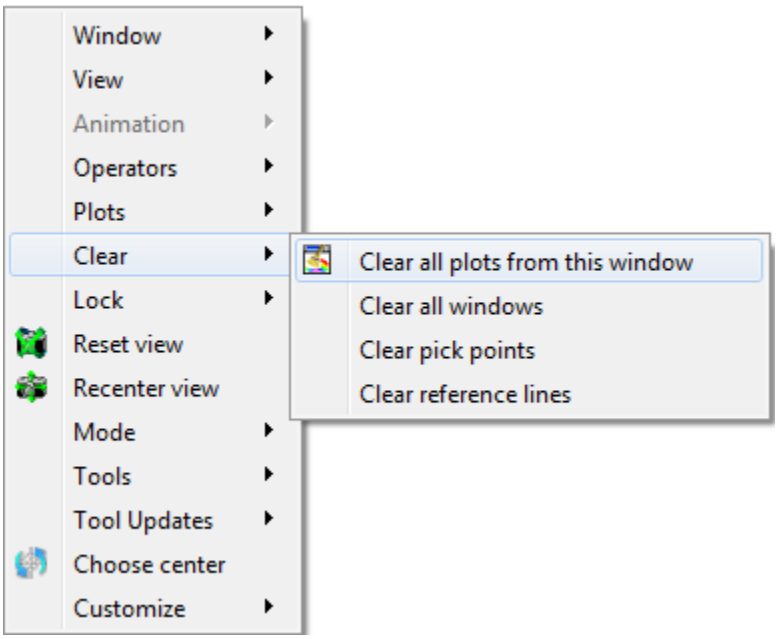


Fig. 4.224: Clear menu

material interface reconstruction parameters, you can clear the plots and force them to be completely regenerated by clearing the plots.

Clearing pick points

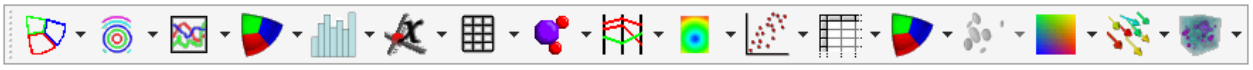
Click on the **Clear** menu's **Clear pick points** option if you want to remove all of the pick labels that were added when you picked on the plots in the vis window. Clearing the pick points also removes any pick information related to those pick points in the **Pick** window.

Clearing reference lines

Click on the **Clear** menu's **Clear reference lines** option if you want to remove all of the reference lines that were added to the vis window when you performed lineouts on the plots in the vis window.

Plot options

The **Plot** toolbar and **Plot** menu let you create new plots using variables from the open databases and also let you hide, delete, and draw the plots that correspond to the selected plot entries in VisIt's **Plot list**. The **Plot** menu is always available in the **Popup menu** but the **Plot** toolbar is not visible by default. If you want to make the **Plot** toolbar visible, you can turn it on in the **Popup menu's Customize** menu. The **Plot** menu and toolbar are shown in Figure 4.225.



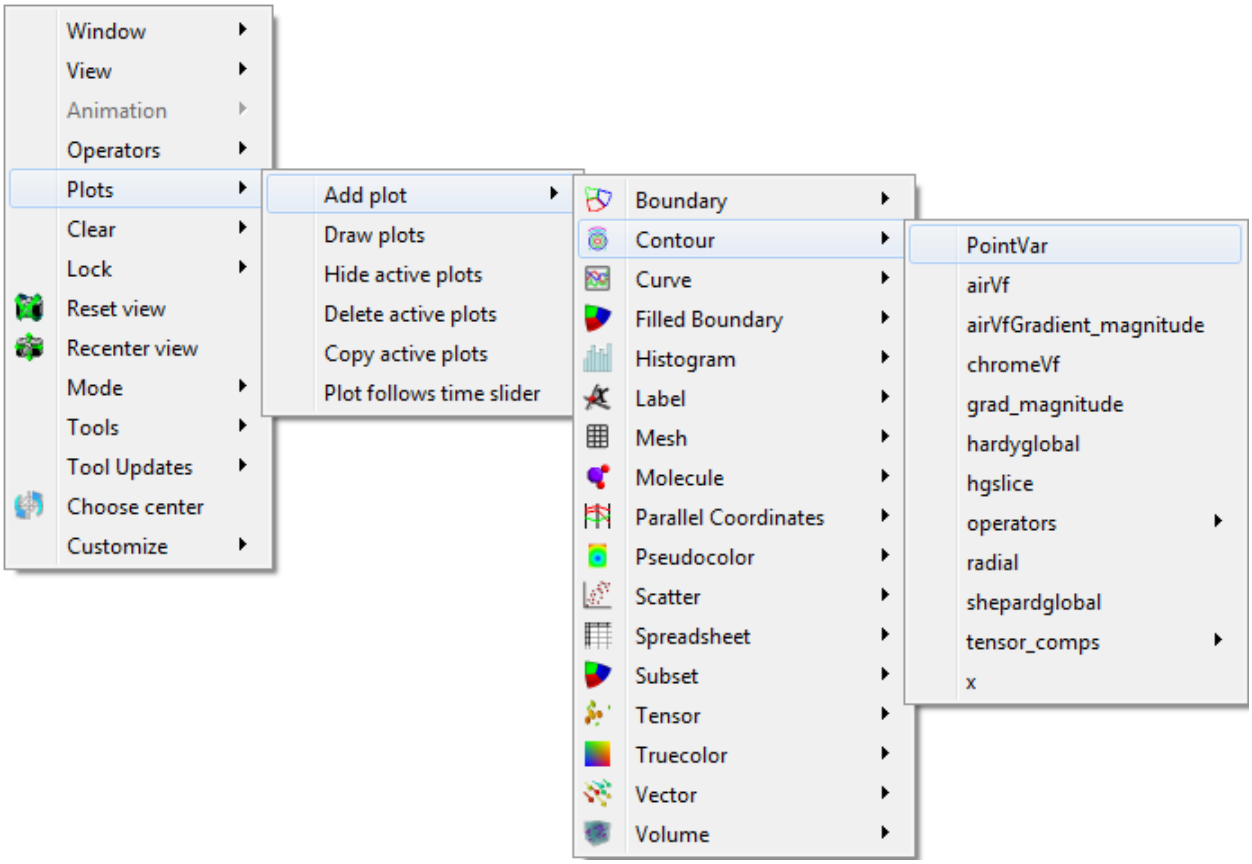


Fig. 4.225: Plot toolbar and menu

Adding a plot

The **Plot** menu and toolbar both provide options for you to add new plots. Each plot has its own menu option or icon that contains the variables that can be plotted from the open database. To add a new plot using the **Plot** menu, click the **Add plot** option to activate the list of available plots and then select a variable for the desired plot type. To add a new plot using the **Plot** toolbar, click on the icon for the desired plot type and select a variable from its variable menu. A new plot will appear in the **Main Window's Plot list** and it will be in the new state. To draw the plot, click the **Draw** button.

Drawing a plot

All plots added using the **Plot** menu or toolbar are in the new state, indicating that they have not been generated yet. To generate a plot once it has been created, click the **Draw** plots option in the **Plot** menu.

Hiding active plots

To hide the active plots, which are the plots that are highlighted in the **Main Window's Plot list**, click the **Plot** menu's **Hide active plots** option. Once clicked, the selected plots are made invisible until you hide them again to show them.

Deleting active plots

To delete the active plots, which are the plots that are highlighted in the **Main Window's Plot list**, click the **Plot** menu's Hide active plots option. Once a plot has been deleted, you can't get it back.

Operator options

The **Operator** menu and toolbar allow you to add new operators and remove operators from plots. The **Operator** menu is always available in the **Popup menu** but the **Operator toolbar** is not visible by default. If you want to make the **Operator toolbar** visible, you can turn it on in the **Popup menu's Customize menu**. The **Operator menu** and **Operator toolbar** are shown in Figure 4.226.

Adding an operator

The **Operator** menu and toolbar both provide options for you to add new operators. Each operator has its own menu option or icon that adds an operator of that type to the selected plots when you click its menu option or icon.

Removing the last operator

The **Operator** menu and toolbar both have options for you to remove the last operator from a plot. Each plot has a list of applied operators and clicking the **Remove last operator** menu option or icon will remove the last operator from each plot that is selected in the **Plot list**. Plots that have been drawn are regenerated.

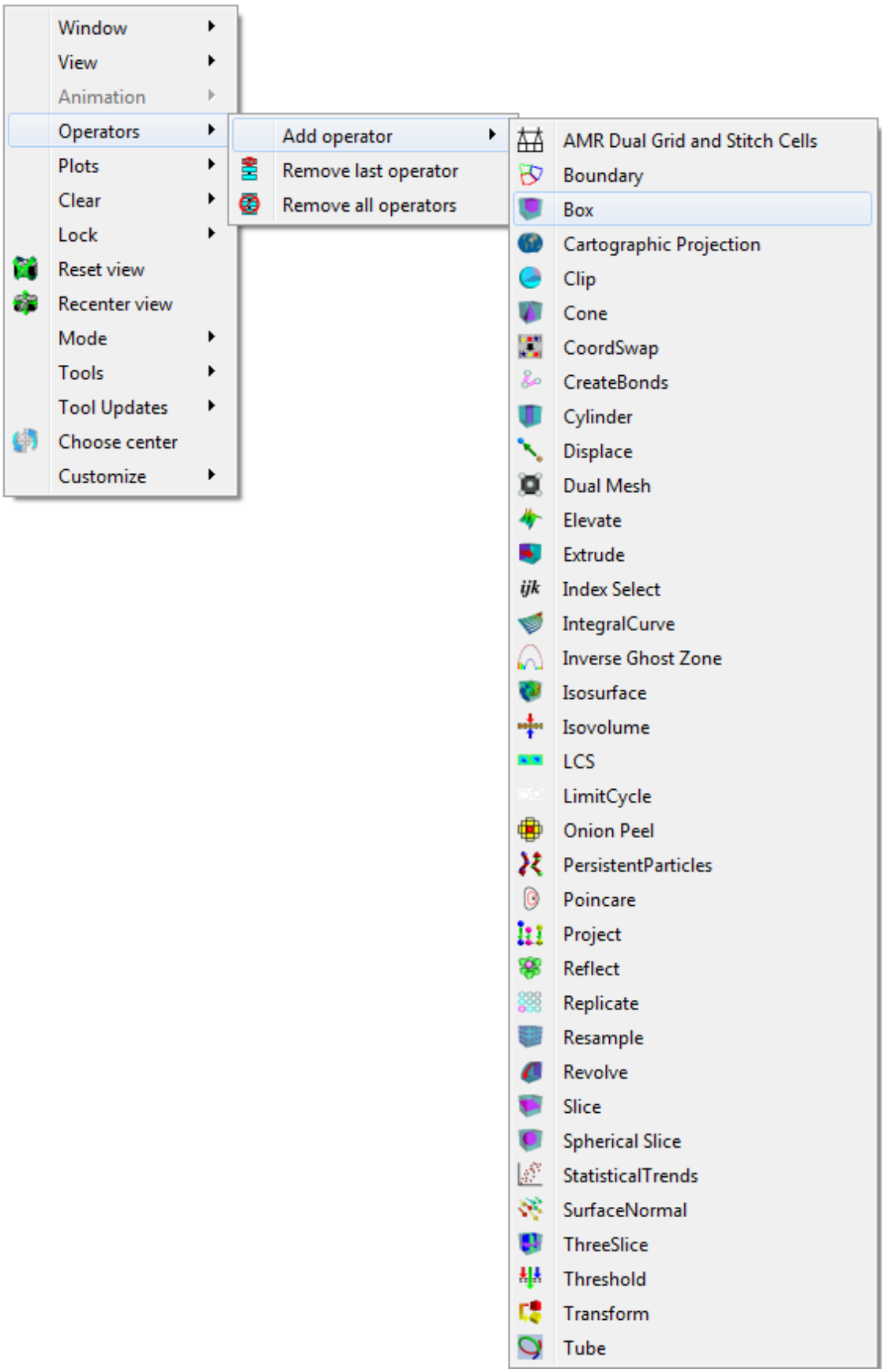


Fig. 4.226: Operator toolbar and menu

Removing all operators

The **Operator** menu and toolbar both have options for you to remove all operators from a plot. Each plot has a list of applied operators and clicking the **Remove all operators** menu option or icon will remove all operators from each plot that is selected in the **Plot list**. Plots that have been drawn are regenerated.

Lock options

The **Lock menu** and toolbar, both shown in Figure 4.227, allow you to lock certain visualization window attributes so that when you change them, other locked visualization windows also update. Currently, you can lock the view, time and tools. See *Locking Windows* for more information on how to use the lock options.

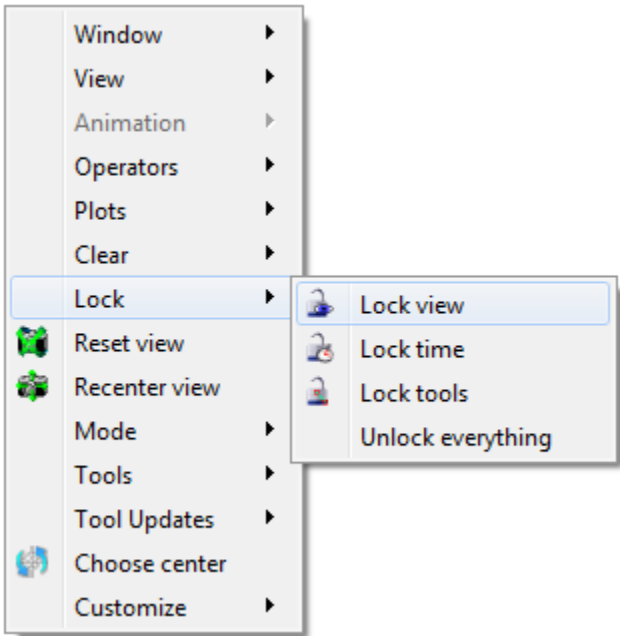


Fig. 4.227: Lock toolbar and menu

4.7 Subsetting

Meshes are frequently composed of a variety of subsets that represent different portions of the mesh. Common examples are domains, groups (of domains), AMR patches and levels, part assemblies, boundary conditions, node sets and zone sets, materials and even material species.

Users often find it useful to *restrict* which subsets are used in any given operation to focus their analyses on only certain regions of interest. This is handled through VisIt's **Subset Window**. Here, we describe VisIt's subsetting functionality and **Subset Window** in detail.

What is described here is primarily about *pre-defined, first-class, named subsets* as created by the data producer and supported within VisIt. Nonetheless, It is important to keep in mind that there are *other* ways that the data producer can organize data within VisIt's GUI or that users can employ VisIt's *Expressions* and *Operators* to create and manage

subsets. However, using these other approaches for the sole purpose of subsetting is often cumbersome through VisIt's GUI. To understand why as well as read about other issues related to subsetting, please see these developer notes.

4.7.1 What is a subset?

VisIt has first-class support for four different kinds of subsets; *Domains*, *Groups* (also called *Blocks*), *Materials* and material *Species*. In particular, as currently designed, any given mesh in VisIt can have only **one** decomposition into each of these kinds of subsets. That is, a mesh can have only one *Domain* decomposition, one *Group* decomposition, one *Material* decomposition and one material *Species* decomposition. A fifth kind of subset, *Enumerated*, is also supported and provides some additional generality but cannot be used in combination with the other four or even with other *Enumerated* subsets.

Data producers as well as the database plugins that read data into VisIt often have flexibility in deciding how to utilize these various kinds of subsets in representing their data. We describe each of these kinds of subsets and constraints in their use below.

Domain Subsets

VisIt's concept of a *Domain* subset is fundamental to its parallel programming and execution model. A domain in VisIt represents a *chunk* of mesh plus its variables that is **both** stored (in files and in memory) **and** processed *coherently* as a single, self-contained unit. Large meshes in VisIt are typically decomposed into *Domain* subsets for parallel processing. In fact, except in rare cases, the maximum number of MPI tasks VisIt may use is determined by the number of *Domain* subsets created by the data producer. VisIt's approach to processing a mesh in parallel is often described as *piggy-backing* off of the parallel decomposition created by the data producer.

Domain subsets also represent the *unit of work* VisIt allocates in its load balancing algorithms. If VisIt is running on *M* processors and reading a mesh of *N* domains, then if N<M, N-M processors will idle for operations involving that mesh. On the other hand, if N>kM (k an integer), some processors will be assigned *k* domains and some *k+1* domains.

In almost all cases, if a mesh is to be processed in parallel by VisIt, it must have been decomposed into *Domain* subsets by the data producer prior to reading the data into VisIt. In general, VisIt does not perform any on-the-fly domain decomposition of data it is reading. However, there is one, special case where VisIt can perform on-the-fly domain decomposition of a large, monolithic mesh; a structured mesh stored in a file format that supports hyper-slabbed I/O. In this simple case, VisIt will try to evenly decompose the 2 or 3D mesh into roughly equal sized hyper-slabs whose number is determined by the number of parallel tasks. VisIt will also then utilize the file format's hyper-slab I/O routines to read into each parallel task only the part(s) of the mesh assigned to that task.

A mesh is **required** to have domains if it is ever to be processed in parallel by VisIt.

Group or Block Subsets

Groups (or *Blocks*) are just unions of *Domains*. They are optional. A mesh is not **required** to have groups. On the other hand, if a mesh has *Groups*, then **every** domain in the mesh must be assigned to one and only one *Group* subset. *Groups* may be used to represent, for example, the files in which multiple domains are stored or sets of neighboring domains that share a common logical/structured indexing arrangement in an otherwise globally unstructured mesh.

The key constraint about group subsets is that they can represent only unions of the *domain* subsets. Internally in VisIt, a group subset is implemented as a list of domain subset ids.

Material Subsets

Material subsets are used to represent the decomposition of a mesh into various materials. For example, a mesh may be composed of steel, brass, and aluminum materials. If these materials are given integer ids 83 (`int('S')`), 66

(`int('B')`) and 65 (`int('A')`), then each zone (or cell) in the mesh can be assigned a value of 83, 66 or 65 to indicate the zone is composed of steel, brass or aluminum. This would be equivalent to an integer valued (with 3 unique values), zone-centered variable on the mesh.

For material subsets, however, VisIt also supports a notion of *mixing* where a single zone (or cell) can be composed of multiple materials each occupying some fractional volume of a whole zone (or cell). From a sub-setting perspective, a more formal way of thinking about *mixing* is that it is way of supporting *partial inclusion* of a mesh zone (or cell) within a given material subset.

Material subsets are optional. Furthermore, if material subsets are defined additionally supporting *mixing* is also optional. Only some data producers that involve *Material* subsets also involve *mixing*.

When *mixing materials* are involved, VisIt can employ a variety of sophisticated *Material Interface Reconstruction (MIR)* algorithms to draw the interfaces between materials based on the volume fractions of the *mixing*. The main point about *MIR* is that it represents an additional computational burden when manipulating *Material* subsets. Manipulating *Group* or *Domain* subsets has no such equivalent computational cost.

Mesh Variables with Material Specific Properties

For some mesh variables, data producers may have different values of the variable for each of the materials within various zones (or cells) of the mesh where *mixing* is occurring. When such a variable is being plotted, for example with the *Pseudocolor Plot*, what value/color should VisIt show for such zones? The fact is, depending on the user's needs, VisIt is capable of showing either an *overall* value for the zone or showing the material-specific values in the zone. This can be handled through appropriate use of VisIt's *(MIR)* algorithms and **Subset Window** controls.

Species Subsets

In addition to *mixing*, another feature *Materials* subsets support is a notion of *Species*. For example, there are many different varieties of brass and steel depending on the alloys used. Neither brass nor steel are themselves pure elements on the periodic table. They are instead *alloys* of other pure metals. Common Yellow Brass is, nominally, a mixture of Copper (Cu) and Zinc (Zn) while Tool Steel is composed primarily of Iron (Fe) but mixed with some Carbon (C) and a variety of other elements.

Lets suppose we are dealing with the following alloys and species compositions…

Material	Species composition
Brass	Cu:65%, Zn:35%
T-1 Steel	Fe:76.3%, W:18%, Cr:4.0%, C:0.7%, V:1%
O-1 Steel	Fe:96.2%, W:0.5%, Cr:0.5%, C:0.9%, Mn:1.4%, Ni:0.5%

The *Materials* decomposition would consist of 3 subsets for Brass, T-1 Steel and O-1 Steel. For the *Species* decomposition, Brass would be further decomposed into 2 *Species* subsets, T-1 Steel into 5 *Species* subsets and O-1 Steel, 6 *Species* subsets.

Alternatively, one could opt to characterize both T-1 Steel and O-1 Steel has a single, non-specific *Steel* having 7 *Species* subsets, Fe, W, Cr, C, V, Mn, Ni where for T-1 Steel, the Mn and Ni *Species* subsets are always empty and for O-1 Steel the V *Species* subset is always empty. In that case, there would only be 2 *Materials* subsets for Brass and non-specific *Steel*.

Species subsets are optional. A mesh does not need to have them defined. However, as currently designed, a data producer cannot define *Species* subsets without also defining *Materials* subsets (even if there is only one material subset for the whole mesh).

A final thing to note about *Species* subsets is that they do not represent spatially distinct parts of the mesh like *Domains*, *Groups*, or *Materials*. Instead, *Species*, if they are defined are ever present, everywhere in the mesh. Only their relative

concentrations vary at any given point in the mesh. But, *Species* do permit subsetting a particular physical quantity's *value* in that, for example the *total pressure* in a zone can be decomposed into partial pressures on each of the species comprising the materials in the zone. Furthermore, using the **Subset Window**, VisIt can then control which partial value(s) are used in a particular plot.

Domains, Groups, Materials and Species In Combination

A given mesh may involve any combination of *Domain*, *Group* and *Material* subsets. Furthermore, VisIt's **Subset Window** makes it possible to manipulate these four kinds of subset *in combination*. That is, a user can simultaneously control which domains, which materials and which groups VisIt should process in any given operation. However, manipulating subsets in combination works only for these kinds of subsets. Other kinds of sub-setting, such as Enumerated subsets which are discussed next, are not as well integrated.

Enumerated Subsets

A key constraint of the other kinds of subsets is that any given mesh can have only **one** decomposition into domains and **one** decomposition into groups and **one** decomposition into materials. However, a mesh can be composed of any number of *Enumerated* subsets. Enumerated subsets are defined by first defining the enumeration *class* and then creating a *bitmap* like variable over the mesh to indicate which mesh entities (nodes, edges, faces or volumes) belong to which subsets of the enumeration class.

Within an enumeration class, the sets can be arranged hierarchically so that some sets contain other sets as in a part assembly.

Enumerated subsets do not work in combination with domains, groups or materials or in combination with other classes of *Enumerated* subsets. On the other hand, for any given mesh, there can be any number of enumeration classes, each defining a collection of related subsets. For example, if a mesh has defined two enumeration classes, one for *node sets* and one for *face sets*, then different subsets of nodes can be manipulated simultaneously or different subsets of faces can be manipulated simultaneously but different sets of nodes cannot simultaneously be manipulated in combination with different sets of faces. Finally, manipulating enumerated subsets can also incur small a computational burden due to the work involved in finding the mesh entities within a given subset.

4.7.2 Subset Inclusion Lattice

VisIt relates all possible subsets in a database using what is called a *Subset Inclusion Lattice* (SIL). Ultimately the subsets in a database are cells that can be grouped into different categories such as material region, domain, patch, refinement level, etc. Each category has some number of possible values when taken together form a collection. A collection lets you group the subsets that have different values but are still part of the same category. For example, the mesh shown in Figure 4.228 is broken down into domain and material categories and there are 3 domain subsets in the domain category. VisIt uses the SIL to remove pieces of a database from a plotted visualization by turning off bottom level subsets that are arrived at through turning off members in various collections or turning off entire collections. When various subsets have been turned off in a SIL, the collective on/off state for each subset is known as a SIL restriction.

4.7.3 Using the Subset Window

Users can open the **Subset Window**, shown in Figure 4.229, by clicking on the **Subset** option in the **Main Window's Controls** menu or by clicking on the **Subset** Venn Diagram-looking icon next to the name of a plot in the **Plot list**. VisIt's **Subset Window** shows the relationships between subsets and provides controls that allow users to turn subsets on and off.

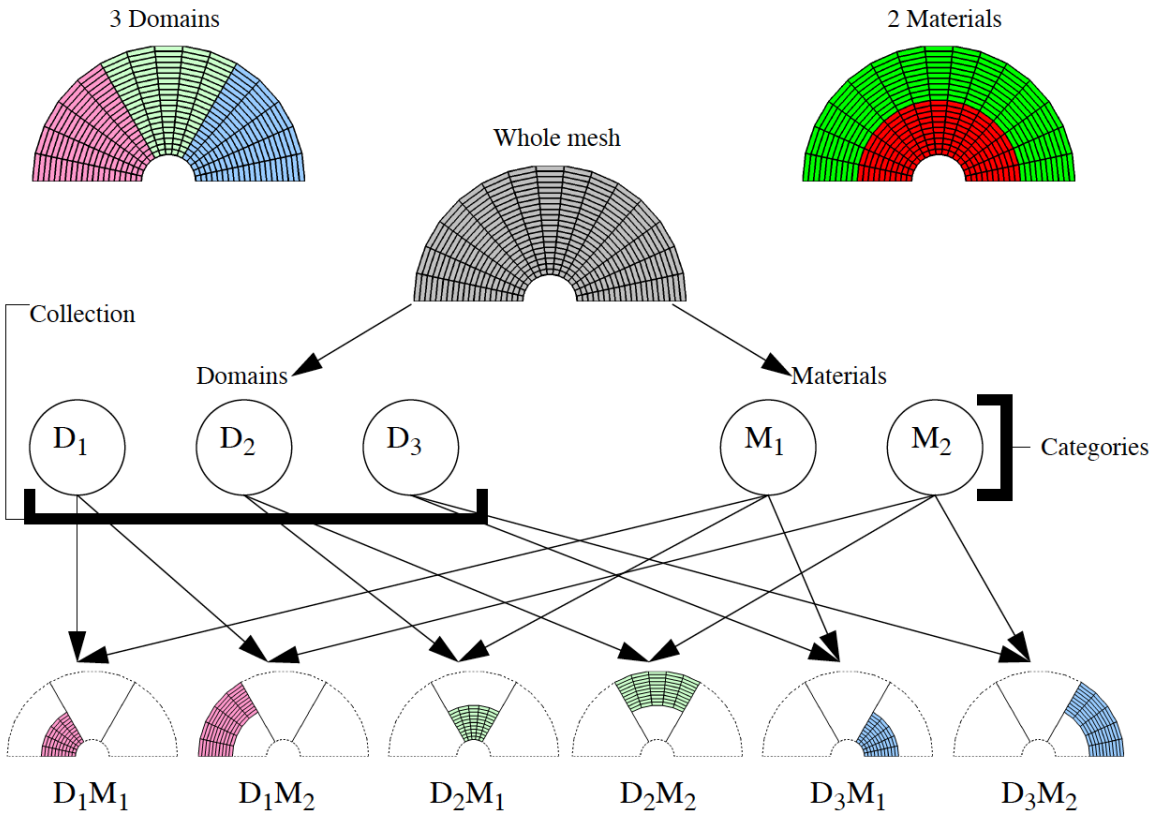


Fig. 4.228: Whole mesh divided up into domains and materials

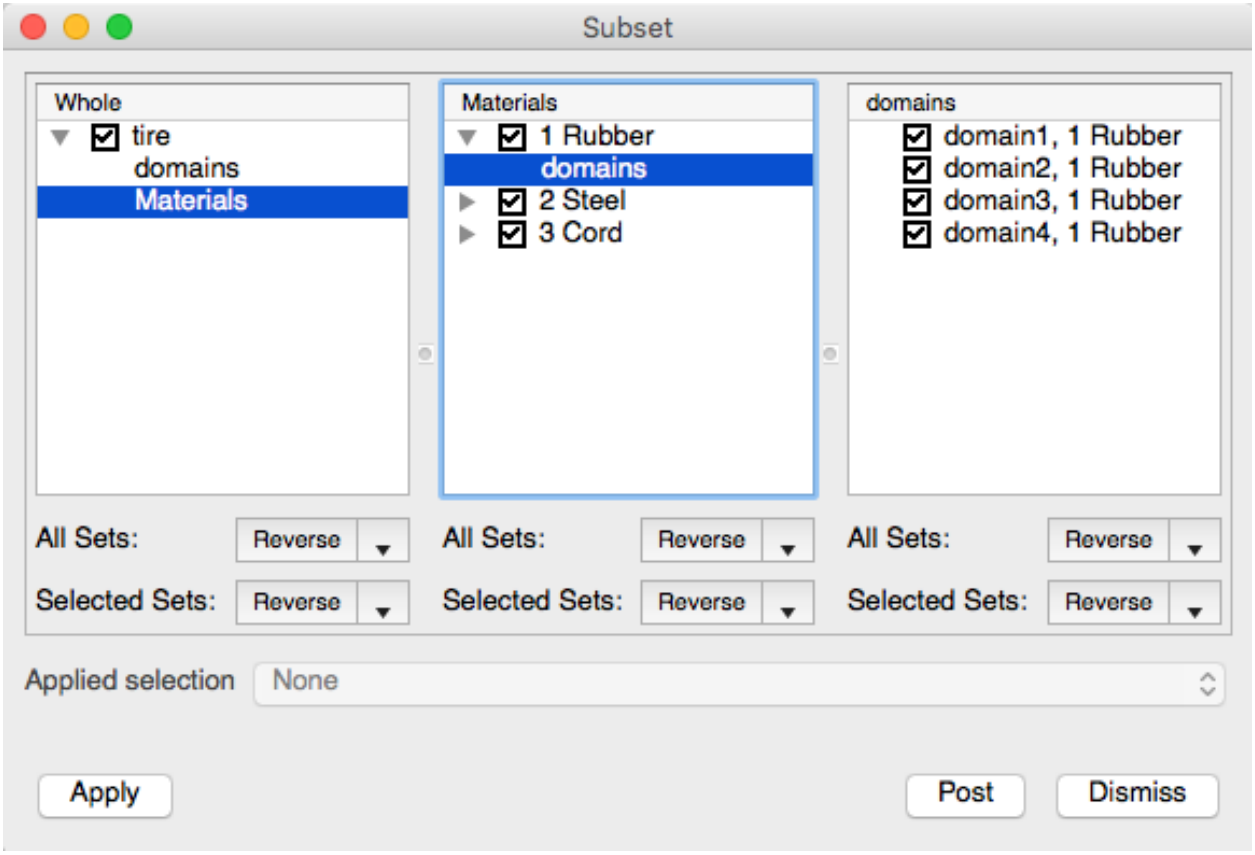


Fig. 4.229: Subset window

The **Subset Window** initially has three panels that display the sets associated with mesh of the currently active plot. The window will grow more panels to the right, when necessary as the subset structure of a mesh is browsed. Each successive panel shows the *next* level of subsets in the mesh. The leftmost panel contains the top level (e.g. *whole*) set for the whole mesh of the currently active plot. The top level or *whole* set, which includes all subsets in the mesh, is usually decomposed into the various kinds of subsets described in the section *What is a subset?*. For example, it can be decomposed by material, processor domain, etc. The various ways in which a database can be decomposed are called *subset categories*. The subset categories will vary depending on how the data producer(s) create the database(s).

Browsing subsets

To browse the subsets for a database, users must first have created a plot. Once a plot is created and selected, open the **Subset Window**. The left panel in the **Subset Window** contains the database's top level set and may also list some subset categories. Some simple databases don't include any subset and so VisIt will not show any subsets for them. To start browsing the available subsets, users can click on one of the subset categories to display the subsets in that category. For instance, clicking on a "Material" subset category will list all of the mesh's materials in the next panel to the right. The materials are subsets of the top level set. Double clicking on a set or clicking on an expand arrow lists any subset categories that can be used to further break down the set.

Changing a SIL restriction

Each set in the **Subset Window** has a small check box next to it that allows users to turn the set on or off. The check box not only displays whether a set is on or off, but it also displays whether or not a set is partially on. When a set is partially on, it means that at least one (but not all) of the subsets it contains is turned on. When a set is partially on, its check box shows a small slash instead of a check or an empty box. Uncheck the check box next to a set name to turn the set off.

Suppose a user has a database that contains 4 domains, numbered 1 through 4. If the user wants to turn off the subset named "domain1", first click on the "domains" category to list the subsets in that category. Next, click the check box next to the subset name "domain1" and click the **Apply** button. The result of this operation, shown in Figure 4.230, removes the "domain1" subset from the visualization. Note that the **Subset Window** "domain1" set's check box is unchecked and the top level set's check box has a slash through it to show that some subsets are turned off.

Creating complex subsets

When visualizing a database, it is often useful to look at *combinations* of subsets. Suppose a user has a database that has two subset categories: "Materials", and "Domains" and that the user wants to turn off the "domain1" subset but also wants to turn off a material in the "domain4" subset. Users can do this by clicking on the "Domains" category and then unchecking the "domain1" check box in the second panel. Now, to turn off a material in the "domain4" subset, the user clicks on the "domains" category in the left panel. Next, double-click on the "domain4" subset in the second panel. Select the "Materials" subset category in the second panel to make the third panel list the materials that can be removed from the "domain4" subset. Turning off a couple materials from the list in the third panel will only affect the "domain4" subset. An example of a complex subset selection is shown in Figure 4.232 and the state of the **Subset window** is shown in Figure 4.233.

Turning multiple sets on and off

When databases contain large numbers of subsets, it is convenient to turn many of them on and off at the same time. Users can select ranges of subsets by clicking on the name of a subset using the left mouse button and dragging the mouse up or down to other subsets in the list while still holding down the left mouse button. Alternatively, users can click on a subset to select it and then click on another subset while holding down the *Shift* key to select all of the subsets in the middle. Finally, users can select a group of multiple nonconsecutive subsets by holding down the *Ctrl* key while clicking on the subsets to be selected.

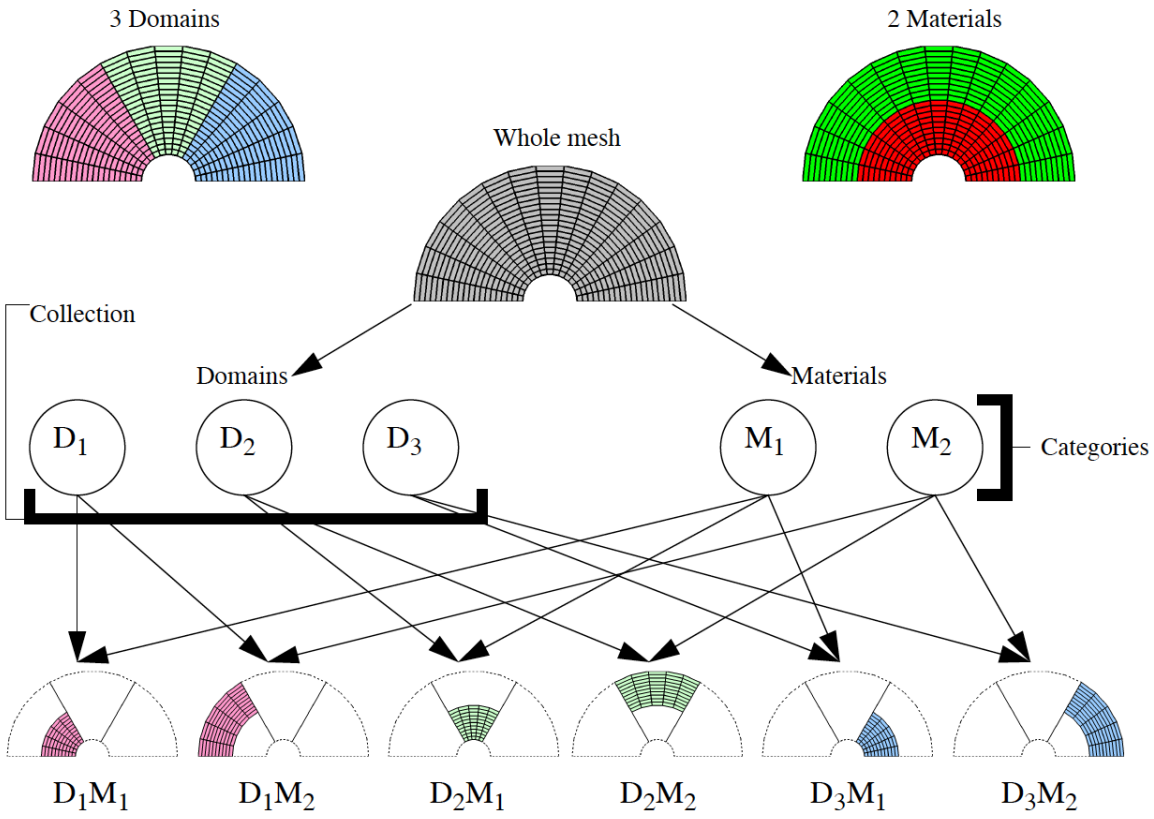


Fig. 4.230: Removing one subset.

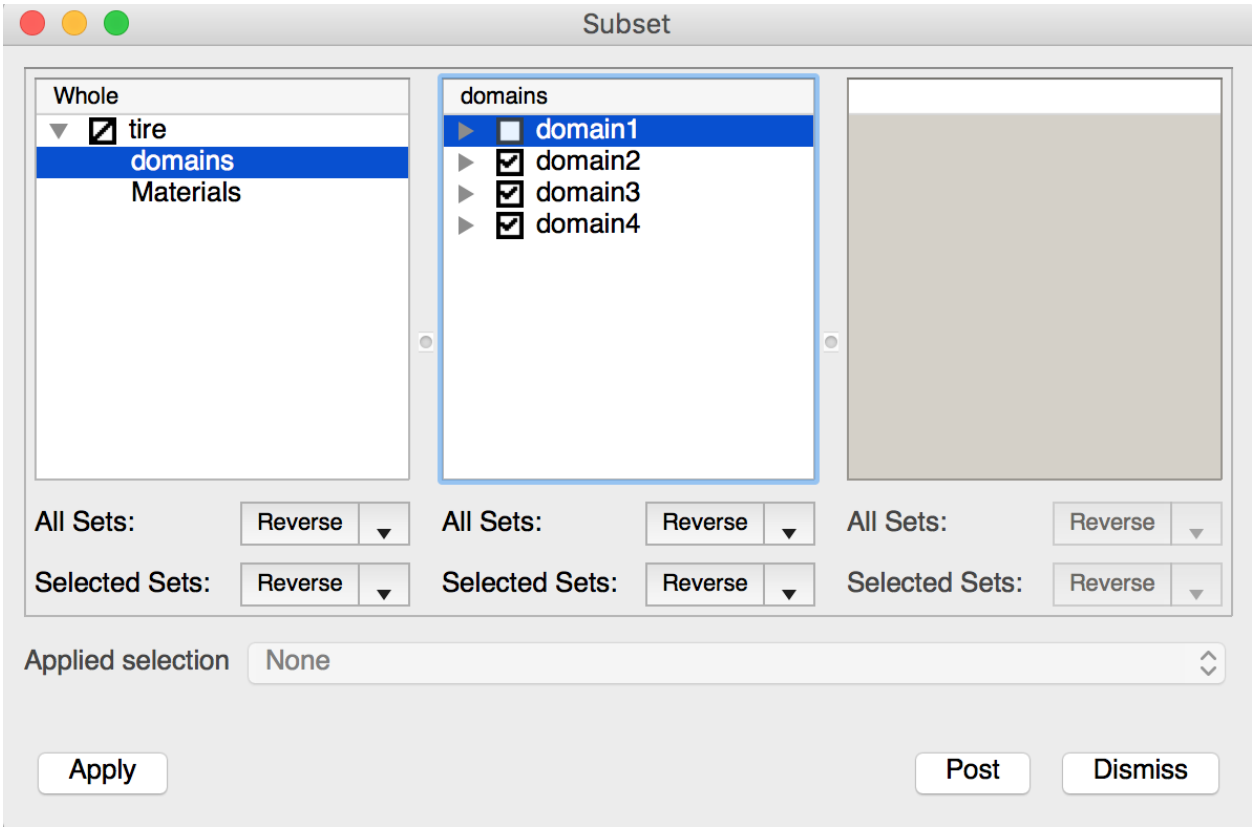


Fig. 4.231: Subset window with one subset removed.

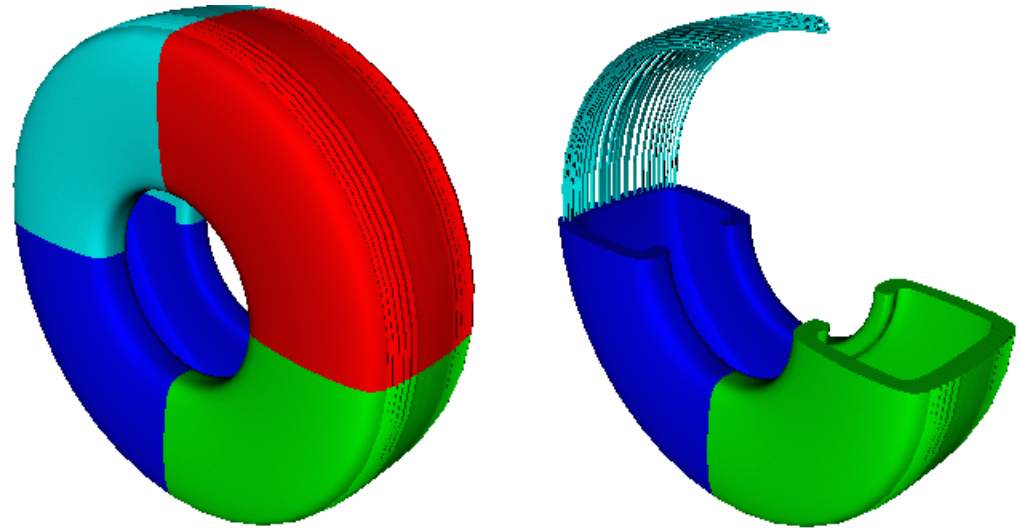


Fig. 4.232: Example of a complex subset.

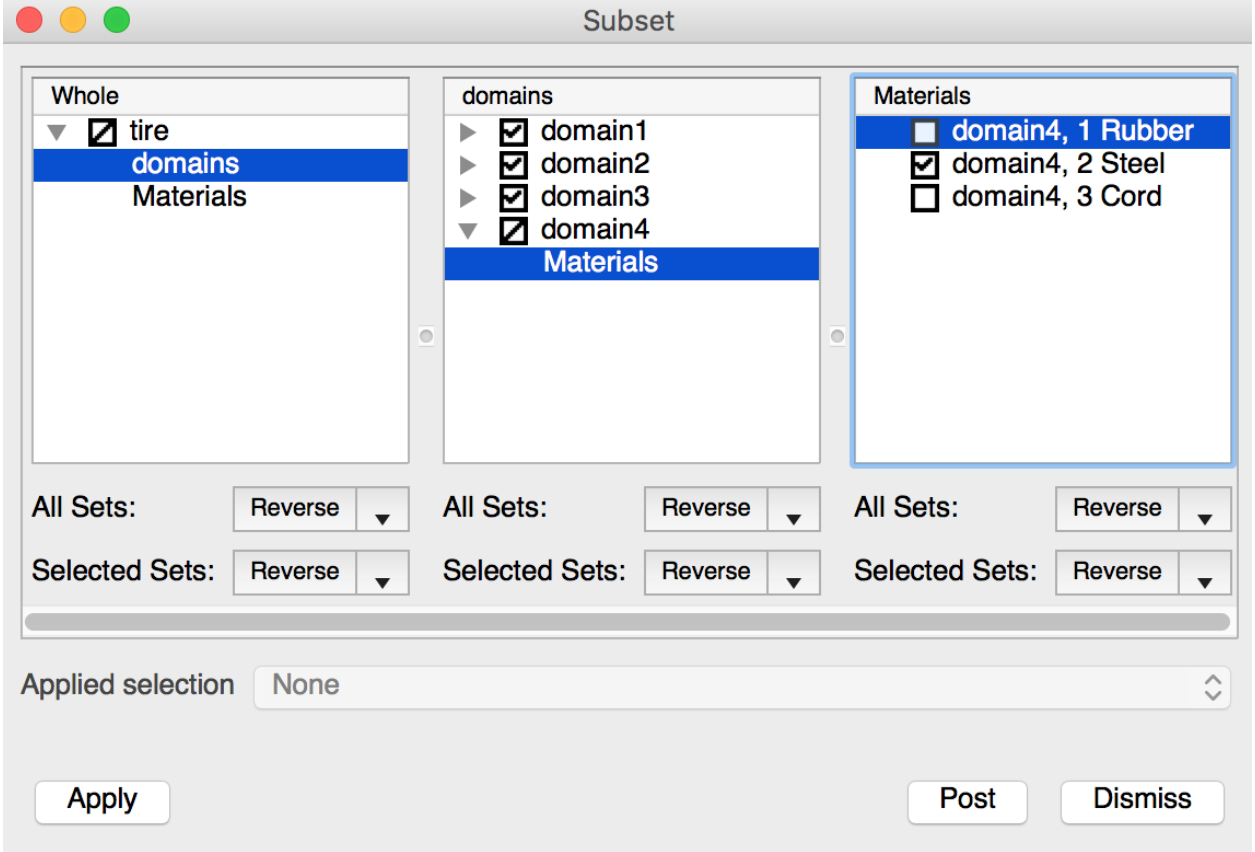


Fig. 4.233: Subset window for complex subset example.

Once a group of subsets has been selected, the buttons at the bottom of the pane can be used to adjust the selection in various ways. The top button applies an action to all of the sets in the pane regardless of how they have been selected. The bottom button applies an action to only the subsets that have been selected. Each action button has three possible actions: Turn on, Turn off, and Reverse. Users can change the action for an action button by clicking on the down-arrow button to its right and selecting one of the **Turn on**, **Turn off**, and **Reverse** menu options. When the **Turn on** action is used, the appropriate subsets will be turned on. When the **Turn off** action is used, they will be turned off. When the **Reverse** action is used, the on/off state of the sets will be reversed (or toggled).

4.7.4 Material Interface Reconstruction

Many data producers create meshes with material subsets. In some cases, materials include *mixing* where multiple materials exist within each mesh zone and in other cases materials are *clean* in each zone (e.g. no *mixing*).

The materials are often used to break meshes into subsets that correspond to physical parts of a model. Materials are commonly stored out as a list of materials and material volume fractions for each cell in the database. If a cell has only one material then is a clean cell. If a cell has more than one material, it has some fraction of each of the materials and it is known as a mixed cell. The fraction of the material in a cell if accounted for by the material volume fraction. Since only the volume fractions are known, and not any information about how the materials are distributed in the cell, VisIt must make a guess at the location of the boundaries between materials.

Material interface reconstruction (MIR) is the process of constructing the boundaries between materials, in cells with mixed materials, from the material volume fraction information stored in the database. MIR is not usually needed when you visualize the entire database but when you start to subset the database by removing materials, VisIt must perform MIR to remove only the parts of the database that contain the material to be removed. Without MIR, visualizations containing mixed materials would be very blocky when materials are removed. VisIt's MIR algorithms have several settings, which you can change using the controls in the **Material Reconstruction Options Window** (see Figure 4.234), that influence the appearance of the final plot. To open the **Material Reconstruction Options Window**, click on the **Materials** option in the **Main Window's Controls menu**.

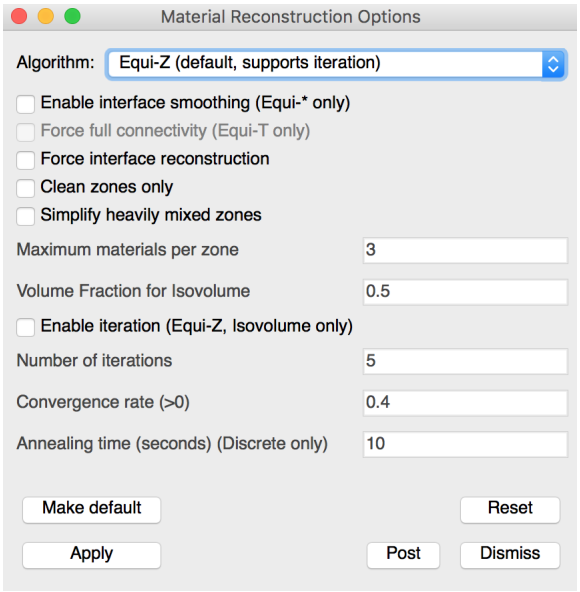


Fig. 4.234: Material Reconstruction Options Window

Choosing a MIR algorithm

VisIt currently provides three MIR algorithms: Tetrahedral, Zoo-based, and Isovolume. Each MIR algorithm reconstructs the interfaces between materials using a different method and one method may work better or worse than another based on the complexities of the input data. You can select your preferred MIR algorithm by choosing from the **Algorithm** combo box in the **Material Reconstruction Options Window**. Note that if you have plots that have already been generated, the new material options will not take effect for those plots unless you clear the plots and redraw them.

The Tetrahedral algorithm breaks up each mixed cell into tetrahedra and computes the interfaces through the original cell by recursively subdividing the tetrahedra until the approximate volume fractions, which determine the amount of material in a cell, are reached. The Tetrahedral MIR algorithm results in a high cell count so it is not often used.

The Zoo-based MIR algorithm breaks up mixed cells into elements based on supported finite elements (tetrahedra, prisms, pyramids, wedges, cubes). The resulting reconstruction results in far fewer cells than other methods while also producing superior material boundaries. The Zoo-based algorithm is the default because of the quality of the material boundaries and because the zoo-based cell representation saves memory and ultimately leads to faster pipeline execution due to the smaller cell count.

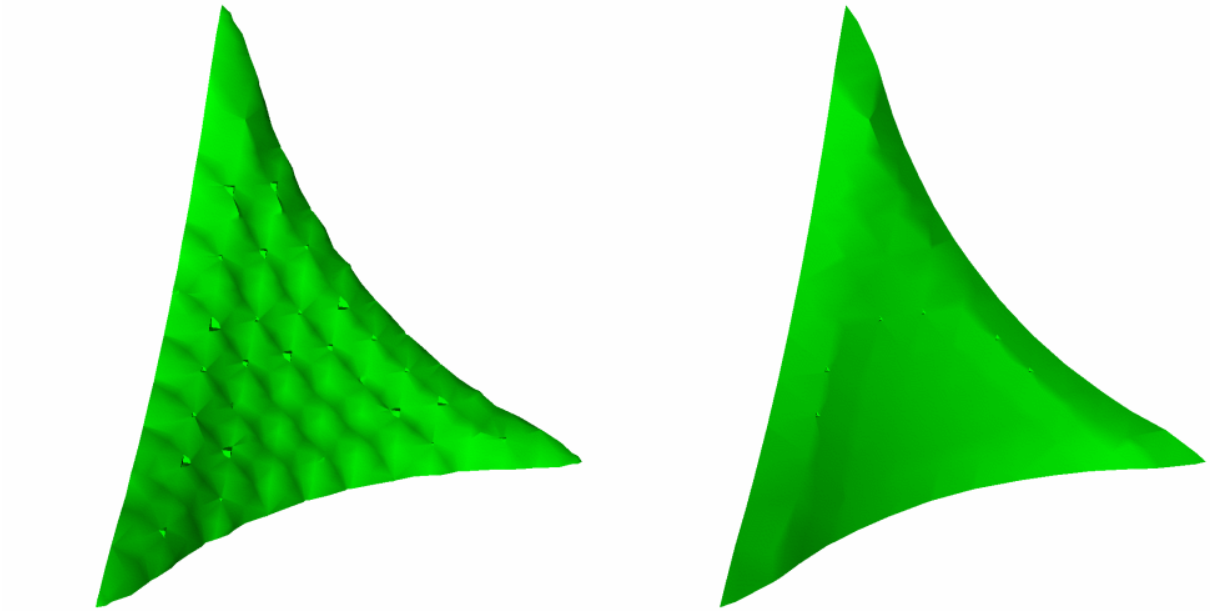


Fig. 4.235: Tetrahedral MIR vs. Zoo-based MIR

The Isovolume algorithm computes an isovolume containing portions of cells that contain a user-specified fraction of materials. The Isovolume approach to MIR does not generally produce very good looking results since there are gaps where several materials join. However, the Isovolume algorithm does do a better job than the other two algorithms when it comes to finding cells that contain very small fractions of a certain material when the cells are heavily mixed. If you use the Isovolume MIR algorithm, you can specify the amount of material required to be present before VisIt creates a material interface for a material. The amount of material is specified as a volume fraction in the range [0,1]. Specifying smaller values in the **Volume Fraction for Isovolume** text field will find materials that may be omitted by other MIR algorithms.

Finding materials with low volume fractions

When mixed cells contain several materials, the Zoo-based MIR algorithm will often omit materials with very small volume fractions, leaving only the materials in the mixed cell that had the highest volume fractions. If you want to plot

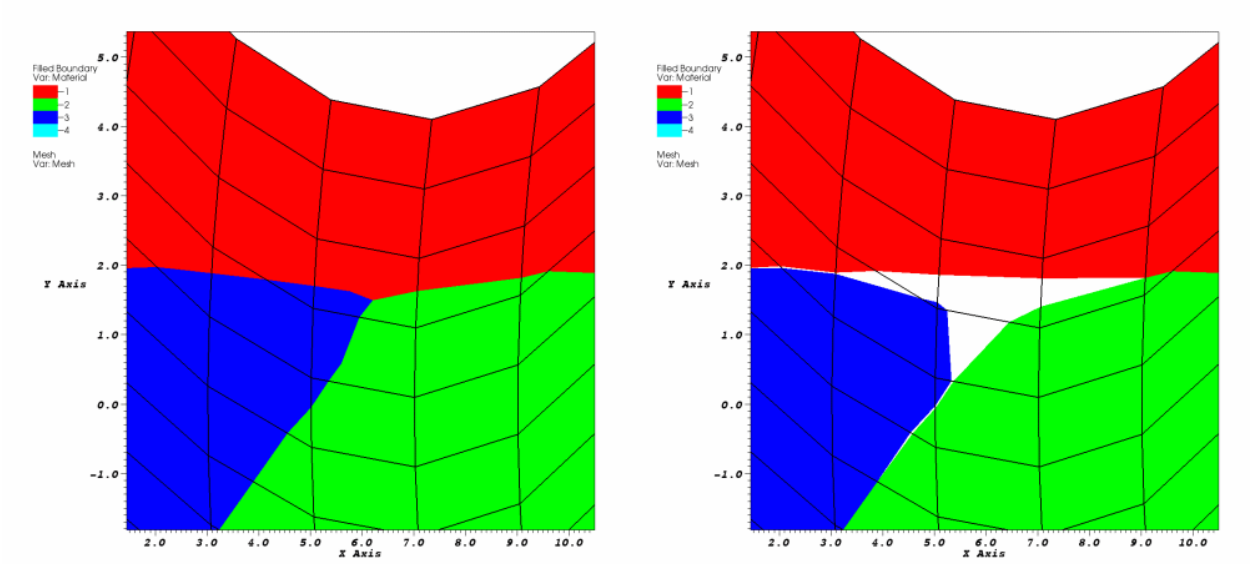


Fig. 4.236: Zoo-based MIR vs. Isovolume MIR

materials in mixed cells where the volume fraction is very small then you can try using the Isovolume MIR algorithm since it can be used to find materials whose volume fractions are above a user-specified threshold. Figure 4.237 shows an example of a dataset containing five mixed materials where the first four mixed materials are roughly equal in the amount of area that they occupy. The fifth material has a volume fraction that never exceeds 0.08 so it is omitted by the Zoo-based MIR algorithm due to its comparatively low volume fraction. To ensure that VisIt plots the fifth material, the Isosurface MIR algorithm is used with a **Volume Fraction for Isovolume** setting of 0.02. Using the Isovolume MIR algorithm with a low **Volume Fraction for Isovolume** value can find materials that have been distributed into many heavily mixed cells.

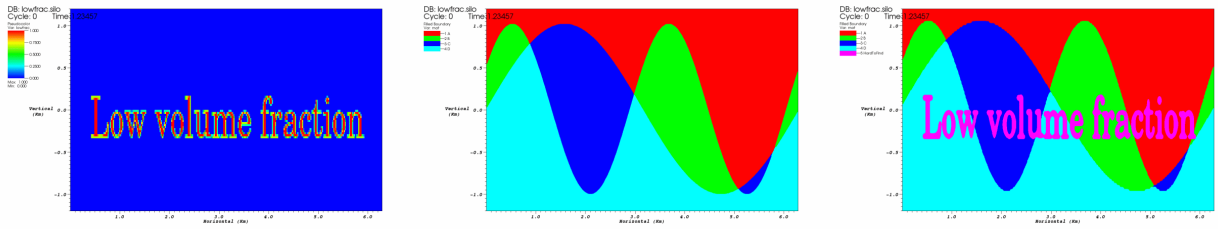


Fig. 4.237: Materials with low volume fractions can be found with the isosurface MIR algorithm

Simplifying heavily mixed cells

VisIt provides the **Simplify heavily mixed cells** check box in the **Material Reconstruction Options Window** so you can tell VisIt to throw away information materials that have low volume fractions. When you tell VisIt to omit these materials, VisIt will use less memory and will also finish MIR faster because fewer materials have to be considered. The **Simplify heavily mixed cells** check box is especially useful for databases where most of the cells are mixed or where there are many cells that contain tens of materials. When you tell VisIt to simplify heavily mixed cells, you can tell VisIt how many of the top materials to keep from each cell by entering a new number of materials into the **Maximum materials per zone** text field. By keeping the N top materials, VisIt will be sure to preserve the features that are contributed by the most dominant materials.

Smoother material boundary interfaces

VisIt's material interface reconstruction algorithm sometimes produces small, pointy outcroppings on reconstructed material boundaries next to where clean cells are located. Since these are often distracting features when looking at a visualization, VisIt provides an interface smoothing option that allows materials to bleed a little bit into clean cells to improve how they look when their material boundary is reconstructed. Figure 4.238 shows a plot that has not been smoothed next to a plot that has been smoothed. To enable interface smoothing, check the **Enable interface smoothing** check box. Note that changing this setting will not affect plots that have already been generated. If you want to make your current plots regenerate with smoother interfaces, you must also clear them out of the visualization window by choosing the **Plots** option from the **Clear** submenu located in the **Main Window's Windows** menu.

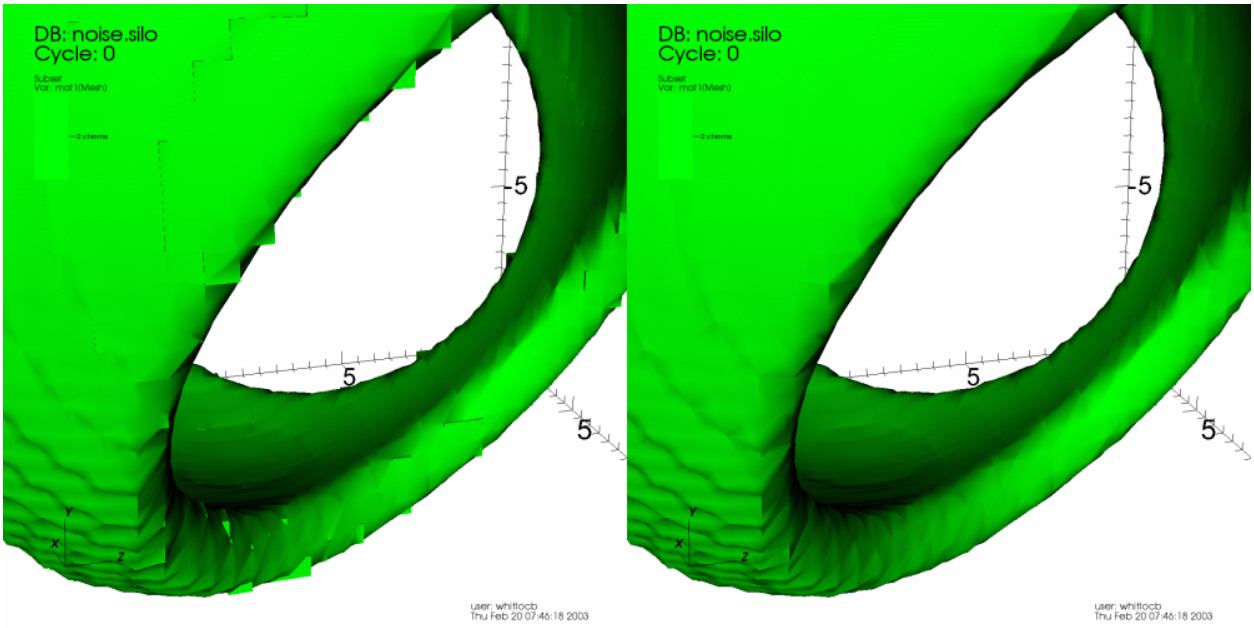


Fig. 4.238: Effect of material interface smoothing

Forcing material interface reconstruction

VisIt tries to minimize the amount of work that it must do to generate a plot so that it can be done quickly. Sometimes databases have variable information for each material in a cell instead of just having a single value for each cell or node. Because the variable is defined for each material in the cell, these variables are known as mixed variables. VisIt tends to just plot the value for the entire cell since it is more work to go through the material interface reconstruction (MIR) stage, which is usually only done when removing material subsets but is required to plot mixed variables correctly. You can force VisIt to always do MIR by checking the **Force interface reconstruction** check box. This will make mixed variables plot correctly even when you are not removing any material subsets.

Mixed variables

Some simulations write out multiple scalar values for cells that contain mixed materials so each material in the cell can have its own scalar value. Once a cell has undergone MIR, it is split into multiple cells if the original cell contained more than one material. Each split cell gets its corresponding scalar value from the original mixed variable data. The resulting plot can then display each split cell's actual value, taking into account the material boundaries. Suppose you are simulating the interaction between hot lava and ice and you have a material interface that happens to cross in the middle of a cell. Obviously each material in the cell has its own temperature. Plotting mixed variables allows the visualization to more faithfully depict the material boundaries while preserving the actual data so the multiple mix

values do not have to be averaged in the cell (see Figure 4.239). Note that VisIt does not use mixed variable values for variables that have them unless the **Force interface reconstruction** check box is enabled because most scalar fields are not mixed variables and automatically performing MIR can be expensive. If your scalars are mixed variables and you want to visualize them as such, be sure to enable the **Force interface reconstruction** check box.

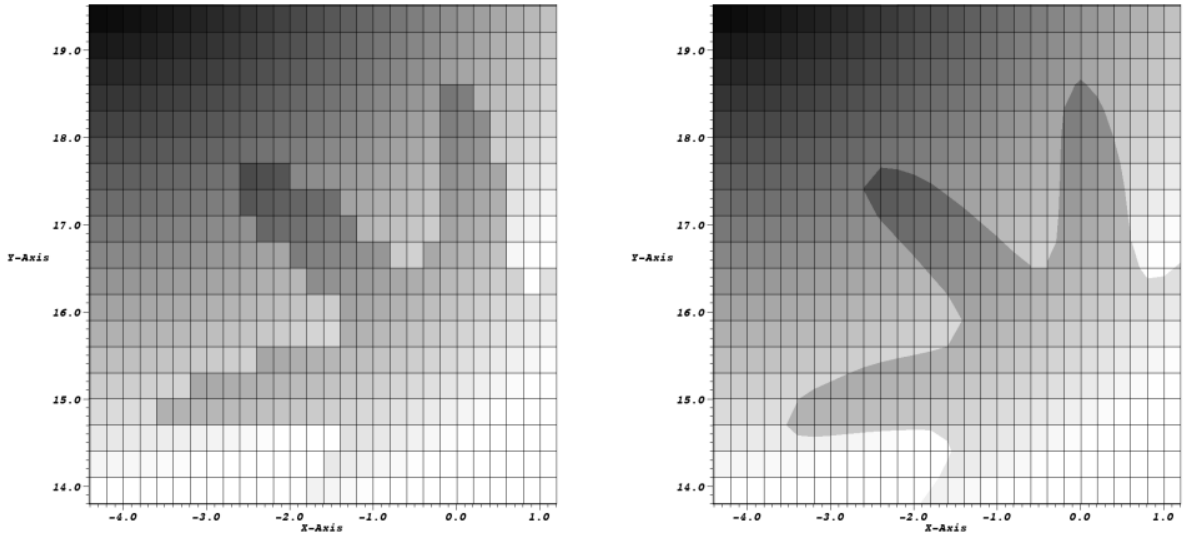


Fig. 4.239: Mixed variables can improve a visualization

4.7.5 Species

VisIt adds species, which are components of materials, to the *SIL* when they are defined by the data producer. Air is a common material in simulations since many things in the real world are surrounded by air. The chemical composition of air on Earth is roughly 78% Nitrogen, 21% oxygen, 1% Argon. One can say that if air is a material then it has species: Nitrogen, Oxygen, and Argon with mass fractions 78%, 21%, 2%, respectively. Suppose one of the calculated quantities in a database with the aforementioned air material is atmospheric temperature. Now suppose that we are examining one cell that contains only the air material from the database and its atmospheric temperature is 100 degrees Fahrenheit. If we wanted to know how much the Nitrogen contributed to the atmospheric temperature, we could multiply its concentration of 78% times the 100 degrees Fahrenheit to yield: 78 degrees Fahrenheit. Species are often used to track chemical composition of materials and their effects on various calculated quantities.

When species are defined, VisIt creates a scalar variable called *Species* and it is available in the variable menus for each plot that can accept scalar variables. The Species variable is a cell-centered scalar field defined over the whole mesh. When all species are turned on, the Species variable has the value of 1.0 over the entire mesh. When species are turned off, the Species variable is set to 1.0 minus the mass fraction of the species that was turned off. Using the previous example, if we plotted the Species variable and then turned off the air material's Nitrogen species, we would be left with only Oxygen's 21% and Argon's 1% so the species variable would be reduced to 22% or 0.22. When species are turned off, the amount of mass left to be multiplied by the plotted variable drops so the plotted variable's value in turn drops.

VisIt adds species to the *SIL* as a category that contains the various chemical constituents for all materials that have species. Since species are handled using the *SIL*, one can use VisIt's **Subset Window** to turn off species. Turning off species has quite a different effect than turning off entire materials. When materials are turned off, they no longer appear in the visualization. When species are turned off, no parts of the visualization disappear but the plotted data values may change due to drops in the Species variable.

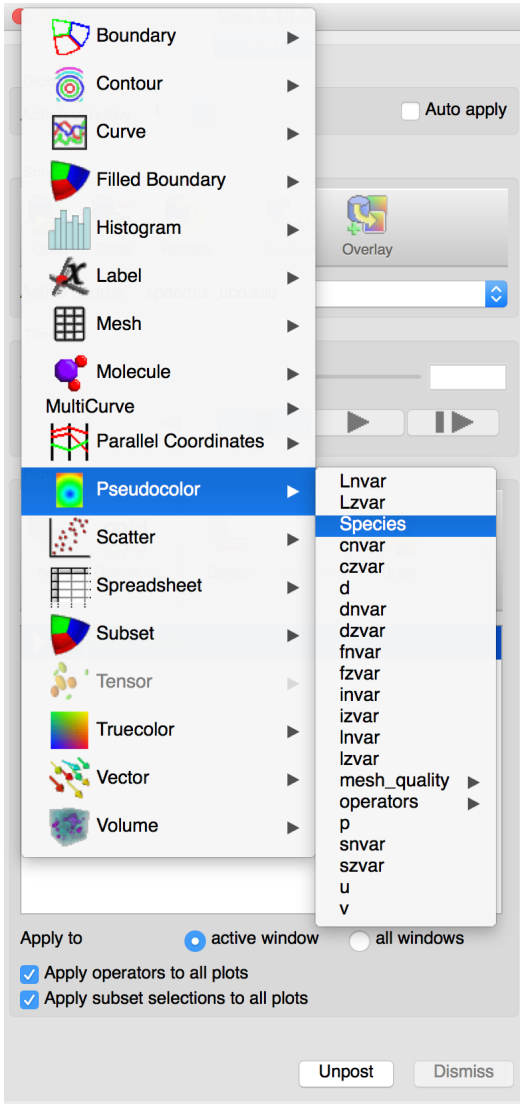


Fig. 4.240: Species variable

Plotting species

VisIt provides the Species scalar variable so users can plot or create expressions that involve species. If the user creates a Pseudocolor plot of the Species variable, the resulting plot will have a constant value of 1.0 over the entire mesh because when no species have been removed, they all sum to 1.0. Once species are removed by turning off species subsets in the **Subset Window**, the plotted value of Species changes, causing plots that use it to also change. If all but one species are removed, the plots that use the Species variable will show zero for all areas that do not contain the one selected species (see Figure 4.241). For example, if a user had air for a material and then removed every species except for oxygen, the plots that use the Species variable would show zero for every place that had no oxygen.

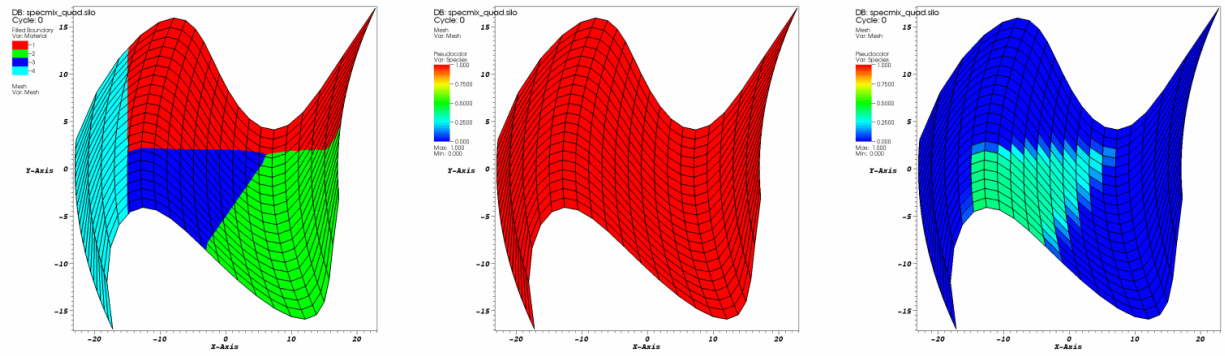


Fig. 4.241: Plots of materials and species

Turning off species

VisIt adds species information to the *SIL* as new subsets under a category called: Species. Since species are part of the SIL, users can use the **Subset Window** (see Figure 4.242) to turn off species. To access the list of species, select the Species category under the whole mesh. Once the Species category is clicked, the second pane in the **Subset Window** is populated with the species for all materials. Users can turn off the species that are not needed to look at by clicking off the check box next to the name of the species subset. When the user applies these changes, the values for the Species variable are recalculated to include only the mass fractions for the species that are still turned on.

4.8 Quantitative Analysis

Simulation data must often be compared to experimental data so VisIt provides a number of features that allow quantitative information to be extracted from simulation databases. This chapter explains how to visualize derived variables created with expressions and query information about a database. This chapter also explains VisIt's Pick, Query and Lineout capabilities which allow users to compute highly sophisticated quantitative, as opposed to visual, results.

4.8.1 Expressions

Scientific simulations often keep track of several dozen variables as they run. However, only a small subset of those variables are usually written to a simulation database to save disk space. Sometimes variables can be derived from other variables using an *expression*. VisIt provides expressions to allow scientists to create derived variables using variables that are stored in the database. Expressions are extremely powerful because they allow users to analyze new data without necessarily having to rerun a simulation. Variables created using expressions behave just like variables stored in a database; they appear in menus where database variables appear and can be visualized like any other database variable.

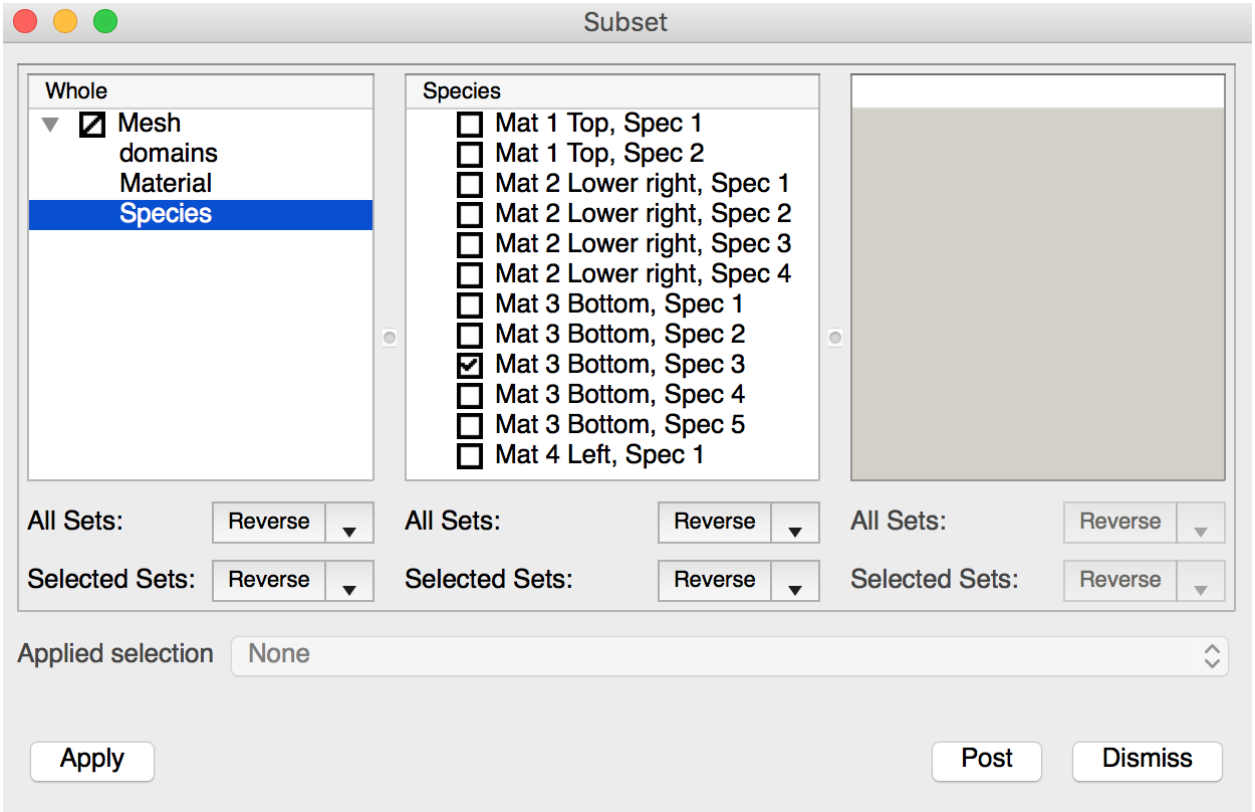


Fig. 4.242: Turning off species in the Subset Window

Expression Window

VisIt provides an **Expression Window**, shown in Figure 4.243, that allows users to create new variables that can be used in visualizations. Users can open the **Expression Window** by clicking on the **Expressions** option in the **Main Window's Controls** menu. The **Expression Window** is divided vertically into two main areas with the **Expression list** on the left and the **Definition** area on the right. The **Expression list** contains the list of expressions. The **Definition** area displays the definition of the expression that is highlighted in the **Expression list** and provides controls to edit the expression definition.

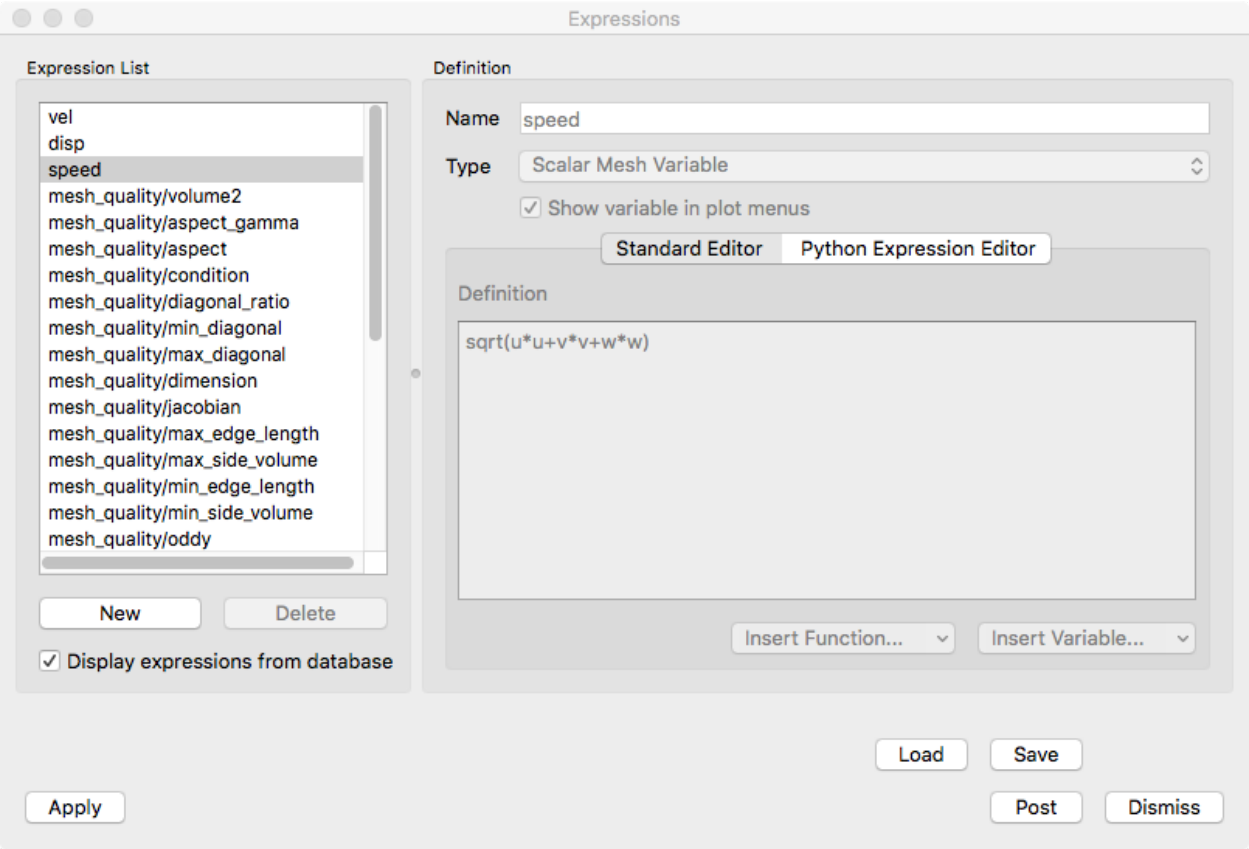


Fig. 4.243: Expression Window

Expressions in VisIt are created either manually by the user or automatically by various means including...

- Preferences
  - Mesh quality expressions
  - Time derivative expressions
  - Vector magnitude expressions
- GUI wizards
- Operators
- Databases

By default, the **Expression list** will display only those expressions created manually by the user. A check box near the bottom of the **Expression list** controls the display of automatically created expressions. When this box is checked, the **Expression list** will also include expressions created automatically by **Preferences** and **Databases** but not expressions created automatically by **GUI wizards** or **Operators**.

Creating a new expression

Users can create a new expression by clicking on the **Expression Window's New** button. When the user clicks on the **New** button, VisIt adds a new expression and shows its new, empty definition in the **Definitions** area. The initial name for a new expression is *"unnamed"* followed by some integer suffix. After the user types a new name for the expression into the **Name** text field, the expression's name in the **Expression list** will update. If the user types a name that already exists in the expression list, then Visit will automatically append a number to the end of the name to avoid duplicate expression names.

Each expression also has a **Type** that specifies the type of variable the expression produces. The available types are:

- Scalar
- Vector
- Tensor
- Symmetric Tensor
- Array
- Curve

Users must be sure to select the appropriate type for any expression they create. The selected type determines the menu in which the variable appears and subsequently the plots that can operate on the variable.

To edit an expression's definition, users can type a new expression comprised of constants, variable names, and even other VisIt expressions into the **Definition** text field. The expression definition can span multiple lines as the VisIt expression parser ignores whitespace. For a complete list of VisIt's built-in expressions, refer to the table in section *Built-in expressions*. Users can also use the **Insert Function...** menu, shown in Figure 4.244, to insert any of VisIt's built-in expressions directly into the expression definition. The list of built-in expressions is divided into certain categories as shown by the structure of the **Insert Function...** menu.

In the example shown in Figure 4.244, the **Insert Function...** operation inserted a sort of *template* for the function giving some indication of the argument(s) to the function and their meanings. Users can then simply edit those parts of the function template that need to be specified.

In addition to the **Insert Function...** menu, which lets users insert built-in functions into the expression definition, VisIt's **Expression Window** provides an **Insert Variable...** menu that allows users to insert variables from the active database into the expression definition. The **Insert Variable...** menu, shown in Figure 4.245, is broken up into Scalars, Vectors, Meshes, etc. and has the available variables under the appropriate heading so they are easy to find.

Some variables can only be expressed as very complex expressions containing several intermediate subexpressions that are only used to simplify the overall expression definition. These types of subexpressions are seldom visualized on their own. If users want to prevent them from being added to the **Plot** menu, turn off the **Show variable in plot menus** check box.

Deleting an expression

Users can delete an expression by clicking on it in the **Expression list** and then clicking on the **Delete** button. Deleting an expression removes it from the list of defined expressions and will cause unresolved references for any other expressions that use the deleted expression. If a plot uses an expression with unresolved references, VisIt will not be able to generate it until the user resolves the reference.

Expression grammar

VisIt allows expressions to be written using a host of unary and binary math operators as well as built-in and user-defined functions. VisIt's expressions follow C-language syntax, although there are a few differences. The following

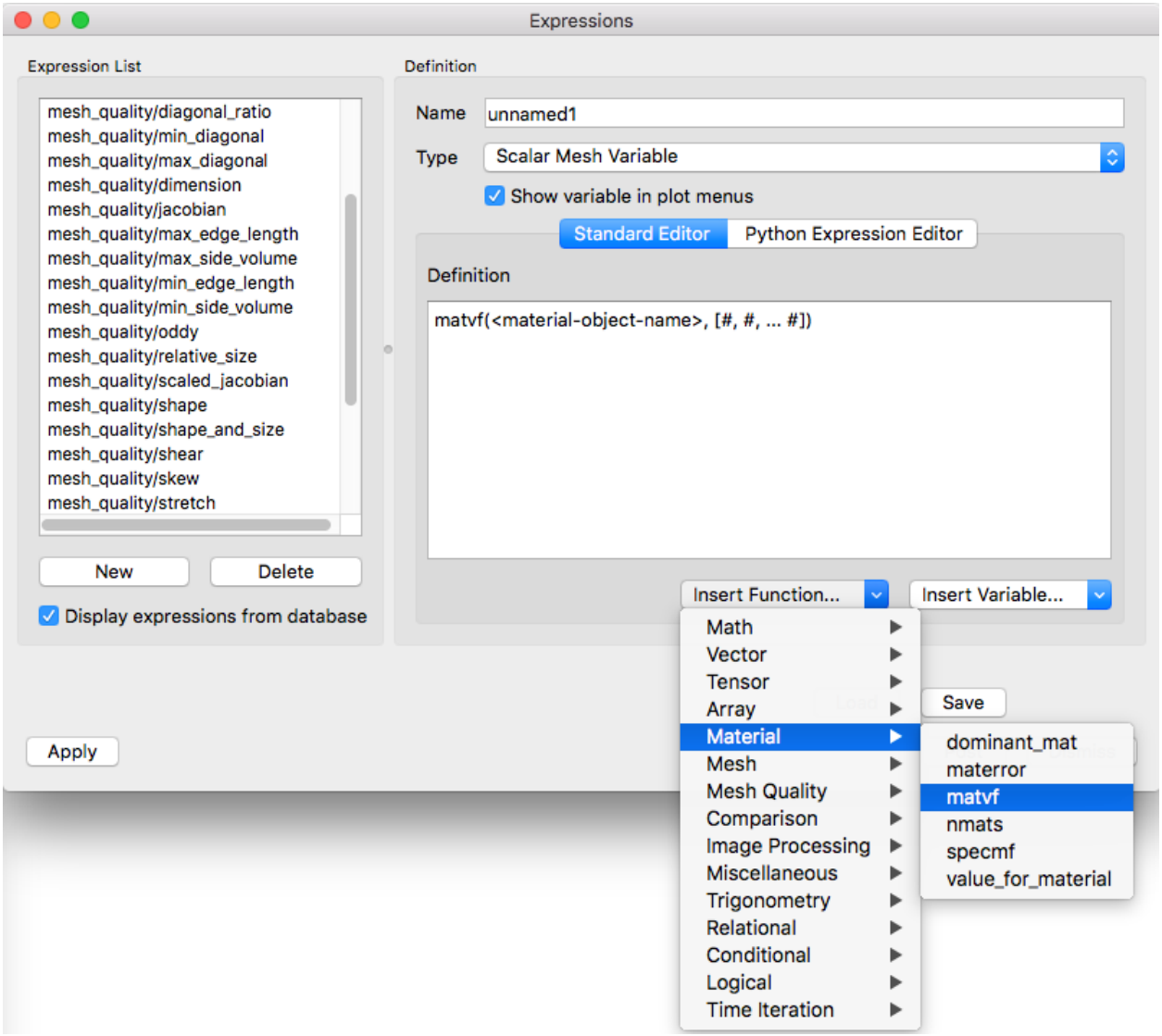


Fig. 4.244: Expression Window's Insert Function... menu

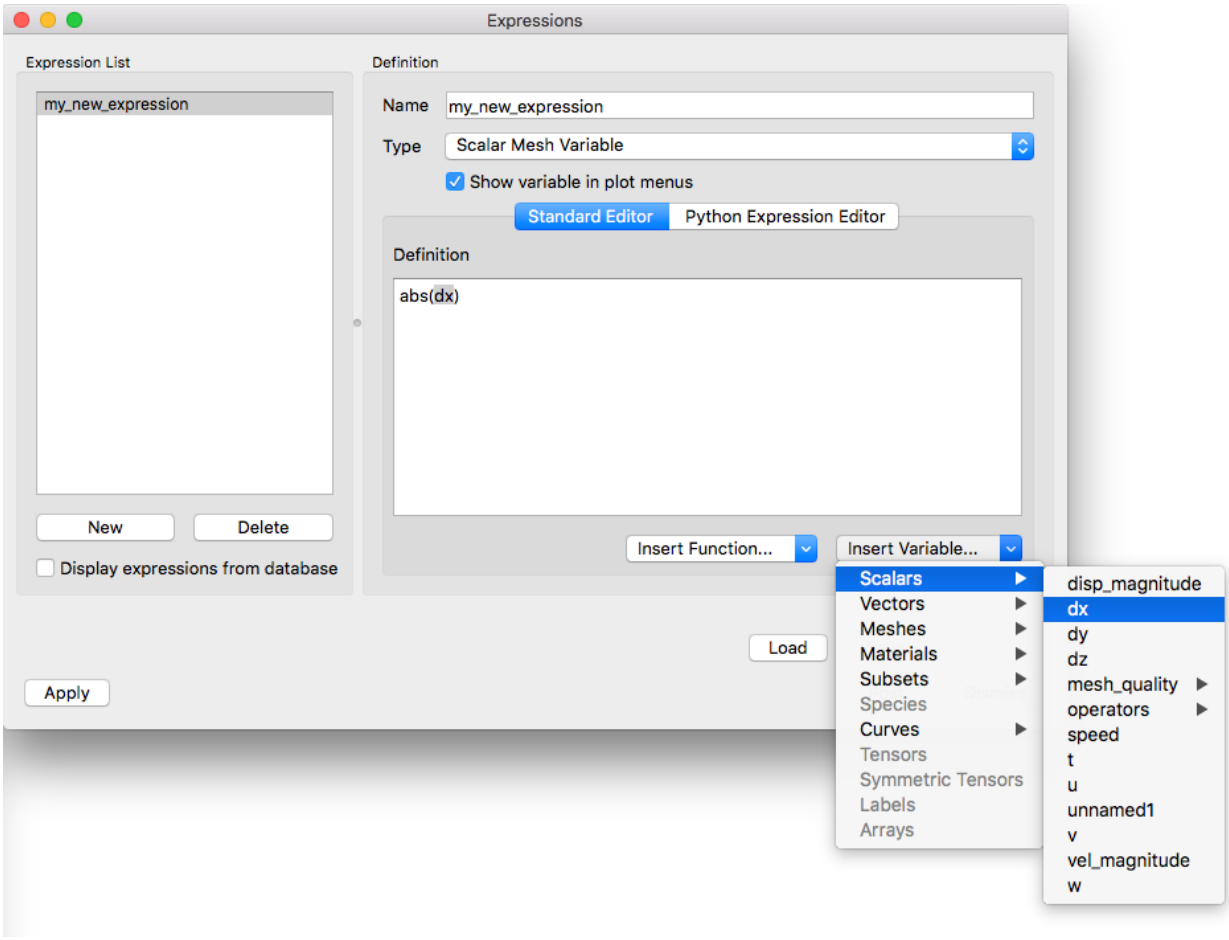


Fig. 4.245: Expression Window's Insert Variable... menu

paragraphs detail the syntax of VisIt expressions.

Math operators

These include use of +, -, *, /, ^ as addition, subtraction, multiplication, division, and exponentiation as infix operators, as well as the unary minus, in their normal precedence and associativity. Parentheses may be used as well to force a desired associativity.

*Examples: a+b^-c (a+b)*c*

Constants

Scalar constants include floating point numbers and integers, as well as booleans (true, false, on, off) and strings.

Examples: 3e4 10 "mauve" true false

Vectors

Expressions can be grouped into two or three dimensional vector variables using curly braces.

Examples: {xc, yc} {0,0,1}

Lists

Lists are used to specify multiple items or ranges, using colons to create ranges of integers, possibly with strides, or using comma-separated lists of integers, integer ranges, floating points numbers, or strings.

Examples: [1,3,2] [1:2, 10:20:5, 22] [silver, gold] [1.1, 2.5, 3.9] [level1, level2]

Identifiers

Identifiers include function names, defined variable and function names, and file variable names. They may include alphabetic characters, numeric characters, and underscores in any order. Identifiers should have at least one non-numeric character so that they are not confused with integers, and they should not look identical to floating point numbers such as 1e6.

Examples: density x y z 3d_mesh

Functions

These are used for built in functions, but they may also be used for functions/macros defined by the user. They take specific types and numbers of arguments within the parentheses, separated by commas. Some functions may accept named arguments in the form identifier=value.

Examples: sin(pi / 2) cross(vec1, {0,0,1}) my_xform(mesh1) subselect(materials=[a,b])

Database variables

These are like identifiers, but may also include periods, plus, and minus characters. A normal identifier will map to a file variable when it is not defined as another expression. To force variables that look like integers or floating point numbers to be interpreted as variable names, or to force variable names which are defined by another expression to map to a variable in a file, they should be enclosed with < and >, the left and right carats/angle brackets. Note that quotation marks will cause them to be interpreted as string constants, not variable names. In addition, variables in files may be in directories within a file, so they may include slashes in a path when in angle brackets.

Examples: density <pressure> <a.001> <a.002> <domain1/density>

Databases

A database specification looks similar to a database variable contained in angle brackets, but it is followed by a colon before the closing angle bracket, and it may also contain extra information. A database specification includes a file specification possibly followed a machine name, a time specification by itself, or a file/machine specification followed by a time specification. A file specification is just a file name with a path if needed. A machine specification is an at-sign @ followed by a host name. A time specification looks much like a list in that it contains integer numbers or ranges, or floating point numbers, separated by commas and enclosed in square brackets. However, it may also be followed by a letter c, t, or i to specify if the time specification refers to cycles, times, or indices, respectively. If no letter is specified, then the parser guesses that integers refer to cycles, floating point numbers refer to times. There is also an alternative to force indices which is the pound sign # after the opening square bracket.

Examples: </dir/file:> <file@host.gov:> <[# 0:10]:> <file[1.234]:> <file[000, 023, 047]:> <file[10]c:>

Qualified file variables

Just like variables may be in directories within a file, they may also be in other timesteps within the same database, within other databases, and even within databases on other machines. To specify where a variable is located, use the angle brackets again, and prefix the variable name with a database specification, using the colon after the database specification as a delimiter.

Examples: <file:var> </dir/file:/domain/var> <file@192.168.1.1:/var> <[#0]:zerocyclevar>

Built-in expressions

The following table lists built-in expressions that can be used to create more advanced expressions. Unless otherwise noted in the description, each expression takes scalar variables as its arguments.

Arithmetic Operator Expressions (Math Expressions)

In binary arithmetic operator expressions, each operand must evaluate to the same type field. For example, both must evaluate to a *scalar* field or both must evaluate to a *vector* field.

In addition, if the two expressions differ in centering (e.g. one is *zone* or *cell* centered or *piecewise-constant* over mesh cells while the other is *node* or *point* centered or *pieceiwse-linear* over mesh cells), VisIt will *recenter* any *node-centered* fields to *zone* centering to compute the expression. This may not always be desirable. When it is not, the *recenter()* may be used to explicitly control the centering of specific operands in an expression.

Sum Operator (+) [`exprL + exprR`] Creates a new expression which is the sum of the `exprL` and `exprR` expressions.

Difference Operator (-) [exprL - exprR] Creates a new expression which is the difference of the exprL and exprR expressions.

Product Operator (*) [exprL * exprR] Creates a new expression which is the product of the exprL and exprR expressions.

Division Operator (/) [exprL / exprR] Creates a new expression which is the quotient after dividing the exprL expression by the exprR expression.

Division Operator [divide(val_numerator, val_denominator, [div_by_zero_value, tolerance])] Creates a new expression which is the quotient after dividing the val_numerator expression by the val_denominator expression. The div_by_zero_value is used wherever the val_denominator is within tolerance of zero.

Exponent Operator (^) [exprL ^ exprR] Creates a new expression which is the product after multiplying the exprL expression by itself exprR times.

Logical AND Operator (&) [exprL & exprR] Creates a new expression which is the logical *AND* of the exprL and exprR expressions treating each value as a binary bit field. It is probably most useful for expressions involving integer data but can be applied to expressions involving any type.

Associative Operator (()) [(expr0 OP expr1)] Parenthesis, *()* are used to explicitly group partial results of sub expressions and control evaluation order.

For example, the expression (a + b) / c first computes the sum, a+b and then divides by c.

Absolute Value Function (abs()) [abs(expr0)] Creates a new expression which is everywhere the absolute value if its argument.

Ceiling Function (ceil()) [ceil(expr0)] Creates a new expression which is everywhere the *ceiling* (smallest integer greater than or equal to) of its argument.

Exponent Function (exp()) [exp(expr0)] Creates a new expression which is everywhere *e* (base of the natural logarithm) raised to the power of its argument.

Floor Function (floor()) [floor(expr0)] Creates a new expression which is everywhere the *floor* (greatest integer less than or equal to) of its argument.

Natural Logarithm Function (ln()) [ln(expr0)] Creates a new expression which is everywhere the natural logarithm of its argument.

Base 10 Logarithm Function (log10()) [log10(expr0)] Creates a new expression which is everywhere the base 10 logarithm of its argument.

Max Function (max()) [max(expr0, exrp1 [, ...])] Creates a new expression which is everywhere the maximum among all input variables.

Min Function (min()) [min(expr0, exrp1 [, ...])] Creates a new expression which is everywhere the minimum among all input variables.

Modulo Function (mod()) [mod(expr0,exrp1)] Creates a new expression which is everywhere the first argument, expr0, modulo the second argument, expr1.

Random Function (random()) [random(expr0)] Creates a new expression which is everywhere a random floating point number between 0 and 1, as computed by $(\text{rand}()\%1024) \div 1024$ where rand() is the standard C library rand() random number generator. The argument, expr0, must be a mesh variable. The seed used on each block of the mesh is the absolute domain number.

Round Function (round()) [round(expr0)] Creates a new expression which is everywhere the result of rounding its argument.

Square Function (sqr()) [sqr(expr0)] Creates a new expression which is everywhere the result of squaring its argument.

Square Root Function (`sqrt()`) [`sqrt(expr0)`] Creates a new expression which is everywhere the square root of its argument.

Relational, Conditional and Logical Expressions

The `if()` conditional expression is designed to be used in concert with the **Relational** and **Logical** expressions. Together, these expressions can be used to build up more complex expressions in which very different evaluations are performed depending on the outcome of other evaluations. For example, the `if()` conditional expression can be used together with one or more relational expressions to create a new expression which evaluates to a dot-product on part of a mesh and to the magnitude of a divergence operator on another part of a mesh. However, the **Relational** and **Logical** expressions alone (e.g. when not used *within* an `if()` expression) do not produce a useful result.

If Function (`if()`) [`if(exprCondition, exprTrue, exprFalse)`] Creates a new expression which is equal to `exprTrue` wherever the condition, `exprCondition` is non-zero and which is equal to `exprFalse` wherever `exprCondition` is zero.

For example, the expression `if(and(gt(pressure, 2.0), lt(pressure, 4.0)), pressure, 0.0)` combines the `if` expression with the `gt` and `lt` expressions to create a new expression that is equal to `pressure` wherever it is between 2.0 and 4.0 and 0 otherwise.

Equal Function (`eq()`) [`eq(exprL,exprR)`] Creates a new expression which is everywhere a boolean value (1 or 0) indicating whether its two arguments are equal. A value of 1 is produced everywhere the arguments *are* equal and 0 otherwise.

Greater Than Function (`gt()`) [`gt(exprL,exprR)`] Creates a new expression which is everywhere a boolean value (1 or 0) indicating whether `exprL` is greater than `exprR`. A value of 1 is produced everywhere `exprL` is greater than `exprR` and 0 otherwise.

Greater Than or Equal Function (`ge()`) [`ge(exprL,exprR)`] Creates a new expression which is everywhere a boolean value (1 or 0) indicating whether `exprL` is greater than or equal to `exprR`. A value of 1 is produced everywhere `exprL` is greater than or equal to `exprR` and 0 otherwise.

Less Than Function (`lt()`) [`lt(exprL,exprR)`] Creates a new expression which is everywhere a boolean value (1 or 0) indicating whether `exprL` is less than `exprR`. A value of 1 is produced everywhere `exprL` is less than `exprR` and 0 otherwise.

Less Than or Equal Function (`le()`) [`le(exprL,exprR)`] Creates a new expression which is everywhere a boolean value (1 or 0) indicating whether `exprL` is less than or equal to `exprR`. A value of 1 is produced everywhere `exprL` is less than or equal to `exprR` and 0 otherwise.

Equal Function (`ne()`) [`ne(exprL,exprR)`] Creates a new expression which is everywhere a boolean value (1 or 0) indicating whether its two arguments are *not* equal. A value of 1 is produced everywhere the arguments are *not* equal and 0 otherwise.

Logical And Function (`and()`) [`and(exprL,exprR)`] Creates a new expression which is everywhere the logical *and* of its two arguments. Non-zero values are treated as true whereas zero values are treated as false.

Logical Or Function (`or()`) [`or(exprL,exprR)`] Creates a new expression which is everywhere the logical *or* of its two arguments. Non-zero values are treated as true whereas zero values are treated as false.

Logical Not Function (`not()`) [`not(expr0)`] Creates a new expression which is everywhere the logical *not* of its argument. Non-zero values are treated as true whereas zero values are treated as false.

Trigonometric Expressions

Arc Cosine Function (`acos()`) [`acos(expr0)`] Creates a new expression which is everywhere the arc cosine of its argument. The returned value is in *radians*.

Arc Sine Function (`asin()`) [`asin(expr0)`] Creates a new expression which is everywhere the arc sine of its argument. The returned value is in *radians*.

Arc Tangent Function (`atan()`) [`atan(expr0)`] Creates a new expression which is everywhere the arc tangent of its argument. The returned value is in *radians*.

Arc Tangent 2 Function (`atan2()`) [`atan2(expr0,expr1)`] Creates a new expression which is everywhere the arc tangent of its arguments. The returned value is in *radians*.

Cosine Function (`cos()`) [`cos(expr0)`] Creates a new expression which is everywhere the cosine of its argument. The argument is treated as in units of *radians*.

Hyperbolic Cosine Function (`cosh()`) [`cosh(expr0)`] Creates a new expression which is everywhere the hyperbolic cosine of its argument. The argument is the *hyperbolic angle*.

Sine Function (`sin()`) [`sin(expr0)`] Creates a new expression which is everywhere the sine of its argument. The argument is treated as in units of *radians*.

Hyperbolic Sine Function (`sinh()`) [`sinh(expr0)`] Creates a new expression which is everywhere the hyperbolic sine of its argument. The argument is the *hyperbolic angle*.

Tangent Function (`tan()`) [`tan(expr0)`] Creates a new expression which is everywhere the tangent of its argument. The argument is treated as in units of *radians*.

Hyperbolic Tangent Function (`tanh()`) [`tanh(expr0)`] Creates a new expression which is everywhere the hyperbolic tangent of its argument. The argument is the *hyperbolic angle*.

Degrees To Radians Conversion Function (`deg2rad()`) [`deg2rad(expr0)`] Creates a new expression which is everywhere the conversion from degrees to radians of its argument. The argument should be a variable defined in units of degrees.

Radians To Degrees Conversion Function (`rad2deg()`) [`rad2deg(expr0)`] Creates a new expression which is everywhere the conversion from radians to degrees of its argument. The argument should be a variable defined in units of radians.

Vector and Color Expressions

Vector Compose Operator (`{}`) [`{expr0, expr1, ... , exprN-1}`] Curly braces, `{}`, are used to create a vector from scalars or a tensor from vectors. A common use is to compose 3 scalar expressions to form a vector expression as in `{a, b, c}`. For 2D data, the 3rd component must still be provided and must be zero as in `{a,b,0}`. The component expressions, `expr0`, `expr1`, etc. must all be the same type (e.g. scalar, vector) and must all be the same centering. Scalars compose into (row) vectors and (row) vectors compose into tensors, row-by-row.

If constant values (e.g. `1` or `0`) are needed in composing a vector expression, then use the expression functions designed to create constant expressions such as `nodal_constant(<mesh>,value)` (for node-centered constant expressions) or `zonal_constant(<mesh>,value)` (for zone-centered consntant expressions). Using the constant values themselves (e.g. `0` or `1`) directly in the compose operator does not always work as expected depending on VisIt's ability to infer the intended *mesh* and/or *centering*.

Vector Component Operator (`[]`) [`expr[I]`] Square brackets, *[]*, are used to create a new expression of lower tensor rank by extracting a component from an expression of higher tensor rank. Components are indexed starting from 0. If `expr` is a tensor of rank 2, the result will be a tensor of rank 1 (e.g. a vector). If `expr` is a tensor of rank 1, the result will be a tensor of rank 0 (e.g. a scalar). To obtain the `J`-th component of the `I`-th row of a tensor of rank 2, the expression would be `expr[I][J]`

Color Function (`color()`) [`color(exprR,exprG,exprB)`] Creates a new, RGB *vector*, expression which defines a *color* vector where `exprR` defines the *red* component, `exprG` defines the *green* component and `exprB`

defines the *blue* component of the color vector. The resulting expression is suitable for plotting with the *True-color Plot*. The arguments are used to define color values in the range 0...255. Values outside that range are clamped. No normalization is performed. If the arguments have much smaller or larger range than [0...255], it may be appropriate to select a suitable multiplicative scale factor.

Color4 Function (`color4()`) [`color4(exprR,exprG,exprB,exprA)`] See *color()*. This function is similar to the `color()` function but also supports *alpha-transparency* as the fourth argument, again in the range 0...255.

Color lookup Function (`colorlookup()`) [`colorlookup(expr0,tabname,scalmode,skewfac)`] Creates a new *vector* expression that is the color that each value in `expr0` maps to. The `tabname` argument is the name of the color table. The `expr0` and `tabname` arguments are *required*. The `scalmode` and `skewfac` arguments are optional. Possible values for `scalmode` are `0` (for *linear* scaling mode), `1` (for *log* scaling mode) and `2` (for *skew* scaling mode). The `skewfac` argument is *required* only for a `scalmode` of `2`.

Cross Product Function (`cross()`) [`cross(exprV0,exprV1)`] Creates a new *vector* expression which is the vector cross product created by crossing `exprV0` *into* `exprv1`. Both arguments must be *vector* expression.

Dot Product Function (`dot()`) [`dot(exprV0,exprV1)`] Creates a new *scalar* expression which is the vector dot product of `exprV0` with `exprV1`.

HSV Color Function (`hsvcolor()`) [`hsvcolor(exprH,exprS,exprV)`] See *color()*. This function is similar to the `color()` function but takes *Hue*, *Saturation* and *Value* (Lightness) arguments as inputs and produces an RGB *vector* expression.

Magnitude Function (`magnitude()`) [`magnitude(exprV0)`] Creates a new *scalar* expression which is everywhere the magnitude of the `exprV0`.

Normalize Function (`normalize()`) [`normalize(exprV0)`] Creates a new *vector* expression which is everywhere a normalized vector (e.g. same direction but unit magnitude) of `exprV0`.

Curl Function: `curl()` [`curl(expr0)`] Creates a new *vector* expression which is everywhere the curl of its input argument, which must be vector valued. In a 3D context, the result is also a vector. However, in a 2D context the result *vector* would always be `[0,0,V]` so expression instead returns only the *scalar* V.

Divergence Function: `divergence()` [`divergence(expr0)`] Creates a new *scalar* expression which is everywhere the divergence of its input argument, which must be vector valued.

Gradient Function: `gradient()` [`gradient(expr0)`] Creates a new *vector* expression which is everywhere the gradient of its input argument, which must be *scalar*. The method of calculation varies depending on the type of mesh upon which the input is defined. See also *ij_gradient()* and *ijk_gradient()*.

IJ_Gradient Function: `ij_gradient()` [`ij_gradient(expr0)`] No description available.

IJK_Gradient Function: `ijk_gradient()` [`ijk_gradient(expr0)`] No description available.

Laplacian Function: `laplacian()` [`laplacian(expr0)`] No description available.

Surface Normal Function: `surface_normal()` [`surface_normal(expr0)`] This function is an *alias* for *cell_surface_normal()*

Point Surface Normal Function: `point_surface_normal()` [`point_surface_normal(expr0)`] Like *cell_surface_normal()* except that after computing face normals, they are averaged to the nodes.

Cell Surface Normal Function: `cell_surface_normal()` [`cell_surface_normal(<Mesh>)`] Computes a *vector* variable which is the normal to a *surface*. The input argument is a *Mesh* variable. In addition, this function cannot be used in isolation. It must be used in combination the *external surface*, *first*, and the *defer expression*, *second*, operators.

Edge Normal Function: `edge_normal()` [`edge_normal(expr0)`] No description available.

Point Edge Normal Function: `point_edge_normal()` [`point_edge_normal(expr0)`] No description available.

Cell Edge Normal Function: `cell_edge_normal()` [`cell_edge_normal(expr0)`] No description available.

Tensor Expressions

Tensor expressions can be constructed either by direct *composition* (e.g. using the compose operator, `{}`) or by using a tensor expression function.

When *composing* tensors with the compose operator, `{}`, 9 scalar (e.g. `{{xx,xy,xz},{yx,yy,yz},{zx,zy,zz}}` or 3 vector expressions (e.g. `{r1,r2,r3}` are typically used. However, tensors can be composed from combinations of these as well (e.g. `{r1,{a,b,c},r3}`). Note that in the preceding example expressions, the compose operator is used in a nested manner twice. The inner instances compose sets of scalars into row vectors and the outer instance composes the row vectors into the final tensor. Tensor expressions in 2D still require 9 scalar components but those in the 3rd row and column must be all zeros. Symmetric tensor expressions also still require 9 scalar components but must also exhibit symmetry.

If constant values (e.g. `1` or `0`) are needed in composing a tensor expression, then use the expression functions designed to create constant expressions such as `nodal_constant(<mesh>,value)` (for node-centered constant expressions) or `zonal_constant(<mesh>,value)` (for zone-centered consntant expressions). Using the constant values themselves (e.g. `0` or `1`) directly in the compose operator does not always work as expected depending on VisIt's ability to infer the intended *mesh* and/or *centering*.

Often, using the tensor expression functions described here necessitates a detailed understanding of the actual numerical calculations VisIt uses in evaluating the expressions. Therefore, in many cases here, we provide collapsible sections that can be expanded to show the actual C++ source code VisIt is compiled with to compute a given tensor expression. The code displayed in these sections is derived from links to the actual source code files. So, the reader can be assured it is a faithful representation of the numerical operations VisIt is actually performing.

Contraction Function: `contraction()` [`contraction(expr0)`] Creates a *scalar* expression which is everywhere the *contraction* of `expr0` which must be a *tensor* valued expression. The contraction is the sum of pairwise dot-products of each of the column vectors of the tensor with itself as shown in the code snip-it below.

Show/Hide Code for `contraction()`

```
// Conceptually it is like as doting each column vector with
// itself and adding the column results
//

ctract +=vals[0] * vals[0] + vals[1] * vals[1] + vals[2] * vals[2];
ctract +=vals[3] * vals[3] + vals[4] * vals[4] + vals[5] * vals[5];
ctract +=vals[6] * vals[6] + vals[7] * vals[7] + vals[8] * vals[8];
```

Determinant Function: `determinant()` [`determinant(expr0)`] Creates a *scalar* expression which is everywhere the determinant of `expr0` which must be *tensor* valued.

Effective Tensor Function: `effective_tensor()` [`effective_tensor(expr0)`] Creates a *scalar* expression which is everywhere the square root of three times the *second principal invariant of the stress deviator tenosr*, $\sqrt{3 * J_2}$, where J_2 is the *second principal invariant of the stress deviator tensor*. This is also known as the *von Mises stress* or the *Huber-Mises stress* or the *Mises effective stress*.

Show/Hide Code for `effective_tensor()`

```
double s11 = vals[0], s12 = vals[1], s13 = vals[2];
double s21 = vals[3], s22 = vals[4], s23 = vals[5];
double s31 = vals[6], s32 = vals[7], s33 = vals[8];

// First invariant of the stress tensor
```

(continues on next page)

(continued from previous page)

```
// aka "pressure" of incompressible fluid in motion
// aka "mean effective stress"
double trace = (s11 + s22 + s33) / 3.;

// components of the deviatoric stress
double dev0 = s11 - trace;
double dev1 = s22 - trace;
double dev2 = s33 - trace;

// The second invariant of the stress deviator
// aka "J2"
double out2 = 0.5*(dev0*dev0 + dev1*dev1 + dev2*dev2) +
              s12*s12 + s13*s13 + s23*s23;

// stress deviator
out2 = sqrt(3.*out2);
```

Eigenvalue Function: `eigenvalue()` [`eigenvalue(expr0)`] The `expr0` argument must evaluate to a 3x3 *symmetric* tensor. The eigenvalue expression returns the eigenvalues of the 3x3 *symmetric* matrix argument as a vector valued expression where each eigenvalue is a component of the vector. Use the vector component operator, *[]*, to access individual eigenvalues. If a non-symmetric tensor is supplied, results are indeterminate.

Eigenvector Function: `eigenvector()` [`eigenvector(expr0)`] The `expr0` argument must evaluate to a 3x3 *symmetric* tensor. The eigenvector expression returns the eigenvectors of the 3x3 matrix argument as a tensor (3x3 matrix) valued expression where each column in the tensor is one of the eigenvectors.

In order to use the vector component operator *[]*, to access individual eigenvectors, the result must be *transposed* with the *transpose()*, expression function.

For example, if `evecs = transpose(eigenvector(tensor))`, the expression `evecs[1]` will return the second eigenvector.

Inverse Function: `inverse()` [`inverse(expr0)`] Creates a new tensor expression which is everywhere the inverse of its input argument, which must also be a tensor.

Principal Deviatoric Tensor Function: `principal_deviatoric_tensor()`
[`principal_deviatoric_tensor(expr0)`] Deviatoric stress is the stress tensor which results after subtracting the hydrostatic stress tensor. Hydrostatic stress is a *scalar* quantity also often referred to as *average pressure* or just *pressure*. However, it is often characterized in *tensor* form by multiplying it through a 3x3 identity matrix.

The `principal_deviatoric_tensor()` expression function creates a new *vector* expression which is everywhere the principal components of the deviatoric stress tensor computed from the *symmetric* tensor argument `expr0`. In other words, the *eigenvalues* of the deviatoric stress tensor.

Potentially, it would be more appropriate to create a new *tensor* field here with all zeros for off-diagonal elements and the eigenvalues on the main diagonal.

This expression can also be computed by using a combination of the `trace()` and `principal_tensor()` expression functions. The `trace()` (divided by 3) would be used to subtract out hydrostatic stress and the result could be used in the `principal_tensor()` expression to arrive at the same result.

Show/Hide Code for `principal_deviatoric_tensor()`

```
double pressure = -(vals[0] + vals[4] + vals[8]) / 3.;
double dev0 = vals[0] + pressure;
double dev1 = vals[4] + pressure;
double dev2 = vals[8] + pressure;
```

(continues on next page)

(continued from previous page)

```
// double invariant0 = dev0 + dev1 + dev2;
double invariant1 = 0.5*(dev0*dev0 + dev1*dev1 + dev2*dev2);
invariant1 += vals[1]*vals[1] + vals[2]*vals[2] + vals[5]*vals[5];
double invariant2 = -dev0*dev1*dev2;
invariant2 += -2.0 *vals[1]*vals[2]*vals[5];
invariant2 += dev0*vals[5]*vals[5];
invariant2 += dev1*vals[2]*vals[2];
invariant2 += dev2*vals[1]*vals[1];

double princ0 = 0.;
double princ1 = 0.;
double princ2 = 0.;
if (invariant1 >= 1e-100)
{
    double alpha = -0.5*sqrt(27./invariant1)
                       *invariant2/invariant1;
    if (alpha < 0.)
        alpha = (alpha < -1. ? -1 : alpha);
    if (alpha > 0.)
        alpha = (alpha > +1. ? +1 : alpha);

    double angle = acos((double)alpha) / 3.;
    double value = 2.0 * sqrt(invariant1 / 3.);
    princ0 = value*cos(angle);
    angle = angle - 2.0*vtkMath::Pi()/3.;
    princ1 = value*cos(angle);
    angle = angle + 4.0*vtkMath::Pi()/3.;
    princ2 = value*cos(angle);
}

double out3[3];
out3[0] = princ0;
out3[1] = princ1;
out3[2] = princ2;
```

Principal Tensor Function: `principal_tensor()` `[principal_tensor(expr0)]` Creates a new *vector* expression which is everywhere the principal stress components of the input argument, which must a *symmetric* tensor. The principal stress components are the eigenvalues of the stress tensor. So, the vector expression computed here is the same as *eigenvalue()*.

Potentially, it would be more appropriate to create a new *tensor* field here with all zeros for off-diagonal elements and the eigenvalues on the main diagonal.

Transpose Function: `transpose()` `[transpose(expr0)]` Creates a new tensor expression which is everywhere the transpose of its input argument which must also be a tensor. The first row vector in the input becomes the first column vector in the output, etc.

Tensor Maximum Shear Function: `tensor_maximum_shear()` `[tensor_maximum_shear(expr0)]` Creates a new *Scalar* expression which is everywhere the *maximum shear stress* as defined in *J.C. Ugural and S.K. Fenster "Advanced Strength and Applied Elasticity", Prentice Hall 4th Edition, page 81.* the specific mathematical operations of which are shown in the code snip-it below.

Show/Hide Code for `tensor_maximum_shear()`

```
; Solution of Stress Cubic  S^3 - I1 * S^2 + I2 *S - I3 = 0 for roots S1, S2, and S3.  J.E. Akin, 2007
;Refer: A.C. Ugural & S.K. Fenster, Advanced Strength and Applied Elasticity , Prentice Hall, 4th Ed
I1 = Sxx + Syy + Szz                                                    ; invariants
I2 = Sxx*Syy + Sxx*Szz + Syy*Szz - Txy^2 - Tyz^2 -Txz^2
I3 = Sxx*Syy*Szz + 2*Txy*Tyz*Txz - Sxx*Tyz^2 - Syy*Txz^2  - Szz*Txy^2
Hydro = I1 / 3                                                          ; hydrostatic component

Q = I1*I2 / 3 - I3 - 2*I1^3 /27   ;  work variable
R = I1^2 /3 - I2                  ;  work variable
S = sqrt (R / 3)                  ;  work variable
T = sqrt (R^3 /27)                ;  work variable
α = acosd (-Q/T/2)                ;  work variable

Sa = 2*S*cosd(α/3)          + I1 / 3          ; recover roots
Sb = 2*S*(cosd(α/3 + 120)) + I1 / 3
Sc = 2*S*(cosd(α/3 + 240)) + I1 / 3

S1 = MAX (Sa, Sb, Sc)                         ; rank the roots
S3 = MIN (Sa, Sb, Sc)
S2 = Sa + Sb + Sc - S1 - S3
SD1 = S1 - Hydro                              ; principal deviatoric stress
SD2 = S2 - Hydro                              ; principal deviatoric stress
SD3 = S3 - Hydro                              ; principal deviatoric stress
T21 = (S1 - S2) / 2                           ; define maximum shear stresses
T32 = (S2 - S3) / 2
T13 = (S1 - S3) / 2
```

```
            double *vals = in->GetTuple9(i);
            double s11 = vals[0], s12 = vals[1], s13 = vals[2];
            double s21 = vals[3], s22 = vals[4], s23 = vals[5];
            double s31 = vals[6], s32 = vals[7], s33 = vals[8];

            // Hydro-static component
            double pressure = (s11 + s22 + s33) / 3.;

            // Deviatoric stress components
            double dev0 = s11 - pressure;
            double dev1 = s22 - pressure;
            double dev2 = s33 - pressure;

            // double invariant0 = dev0 + dev1 + dev2;
            // Second invariant of stress deviator
            double invariant1 = 0.5*(dev0*dev0 + dev1*dev1 + dev2*dev2);
            invariant1 += s12*s12 + s13*s13 + s23*s23;

            // Third invariant of stress deviator
            double invariant2 = -dev0*dev1*dev2;
            invariant2 += -2.0*s12*s13*s23;
            invariant2 +=     dev0*s23*s23;
```

(continues on next page)

(continued from previous page)

```
            invariant2 +=     dev1*s13*s13;
            invariant2 +=     dev2*s12*s12;

            // Cubic roots of the characteristic equation
            // http://mathworld.wolfram.com/CubicFormula.html
            double princ0 = 0.;
            double princ2 = 0.;
            if (invariant1 >= 1e-100)
            {
                double alpha = -0.5*sqrt(27./invariant1)
                                   *invariant2/invariant1;
                if (alpha < 0.)
                    alpha = (alpha < -1. ? -1 : alpha);
                if (alpha > 0.)
                    alpha = (alpha > +1. ? +1 : alpha);

                double angle = acos((double)alpha) / 3.;
                double value = 2.0 * sqrt(invariant1 / 3.);
                princ0 = value*cos(angle);
                // Displace the angle for princ1 (which we don't calculate)
                angle = angle - 2.0*vtkMath::Pi()/3.;
                // Now displace for princ2
                angle = angle + 4.0*vtkMath::Pi()/3.;
                princ2 = value*cos(angle);
            }

            // set the output value
```

Trace Function: `trace()` [`trace(expr0)`] Creates a new scalar expression which is everywhere the trace of `expr0` which must be a 3x3 tensor. The trace is the sum of the diagonal elements.

Viscous Stress Function: `viscous_stress()` [`viscous_stress(expr0)`] Creates a new tensor expression which is everywhere the viscous stress. The key difference between *viscous* stress and *elastic* stress (which is the kind of stress many of the other functions here deal with) is that *viscous* stress is related to the *rate of change* of deformation whereas *elastic* stress is related to the *amount* of deformation. These two are related in the same way velocity and distance are related.

The argument here, `expr0` is a *vector* valued velocity. In addition, the current implementation of this function works only for 2D, structured gridded meshes.

Show/Hide Code for `viscous_stress()`

```
    dx[0] = .5 * (px[0] + px[1] - px[2] - px[3]);
    dx[1] = .5 * (px[1] + px[2] - px[3] - px[0]);

    dy[0] = .5 * (py[0] + py[1] - py[2] - py[3]);
    dy[1] = .5 * (py[1] + py[2] - py[3] - py[0]);

    du[0] = .5 * (vx[0] + vx[1] - vx[2] - vx[3]);
    du[1] = .5 * (vx[1] + vx[2] - vx[3] - vx[0]);

    dv[0] = .5 * (vy[0] + vy[1] - vy[2] - vy[3]);
    dv[1] = .5 * (vy[1] + vy[2] - vy[3] - vy[0]);

    div = 1.0 / (dx[0] *dy[1] - dx[1] *dy[0] + tiny);

    dvx[0] = div * (du[0]*dy[1] - du[1] * dy[0]);
```

(continues on next page)

(continued from previous page)

```
    dvx[1] = div * (du[1]*dx[0] - du[0] * dx[1]);

    dvy[0] = div * (dv[0]*dy[1] - dv[1] * dy[0]);
    dvy[1] = div * (dv[1]*dx[0] - dv[0] * dx[1]);

    // create the tensor

    // if rz mesh include extra divergence term
    if( rz_mesh )
    {
        cyl_term = (vy[0] + vy[1] + vy[2] + vy[3]) /
                   (py[0] + py[1] + py[2] + py[3] + tiny);
    }

    // diag terms
    vstress[0] = 1/3.0 * (2.0 * dvx[0] - dvy[1]- cyl_term);
    vstress[4] = 1/3.0 * (2.0 * dvy[1] - dvx[0]- cyl_term);
    vstress[8] = 0.0;
    // other terms
    vstress[1] = 0.5 * (dvy[0]  + dvx[1]);
    vstress[2] = 0.0;
    vstress[5] = 0.0;

    // use symm to fill out remaining terms
    vstress[3] = vstress[1];
    vstress[6] = vstress[2];
    vstress[7] = vstress[5];
}
```

Array Expressions

Array Compose Function: `array_compose()` `[array_compose(expr1, expr2, ..., exprK)]`
Create a new *array* expression variable which is everywhere the array composition of its arguments, which all must be *scalar* type. An array mesh variable is useful when using the label plot or when doing picks and wanting pick values to always return a certain selected set of mesh variables. But, all an array mesh variable really is is a convenient container to hold a group of individual scalar mesh variables. Each argument to the array_compose expression must evaluate to a scalar expression and all of the input expressions must have the same centering. Array variables are collections of scalar variables that are commonly used with certain plots to display the contents of multiple variables simultaneously. For example, the Label plot can display the values in an array variable.

Array Compose With Bins Function: `array_compose_with_bins()` `[array_compose_with_bins(expr1, expr2,...,exprK,[b0,...bK+1])]` This expression combines two related concepts. One is the array concept where a group of individual scalar mesh variables are grouped into an array variable. The other is a set of *coordinate* values (interpreted as histogram bin boundaries), that will be used by VisIt for certain kinds of operations involving the array variable. **Note** that the bin boundaries are specified as a single additional argument to the function as a *list* of values embedded in square brackets. If there are `K` variables in the array, `expr1, expr2,..., exprK`, there are K+1 coordinate values (or bin boundaries), `[b0,b1,...,bK+1]`. When such a variable is picked using one of VisIt's pick operations, VisIt can plot a **Histogram** plot. Each bar in the **Histogram** plot has a height determined by the associated member of the array and a width determined by the associated bin-boundaries.

For example, suppose a user had an array variable, foo, composed of 5 scalar mesh variables, `a1`, `a2`, `a3`, `a4`, and `a5` like so...

`array_compose_with_bins(a1,a2,a3,a4,a5,[0,3.5,10.1,10.7,12,22])`

For any given point on a plot, when the user picked foo, there are 5 values returned, the value of each of the 5 scalar variable members of foo. If the user arranged for a pick to return a bar-graph of the variable using the bin-boundaries, the result might look like...

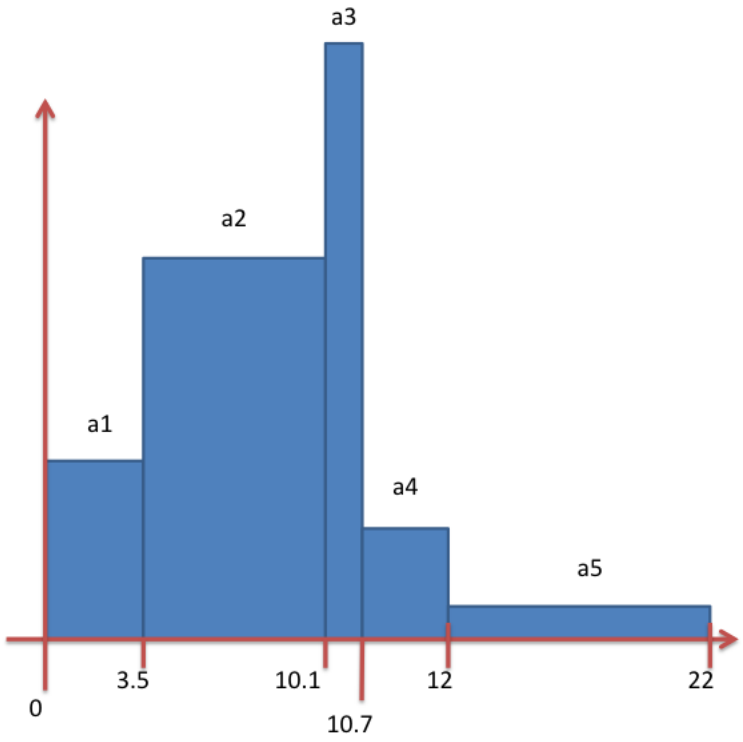


Fig. 4.246: Bar graph created from picking an array variable created with array_compose_with_bins()

Array Decompose Function: `array_decompose()` [`array_decompose(Arr,Idx)`] Creates a new *scalar* expression which is everywhere the scalar member of the *array* input argument at index `Idx` (numbered starting from zero).

Array Decompose 2D Function: `array_decompose2d()` [`array_decompose2d(expr0)`] No description available.

Array Component-wise Division Function: `array_componentwise_division()` [`array_componentwise_division(<Array>,<Divisor>)`] Return a new *array* variable which is the old input `<Array>` variable with each of its components divided by the `<Divisor>`.

Array Component-wise Product Function: `array_componentwise_product()` [`array_componentwise_product(<Array>,<Multiplier>)`] Return a new *array* variable which is the old input `<Array>` variable with each of its components multiplied by the `<Multiplier>`.

Array Sum Function: `array_sum()` [`array_sum(<Array>)`] Return a new *scalar* variable which is the sum of the `<Array>` components.

Material Expressions

Dominant Material Function: `dominant_mat()` [`domimant_mat(<Mesh>)`] Creates a new scalar expression which is for every mesh cell/zone the material having the largest volume fraction.

Material Error Function: `materror()` [`materror(<Mat>,[Const,Const...])`] Creates a new scalar expression which is everywhere the difference in volume fractions as stored in the database and as computed by VisIt's material interface reconstruction (MIR) algorithm. The `<Mat>` argument is a *material variable* from a database and the `Const` argument is one of the material names as an quoted string or a material number as an integer. If multiple materials are to be selected from the *material variable*, enclose them in square brackets as a list.

Examples...

```
materror(materials, 1)
materror(materials, [1,3])
materror(materials, "copper")
materror(materials, ["copper", "steel"])
```

Material Volume Fractions Function: `matvf()` [`matvf(<Mat>,[Const,Const,...])`] Creates a new scalar expression which is everywhere the sum of the volume fraction of the specified materials within the specified material variable. The `<Mat>` argument is a *material variable* from a database and the `Const` argument(s) identify one or more materials within the *material variable*.

Examples...

```
matvf(materials, 1)
matvf(materials, [1,3])
matvf(materials, "copper")
matvf(materials, ["copper", "steel"])
```

NMats Function: `nmats()` [`nmats(<Mat>)`] Creates a new scalar expression which for each mesh cell/zone is the number of materials in the cell/zone. The `<Mat>` argument is a *material variable* from a database.

Specmf Function: `specmf()` [`specmf(<Spec>,<MConst>,[Const,Const,...])`] Performs the analogous operation to `matvf` for species mass fractions. The `<Spec>` argument is a *species variable* from a database. The `<MConst>` argument is a specific material within the *species variable*. The `<Const>` argument(s) identify which species within the *species variable* to select.

Examples:

```
specmf(species, 1, 1)
specmf(species, "copper", 1)
specmf(species, "copper", [1,3])
```

Value For Material Function: `value_for_material()` [`value_for_material(<Var>,<Const>)`] Creates a new scalar expression which is everywhere the material-specific value of the variable specified by `<Var>` for the material specified by `<Const>`. If variable specified by `<Var>` has no material specific values, the values returned from this function will be just the variable's values.

Mesh Expressions

Area Function: `area()` [`area(<Mesh>)`] See the Verdict Manual

cylindrical Function: `cylindrical()` [`cylindrical(<Mesh>)`] Creates a new vector variable on the mesh which is the cylindrical coordinate tuple (R,theta,Z) of each mesh node.

Cylindrical Radius [`cylindrical_radius(<Mesh>)`] Creates a scalar new variable on the mesh which is the radius component of the cylindrical coordinate (from the Z axis) of each mesh node.

cylindrical theta Function: `cylindrical_theta()` [`cylindrical_theta(<Mesh>)`] Creates a new scalar variable on the mesh which is the angle component of the cylindrical coordinate (around the Z axis from the +X axis) of each mesh node.

polar radius Function: `polar_radius()` [`polar_radius(<Mesh>)`] Creates a new scalar variable on the mesh which is the radius component of the polar coordinate of each mesh node.

polar theta Function: `polar_theta()` [`polar_theta(<Mesh>)`] Creates a new scalar variable on the mesh which is the theta component of the polar coordinate of each mesh node.

polar phi Function: `polar_phi()` [`polar_phi(<Mesh>)`] Creates a new scalar variable on the mesh which is the phi component of the polar coordinate of each mesh node.

min coord Function: `min_coord()` [`min_coord(expr0)`] No description available.

max coord Function: `max_coord()` [`max_coord(expr0)`] No description available.

external node Function: `external_node()` [`external_node(expr0)`] No description available.

external cell Function: `external_cell()` [`external_cell(expr0)`] No description available.

Zoneid Function: `zoneid()` [`zoneid(<Mesh>)`] Return a *zone-centered scalar* variable where the value for each zone/cell is local index of a zone, staring from zero, within its domain.

Global Zoneid Function: `global_zoneid()` [`global_zoneid(<Mesh>)`] If global zone ids are specified by the input database, return a *zone-centered scalar* variable where the value for each zone/cell is the *global* index of a zone, as specified by the data producer.

Nodeid Function: `nodeid()` [`nodeid(expr0)`] Return a *node-centered scalar* variable where the value for each node/vertex/point is local index of a node, staring from zero, within its domain.

Global Nodeid Function: `global_nodeid()` [`global_nodeid(expr0)`] If global node ids are specified by the input database, return a *node-centered scalar* variable where the value for each node/vertex/point is the *global* index of a node, as specified by the data producer.

Ghost Zoneid Function: `ghost_zoneid()` [`ghost_zoneid(<Mesh>)`] Returns the ghost zone id of each zone in the mesh. The ghost zone id could be any combination of the following:

```
DUPLICATED_ZONE_INTERNAL_TO_PROBLEM = 0,
ENHANCED_CONNECTIVITY_ZONE = 1,
REDUCED_CONNECTIVITY_ZONE = 2,
REFINED_ZONE_IN_AMR_GRID = 3,
ZONE_EXTERIOR_TO_PROBLEM = 4,
ZONE_NOT_APPLICABLE_TO_PROBLEM = 5
```

where each flag represents a bit shift by the specified number of bits. So if a zone is not a ghost zone, the value returned would be 0, while if it was a `DUPLICATED_ZONE_INTERNAL_TO_PROBLEM` and a `REFINED_ZONE_IN_AMR_GRID`, the value returned would be 1001 in binary, or 9 in decimal.

Volume Function: `volume()` [`volume(<Mesh>)`] No description available.

Volume2 Function: `volume2()` [`volume2(<Mesh>)`] No description available.

Revolved Volume Function: `revolved_volume()` [`revolved_volume(<Mesh>)`] No description available.

Revolved Surface Area Function: `revolved_surface_area()` [`revolved_surface_area(<Mesh>)`] No description available.

Zone Type Function: `zonetype()` [`zonetype(<Mesh>)`] Return a *zone* centered, character valued variable which indicates the *shape type* of each zone suitable for being used within the *label* plot. Upper case characters generally denote 3D shapes (e.g. `T` for `tet`) while lower case characters denote 2D shapes (e.g. `t` for `triangle`).

Zone Type Rank Function: `zonetype_rank()` [`zonetype_rank(<Mesh>)`] Return a *zone* centered, integer valued variable which indicates the *VTK shape type* of each zone. This expression is often useful with the threshold operator to select only certain shapes within the mesh to be displayed.

Mesh Quality Expressions

VisIt employs the *Verdict Mesh Quality Library* to support a number of expressions related to computing cell-by-cell mesh quality metrics. The specific definitions of the various mesh quality metrics defined by the *Verdict Mesh Quality Library* are amply explained in the `Verdict Manual`. Below, we simply list all the mesh quality metrics and describe in detail only those that are not part of the *Verdict Mesh Quality Library*

In all cases in the **Mesh Quality Expressions**, the input argument is a *mesh variable* from a database and the output is a *scalar* expression.

Neighbor Function: `neighbor()` [`neighbor(<Mesh>)`] See the Verdict Manual

Node Degree Function: `node_degree()` [`node_degree(<Mesh>)`] Return a *node* centered, integer valued variable which indicates the *number* of mesh zones/cells that share each node.

Degree Function: `degree()` [`degree(expr0)`] Return a *node* centered, integer valued variable which indicates the *number* of mesh edges incident to each node.

Aspect Function: `aspect()` [`aspect(<Mesh>)`] See the Verdict Manual

Skew Function: `skew()` [`skew(<Mesh>)`] See the Verdict Manual

Taper Function: `taper()` [`taper(<Mesh>)`] See the Verdict Manual

Minimum Corner Angle Function: `min_corner_angle()` [`min_corner_angle(<Mesh>)`] See the Verdict Manual

Maximum Corner Angle Function: `max_corner_angle()` [`max_corner_angle(<Mesh>)`] See the Verdict Manual

Minimum Edge Length Function: `min_edge_length()` [`min_edge_length(<Mesh>)`] See the Verdict Manual

Maximum Edge Length Function: `max_edge_length()` [`max_edge_length(<Mesh>)`] See the Verdict Manual

Minimum Side Volume Function: `min_side_volume()` [`min_side_volume(<Mesh>)`] See the Verdict Manual

Maximum Side Volume Function: `max_side_volume()` [`max_side_volume(<Mesh>)`] See the Verdict Manual

Stretch Function: `stretch()` [`stretch(<Mesh>)`] See the Verdict Manual

Diagonal Ratio Function: `diagonal_ratio()` [`diagonal_ratio(<Mesh>)`] See the Verdict Manual

Maximum Diagonal Function: `max_diagonal()` [`max_diagonal(<Mesh>)`] See the Verdict Manual

Minimum Diagonal Function: `min_diagonal()` [`min_diagonal(<Mesh>)`] See the Verdict Manual

Dimension Function: `dimension()` [`dimension(<Mesh>)`] See the Verdict Manual

Oddy Function: `oddy()` [`oddy(<Mesh>)`] See the Verdict Manual

Condition Function: `condition()` [`condition(<Mesh>)`] See the Verdict Manual

Jacobian Function: `jacobian()` [`jacobian(<Mesh>)`] See the Verdict Manual

Scaled Jacobian Function: `scaled_jacobian()` [`scaled_jacobian(<Mesh>)`] See the Verdict Manual

Shear Function: `shear()` [`shear(<Mesh>)`] See the Verdict Manual

Shape Function: `shape()` [`shape(<Mesh>)`] See the Verdict Manual

Relative Size Function: `relative_size()` [`relative_size(<Mesh>)`] See the Verdict Manual

Shape and Size Function: `shape_and_size()` [`shape_and_size(<Mesh>)`] See the Verdict Manual

Aspect Gamma Function: `aspect_gamma()` [`aspect_gamma(<Mesh>)`] See the Verdict Manual

Warpage Function: `warpage()` [`warpage(<Mesh>)`] See the Verdict Manual

Maximum Angle Function: `maximum_angle()` [`maximum_angle(<Mesh>)`] See the Verdict Manual

Minimum Angle Function: `minimum_angle()` [`minimum_angle(<Mesh>)`] See the Verdict Manual

Minimum Corner Area Function: `min_corner_area()` [`min_corner_area(<Mesh>)`] See the Verdict Manual

Minimum Sin Corner Function: `min_sin_corner()` [`min_sin_corner(<Mesh>)`] See the Verdict Manual

Minimum Sin Corner CW Function: `min_sin_corner_cw()` [`min_sin_corner_cw(<Mesh>)`] See the Verdict Manual

Face Planarity Function: `face_planarity()` [`face_planarity(<Mesh>)`] Creates a new expression which is everywhere a measure of how close to *planar* all the points comprising a face are. This is computed for each face of a cell and the maximum over all faces is selected for each cell. Planarity is measured as the maximum distance from an arbitrary plane defined by the first 3 points of a face of the remaining points of the face. Values closer to zero are *better*. A triangle face will always have a planarity measure of zero. This mesh quality expression is not part of the Verdict library.

Relative Face Planarity Function: `relative_face_planarity()` [`relative_face_planarity(<Mesh>)`] Performs the same computation as the *face_planarity()*, except where each face's value is normalized by the average edge length of the face.

Comparison Expressions

Comparing variables defined on the *same* mesh is often as simple as taking their difference. What about comparing variables when they are defined on different meshes? A common example is taking the difference between results from two runs of the same simulation application. Even if the two runs operate on computationally *identical* meshes, the fact that each run involves its own *instance* of that mesh means that as far as VisIt is concerned, they are different meshes.

In order to compose an expression involving variables on different meshes, the *first* step is to *map* the variables onto a *common* mesh. The position-based CMFE function and its friend, the connectivity-based CMFE function, *conn_cmfe()* are the work-horse methods needed when working with variables from *different* meshes in the *same* expression. *CMFE* is an abbreviation for *cross-mesh field evaluation*.

The syntax for specifying CMFE expressions can be complicated. Therefore, the GUI supports a *wizard* to help create them. See the *Data-Level Comparisons Wizard* for more information. It sometimes makes sense to use the wizard to create an *initial* CMFE expression and then modify it manually, often to adjust the state indexing. Here, we describe the details of creating CMFE expressions manually.

All of the comparison expressions involve the concepts of a *donor variable* and a *target mesh*. The donor variable (e.g. *pressure*) is the variable to be mapped. The target mesh is the mesh onto which the donor variable is to be mapped. In addition, the term *donor mesh* refers to the mesh upon which the donor variable is defined.

Position-Based CMFE Function: `pos_cmfe()` [`pos_cmfe(<Donor Variable>,<Target Mesh>,<Fill>)`] The `pos_cmfe()` function performs the mapping assuming the two meshes, that is the `<Target Mesh>` and the mesh upon which the `<Donor Variable>` (e.g. the *donor mesh*) is defined, share *only* a common spatial (positional) extent. Its friend, the *conn_cmfe()* function is *optimized* to perform the mapping when the two meshes are also *topologically identical*. In other words, their *coordinate* **and** *connectivity* arrays are 1:1. In this case, the mapping can be performed with more efficiency and numerical accuracy. Therefore, when it is possible and makes sense to do so, it is always best to use `conn_cmfe()`.

We'll describe the arguments to `pos_cmfe()` working backwards from the last.

The last, `<Fill>` argument is a numerical constant that VisIt will use to determine the value of the result in places on the target mesh that do not spatially overlap with the mesh of the donor variable. Note that if a value is chosen within the range of the donor variable, it may by difficult to distinguish regions VisIt deemed were non-overlapping. On the other hand, if a value outside the range is chosen, it will effect the range of the mapped variable. A common practice is to choose a value that is an extremum of the donor variable's range. Another practice is to choose a value that is easily distinguishable and then apply a threshold operator to remove those portions of the result. If the `Fill` argument is not specified, zero is assumed.

Working backwards, the next argument, is the `<Target Mesh>`. The `<Target Mesh>` argument in `pos_cmfe()` is always interpreted as a mesh *within* the currently *active* database. The CMFE expressions are always mapping data from *other* meshes, possibly in *other* databases onto the `<Target Mesh>` which is understood to be in the currently *active* database. When mapping data between meshes *in different databases*, the additional information necessary to specify the other database is encoded with a special syntax prepending the `Donor Variable` argument.

The `Donor Variable` argument is a string argument of the form:

```
<PATH-TO-DATABASE-FROM-CWD[SSS]MM:VARNAME>
```

consisting of the donor variable's name and up to three pre-pending sub-strings which may be optionally needed to specify...

1. ...the *Database* (`PATH-TO-DATABASE-FROM-CWD`) in which the donor variable resides,
2. ...the *State Id* (`[SSS]`) from which to take the donor variable,
3. ...the *Modality* (`MM`) by which states are identified in the *State Id* sub-string.

Depending on circumstances, specifying the `Donor-Variable` argument to the CMFE functions can get cumbersome. For this reason, CMFE expressions are typically created using the *Data-Level Comparisons Wizard* under the *Controls* menu. Nonetheless, here we describe the syntax and provide examples for a number of cases of increasing complexity in specifying where the `Donor Variable` resides.

When the donor variable is in the same database and state as the target mesh, then only the variable's name is needed. The optional substrings are not. See case A in the examples below.

When the donor variable is in a different database **and** the databases do not have multiple time states, then only sub-string 1, above, is needed to specify the path to the database in the file system. The path to the database can be specified using either *absolute* or *relative* paths. *Relative* paths are interpreted relative to the current working directory in which the VisIt session was started. See cases B and C in the examples below.

When the donor variable is in a different database **and** the databases have multiple states, then all 3 sub-strings, above, are required. The `State Id` substring is a square-bracket enclosed number used to identify *which state* from which to take the donor variable. The `Modality` substring is a one- or two-character moniker. The first character indicates whether the number in the the `State Id` substring is a cycle (`c`), a time (`t`), or an index (`i`). The second character, if present, is a `d` character to indicate the cycle, time or index is *relative* (e.g. a *delta*) to the current state. For example, the substring `[200]c` means to treat the `200` as a *cycle* number in the donor database whereas the the substring `[-10]id` means to treat the `-10` as an (`i`) index (`d`) delta. So, `[200]c` would map the *donor* at cycle 200 to the *current* cycle of the *target* and `[-10]id` would map the *donor* at the current *index minus 10* to the *current* index of the *target*. In particular, the string `[0]id` is needed to create a CMFE that keeps *donor* and *target* in lock step. Note that in cases where the donor database does not have an exact match for the specified cycle or time, VisIt will chose the state with the cycle or time which is closest in absolute distance. For the *index* modality, if there is no exact match for the specified index, an error results. See cases D-I in the examples below.

Note that the *relative* form of specifying the *State Id* is needed even when working with different states *within the same database*. In particular, to create an expression representing a *time derivative* of a variable

in a database, the key insight is to realize it involves mapping a donor variable from one state in the database onto a mesh at another state. In addition, the value in using the *relative* form of specifying the `State Id` of the donor variable is that as the current time is changed, the expression properly identifies the different states of the donor variable instead of always mapping a fixed state.

Examples...

```
# Case A: Donor variable, "pressure" in same database as mesh, "ucdmesh"
# Note that due to a limitation in Expression parsing, the '[0]id:' is
# currently required in the donor variable name as a substitute for
# specifying a file system path to a database file. The syntax '[0]id:'
# means a state index delta of zero within the active database.
pos_cmfe(<[0]id:pressure>,<ucdmesh>,1e+15)

# Case B: Donor variable in a different database using absolute path
pos_cmfe(</var/tmp/foo.silo:pressure>,<ucdmesh>,1e+15)

# Case C: Donor variable in a different database using relative path
pos_cmfe(<foo/bar.silo:pressure>,<ucdmesh>,1e+15)

# Case D: Map "p" from wave.visit at state index=7 onto "mesh"
pos_cmfe(<./wave.visit[7]i:p>, mesh, 1e+15)

# Case E: Map "p" from wave.visit at state index current-1 onto "mesh"
pos_cmfe(<./wave.visit[-1]id:p>, mesh, 1e+15)

# Case F: Map "p" from wave.visit at state with cycle~200 onto "mesh"
pos_cmfe(<./wave.visit[200]c:p>, mesh, 1e+15)

# Case G: Map "p" from wave.visit at state with cycle~cycle(current)-200␣
↪onto "mesh"
pos_cmfe(<./wave.visit[-200]id:p>, mesh, 1e+15)

# Case H: Map "p" from wave.visit at state with time~1.4 onto "mesh"
pos_cmfe(<./wave.visit[1.4]t:p>, mesh, 1e+15)

# Case I: Map "p" from wave.visit at state with time~time(current)-0.8 onto
↪"mesh"
pos_cmfe(<./wave.visit[-0.8]td:p>, mesh, 1e+15)
```

Connectivity-Based CMFE Function: `conn_cmfe()` [`conn_cmfe(<Donor Variable>,<Target Mesh>)`] The connectivity-based CMFE is an *optimized* version of *pos_cmfe()* for cases where the `Target Mesh` and the mesh of the `Donor Variable` are *topologically and geometrically identical*. In such cases, there is no opportunity for the two meshes to fail to overlap perfectly. Thus, there is no need for the third, `<Fill>` argument. In all other respects, `conn_cmfe()` performs the same function as *pos_cmfe()* except that `conn_cmfe()` *assumes* that any differences in the coordinates of the two meshes are numerically insignificant to the resulting mapped variable. In other words, differences in the coordinate fields, if they exist, are not incorporated into the resulting mapping.

Curve CMFE Function: `curve_cmfe()` [`curve_cmfe(<Donor Curve>,<Target Curve>)`] The curve-based CMFE performs the same function as *pos_cmfe()* except for curves. The arguments specify the `Target Curve` and `Donor Curve` and the same syntax rules apply for specifying the `Donor Curve` as for the `Donor Variable` in *pos_cmfe()*. However, if curves represent different spatial extents or different numbers of samples or sample spacing, no attempt is made to unify them.

Symmetric Difference By Point Function: `symm_point()` [`symm_point(<Scalar>,<Fill>,[Px,Py,Pz])`] Return a new *scalar* variable which is the symmetric difference of `<Scalar>` reflected about the point `[Px, Py, Pz]`. In 2D, `Pz` is still required but ignored. The `<Fill>` argument is a numerical constant that

VisIt will use to determine the value of the result in places symmetry about the point doesn't overlap with the donor mesh. This operation involves **both** the reflection about the point **and** taking the difference. If the input `<Scalar>` is indeed symmetric about the point, the result will be a constant valued variable of zero.

Symmetric Difference By Plane Function: `symm_plane()` [`symm_plane(<Scalar>,<Fill>,[Nx,Ny,Nz,Px,Py,Pz])`] Return a new *scalar* variable which is the symmetric difference of `<Scalar>` reflected about the plane defined by the point `[Px, Py, Pz]` and normal `[Nx, Ny, Nz]`. In 2D, the `Nz` and `Pz` arguments are still required but ignored. The `<Fill>` argument is a numerical constant that VisIt will use to determine the value of the result in places symmetry about the plane doesn't overlap with the donor mesh. This operation involves **both** the reflection about the plane **and** taking the difference. If the input `<Scalar>` is indeed symmetric about the plane, the result will be a constant valued variable of zero.

Symmetric Difference By Transform Function: `symm_transform()` [`symm_transform(<Scalar>,<Fill>,[T00,T01,T02,...,T22])`] Return a new *scalar* variable which is the symmetric difference of `<Scalar>` reflected through the 3x3 transformation where each point, `[Px,Py,Pz]`, in the mesh supporting `<Scalar>` is transformed by the transform coefficients, `[T00, T01,...,T22]` as shown below. In 2D, all 9 transform coefficients are still required but the last row and column are ignored. The `<Fill>` argument is a numerical constant that VisIt will use to determine the value of the result in places symmetry through the transform doesn't overlap with the donor mesh. This operation involves **both** the transform **and** taking the difference. If the input `<Scalar>` is indeed symmetric through the transform, the result will be a constant valued variable of zero.

$$\begin{bmatrix} T_{00} & T_{01} & T_{02} \\ T_{10} & T_{11} & T_{12} \\ T_{20} & T_{21} & T_{22} \end{bmatrix} * \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} T_{00} * P_x + T_{01} * P_y + T_{02} * P_z \\ T_{10} * P_x + T_{11} * P_y + T_{12} * P_z \\ T_{20} * P_x + T_{21} * P_y + T_{22} * P_z \end{bmatrix}$$

Evaluate Point Function: `eval_point()` [`eval_point(<Scalar>,<Fill>,[Px,Py,Pz])`] Performs only the reflection half of the *symm_point()* operation. That is, it computes a new *scalar* variable which is the input `<Scalar>` reflected through the symmetric point. It does not then take the *difference* between with the input `<Scalar>` as *symm_point()* does.

Evaluate Plane Function: `eval_plane()` [`eval_plane(<Scalar>,<Fill>,[Nx,Ny,Nz,Px,Py,Pz])`] Performs only the reflection half of the *symm_plane()* operation. That is, it computes a new *scalar* variable which is the input `<Scalar>` reflected through the symmetric plane. It does not then take the *difference* between with the input `<Scalar>` as *symm_plane()* does.

Evaluate Transform Function: `eval_transform()` [`eval_transform(expr0,<Fill>,[T00,T01,T02...T22])`] Performs only the transform half of the *symm_transform()* operation. That is, it computes a new *scalar* variable which is the input `<Scalar>` mapped through the transform. It does not then take the *difference* between with the input `<Scalar>` as *symm_transform()* does.

Image Processing Expressions

The image processing expressions defined here are not suitable for multi-block data. They do not handle domain boundaries properly even if the input database properly defines suitable layers of *ghost* zones. They do, however, operate on 2 and 3D data.

conservative smoothing Function: `conservative_smoothing()` [`conservative_smoothing(expr0)`] No description available.

Mean Filter Function: `mean_filter()` [`mean_filter(<Scalar>,<Int>)`] Return a filtered version of the input *scalar* variable using the mean filter of width specified by `<Int>` argument. By default, the filter width is 3 (3x3). The input scalar must be defined on a structured mesh.

Median Filter Function: `median_filter()` [`median_filter(expr0)`] Return a filtered version of the input *scalar* variable using a 3x3 median filter. The input scalar must be defined on a structured mesh.

Abel Inversion Function: `abel_inversion()` [`abel_inversion(expr0)`] No description available.

Miscellaneous Expressions

Zonal Constant Function: `zonal_constant()` [`zonal_constant(expr0)`] Return a *scalar*, *zone-centered* field that is everywhere on `<Mesh>` the constant value `<Const>`.

Zone Constant Function: `zone_constant()` [`zone_constant(<Mesh>,<Const>)`] An alias for *zonal_constant()*

Cell Constant Function: `cell_constant()` [`cell_constant(expr0)`] An alias for *zonal_constant()*

Nodal Constant Function: `nodal_constant()` [`nodal_constant(<Mesh>,<Const>)`] Return a *scalar*, *node-centered* field that is everywhere on `<Mesh>` the constant value `<Const>`.

Node Constant Function: `node_constant()` [`node_constant(expr0)`] An alias for *nodal_constant()*

Point Constant Function: `point_constant()` [`point_constant(expr0)`] An alias for *nodal_constant()*

Time Function: `time()` [`time(expr0)`] Return a *constant scalar* variable which is everywhere the time of the associated input argument within its time-series.

Cycle Function: `cycle()` [`cycle(expr0)`] Return an integer *constant scalar* variable which is everywhere the cycle of the associated input argument within its time-series.

Timestep Function: `timestep()` [`timestep(expr0)`] Return an integer *constant scalar* variable which is everywhere the index of the associated input argument within its time-series.

curve domain Function: `curve_domain()` [`curve_domain(expr0)`] No description available.

curve integrate Function: `curve_integrate()` [`curve_integrate(expr0)`] No description available.

curve swapxy Function: `curve_swapxy()` [`curve_swapxy(expr0)`] No description available.

curve Function: `curve()` [`curve(expr0)`] No description available.

Enumerate Function: `enumerate()` [`enumerate(<Int-Scalar>,<[Int-List]>)`] Map an integer valued *scalar* variable to a new set of integer values. If *K* is the maximum value in the `Int-Scalar` input argument, the `[Int-List]` argument must be a square bracketed list of *K+1* integer values. Value *i* in the `Int-Scalar` input argument is used to index the *ith* entry in the `[Int-List]` to produce mapped value.

Map Function: `map()` [`map(<Scalar>,<[Input-Value-List]>,<[Output-Value-List]>, fill_value)`] A more general form of *enumerate()* which supports non-integer input *scalar* variables and input and output maps which are not required to include all values in the input *scalar* variable. The `[Input-Value-List]` and `[Output-Value-List]` must have the same number of entries. A value in the input *scalar* variable that matches the *ith* entry in the `[Input-Value-List]` is mapped to the new value at the *ith* entry in the `[Output-Value-List]`. Values that do not match any entry in the `[Input-Value-List]` are mapped to `fill_value`, which is `-1` by default.

Resample Function: `resample()` [`resample(<Var>,Nx,Ny,Nz)`] Resample `<Var>` onto a regular grid defined by taking the X, Y and for 3D, Z spatial extents of the mesh `<Var>` is defined on and taking `Nx` samples over the spatial extents in X, `Ny` samples over the spatial extents in Y, and, for 3D, `Nz` samples over the spatial extents in Z. Any samples that *miss* the mesh `<Var>` is defined on are assigned the value `-FLT_MAX`. For 2D, the `Nz` argument is still required but ignored.

Recenter Expression Function [`recenter(expr, ["nodal", "zonal", "toggle"])`] This function can be used to recenter `expr`. The second argument is optional and defaults to *"toggle"* if it is not specified. A value of *"toggle"* for the second argument means that if `expr` is *node* centered, it is recentered to *zone* centering and if `expr` is *zone* centered, it is recentered to *node* centering. Note that the quotes are required for the second argument. This function is typically used to force a specific centering among the operands of some other expression.

Process Id Function: `procid()` [`procid(<Var>)`] Return an integer *scalar* variable which is everywhere the MPI rank associated with each of the blocks of the possibly parallel decomposed mesh upon which `<Var>` is defined. For serial execution or for parallel execution of a single-block mesh, this will produce a constant zero variable. Otherwise, the values will vary block by block.

Thread Id Function: `threadid()` [`threadid(expr0)`] Return an integer *scalar* variable which is everywhere the local thread id associated with each of the blocks of the possibly parallel decomposed mesh upon which `<Var>` is defined. For non-threaded execution, this will produce a constant zero variable. Otherwise, the values will vary block by block.

isnan Function: `isnan()` [`isnan(expr0)`] No description available.

q criterion Function: `q_criterion()` [`q_criterion(<gradient(velocity[0])>, <gradient(velocity[1])>, <gradient(velocity[2])>)`] Generates the Q-criterion value developed by Hunt et. al.. It is based on the observation that, in regions where the Q-criterion is greater than zero, rotation exceeds strain and, in conjunction with a pressure min, indicates the presence of a vortex. The three arguments to the function are gradient vectors of the x-, y-, and z-velocity. The gradient function (see *gradient()*) can be used to create the gradient vectors.

lambda2 Function: `lambda2()` [`lambda2(<gradient(velocity[0])>, <gradient(velocity[1])>, <gradient(velocity[2])>)`] Generates the Lambda-2 criterion. It is based on the observation that, in regions where Lambda-2 is less than zero, rotation exceeds strain and, in conjunction with a pressure min, indicates the presence of a vortex. The three arguments to the function are gradient vectors of the x-, y-, and z-velocity. The gradient function (see *gradient()*) can be used to create the gradient vectors.

mean curvature Function: `mean_curvature()` [`mean_curvature(expr0)`] No description available.

Gauss Curvature Function: `gauss_curvature()` [`gauss_curvature(expr0)`] No description available.

agrad Function: `agrad()` [`agrad(expr0)`] No description available.

key aggregate Function: `key_aggregate()` [`key_aggregate(expr0)`] No description available.

rectilinear Laplacian Function: `rectilinear_laplacian()` [`rectilinear_laplacian(expr0)`] No description available.

conn components Function: `conn_components()` [`conn_components(expr0)`] No description available.

resrad Function: `resrad()` [`resrad(expr0)`] No description available.

crack width Function: `crack_width()` : `crack_width(crack_num, <crack1_dir>, <crack2_dir>, <crack3_dir>, <strain_tensor>, volume2(<mesh_name>))`

Calculates crack width using the following formula:

```
crackwidth = L * (1 - (exp(-delta))
where:
  L = ZoneVol / (Area perpendicular to crack_dir)
    find Area by slicing the cell by plane with origin == cell center
    and Normal == crack_dir.  Take area of that slice.

delta =
    T11 for crack dir1 = component 0 of strain_tensor
    T22 for crack dir2 = component 4 of strain_tensor
    T33 for crack dir3 = component 8 of strain_tensor
```

Time Iteration Expressions

Average Over Time Function: `average_over_time()` [`average_over_time(<Scalar>,<Start>, <Stop>,<Stride>)`] Return a new *scalar* variable in which each zonal or nodal value is the average over

the times indicated by `Start`, `Stop` and `Stride`.

Min Over Time Function: `min_over_time()` [`min_over_time(<Scalar>,<Start>,<Stop>,<Stride>)`] Return a new *scalar* variable in which each zonal or nodal value is the minimum value the variable, `<Scalar>`, attained over the times indicated by `Start`, `Stop` and `Stride`.

Max Over Time Function: `max_over_time()` [`max_over_time(<Scalar>,<Start>,<Stop>,<Stride>)`] Return a new *scalar* variable in which each zonal or nodal value is the maximum value the variable, `<Scalar>`, attains over the times indicated by `Start`, `Stop` and `Stride`.

Sum Over Time Function: `sum_over_time()` [`sum_over_time(<Scalar>,<Start>,<Stop>,<Stride>)`] Return a new *scalar* variable in which each zonal or nodal value is the sum of the values the variable, `<Scalar>` attains over the times indicated by `Start`, `Stop` and `Stride`.

First Time When Condition Is True Function: `first_time_when_condition_is_true()` [`first_time_when_condition_is_true(<Cond>,<Fill>,<Start>,<Stop>,<Stride>)`] Return a new *scalar* variable in which each zonal or nodal value is the *first* time (not cycle and not time-index, but floating point time) at which the true/false condition, `<Cond>` is true. The `<Fill>` value is used if there is no *first* time the condition is true.

Last Time When Condition Is True Function: `last_time_when_condition_is_true()` [`last_time_when_condition_is_true(<Cond>,<Fill>,<Start>,<Stop>,<Stride>)`] Return a new *scalar* variable in which each zonal or nodal value is the *last* time (not cycle and not time-index, but floating point time) at which the true/false condition, `<Cond>` is true. The `<Fill>` value is used if there is no *last* time the condition is true.

First Cycle When Condition Is True Function: `first_cycle_when_condition_is_true()` [`first_cycle_when_condition_is_true(<Cond>,<Fill>,<Start>,<Stop>,<Stride>)`] Return a new integer valued *scalar* variable in which each zonal or nodal value is the *first* cycle (not time and not time-index, but integer cycle) at which the true/false condition, `<Cond>` is true. The `<Fill>` value is used if there is no *first* cycle the condition is true.

Last Cycle When Condition Is True Function: `last_cycle_when_condition_is_true()` [`last_cycle_when_condition_is_true(<Cond>,<Fill>,<Start>,<Stop>,<Stride>)`] Return a new integer valued *scalar* variable in which each zonal or nodal value is the *last* cycle (not time and not time-index, but integer cycle) at which the true/false condition, `<Cond>` is true. The `<Fill>` value is used if there is no *last* cycle the condition is true.

First Time Index When Condition Is True Function: `first_time_index_when_condition_is_true()` [`first_time_index_when_condition_is_true(<Cond>,<Fill>,<Start>,<Stop>,<Stride>)`] Return a new integer valued *scalar* variable in which each zonal or nodal value is the *first* time index (not cycle and not time, but integer time-index) at which the true/false condition, `<Cond>` is true. The `<Fill>` value is used if there is no *first* time-index the condition is true.

Last Time Index When Condition Is True Function: `last_time_index_when_condition_is_true()` [`last_time_index_when_condition_is_true(<Cond>,<Fill>,<Start>,<Stop>,<Stride>)`] Return a new integer valued *scalar* variable in which each zonal or nodal value is the *last* time index (not cycle and not time, but integer time-index) at which the true/false condition, `<Cond>` is true. The `<Fill>` value is used if there is no *last* time-index the condition is true.

var when condition is first true Function: `var_when_condition_is_first_true()` [`var_when_condition_is_first_true(expr0)`] No description available.

var when condition is last true Function: `var_when_condition_is_last_true()` [`var_when_condition_is_last_true(expr0)`] No description available.

time at minimum Function: `time_at_minimum()` [`time_at_minimum(expr0)`] No description available.

cycle at minimum Function: `cycle_at_minimum()` [`cycle_at_minimum(expr0)`] No description available.

time index at minimum Function: **time_index_at_minimum()** [time_index_at_minimum(expr0)] No description available.

value at minimum Function: **value_at_minimum()** [value_at_minimum(expr0)] No description available.

time at maximum Function: **time_at_maximum()** [time_at_maximum(expr0)] No description available.

cycle at maximum Function: **cycle_at_maximum()** [cycle_at_maximum(expr0)] No description available.

time index at maximum Function: **time_index_at_maximum()** [time_index_at_maximum(expr0)] No description available.

value at maximum Function: **value_at_maximum()** [value_at_maximum(expr0)] No description available.

localized compactness Function: **localized_compactness()** [localized_compactness(expr0)] No description available.

merge tree Function: **merge_tree()** [merge_tree(expr0)] No description available.

split tree Function: **split_tree()** [split_tree(expr0)] No description available.

local threshold Function: **local_threshold()** [local_threshold(expr0)] No description available.

python Function: **python()** [python(expr0)] No description available.

relative difference Function: **relative_difference()** [relative_difference(expr0)] No description available.

var skew Function: **var_skew()** [var_skew(expr0)] No description available.

apply data binning Function: **apply_data_binning()** [apply_data_binning(expr0)] No description available.

distance to best fit line Function: **distance_to_best_fit_line()** [distance_to_best_fit_line(expr0)] No description available.

distance to best fit line2 Function: **distance_to_best_fit_line2()** [distance_to_best_fit_line2(expr0)] No description available.

geodesic vector quantize Function: **geodesic_vector_quantize()** [geodesic_vector_quantize(expr0)] No description available.

bin Function: **bin()** [bin(expr0)] No description available.

biggest neighbor Function: **biggest_neighbor()** [biggest_neighbor(expr0)] No description available.

smallest neighbor Function: **smallest_neighbor()** [smallest_neighbor(expr0)] No description available.

neighbor average Function: **neighbor_average()** [neighbor_average(expr0)] No description available.

Expression Compatibility Gotchas

VisIt will allow you to define expressions that it winds up determining to be invalid later when it attempts to execute those expressions. Some common issues are the mixing of incompatible mesh variables in the same expression *without* the necessary additional functions to make them compatible.

Tensor Rank Compatibility

For example, what happens if you mix scalar and vector mesh variables in the same expression? VisIt will allow users to define such an expression. But, when it is plotted, the plot will fail.

As an aside, as the user goes back and forth between the Expressions window creating and/or adjusting expression definitions, VisIt makes no attempt to keep track of all the changes made in expressions and automatically update plots as expressions change. Users have to manually clear or delete plots to force VisIt to re-draw plots in which the expressions changed.

If what is really intended was a scalar mesh variable, then users must use one of the expression functions that converts a vector to a scalar such as the magnitude() built-in expression or the array de-reference operator.

Centering Compatibility

Some variables are zone centered and some are node centered. What happens if a user combines these in an expression? VisIt will default to zone centering for the result. If this is not the desired result, the *recenter()* expression function should be used, where appropriate, to adjust centering of some of the terms in the expression. Note that ordering of operations will probably be important. For example

```
node_var + recenter(zone_var)
recenter(zone_var + node_var)
```

both achieve a *node-centered* result. But, each expression is subtly (and numerically) different. The first recenter's *zone_var* to the nodes and then performs the summation operator at each node. In the second, there is an implied recentering of *node_var* to the zones first. Then, the summation operator is applied at each zone center and finally the results are recentered back to the nodes. In all likelihood this creates in a numerically lower quality result. The moral is that in a complex series of expressions be sure to take care where you want recentering to occur.

Mesh Compatibility

In many cases, especially in Silo databases, all the available variables in a database are not always defined on the same mesh. This can complicate matters involving expressions in variables from different meshes.

Just as in the previous two examples of incompatible variables where the solution was to apply some function to make the variables compatible, we have to do the same thing when variables from different meshes are combined in an expression. The key expression functions which enable this are called **Cross Mesh Field Evaluation** or **CMFE** expression functions. We will only briefly touch on these here. **CMFEs** will be discussed in much greater detail elsewhere.

Just as for centering, we have two options when dealing with variables from two different meshes. Each of which involves *mapping* one of the variables onto the other variable's mesh using one of the CMFE expression functions.

Deferring expression evaluation

Expressions are generally evaluated before any operators are applied to the data. In cases where an expression involves the *output* of an operator, or the operator behaves in such a way as to change the outcome of an expression, then it is necessary to *defer* evaluation of such an expression until *after* operators have been applied. The *DeferExpression operator* is designed for this purpose. It will cause expressions in its list to be evaluated at the time of it's own execution in the pipeline.

Automatic expressions

4.8.2 Query

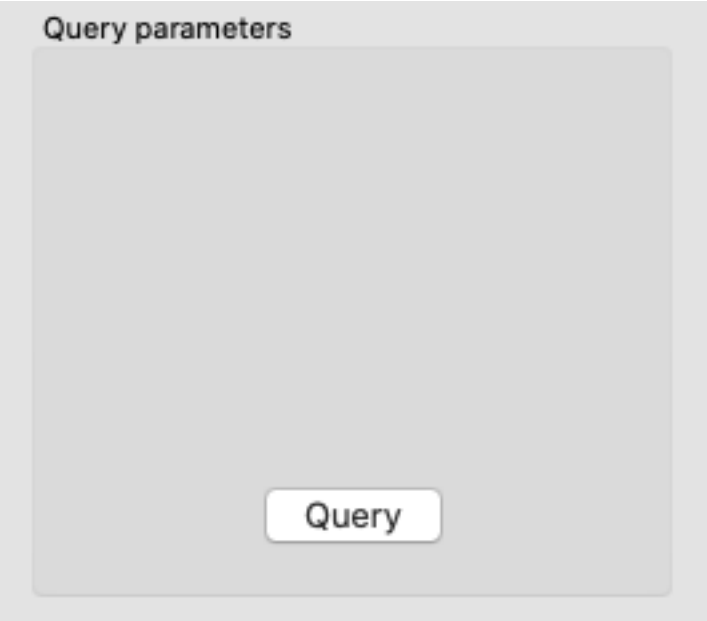
VisIt allows you to gather quantitative information from the database being visualized through the use of queries. A query is a type of calculation that can either return values from the database or values that are calculated from data in the database. For example, VisIt's Pick and Lineout capabilities (described later in this chapter) are specialized point and line queries that print out the values of variables in the database at points or along lines. In addition to point and line queries, VisIt provides database queries that return values that are based on all of the data in a database.

Some queries can even be executed for all of the time states in a database to yield a Curve plot of the query's behavior over time. This feature will be covered in more detail a little later.

VisIt's queries are available in the **Query Window** (shown in Figure 4.247), which you can open by clicking the **Query** option in the **Main Window's Control** menu. The **Query Window** consists of upper and lower areas where the upper area allows you to select a query and set its query parameters. The controls for setting a query's parameters change as required and some queries have no parameters and thus have no controls for setting parameters. The bottom area of the window displays the results of the query once VisIt has finished processing it. The results for new queries are appended to the output from previous queries until you clear the **Query results** by clicking the **Clear results** button.

Query types

VisIt's queries can be divided into three types: database queries, point queries, and line queries. Database queries usually calculate information for the database as a whole instead of concentrating on a single zone or node but some Pick-related database queries do concentrate on cells and nodes. Point queries calculate information for a point in the database and several types of variable picking queries fall into this category. Line queries calculate information along a line. Each type of query has different controls in the **Query parameters** area (see Figure 4.248) and as you highlight different queries, the controls in the **Query parameters** area may change.



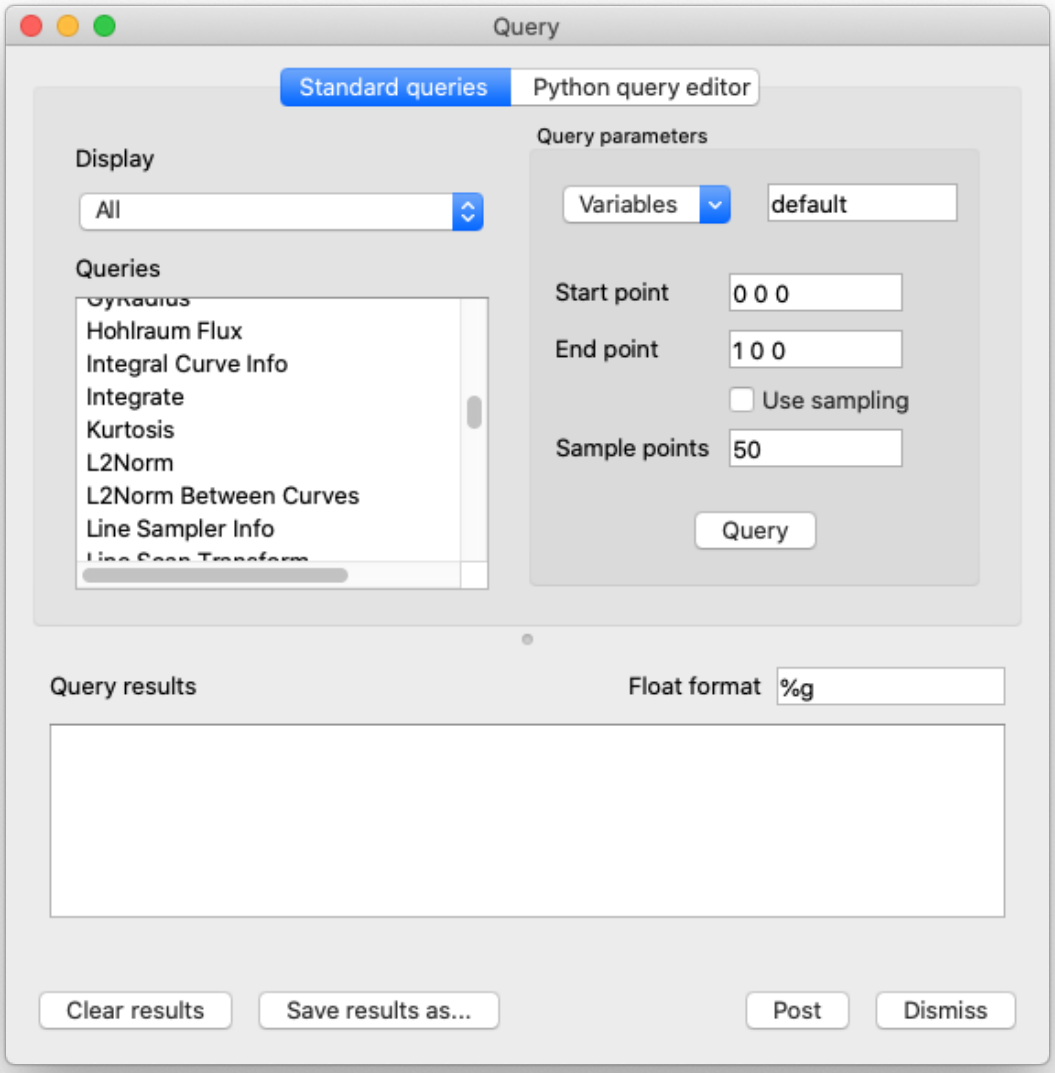
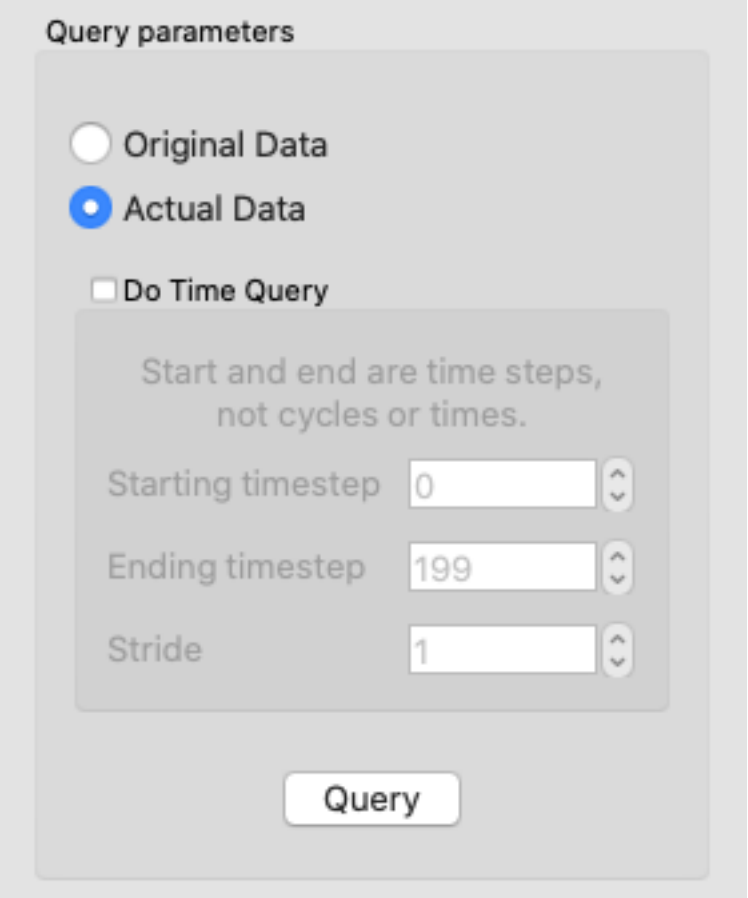
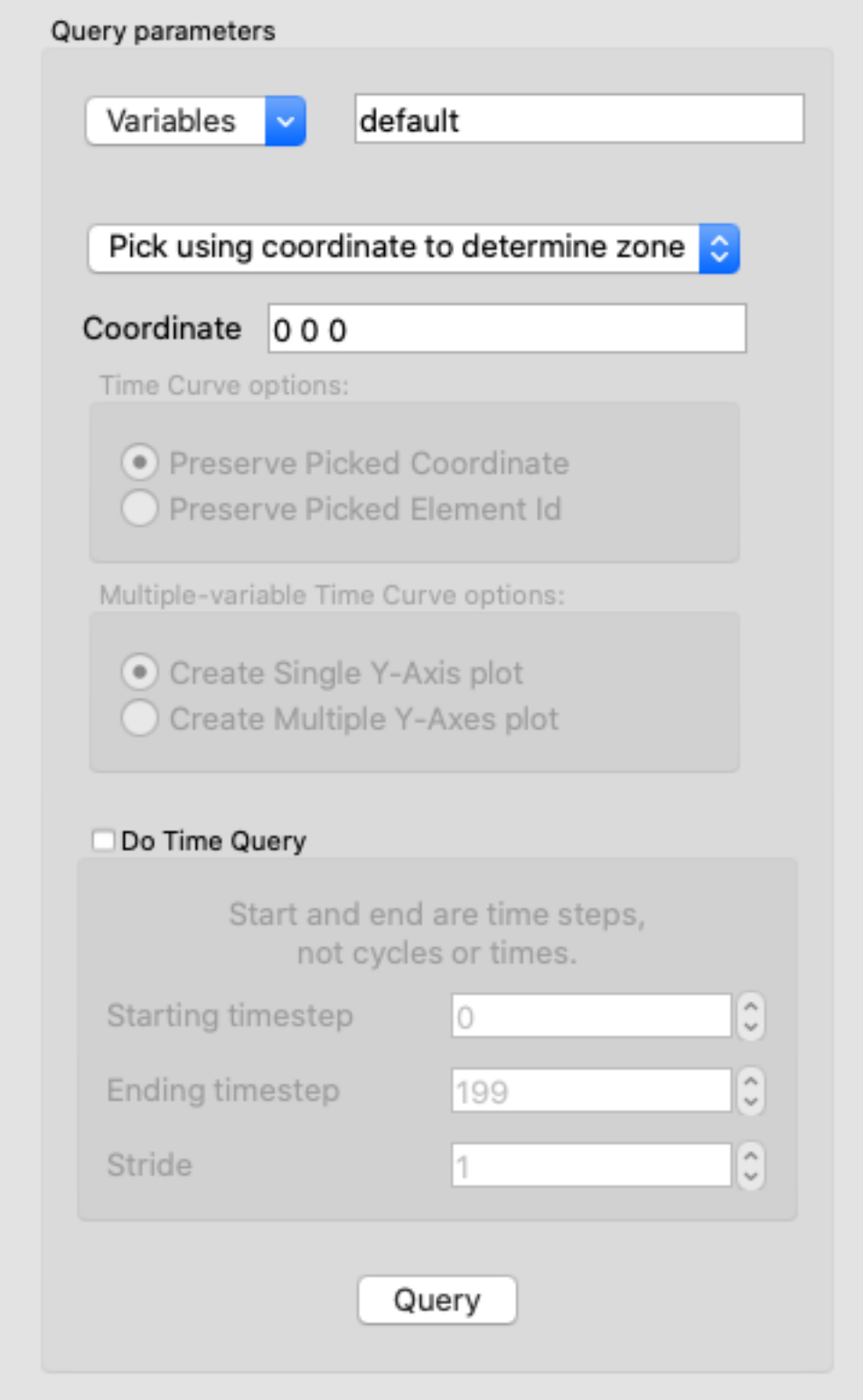


Fig. 4.247: Query window


Query parameters
Original Data
Actual Data
Do Time Query
Start and end are time steps,
not cycles or times.
Starting timestep
0
Ending timestep
199
Stride
1
Query



Database queries provide a few different interfaces depending on the query. Many database queries require no additional input so they have no controls except for the **Query** button. Other database queries ask whether the query is to be performed with respect to the original data or the actual data, which is that data that is left in the plot after subsets have been removed and operators have transformed the data. Finally, some database queries ask for a specific domain number and zone or node number.

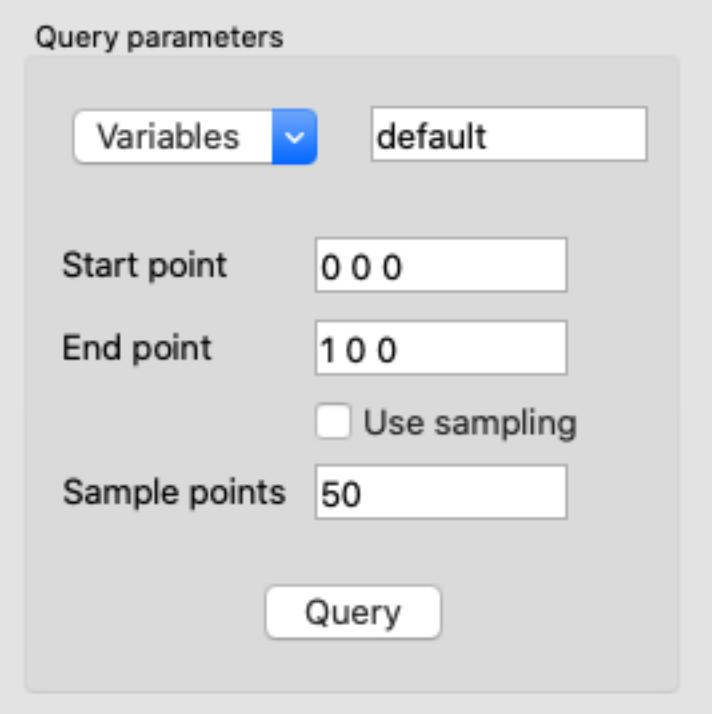


Fig. 4.248: Query parameters area

Point queries provide interfaces in the **Query parameters** area that allow you to enter a 3D point or a screen space point to use as the point for the query. Line queries provide an interface that lets you specify the start and end positions of the line as well as the number of sample points to consider along the length of the line. Nearly all query types allow you to provide additional variables to query in a **Variables** text field.

Built-in queries

Database Queries

2D Area The 2D area query calculates the area of the 2D plot highlighted in the **Plot list** and prints the result to the **Query results**. VisIt can produce a Curve plot of this query with respect to time.

3D Surface Area The 3D surface area calculates the area of the plot highlighted in the **Plot list** and prints the result to the **Query results**. VisIt can produce a Curve plot of this query with respect to time.

Connected Component Area Performs the same operation as either 2/3D area query except individually for each *component* of a disconnected mesh. The query result is a list of values, one for each component.

Connected Component Length Performs an operation similar to *Connected Component Area* except that it works only for 1D components and returns their length. The query result is a list of values, one for each component.

Area Between Curves The Area Between Curves query calculates the area between 2 curve plots. The plots that will serve as input to this query must both be highlighted in the **Plot list** or VisIt will issue an error message. Once the area has been calculated, the result is printed to the **Query results**.

Centroid This query can calculate a centroid (geometric center) or center-of-mass of a dataset depending on the plot (and variable) upon which the query is performed. On a Pseudocolor plot, the plot's variable will be treated as a *density* field. The value of this field at the *center* of each cell will be multiplied by the cell's volume to compute a cell-centered mass contribution for each cell. If the plot's variable is indeed a true density variable,

then the result will be the center-of-mass. If the plot's variable is not a true density variable (e.g. temperature), the result may be nonsensical. If the plot's variable is constant over the whole object, the result will be a centroid (geometric center). If the query is performed on a Mesh or FilledBoundary plot, constant density will be assumed and the result will be a centroid. The results are printed to the **Query results**.

Connected Component Centroid Performs the same operation as either *Centroid* query except individually for each *component* of a disconnected mesh. The query result is a list of values, one for each component.

Chord Length Distribution The Chord Length Distribution query calculates a probability density function of chord length over a two or three dimensional object. Axially symmetric objects (RZ-meshes) are treated as 3D meshes and chords are calculated over the revolved, 3D object. A statistical approach, casting uniform density, random lines, is used. The result of this query is a curve, which is outputted as a separate file. This curve is a probability density function over length scale. The name of the resulting file is printed to the **Query results**.

Compactness The Compactness query calculates mesh metrics and prints them in the **Query results**.

Cycle The Cycle query prints the cycle for the plot that is highlighted in the **Plot list** to the **Query results**.

Distance from Boundary The Distance From Boundary query calculates how much mass is at a given distance away from the boundary of a shape. An important distinction for this query is that distance from the boundary (for a given point) is not defined as the shortest distance to the boundary, but simultaneously as all surrounding distances. Axially symmetric objects (RZ-meshes) are treated as 3D meshes and length scales are calculated over the revolved, 3D object. The implementation employs a statistical approach, with the casting of uniform density, random lines. The result of this query is a curve, which is outputted as a separate file. This curve contains the amount of mass as a function of length scale. Integrating the curve between P0 and P1 will give the total mass at distance between P0 and P1 (given the interpretation above). The name of the resulting file is printed to the **Query results**.

Eulerian The Eulerian query calculates the Eulerian number for the mesh that is used by the highlighted plot in the Plot list. The results are printed to the **Query results**.

Expected Value The Expected Value query calculates the integral of $xf(x)dx$ for some curve f(x). The curve should be highlighted in the **Plot list** and prints the result to the **Query results**. This query is intended for distribution functions.

Grid Information The Grid Information query prints information for each domain in a multi- domain mesh. The mesh type is printed as well as the mesh sizes. For structured meshes the size information contains the logical mesh dimensions (IJK sizes) and for unstructured meshes the size information contains the number of nodes and number of cells in the mesh. The query can optionally accept a *get_extents* parameter that will cause the spatial extents for each domain to be obtained. The query also accepts an optional *get_ghosttype* parameter that causes the ghost zone information for each domain to be obtained. Both the numerical value and list of or'd values for ghost values are obtained. All query outputs are printed to the **Queryresults**.

Integrate The Integrate query calculates the area under the Curve plot that is highlighted in the Plot list and prints the result to the **Query results**.

Kurtosis The Kurtosis query calculates the kurtosis of a normalized distribution function. The normalized distribution function must be represented as a Curve plot in VisIt. Kurtosis measures the variability of a distribution by comparing the ratios of the fourth and second central moments. The results are print to the **Query results**.

L2Norm The L2Norm query calculates the L2Norm, or square of the integrated area, of a Curve plot. The Curve plot must be highlighted in the **Plot list**. The results are printed to the **Query results**.

L2Norm Between Curves The L2Norm query takes two Curve plots as input and calculates the L2Norm between the 2 curves. Both Curve plots must be highlighted in the **Plot list** or VisIt will issue an error message. The results are printed to the **Query results**.

Min The Min query calculates the minimum value for the variable used by the highlighted plot in the **Plot list** and prints the value and the logical and physical coordinates where the minimum value was found to the **Query results**.

Mass Distribution The Mass Distribution query calculates how much mass occurs at different length scales over a two or three dimensional object. Axially symmetric objects (RZ-meshes) are treated as 3D meshes and length scales are calculated over the revolved, 3D object. The implementation employs a statistical approach, with the casting of uniform density, random lines. The result of this query is a curve, which is outputted as a separate file. This curve contains the amount of mass as a function of length scale. Integrating the curve between P0 and P1 will give the total mass between length scale P0 and length scale P1. The name of the resulting file is printed to the **Query results**.

Max The Max query calculates the maximum value for the variable used by the highlighted plot in the Plot list and prints the value and the logical and physical coordinates where the maximum value was found to the **Query results**.

MinMax The MinMax query calculates the minimum and maximum values for the variable used by the highlighted plot in the Plot list and prints the values and their logical and physical coordinates in the **Query results**.

Moment of inertia This query will calculate the moment of inertia tensor for each cell in a three-dimensional dataset. The contribution of each cell is calculated assuming its mass all lies at the center of the cell. If the query is performed on a Pseudocolor plot, the plot's variable will be assumed to be density. If the query is performed on a plot such as a mesh plot or FilledBoundary plot, uniform density will be used. The results are printed to the **Query results**.

NodeCoords The NodeCoords query prints the node coordinates for the specified node and prints the values in the **Query results**.

NumNodes The NumNodes query prints the number of nodes for the mesh used by the highlighted plot in the **Plot list** to the **Query results**.

NumZones The NumZones query prints the number of zones for the mesh used by the highlighted plot in the **Plot list** to the **Query results**.

Revolved surface area The Revolved surface area query revolves the mesh used by the highlighted plot in the **Plot list** about the X-axis and prints the plot's revolved surface area to the **Query results**.

Revolved volume The Revolved volume area query revolves the mesh used by the highlighted plot in the **Plot list** about the X-axis and print's the plot's volume to the **Query results**.

Skewness The Skewness query calculates the skewness of a normalized distribution function. The normalized distribution function must be represented as a Curve plot in VisIt. Skewness measures the symmetry of a distribution using its second and third central moments. The results are print to the **Query results**

Spatial Extents The Spatial Extents query calculates the original or actual spatial extents for the plot that is highlighted in the **Plot list**. Whether the original or actual extents are calculated is determined by setting the options in the **Query parameters** area. The spatial extents are printed to the **Query results** when the query has finished.

Spherical compactness factor This query attempts to measure how spherical a three dimensional shape is. The query first determines what the volume of a shape is. It then constructs a sphere that has that same volume. Finally, the query positions the sphere so that the maximum amount of the original shape is within the sphere. The query returns the percentage of the original shape that is contained within the sphere. The results are print to the **Query results**. VisIt can produce a Curve plot of this query with respect to time.

Time The Time query prints the time for the plot that is highlighted in the Plot list to the **Query results**.

Variable Sum The Variable Sum query adds up the variable values for all cells using the plot highlighted in the **Plot list** and prints the results to the **Query results**. VisIt can produce a Curve plot of this query with respect to time.

Connected Component Variable Sum Performs the same operation as *Variable Sum* query except individually for each *component* of a disconnected mesh. The query result is a list of values, one for each component.

Volume The Volume query calculates the volume of the mesh used by the plot highlighted in the **Plot list** and prints the value to the **Query results**. VisIt can use this query to produce a Curve plot of volume with respect to time.

Connected Component Volume Performs the same operation as *Volume* query except individually for each *component* of a disconnected mesh. The query result is a list of values, one for each component.

Watertight The Watertight query determines if a three-dimensional surface mesh, of the plot highlighted in the **Plot list**, is "watertight", meaning that it is a closed volume with mesh connectivity such that every edge is incident to exactly two faces. This means that no edge can have a duplicate in the exact same position. The result of the query is printed in the **Query results**.

Weighted Variable Sum The Weighted Variable Sum query adds up the variable values, weighted by cell size (volume in 3D, area in 2D, length in 1D), for all cells using the plot highlighted in the **Plot list** and prints the results to the **Query results**. VisIt can produce a Curve plot of this query with respect to time.

Connected Component Weighted Variable Sum Performs the same operation as *Weighted Variable Sum* query except individually for each *component* of a disconnected mesh. The query result is a list of values, one for each component.

XRay Image See *XRayQuery*.

ZoneCenter The ZoneCenter query calculates the zone center for a certain cell in the database used by the highlighted plot in the Plot list. The cell center is printed to the **Query results** and the **Pick Window**.

Point Queries

Pick In general, the Pick query allows users to query a single zone or node at a user specified location in the dataset. There are several options for determining how this zone or node is chosen:

1. **Pick using coordinates**
2. **Pick using domain and element id**
3. **Pick using unique element label**

It's important to make sure that the plot you wish to query is highlighted in the **Plot list**. Information from your picked element, when available, will appear in both the **Pick Window** and the **Query results** window. If querying a 3D dataset, the queried element need not be on the surface of the mesh.

The Pick query also provides the option to generate a curve with respect to time, allowing the user to set the start time, stop time, and stride. **Note on performance**: when generating a curve over time, users have the option to preserve either the picked *coordinate* or the picked *element*. While each of these choices will produce very different results, it's worth keeping in mind that preserving the picked *element* will be substantially faster than preserving the picked *coordinate* when working with datasets with large numbers of time steps.

TrajectoryByNode and TrajectoryByZone The TrajectoryByNode and TrajectoryByZone queries first perform a Pick using domain and element id on their respective elements, and they then generate a curve *plotting one variable with respect to another*. You'll notice that, next to the **Variables** parameter, there is a text box containing default variables **var_for_x** and **var_for_y**. Replace these defaults with your desired variables for the query, and the resulting curve will plot your replacement for **var_for_x** with respect to **var_for_y**.

Line Queries

Lineout The Lineout query creates a new instance of the highlighted plot in the **Plot list**, applies a Lineout operator, and copies the plot to another vis window. The properties of the Lineout operator such as the start and end points are set using the controls in the **Query parameters area** of the **Query Window**. Creating Lineouts in this manner instead of using VisIt's interactive lineout allows you to create 1D Curve plots from 3D databases.

Executing a query

VisIt has many queries from which to choose. You can choose the type of query to execute by clicking on the name of the query in the **Queries list**. The **Queries list** usually displays the names of all of the queries that VisIt knows how to execute. If you instead want to view a subset of the queries, grouped by function, you can make a selection from the **Display as** combo box. Once you have clicked on a query in the **Query list**, the **Query parameters** area updates to show the controls that you need to edit the parameters for the query. In the case of a point query like Pick, the only parameters you need to specify are the 3D point where VisIt will extract values and the names of the variables that you want to examine. Once you specify the query parameters, click the **Query** button to tell VisIt to process the query. Once VisIt has fulfilled your request, the query results are displayed in the **Query results** at the bottom of the **Query Window**.

Querying over time

Many of VisIt's queries can be executed for every time state in the database used by the queried plot. The results from a query over time is a Curve plot that plots the query results with respect to time. The **Query parameters** area contains a **Time Curve** button when the selected query can be plotted over time. Clicking the **Time Curve** button executes the selected query for each time state in the database used by the plot highlighted in the **Plot list**. VisIt then creates a new Curve plot in a new vis window and uses the query results versus time as the curve data.

By default, querying over time will force VisIt to execute the selected query on every time state in the relevant database. If you want to restrict the number of time states used when querying over time or if you want to set some general options that also affect how time curves are created, you can set additional options in the **Query Over Time Window** (see Figure 4.249). If you want to open the **Query Over Time Window**, click on the **Query over time** option in the **Controls** menu in VisIt's **Main Window**.

Querying over a time range

You can restrict the range of time states that are considered when VisIt is performing a query over time if you specify a start or end time state in the **Query Over Time Window**. To set a starting time state, click the **Starting timestep** check box and enter a new time state into the adjacent text field. To set an ending time state, click the **Ending timestep** check box and enter a new ending time state into the adjacent text field.

In addition to setting the starting and ending time states, you can also specify a stride so VisIt can skip frames in the middle and consider every Nth frame instead of every frame. If you want to specify a stride, enter a new stride into the **Stride** text field in the **Query Over Time Window** and click the **Apply** button.

Setting the axis title

When VisIt creates a new Curve plot, after having calculated a query over time, the horizontal axis label is labeled with the database cycles. If you prefer to think about time in terms of time state or simulation time then you can change the axis label by clicking one of the following radio buttons in the **Query Over Time Window** : **Cycle**, **Time**, **Timestep**.

Setting the time curve's destination window

When VisIt creates a Curve plot using the results of a query over time, the Curve plot is placed in a vis window designated for Curve plots. If there is no vis window into which the Curve plot can be added, VisIt creates a new vis window to contain the Curve plot. If you want VisIt to always place the new Curve plot in a specific window, turn off the **Use 1st unused window or create new one** check box and enter a new window number into the **Window#** text field. After setting these options, subsequent Curve plots created by querying over time will be added to the specified vis window.

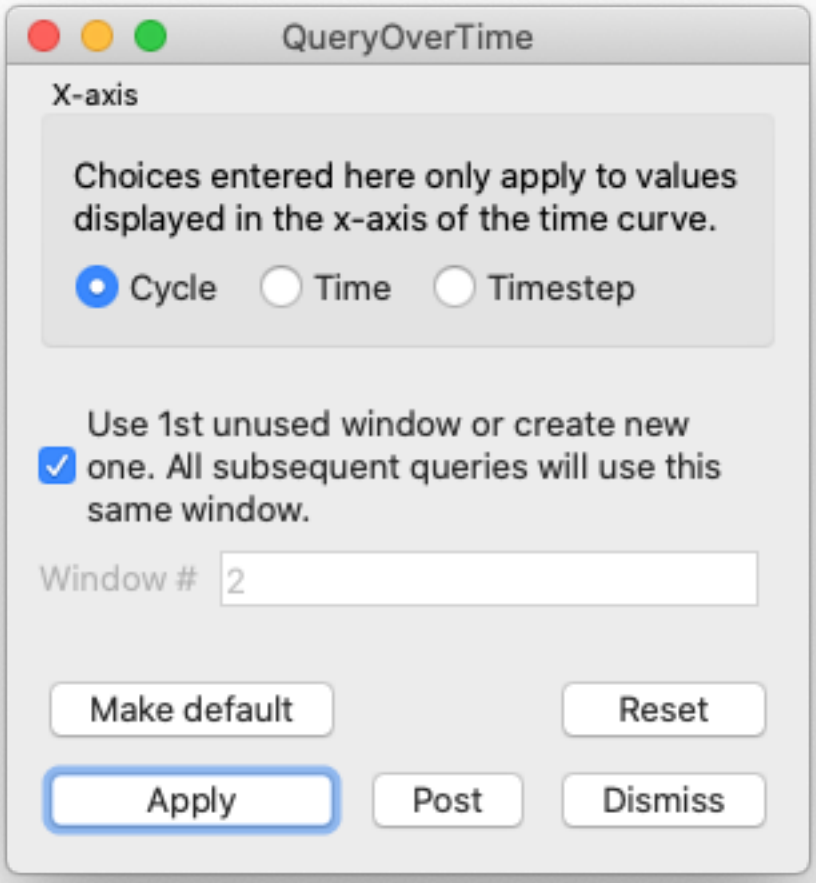


Fig. 4.249: Query Over Time Window

Python Queries

Python Queries allow you to use a Python script to define a custom query. You can use the Python Query tab in the Query Window to create a Python query:

In your query you can access and process Python wrapped versions of the VTK objects that represent your data. To demonstrate this, here is an example Python Query that computes the average of a zonal (or cell-centered) field:

```
# simple cell average query
class CellAvgQuery(SimplePythonQuery):
    def __init__(self):
        SimplePythonQuery.__init__(self)
        self.name = "CellAvgQuery"
        self.description = "Calculate the average cell value."
    def pre_execute(self):
        # init vars used to compute the average
        self.total_ncells = 0
        self.total_sum    = 0.0
    def execute_chunk(self,ds_in,domain_id):
        # sum over cell data array passed to query args
        ncells = ds_in.GetNumberOfCells()
        self.total_ncells += ncells
        cell_data = ds_in.GetCellData().GetArray(self.input_var_names[0])
        for i in range(ncells):
            self.total_sum += cell_data.GetTuple1(i)
    def post_execute(self):
        # calculate average and set results
        res_val = mpicom.sum(self.total_sum) / mpicom.sum(self.total_ncells)
        res_txt = "The average value = " + self.float_format
        res_txt = res_txt % res_val
        self.set_result_text(res_txt)
        self.set_result_value(res_val)

py_filter = CellAvgQuery
```

This example is from our pyavt examples.

This example inherits from *SimplePythonQuery*. The base classes of VisIt's Python Filters are defined in the pyavt module.

You can select the variables passed to your Query using the Python Query variable list:

In the script, your class needs to implement four methods:

Constructor : Called to initialize the Python Query Filter object. Use this to call the base class constructor and provide a name and description of your custom query.

Pre Execute [`pre_execute(self)`] This method is called on all MPI tasks before any domains have been processed. Use this to initialize any state needed before parallel execution. In this example we initialize variables used to hold the total field value sum and the total number of cells.

Execute Chunk [`execute_chunk(self,ds_in,domain_id)`] This method is called to process each domain. When VisIt runs with MPI, *execute_chunk()* will be called in parallel across MPI tasks. `ds_in` is a Python-wrapped VTK object and `domain_id` provides the domain id of `ds_in`. In this example, for each domain we get the field value array and update the aggregate sum and the total number of cells.

Post Execute [`post_execute(self)`] This method is called on all MPI tasks after all domains have been processed. Use this to finalize results after parallel execution. In this example, we use MPI to combine the local results across MPI tasks.

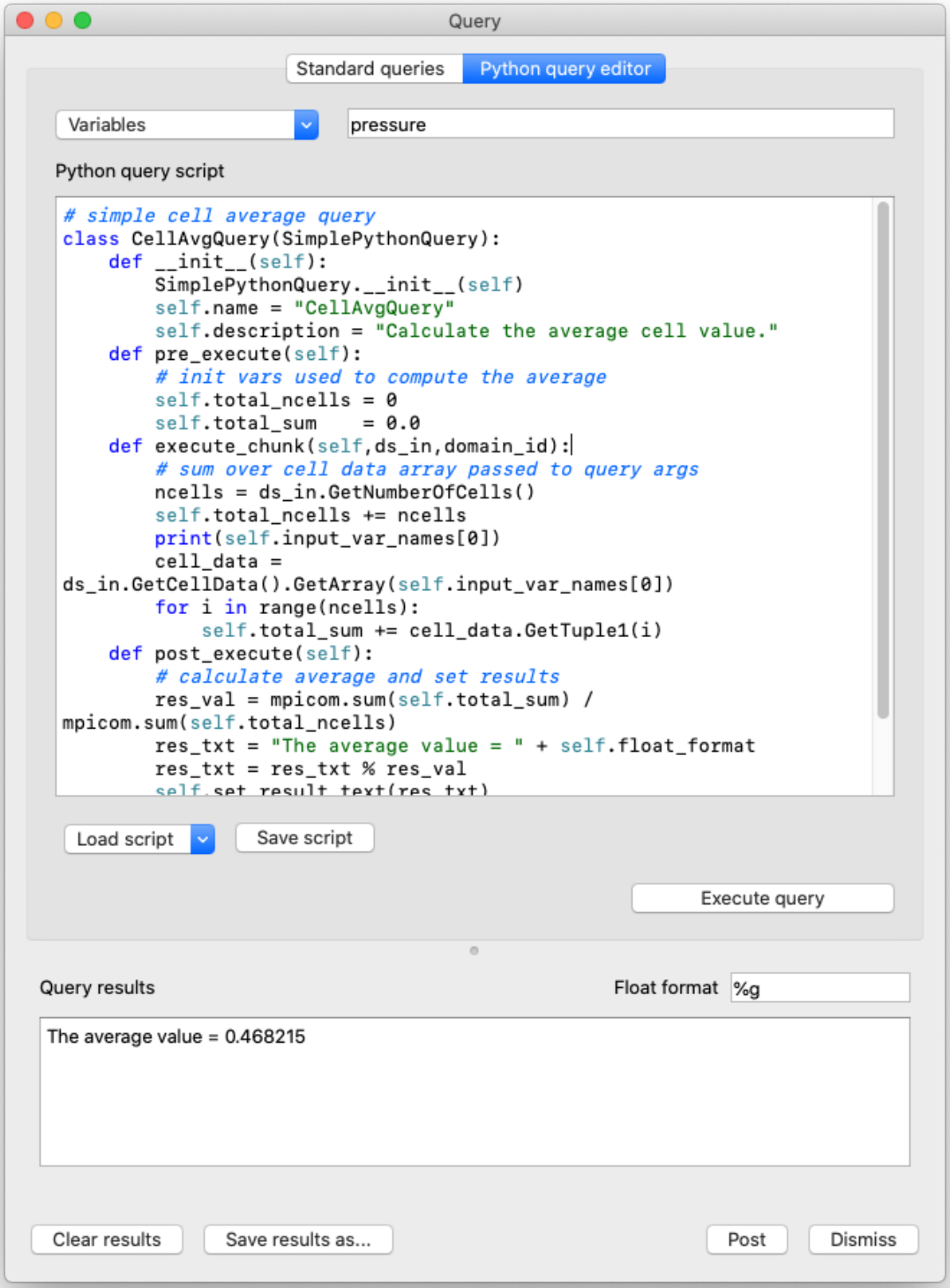


Fig. 4.250: Python Query Editor

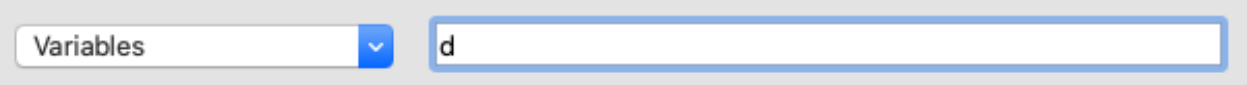


Fig. 4.251: Python Query Variable list

The final aspect required is to bind your new Python Query Filter class to *py_filter*, this is the name VisIt uses to connect your Python script to the Python Filter Runtime in the engine.

When you run your Python Query, results are presented like any other Query: Displayed in the Query window and can be accessed via VisIt's Query output CLI functions.

4.8.3 X Ray Image Query

- *Introduction*
- *Query Arguments*
  - *Standard Arguments*
    - *Output Filenames and Directories*
    - *Output Types*
    - *Units*
  - *Camera Specification*
    - *Complete Camera Specification*
    - *Simplified Camera Specification*
  - *Calling the Query*
- *Examples*
- *Conduit Output*
  - *Why Conduit Output?*
  - *Overview of Output*
    - *Basic Mesh Output*
    - *Metadata*
      - *View Parameters*
      - *Query Parameters*
      - *Other Metadata*
    - *Imaging Planes and Rays Meshes*
      - *Imaging Planes*
      - *Rays Meshes*
    - *Spatial Extents Meshes*
    - *1D Spectra Curves*
  - *Quick Results*

- *Pitfalls*
- *Visualizing with VisIt*
  - *Visualizing the Basic Mesh Output*
  - *Visualizing the Imaging Planes*
  - *Visualizing the Rays Meshes*
  - *Visualizing the Spatial Extents Meshes*
  - *Visualizing the 1D Spectra Curves*
- *Introspecting with Python*
  - *Getting a General Overview of the Output*
  - *Accessing the Basic Mesh Output Data*
  - *Accessing the Metadata*
  - *Accessing the Spatial Extents Meshes Data*
  - *Accessing the 1D Spectra Curves Data*
  - *Accessing Everything Else*
- *Troubleshooting*
  - *Is my image blank?*
  - *Why is my image blank?*
  - *Where are the rays intersecting my geometry?*
  - *What information is the query using to create the output?*
  - *The fields in the Conduit Output are 1D. How can I reshape them to be 3D?*

Introduction

The X Ray Image Query computes the attenuation and self-emission for radiation passing through an object. The query can be used for EUV radiation and, in some cases, for optical light. The attenuation is used when simulating a radiograph (an xray of a broken bone is a radiograph). If the object is hot enough, it emits xrays and the self-emission is of interest.

The query takes as input a mesh with zone-centered opacities (absorbtivities) and emissivities. The absorbtivity and emissivity variables must be zone-centered and can be either scalar variables (special case) or array variables to support multiple energy groups. The query tracks rays through this mesh, using opacity and emissivity information in each zone to generate an image per radiation energy group. The opacity and emissivity in each zone is a function of radiation energy and rays are tracked for all energies. These rays are regularly spaced in the image plane.

The inputs to the query are usually generated by a computer simulation of the temporal evolution of an object, which records the opacity and emissivity at a set of times. The images produced by the query can be convolved with the spatial, spectral, and temporal response of an (xray) detector in a post-processing step. The simulated data generated using the query can be compared with data recorded during an experiment to see if the simulation is accurate.

The query operates on 2D R-Z meshes and 3D meshes. In the case of 2D R-Z meshes, the mesh is revolved around the Z-axis and the rays are tracked in 3 dimensions.

The query performs the following integration as it traces the rays through the volume.

Show/Hide Code for XRay Image Query

```
            for (int j = 0 ; j < numBins ; j++)
            {
                double tmp = exp(-a[j] * segLength);
                intensityBins[j] = intensityBins[j] * tmp + e[j] * (1.0 - tmp);
                pathBins[j] = pathBins[j] + a[j] * segLength;
            }
```

In this code snippet, `a` represents the absorbtivity, and `e` represents the emissivity.

If the `divide_emis_by_absorb` is set, then the following integration is performed.

Show/Hide Code for Absorbtivity-Normalized X Ray Image Query

```
            for (int j = 0 ; j < numBins ; j++)
            {
                double tmp = exp(-a[j] * segLength);
                intensityBins[j] = intensityBins[j] * tmp + (e[j] / a[j]) * (1.0 -␣
↪tmp);
                pathBins[j] = pathBins[j] + a[j] * segLength;
            }
```

When using the `divide_emis_by_absorb` option, beware of the case where zones have zero absorbtivity. This will lead to NaN intensity results.

When the goal of the query is to generate a radiograph, the user supplies a background intensity (using either *background_intensity* or *background_intensities*; see *Standard Arguments*) and sets the emissivity arrays to zero. The self-emission image produced by the query is then a radiograph.

When the goal of the query is to generate a self-emission image, the emissivities should be non-zero and a background intensity should not be supplied.

Sometimes the goal of an experiment is to generate a radiograph, but the object is hot enough that self-emission might "wash out" the radiograph. In this case, the emissivities should be non-zero and the background intensity should be supplied. The background intensity can then be adjusted until the radiograph is not washed out.

The X Ray Image Query can be used as a part of a larger workflow for simulating X Ray Detectors, namely for the National Ignition Facility. For a discussion of how the query fits into this larger workflow as well as additional detail on our efforts to add Conduit Blueprint output to the query, the following presentation is provided: Supporting Simulated Diagnostics with VisIt's X Ray Image Query (DOECGF23) Presented at the DOE Computer Graphics Forum 2023.

Query Arguments

The query takes a few different kinds of arguments:

Standard Arguments

The standard arguments have to do with the query execution, output, debugging, and passing through metadata.

vars	An array of the names of the absorbtivity and emissivity variables.
background_intensity	The background intensity if ray tracing scalar variables. The default is 0.
background_intensities	The background intensities if ray tracing array variables. The default is 0.
divide_emis_by_absorb	Described above. The default is 0.
image_size	The width and height of the image in pixels. The default is 200 x 200.
debug_ray	The ray index for which to output ray tracing information. The default is -1, which turns it off.
output_ray_bounds	Output the ray bounds as a bounding box in a VTK file. The default is 0 (off). The name of the file is `ray_bounds.vtk`.
energy_group_bounds	The energy group bounds can be handed off to the query in a list or tuple. The values will appear in the Spatial Extents Mesh in the Blueprint output.

If using the *Conduit Output*, many of these arguments will appear in the output in a few different places. The `vars` will show up as `abs_var_name` and `emis_var_name` under the *Query Parameters* section of the *Metadata*. `divide_emis_by_absorb` shows up under the *Query Parameters* section of the *Metadata*. `image_size` shows up as `num_x_pixels` and `num_y_pixels` under the *Query Parameters* section of the *Metadata*. The `energy_group_bounds` appear under the `spatial_coords` in the *Spatial Extents Meshes*.

Output Filenames and Directories

output_dir		The output directory. The default is "."
family_files		A flag indicating if the output files should be familied. The default is off. If it is off then the output file is `output.ext`, where `ext` is the file extension. If the file exists it will overwrite the file. If it is on, then the output file is `outputXXXX.ext`, where `XXXX` is chosen to be the smallest integer not to overwrite any existing files. As of VisIt 3.4, it is recommended to use *filename_scheme* in lieu of *family_files*.
filename_scheme		The naming convention for output filenames. This option is available in VisIt 3.4, and is meant to replace the *family_files* option. If both are provided, *filename_scheme* will be used.
	"none" or 0	The default. Output filenames will be of the form `output.ext`, where `ext` is the file extension. If the filename already exists, VisIt will overwrite it.
	"family" or 1	If on, VisIt will attempt to family output files. Output filenames will be of the form `output.XXXX.ext`, where `XXXX` is chosen to be the smallest integer such that the filename is unique.
	"cycle" or 2	VisIt will put cycle information in the filename. Output filenames will be of the form `output.cycle_XXXXXX.ext`, where `XXXXXX` is the cycle number. If another file exists with this name, VisIt will overwrite it.

Output Types

output_type		The format of the image. The default is PNG.
	"bmp" or 0	BMP image format. This is deprecated as of VisIt 3.4.
	"jpeg" or 0 (1 prior to VisIt 3.4)	JPEG image format.
	"png" or 1 (2 prior to VisIt 3.4)	PNG image format.
	"tif" or 2 (3 prior to VisIt 3.4)	TIFF image format.
	"rawfloats" or 3 (4 prior to VisIt 3.4)	File of 32 or 64 bit floating point values in IEEE format.
	"bov" or 4 (5 prior to VisIt 3.4)	BOV (Brick Of Values) format, which consists of a text header file describing a rawfloats file.
	"json" or 5 (6 prior to VisIt 3.4)	Conduit JSON output.
	"hdf5" or 6 (7 prior to VisIt 3.4)	Conduit HDF5 output.
	"yaml" or 7 (8 prior to VisIt 3.4)	Conduit YAML output.

When specifying "bov" or "rawfloats" output, the value can be either 32 or 64 bit floating point values. The number of bits is determined by the number of bits in the data being processed.

When specifying "bov" output, 2 files are created for each variable. One contains the `intensity` and the other the `path_length`. The files are named `output.XX.bof` and `output.XX.bov` with `XX` being a sequence number. The `intensity` variables are first followed by the `path_length` variables in the sequence. For example, if the input array variables were composed of 2 scalar variables, the files would be named as follows:

- output.00.bof
- output.00.bov - `intensity` from the first variable of the array variable.
- output.01.bof
- output.01.bov - `intensity` from the second variable of the array variable.
- output.02.bof
- output.02.bov - `path_length` from the first variable of the array variable.
- output.03.bof
- output.03.bov - `path_length` from the second variable of the array variable.

The Conduit output types provide a plethora of extra features; to learn more see *Conduit Output*.

Units

Units of various quantities can be passed through the query. None of these values are used in any calculations the query does to arrive at its output; all are optional. These units appear in the *Conduit Output* in a few different places.

spatial_units	The units of the simulation in the x and y dimensions.
energy_units	The units of the simulation in the z dimension.
abs_units	The units of the absorbtivity variable passed to the query.
emis_units	The units of the emissivity variable passed to the query.
intensity_units	The units of the intensity output.
path_length_info	Metadata describing the path length output.

The `spatial_units` and `energy_units` appear in the *Spatial Extents Meshes*. The `abs_units` and the `emis_units` appear in the *Query Parameters* section of the *Metadata*. The `intensity_units` and the `path_length_info` appear in the *Basic Mesh Output* and in the 3D Spatial Extents Mesh (*Spatial Extents Meshes*) under the fields.

Camera Specification

The query also takes arguments that specify the orientation of the camera in 3 dimensions. This can take 2 forms. The first is a complete specification that matches the 3D image viewing parameters and the second is a simplified specification that gives limited control over the camera but is easier to use.

Complete Camera Specification

The complete version consists of:

normal	The view normal. The default is (0., 0., 1.).
focus	The focal point. The default is (0., 0., 0.).
view_up	The up vector. The default is (0., 1., 0.).
view_angle	The view angle. The default is 30. This is only used if perspective projection is enabled.
parallel_scale	The parallel scale, or view height. The default is 0.5.
view_width	The view width. The default is 0.5. If this argument is not specified, the query will assume pixels are to be square, and it will use the specified `image_size` and the `parallel_scale` to calculate the correct `view_width`. If this argument *is* specified, the query may produce results with non-square pixels.
near_plane	The near clipping plane. The default is -0.5.
far_plane	The far clipping plane. The default is 0.5.
image_pan	The image pan in the X and Y directions. The default is (0., 0.).
image_zoom	The absolute image zoom factor. The default is 1. A value of 2. zooms the image closer by scaling the image by a factor of 2 in the X and Y directions. A value of 0.5 zooms the image further away by scaling the image by a factor of 0.5 in the X and Y directions.
perspective	Flag indicating if doing a parallel or perspective projection. 0 indicates parallel projection. 1 indicates perspective projection.

If any of the above properties are specified in the parameters, the query will use the complete version and disregard all arguments pertaining to the simplified version.

When a Conduit Blueprint output type is specified, these parameters will appear in the metadata. See *View Parameters* for more information.

Simplified Camera Specification

The simplified version consists of:

width	The width of the image in physical space. The default is 1.0.
height	The height of the image in physical space. The default is 1.0.
origin	The point in 3D corresponding to the center of the image.
theta *phi*	The orientation angles. The default is 0. 0. and is looking down the Z axis. Theta moves around the Y axis toward the X axis. Phi moves around the Z axis. When looking at an R-Z mesh, phi has no effect because of symmetry.
up_vector	The up vector.

During execution, the simplified camera specification parameters are converted to the complete ones.

Calling the Query

There are a couple ways to call the X Ray Image Query, each with their own nuances.

The first is the standard way of calling the query, by using a dictionary to store the arguments. This way is recommended. Here is an example:

```
params = dict()
params["image_size"] = (400, 300)
params["output_type"] = "hdf5"
params["focus"] = (0., 2.5, 10.)
params["perspective"] = 1
params["near_plane"] = -25.
params["far_plane"] = 25.
params["vars"] = ("d", "p")
params["parallel_scale"] = 10.
Query("XRay Image", params)
```

Of course, one could use this to set up the parameters instead:

```
params = GetQueryParameters("XRay Image")
```

The default arguments to the query will be provided here, containing the *Complete Camera Specification* and not the *Simplified Camera Specification* arguments. To use the *Simplified Camera Specification* arguments, one must specify them manually and not specify any arguments from the *Complete Camera Specification*.

The second way to call the query is the old style of argument passing:

```
Query("XRay Image",
   output_type,
   output_dir,
   divide_emis_by_absorb,
   origin_x,
   origin_y,
   origin_z,
   theta,
   phi,
   width,
   height,
   image_size_x,
```

(continues on next page)

(continued from previous page)

```
   image_size_y,
   vars)

# An example
Query("XRay Image", "hdf5", ".", 1, 0.0, 2.5, 10.0, 0, 0, 10., 10., 400, 300, ("d", "p
↪"))
```

This way of calling the query exclusively makes use of the *Simplified Camera Specification*.

Examples

Let's look at some examples, starting with some simulated x rays using curv2d.silo, which contains a 2D R-Z mesh. Here is a pseudocolor plot of the data.

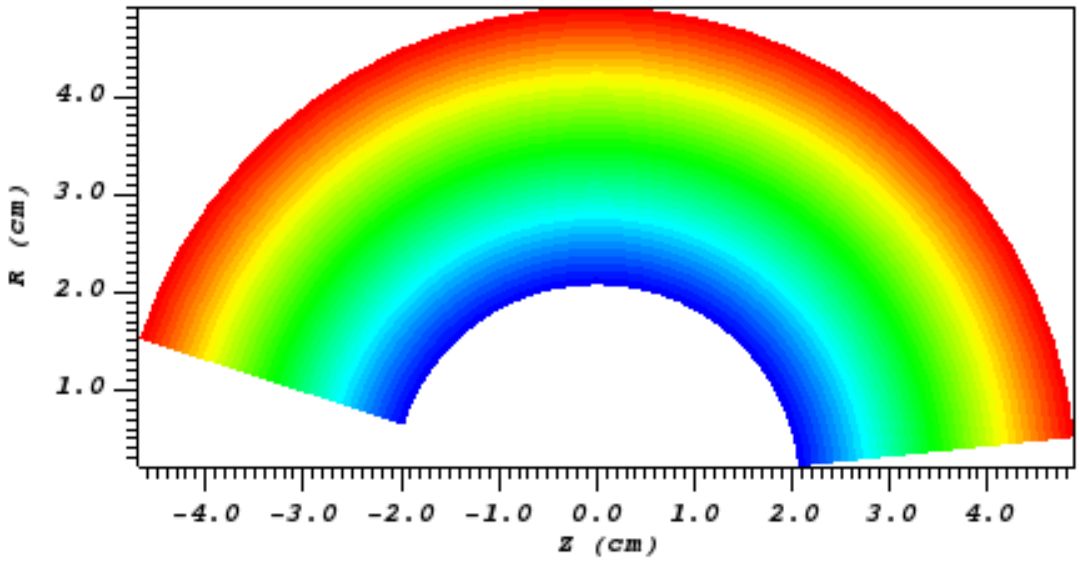


Fig. 4.252: The 2D R-Z data.

Now we will show the Python code to generate a simulated x ray looking down the Z Axis and the resulting image.

```
params = GetQueryParameters("XRay Image")
params['image_size'] = (300, 300)
params['divide_emis_by_absorb'] = 1
params['width'] = 10.
params['height'] = 10.
params['vars'] = ("d", "p")
Query("XRay Image", params)
```

Here is the Python code to generate the same image but looking at it from the side.

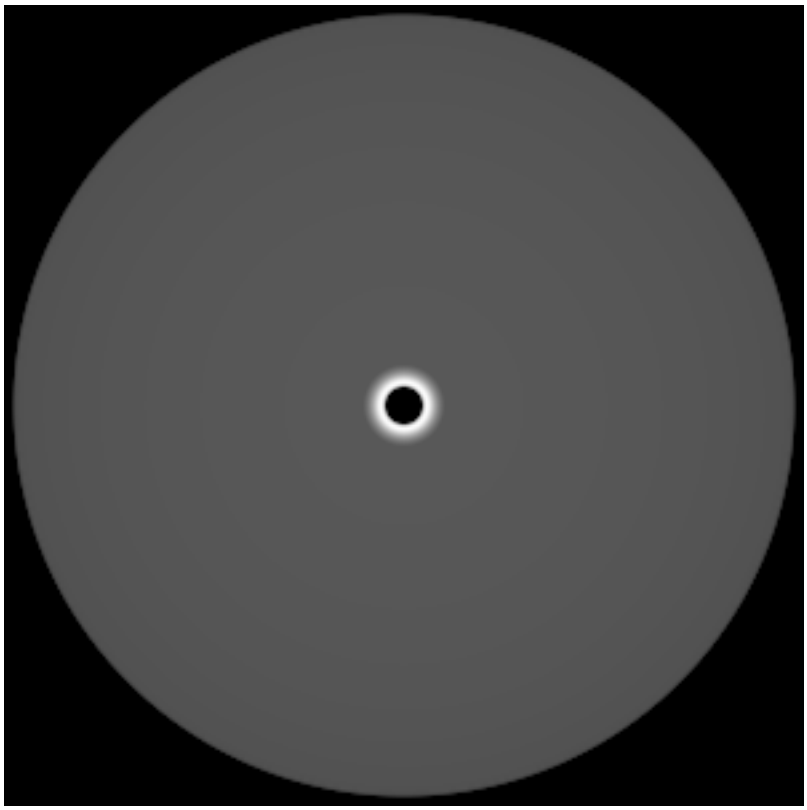


Fig. 4.253: The resulting x ray image.

```
params = GetQueryParameters("XRay Image")
params['image_size'] = (300, 300)
params['divide_emis_by_absorb'] = 1
params['width'] = 10.
params['height'] = 10.
params['theta'] = 90.
params['phi'] = 0.
params['vars'] = ("d", "p")
Query("XRay Image", params)
```

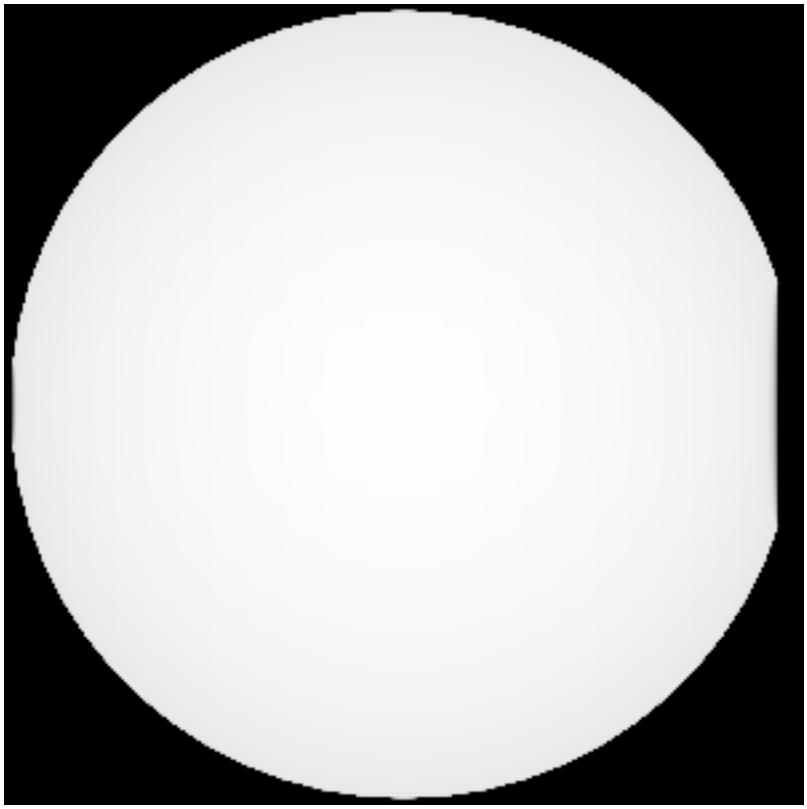


Fig. 4.254: The resulting x ray image.

Here is the same Python code with the addition of an origin that moves the image down and to the right by 1.

```
params = GetQueryParameters("XRay Image")
params['image_size'] = (300, 300)
params['divide_emis_by_absorb'] = 1
params['width'] = 10.
params['height'] = 10.
params['theta'] = 90.
params['phi'] = 0.
params['origin'] = (0., 1., 1.)
params['vars'] = ("d", "p")
Query("XRay Image", params)
```

Now we will switch to a 3D example using globe.silo. Globe.silo is an unstructured mesh consisting of tetrahedra, pyramids, prisms and hexahedra forming a globe. Here is an image of the tetrahedra at the center of the globe that form 2 cones.

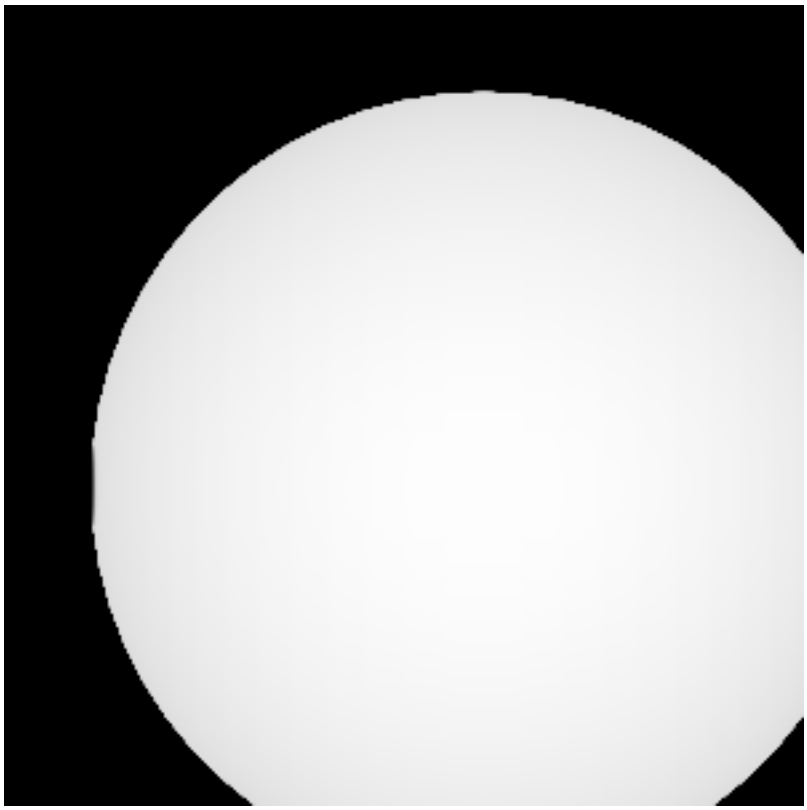


Fig. 4.255: The resulting x ray image.

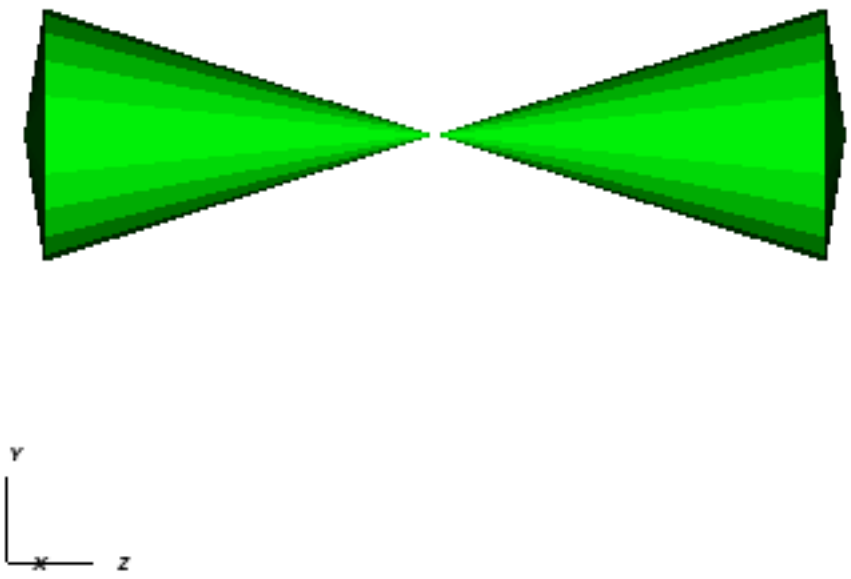


Fig. 4.256: The tetrahedra at the center of the globe.

Here is the Python code for generating an x ray image from the same orientation. Note that we have defined some expressions so that the x ray image shows some variation.

```
DefineScalarExpression("u1", 'recenter(((u+10.)*0.01), "zonal")')
DefineScalarExpression("v1", 'recenter(((v+10.)*0.01*matvf(mat1,1)), "zonal")')
DefineScalarExpression("v2", 'recenter(((v+10.)*0.01*matvf(mat1,2)), "zonal")')
DefineScalarExpression("v3", 'recenter(((v+10.)*0.01*matvf(mat1,3)), "zonal")')
DefineScalarExpression("v4", 'recenter(((v+10.)*0.01*matvf(mat1,4)), "zonal")')
DefineScalarExpression("w1", 'recenter(((w+10.)*0.01), "zonal")')

params = GetQueryParameters("XRay Image")
params['image_size'] = (300, 300)
params['divide_emis_by_absorb'] = 1
params['width'] = 4.
params['height'] = 4.
params['theta'] = 90.
params['phi'] = 0.
params['vars'] = ("w1", "v1")
Query("XRay Image", params)
```

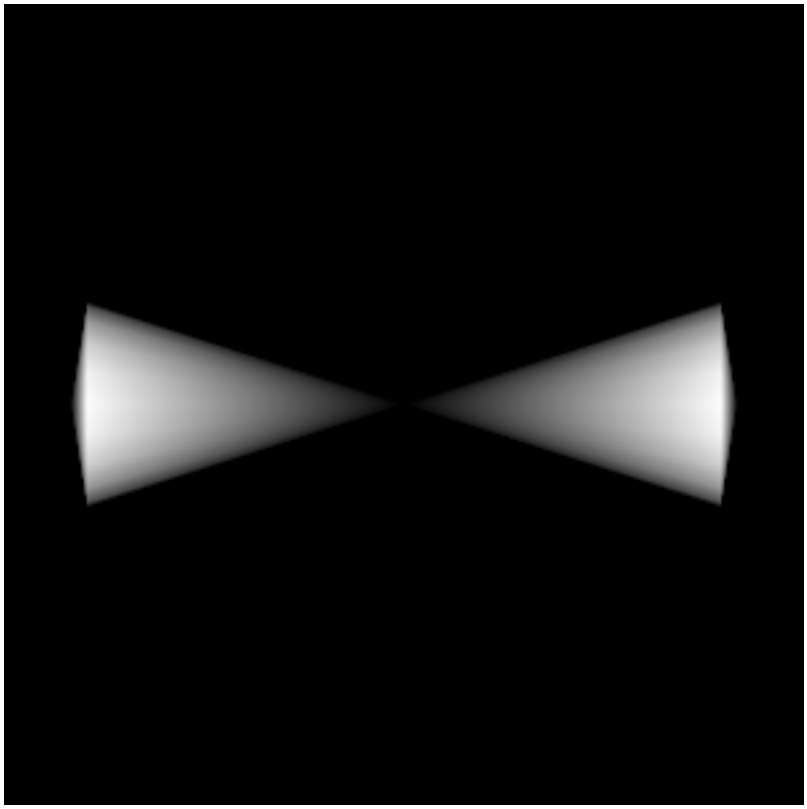


Fig. 4.257: The resulting x ray image.

Now we will look at the pyramids in the center of the globe.

Here is the Python code for generating an x ray image from the same orientation using the full view specification. The view specification was merely copied from the 3D tab on the View window.

```
params = dict(output_type="png")
params['image_size'] = (300, 300)
```

(continues on next page)

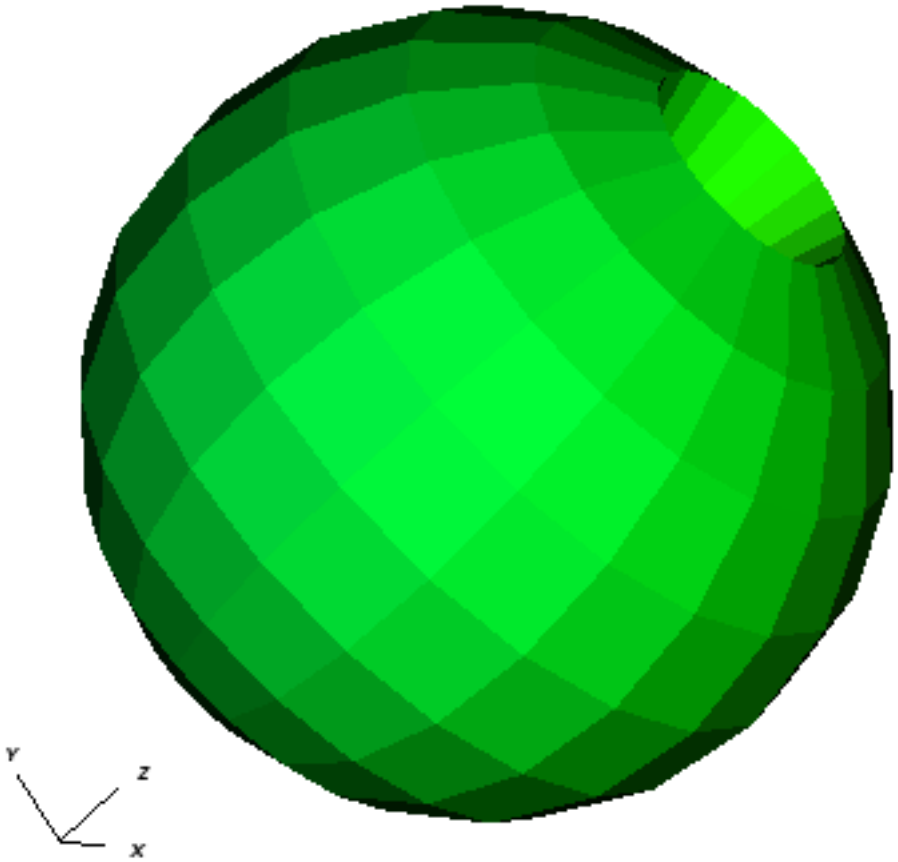


Fig. 4.258: The pyramids at the center of the globe.

(continued from previous page)

```
params['divide_emis_by_absorb'] = 1
params['focus'] = (0., 0., 0.)
params['view_up'] = (-0.0651, 0.775, 0.628)
params['normal'] = (-0.840, -0.383, 0.385)
params['view_angle'] = 30.
params['parallel_scale'] = 17.3205
params['near_plane'] = -34.641
params['far_plane'] = 34.641
params['image_pan'] = (0., 0.)
params['image_zoom'] = 8
params['perspective'] = 0
params['vars'] = ("w1", "v2")
Query("XRay Image", params)
```

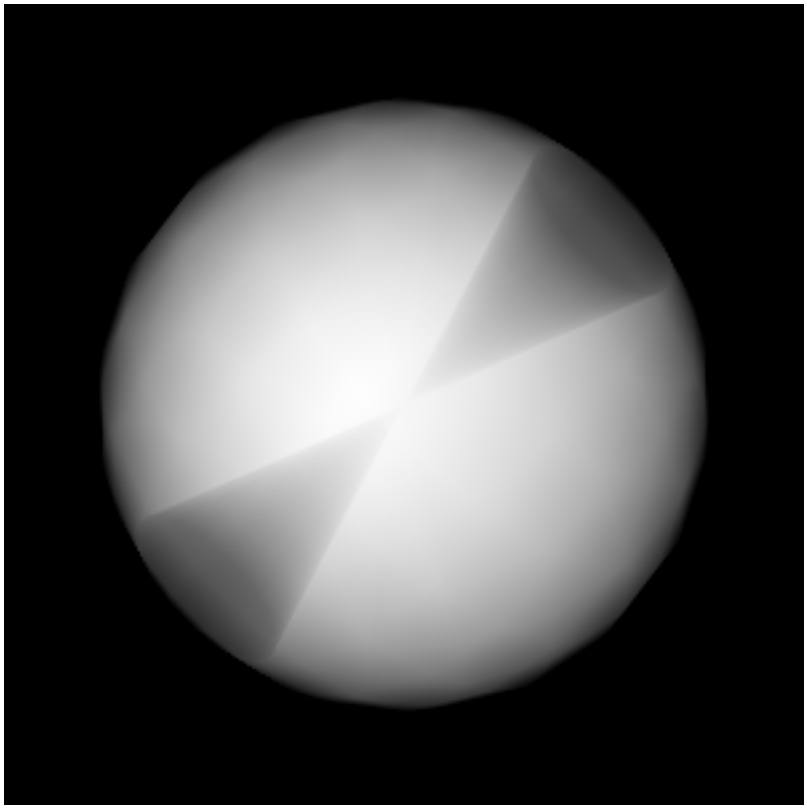


Fig. 4.259: The resulting x ray image.

The next example illustrates use of one of the *Conduit Output* types.

```
# A test file
OpenDatabase("testdata/silo_hdf5_test_data/curv3d.silo")

AddPlot("Pseudocolor", "d")
DrawPlots()
```

We call the query as usual, although there are a few extra arguments we can provide that are used for generating the Conduit output in particular.

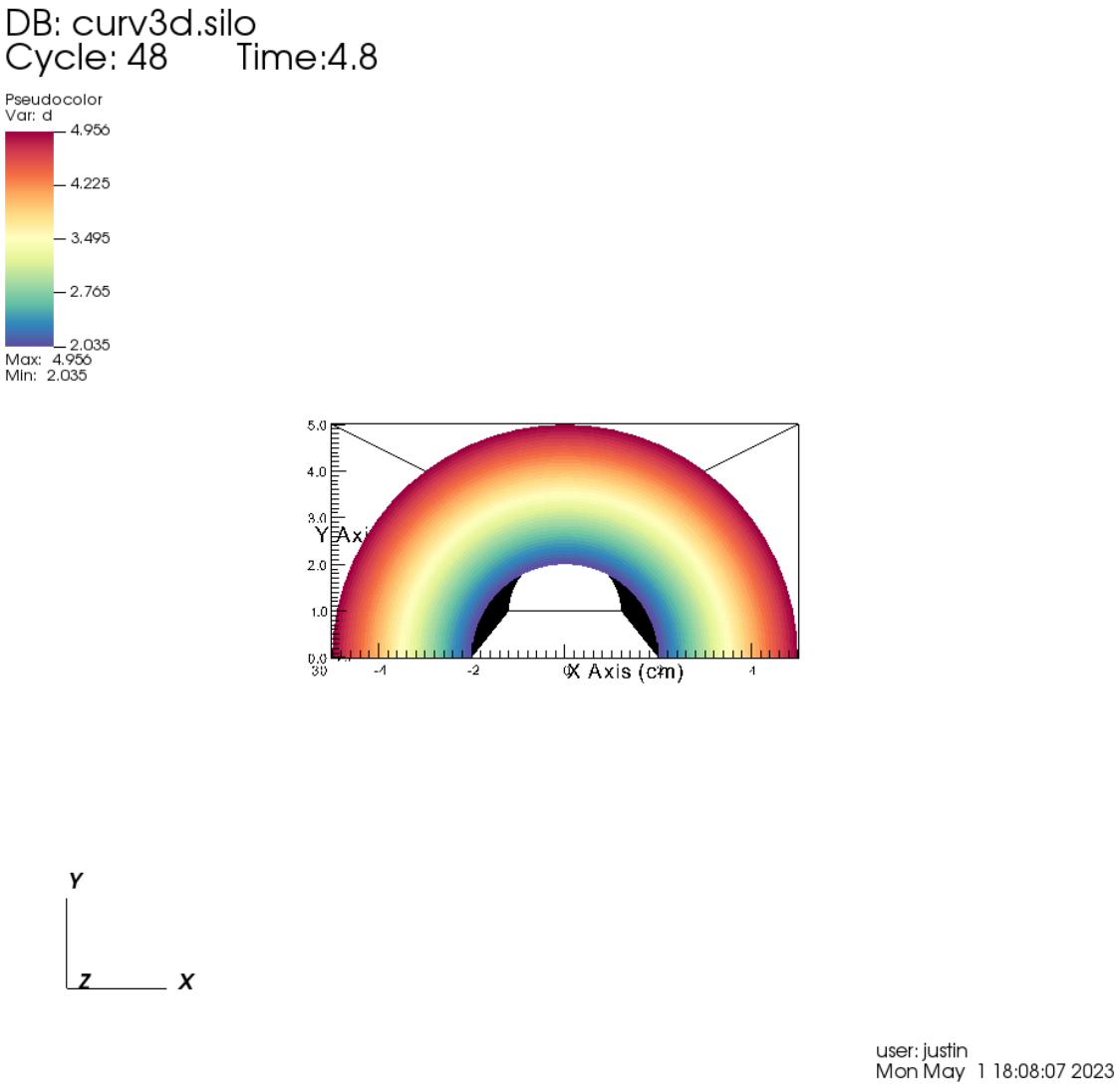


Fig. 4.260: Our input mesh.

```
params = dict()
params["image_size"] = (400, 300)
params["output_type"] = "hdf5"
params["focus"] = (0., 2.5, 10.)
params["perspective"] = 1
params["near_plane"] = -25.
params["far_plane"] = 25.
params["vars"] = ("d", "p")
params["parallel_scale"] = 10.

# ENERGY GROUP BOUNDS
params["energy_group_bounds"] = [2.7, 6.2]

# UNITS
params["spatial_units"] = "cm"
params["energy_units"] = "kev"
params["abs_units"] = "cm^2/g"
params["emis_units"] = "GJ/cm^2/ster/ns/keV"
params["intensity_units"] = "intensity units"
params["path_length_info"] = "path length metadata"

Query("XRay Image", params)
```

To look at the raw data from the query, we run this code:

```
import conduit
xrayout = conduit.Node()

conduit.relay.io.blueprint.load_mesh(xrayout, "output.root")

print(xrayout["domain_000000"])
```

This yields the following data overview. See *Introspecting with Python* for a deeper dive into viewing and extracting the raw data from the *Conduit Output*.

```
state:
  time: 4.8
  cycle: 48
  xray_view:
    normal:
      x: 0.0
      y: 0.0
      z: 1.0
    focus:
      x: 0.0
      y: 2.5
      z: 10.0
    view_up:
      x: 0.0
      y: 1.0
      z: 0.0
    view_angle: 30.0
    ... ( skipped 6 children )
    image_zoom: 1.0
    perspective: 1
    perspective_str: "perspective"
  xray_query:
```

(continues on next page)

(continued from previous page)

```
    divide_emis_by_absorb: 0
    divide_emis_by_absorb_str: "no"
    num_x_pixels: 400
    num_y_pixels: 300
    ... ( skipped 2 children )
    emis_var_name: "pa"
    abs_units: "cm^2/g"
    emis_units: "GJ/cm^2/ster/ns/keV"
  xray_data:
    detector_width: 8.80338743415454
    detector_height: 6.60254037884486
    intensity_max: 1.96578788757324
    intensity_min: 0.0
    path_length_max: 519.428039550781
    path_length_min: 0.0
    image_topo_order_of_domain_variables: "xyz"
  domain_id: 0
coordsets:
  image_coords:
    type: "rectilinear"
    values:
      x: [0, 1, 2, ..., 399, 400]
      y: [0, 1, 2, ..., 299, 300]
      z: [0, 1, 2, 3, 4]
    labels:
      x: "width"
      y: "height"
      z: "energy_group"
    units:
      x: "pixels"
      y: "pixels"
      z: "bins"
  spatial_coords:
    type: "rectilinear"
    values:
      x: [0.0, 0.0220084685853863, 0.0440169371707727, ..., 8.78137896556915, 8.
↪80338743415454]
      y: [0.0, 0.0220084679294829, 0.0440169358589658, ..., 6.58053191091538, 6.
↪60254037884486]
      z: [0.0, 1.0, 2.0, 3.0, 4.0]
    info: "Energy group bounds size mismatch: provided 7 bounds, but 5 in query␣
↪results."
    units:
      x: "cm"
      y: "cm"
      z: "kev"
    labels:
      x: "width"
      y: "height"
      z: "energy_group"
  spatial_energy_reduced_coords:
    type: "rectilinear"
    values:
      x: [0.0, 0.0220084685853863, 0.0440169371707727, ..., 8.78137896556915, 8.
↪80338743415454]
      y: [0.0, 0.0220084679294829, 0.0440169358589658, ..., 6.58053191091538, 6.
↪60254037884486]
```

(continues on next page)

(continued from previous page)

```
    units:
      x: "cm"
      y: "cm"
    labels:
      x: "width"
      y: "height"
  spectra_coords:
    type: "rectilinear"
    values:
      x: [0.0, 1.0, 2.0, 3.0, 4.0]
    units:
      x: "kev"
    labels:
      x: "energy_group"
    info: "Energy group bounds size mismatch: provided 7 bounds, but 5 in query␣
↪results."
  ... ( skipped 2 children )
  far_plane_coords:
    type: "explicit"
    values:
      x: [22.264973744318, -22.264973744318, -22.264973744318, 22.264973744318]
      y: [-14.1987298105776, -14.1987298105776, 19.1987298105776, 19.1987298105776]
      z: [35.0, 35.0, 35.0, 35.0]
  ray_corners_coords:
    type: "explicit"
    values:
      x: [4.40169371707727, 22.264973744318, -4.40169371707727, ..., 4.40169371707727,
↪ 22.264973744318]
      y: [-0.801270189422432, -14.1987298105776, -0.801270189422432, ..., 5.
↪80127018942243, 19.1987298105776]
      z: [-15.0, 35.0, -15.0, ..., -15.0, 35.0]
  ray_coords:
    type: "explicit"
    values:
      x: [-4.39068948278457, -4.39068948278457, -4.39068948278457, ..., 22.
↪2093113099572, 22.2093113099572]
      y: [-0.790265955457691, -0.768257487528208, -0.746249019598725, ..., 19.
↪0317425124718, 19.1430673778756]
      z: [-15.0, -15.0, -15.0, ..., 35.0, 35.0]
topologies:
  image_topo:
    coordset: "image_coords"
    type: "rectilinear"
  spatial_topo:
    coordset: "spatial_coords"
    type: "rectilinear"
  spatial_energy_reduced_topo:
    coordset: "spatial_energy_reduced_coords"
    type: "rectilinear"
  spectra_topo:
    coordset: "spectra_coords"
    type: "rectilinear"
  ... ( skipped 2 children )
  far_plane_topo:
    type: "unstructured"
    coordset: "far_plane_coords"
    elements:
```

(continues on next page)

(continued from previous page)

```
      shape: "quad"
      connectivity: [0, 1, 2, 3]
  ray_corners_topo:
    type: "unstructured"
    coordset: "ray_corners_coords"
    elements:
      shape: "line"
      connectivity: [0, 1, 2, ..., 6, 7]
  ray_topo:
    type: "unstructured"
    coordset: "ray_coords"
    elements:
      shape: "line"
      connectivity: [0, 120000, 1, ..., 119999, 239999]
fields:
  intensities:
    topology: "image_topo"
    association: "element"
    units: "intensity units"
    values: [0.0, 0.0, 0.0, ..., 0.0, 0.0]
    strides: [1, 400, 120000]
  path_length:
    topology: "image_topo"
    association: "element"
    units: "path length metadata"
    values: [0.0, 0.0, 0.0, ..., 0.0, 0.0]
    strides: [1, 400, 120000]
  intensities_spatial:
    topology: "spatial_topo"
    association: "element"
    units: "intensity units"
    values: [0.0, 0.0, 0.0, ..., 0.0, 0.0]
    strides: [1, 400, 120000]
  path_length_spatial:
    topology: "spatial_topo"
    association: "element"
    units: "path length metadata"
    values: [0.0, 0.0, 0.0, ..., 0.0, 0.0]
    strides: [1, 400, 120000]
  ... ( skipped 6 children )
  far_plane_field:
    topology: "far_plane_topo"
    association: "element"
    volume_dependent: "false"
    values: 0.0
  ray_corners_field:
    topology: "ray_corners_topo"
    association: "element"
    volume_dependent: "false"
    values: [0.0, 0.0, 0.0, 0.0]
  ray_field:
    topology: "ray_topo"
    association: "element"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, ..., 119998.0, 119999.0]
```

The next thing we may want to do is to visualize an x ray image using VisIt. The *Visualizing with VisIt* section goes

into more detail on this subject, so for now we will only visualize the *Basic Mesh Output*.

```
# Have VisIt open the Conduit output from the query
OpenDatabase("output.root")

# Give ourselves a clean slate for ensuing visualizations
DeleteAllPlots()

# Add a pseudocolor plot of the intensities
AddPlot("Pseudocolor", "mesh_image_topo/intensities")
DrawPlots()

# Change the color table to be xray
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.colorTableName = "xray"
SetPlotOptions(PseudocolorAtts)
```

Running this code yields the following image:

Conduit Output

The Conduit output types (see *Output Types* for more information) provide advantages over the other output types and include additional metadata and topologies. These output types were added in VisIt 3.3.0, and many of the features discussed here have been added since then.

Why Conduit Output?

Conduit Blueprint output types were added to the X Ray Image Query primarily to facilitate usability and convenience. Before Conduit Blueprint formats were available as output types, the X Ray Image Query would often produce large numbers of output files, particularly when using the BOV or rawfloats output type, which was a popular choice because it provided the raw data. For example, ray tracing 60 energy groups would generate 120 BOV files (one for intensities and one for path lengths for each energy group). These files lack important context and metadata about the details of the ray trace setup. The large number of these files and the inability to control where they were saved led to lots of external data management issues that our users were unfortunately saddled with. Alternatively, users could choose one of the image file output types to generate a picture or pictures. But, without additional post-processing, it was impossible to have both, unless the query was run twice.

We added Conduit Blueprint output as an option to address these problems. Instead of many files coming out of the query, only one comes out. This single output file presents the query output in multiple ways using well-described meshes. We also provide three supported Conduit output types: HDF5, YAML, and JSON. The latter two are useful as they are human-readable options. Instead of having to choose between getting out raw data or getting out an image, Conduit Blueprint provides the best of both worlds. All of the meshes provided in the output can be easily plotted in VisIt (see *Visualizing with VisIt*), and everything in the output (raw intensities and path lengths data, mesh data, metadata, etc.) can be digested in Python using Conduit's Python API (see *Introspecting with Python*).

We have opted to enrich the Blueprint output (see *Basic Mesh Output*) with extensive metadata (see *Metadata*) as well as additional meshes (see *Imaging Planes and Rays Meshes*, *Spatial Extents Meshes*, and *1D Spectra Curves*) to provide extra context and information to the user. These additions should make it easier to troubleshoot unexpected results, make sense of the query output, and pass important information through the query. Blueprint makes it simple to put all of this information into one file, and just as simple to read that information back out and/or visualize.

One of the main reasons for adding the Conduit output was to make it far easier to troubleshoot strange query results. See the *Troubleshooting* section to see a few examples of the kinds of questions the Conduit output can be used to answer.

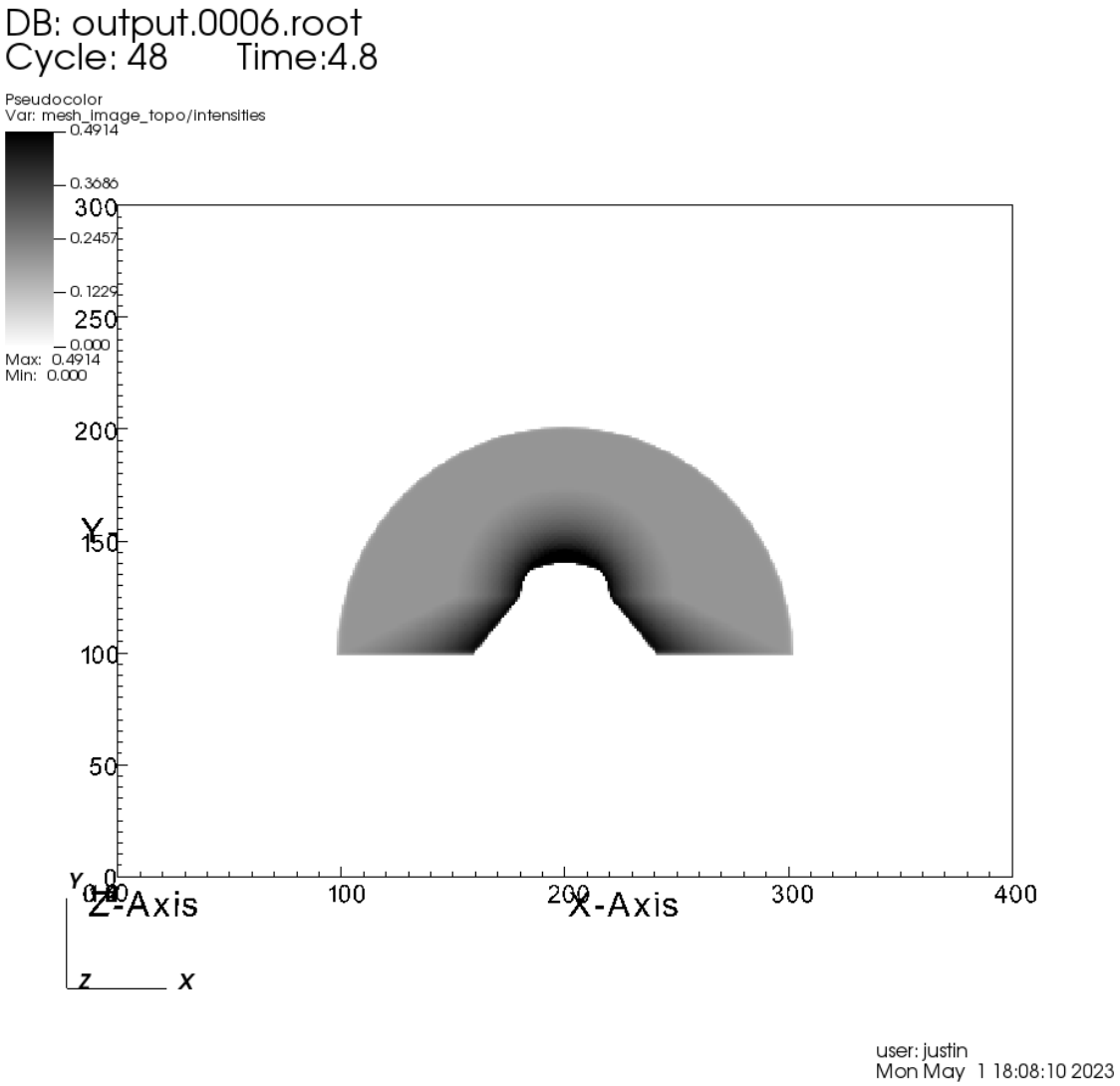


Fig. 4.261: The resulting x ray image, visualized using VisIt.

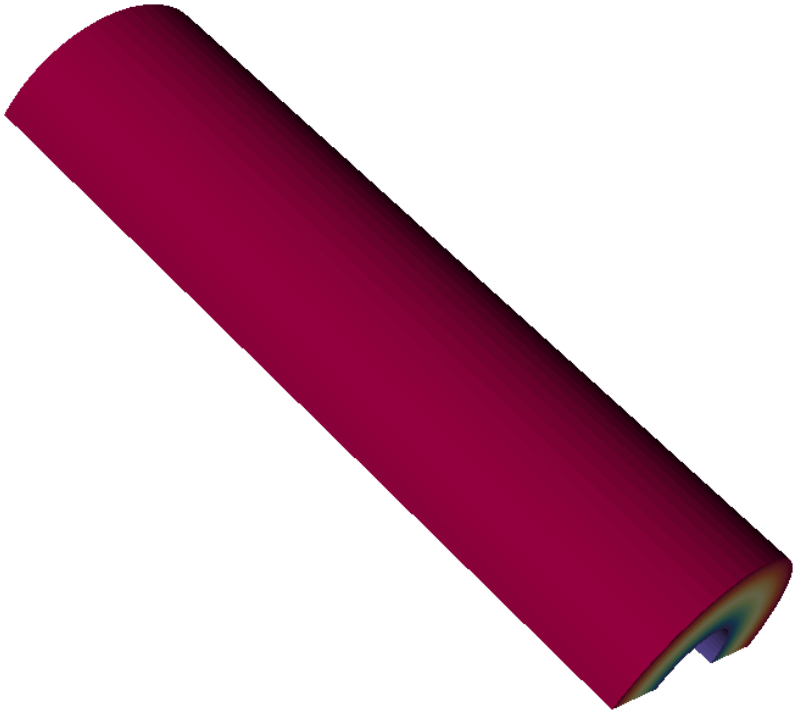


Fig. 4.262: An input mesh.

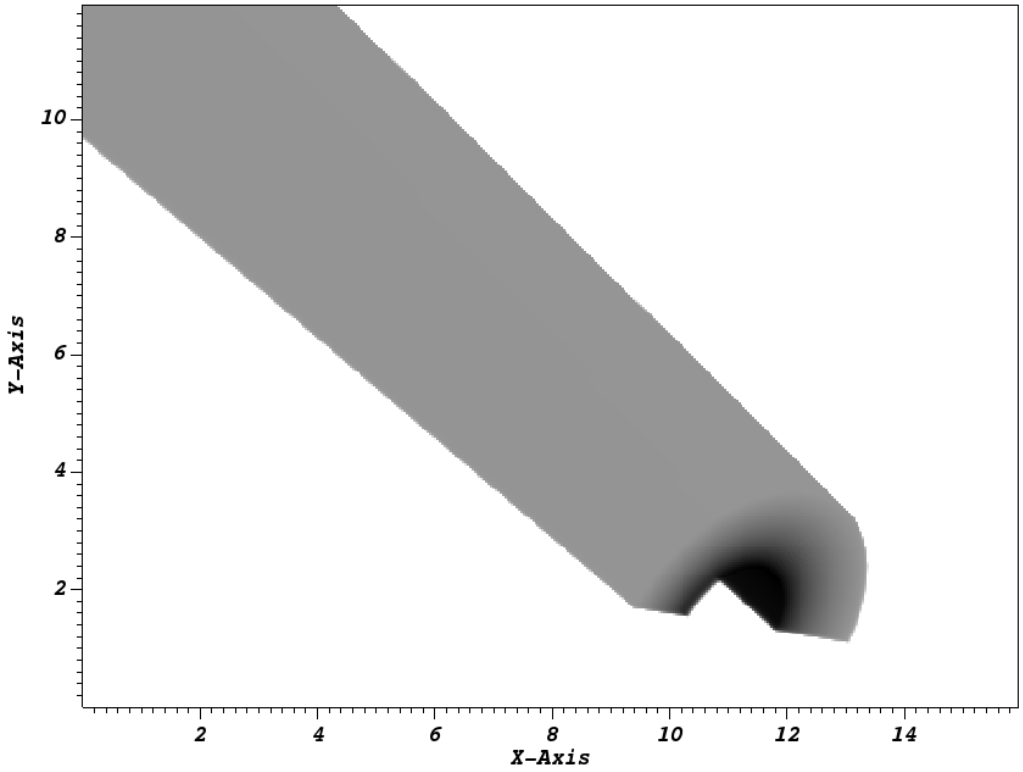


Fig. 4.263: The resulting x ray image from Conduit Blueprint output, visualized by plotting with VisIt.

Overview of Output

So what is actually in the Blueprint output? The Blueprint output provides multiple Blueprint meshes, which are each in turn comprised of a coordinate set, a topology, and fields. These all live within a Conduit tree, along with metadata. Using Conduit allows us to package everything in one place for ease of use.

To extract this data with Python, see *Getting a General Overview of the Output*. Here is a simplified representation of a Conduit tree that is output from the Query:

```
state:
  time: 4.8
  cycle: 48
  xray_view:
    ...
  xray_query:
    ...
  xray_data:
    ...
  domain_id: 0
coordsets:
  image_coords:
    ...
  spatial_coords:
    ...
  spatial_energy_reduced_coords:
    ...
  spectra_coords:
    ...
  near_plane_coords:
    ...
  view_plane_coords:
    ...
  far_plane_coords:
    ...
  ray_corners_coords:
    ...
  ray_coords:
    ...
topologies:
  image_topo:
    ...
  spatial_topo:
    ...
  spatial_energy_reduced_topo:
    ...
  spectra_topo:
    ...
  near_plane_topo:
    ...
  view_plane_topo:
    ...
  far_plane_topo:
    ...
  ray_corners_topo:
    ...
  ray_topo:
    ...
fields:
```

(continues on next page)

(continued from previous page)

```
  intensities:
    ...
  path_length:
    ...
  intensities_spatial:
    ...
  path_length_spatial:
    ...
  intensities_spatial_energy_reduced:
    ...
  path_length_spatial_energy_reduced:
    ...
  intensities_spectra:
    ...
  path_length_spectra:
    ...
  near_plane_field:
    ...
  view_plane_field:
    ...
  far_plane_field:
    ...
  ray_corners_field:
    ...
  ray_field:
    ...
```

There are multiple Blueprint meshes stored in this tree, as well as extensive metadata. Each piece of the Conduit output will be covered in more detail in ensuing parts of the documentation. To learn more about what lives under the `state` branch, see the *Metadata* section. To learn more about the coordinate sets, topologies, and fields, see the *Basic Mesh Output*, *Imaging Planes and Rays Meshes*, *Spatial Extents Meshes*, and *1D Spectra Curves* sections. To extract this data with Python, see *Getting a General Overview of the Output*.

Basic Mesh Output

The most important piece of the Blueprint output is the actual ray trace result. We have taken the image data that comes out of the query and packaged it into a single Blueprint mesh.

To visualize this mesh with VisIt, see *Visualizing the Basic Mesh Output*. To extract this mesh data with Python, see *Accessing the Basic Mesh Output Data*. The following is the example from *Overview of Output*, but with the Blueprint mesh representing the query result fully realized:

```
state:
  time: 4.8
  cycle: 48
  xray_view:
    ...
  xray_query:
    ...
  xray_data:
    ...
  domain_id: 0
coordsets:
  image_coords:
    type: "rectilinear"
```

(continues on next page)

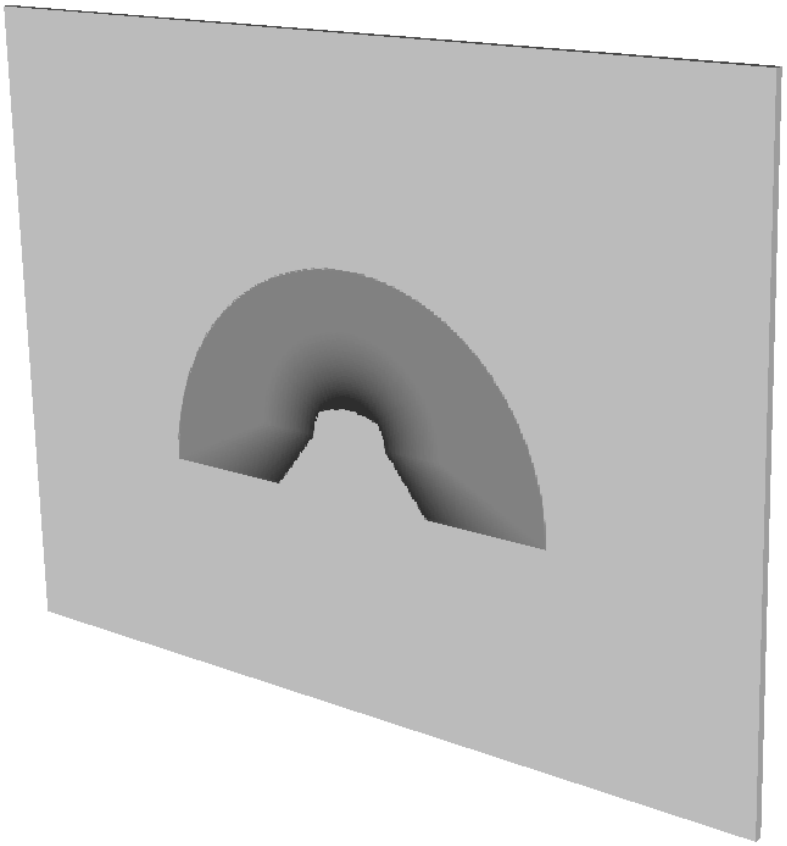


Fig. 4.264: The basic mesh output visualized using VisIt.

(continued from previous page)

```
    values:
      x: [0, 1, 2, ..., 399, 400]
      y: [0, 1, 2, ..., 299, 300]
      z: [0, 1]
    labels:
      x: "width"
      y: "height"
      z: "energy_group"
    units:
      x: "pixels"
      y: "pixels"
      z: "bins"
  spatial_coords:
    ...
  spatial_energy_reduced_coords:
    ...
  spectra_coords:
    ...
  near_plane_coords:
    ...
  view_plane_coords:
    ...
  far_plane_coords:
    ...
  ray_corners_coords:
    ...
  ray_coords:
    ...
topologies:
  image_topo:
    coordset: "image_coords"
    type: "rectilinear"
  spatial_topo:
    ...
  spatial_energy_reduced_topo:
    ...
  spectra_topo:
    ...
  near_plane_topo:
    ...
  view_plane_topo:
    ...
  far_plane_topo:
    ...
  ray_corners_topo:
    ...
  ray_topo:
    ...
fields:
  intensities:
    topology: "image_topo"
    association: "element"
    units: "intensity units"
    values: [0.281004697084427, 0.281836241483688, 0.282898783683777, ..., 0.0, 0.0]
    strides: [1, 400, 120000]
  path_length:
    topology: "image_topo"
```

(continues on next page)

(continued from previous page)

```
    association: "element"
    units: "path length metadata"
    values: [2.46405696868896, 2.45119333267212, 2.43822622299194, ..., 0.0, 0.0]
    strides: [1, 400, 120000]
  intensities_spatial:
    ...
  path_length_spatial:
    ...
  intensities_spatial_energy_reduced:
    ...
  path_length_spatial_energy_reduced:
    ...
  intensities_spectra:
    ...
  path_length_spectra:
    ...
  near_plane_field:
    ...
  view_plane_field:
    ...
  far_plane_field:
    ...
  ray_corners_field:
    ...
  ray_field:
    ...
```

The 3 constituent parts of the Blueprint mesh output are the coordinate set, `image_coords`, the topology, `image_topo`, and the fields, `intensities` and `path_length`.

The `image_coords` represent the x and y coordinates of the 2D image, and the z dimension represents the energy group bounds. In the case of multiple energy groups, previously, the query would have output multiple images, one for each pair of energy group bounds. In the Blueprint output, this is simplified; rather than outputting multiple files, each containing one image, we have opted to "stack" the resulting images on top of one another. This is why the Blueprint output is a 3D mesh; this way, it can account for multiple energy groups, and place resulting images one on top of another. Also included in the `image_coords` are labels and units for disambiguation purposes.

The `image_topo` exists to tell Blueprint that the rectilinear coordinate data stored in `image_coords` can be interpreted as a rectilinear grid.

The fields, `intensities` and `path_length`, can be thought of as containers for the actual image data. This image data is stored in 1-dimensional arrays; for information on reshaping those into 3-dimensions see the following from the *Troubleshooting* section: *The fields in the Conduit Output are 1D. How can I reshape them to be 3D?* Each field also includes units. For path length, the `units` entry is just a way of including metadata or information about the path length, since path length is unitless.

To visualize this mesh with VisIt, see *Visualizing the Basic Mesh Output*. To extract this mesh data with Python, see *Accessing the Basic Mesh Output Data*.

Metadata

The Conduit output types (see *Output Types* for more information) come packaged with metadata in addition to Blueprint-conforming mesh data. The ability to send this metadata alongside the output mesh (and other data) is one of the advantages of using Conduit for outputs from the query. We hope this metadata helps to make it clear exactly what the query is doing, what information it has available to it, and what the output might look like. To extract the metadata from the Blueprint output, see *Accessing the Metadata*.

Metadata is stored under the `state` Node in the resulting Conduit tree. See the example below, which is taken from the example in *Overview of Output*, but this time with only the metadata fully realized:

```
state:
  time: 4.8
  cycle: 48
  xray_view:
    normal:
      x: 0.0
      y: 0.0
      z: 1.0
    focus:
      x: 0.0
      y: 2.5
      z: 10.0
    view_up:
      x: 0.0
      y: 1.0
      z: 0.0
    view_angle: 30.0
    parallel_scale: 5.0
    view_width: 6.667
    non_square_pixels: "no"
    near_plane: -50.0
    far_plane: 50.0
    image_pan:
      x: 0.0
      y: 0.0
    image_zoom: 1.0
    perspective: 1
    perspective_str: "perspective"
  xray_query:
    divide_emis_by_absorb: 0
    divide_emis_by_absorb_str: "no"
    num_x_pixels: 400
    num_y_pixels: 300
    num_bins: 1
    abs_var_name: "d"
    emis_var_name: "p"
    abs_units: "cm^2/g"
    emis_units: "GJ/cm^2/ster/ns/keV"
  xray_data:
    detector_width: 22.3932263237838
    detector_height: 16.7949192423103
    intensity_max: 0.491446971893311
    intensity_min: 0.0
    path_length_max: 120.815788269043
    path_length_min: 0.0
    image_topo_order_of_domain_variables: "xyz"
  domain_id: 0
coordsets:
  ...
topologies:
  ...
fields:
  ...
```

There are three top-level items: `time`, `cycle`, and `domain_id`. The fact that the `domain_id` is present is a side effect of Conduit; all of the output data is single domain and this value has nothing to do with the query. In addition to

the top level items, there are three categories of metadata: *View Parameters*, *Query Parameters*, and *Other Metadata*. The following subsections discuss each of these categories in more detail.

View Parameters

View parameters can be found under "state/xray_view". This metadata represents the view-related values that were used in the x ray image query calculations. Remember from the section on *Camera Specification* options that if the *Simplified Camera Specification* is used, the parameters are converted to the *Complete Camera Specification* during execution. Hence the values output here correspond to those in the *Complete Camera Specification*, as these are the values that were actually used by the query when calculating results. The following is included:

normal	The x, y, and z components represent the view normal vector that was used in the calculations.
focus	The x, y, and z components represent the focal point that was used in the calculations.
view_up	The x, y, and z components represent the up vector that was used in the calculations.
view_angle	The view angle, only used in the calculations if perspective projection was enabled.
parallel_scale	The parallel scale, or view height, that was used in the calculations.
view_width	The view width that was used in the calculations.
non_square_pixels	"yes" means that the output is using non-square pixels, meaning that a view width was specified in the camera setup.
near_plane	The near plane that was used in the calculations.
far_plane	The far plane that was used in the calculations.
image_pan	The x and y components represent the image pan that was used in the calculations.
image_zoom	The absolute image zoom factor that was used in the calculations.
perspective	A flag indicating if parallel or perspective projection was used. 0 indicates parallel projection and 1 indicates perspective projection.
perspective_str	A String representation of the perspective parameter. See above for more information.

An example:

```
xray_view:
  normal:
    x: 0.0
    y: 0.0
    z: 1.0
  focus:
    x: 0.0
    y: 2.5
    z: 10.0
  view_up:
    x: 0.0
    y: 1.0
    z: 0.0
  view_angle: 30.0
  parallel_scale: 5.0
  view_width: 5.0
  non_square_pixels: "no"
  near_plane: -50.0
  far_plane: 50.0
  image_pan:
    x: 0.0
    y: 0.0
```

(continues on next page)

(continued from previous page)

```
  image_zoom: 1.0
  perspective: 1
  perspective_str: "perspective"
```

To extract this metadata from the Blueprint output, see *Accessing the Metadata*.

Query Parameters

Query parameters can be found under "state/xray_query". This metadata represents the query-related values that were used in the x ray image query calculations. This data is available as of VisIt 3.3.2. The following is included:

divide_emis_by_absorb	A flag indicating if emissivity was divided by absorbtivity in the calculations. More details can be found above.
divide_emis_by_absorb_str	A String representation of the divide_emis_by_absorb parameter. See above for more information.
num_x_pixels	The pixel extent in the X dimension in the output image.
num_y_pixels	The pixel extent in the Y dimension in the output image.
num_bins	The number of bins (the Z dimension extent) in the output image.
abs_var_name	The name of the absorbtivity variable that was used in the calculations.
emis_var_name	The name of the emissivity variable that was used in the calculations.
abs_units	The units of the absorbtivity variable that was used in the calculations.
emis_units	The units of the emissivity variable that was used in the calculations.

An example:

```
xray_query:
  divide_emis_by_absorb: 0
  divide_emis_by_absorb_str: "no"
  num_x_pixels: 400
  num_y_pixels: 300
  num_bins: 1
  abs_var_name: "d"
  emis_var_name: "p"
  abs_units: "cm^2/g"
  emis_units: "GJ/cm^2/ster/ns/keV"
```

To extract this metadata from the Blueprint output, see *Accessing the Metadata*.

Other Metadata

Other metadata can be found under "state/xray_data". These values are calculated constants based on the input parameters and output data. This data is available as of VisIt 3.3.2. The following is included:

detector_width	The width of the simulated x ray detector in physical space.
detector_height	The height of the simulated x ray detector in physical space.
intensity_max	The maximum value of the calculated intensities.
intensity_min	The minimum value of the calculated intensities.
path_length_max	The maximum value of the calculated path lengths.
path_length_min	The minimum value of the calculated path lengths.
image_topo_order_of_domain_variables	The intensities and path length field data can be indexed as 3D arrays, even though they are stored in flat 1D arrays. The 3D striding calculation can be fully determined using the shape of the coordinate set the fields are associated with and an optional field-specific stride array. The default case fast varies the first coordinate (x), then the second (y), and finally the third (z). The optional field-specific stride info enables arbitrary striding patterns. We provide striding info for these fields, however the X Ray Image Query always writes data using the `xyz` (fast to slow) default strides. `image_topo_order_of_domain_variables` provides this information as a string, hardcoded to be "xyz", that reflects this.

An example:

```
xray_data:
  detector_width: 22.3932263237838
  detector_height: 16.7949192423103
  intensity_max: 0.491446971893311
  intensity_min: 0.0
  path_length_max: 120.815788269043
  path_length_min: 0.0
  image_topo_order_of_domain_variables: "xyz"
```

The minimum and maximum values that are included for the path length and intensity outputs are useful for quick *Troubleshooting* or sanity checks that the output matches expectations. If both maximums and minimums are zero, for example, the simulated detector may not be facing the right way. In that case, the *Imaging Planes and Rays Meshes* section may be of some use.

To extract this metadata from the Blueprint output, see *Accessing the Metadata*.

Imaging Planes and Rays Meshes

One of our goals with the Conduit output types (see *Output Types* for more information) is to provide rich, easy to understand information about the query to facilitate usability. To that end, these outputs come packaged with meshes representing the imaging planes specified by the user when calling the query. Additionally, they also include meshes representing the rays that were used in the ray tracing. The following subsections discuss both of these in more detail.

To visualize these meshes with VisIt, see *Visualizing the Imaging Planes* and *Visualizing the Rays Meshes*.

Imaging Planes

Users can visualize the near, view, and far planes in physical space alongside the meshes used in the ray trace:

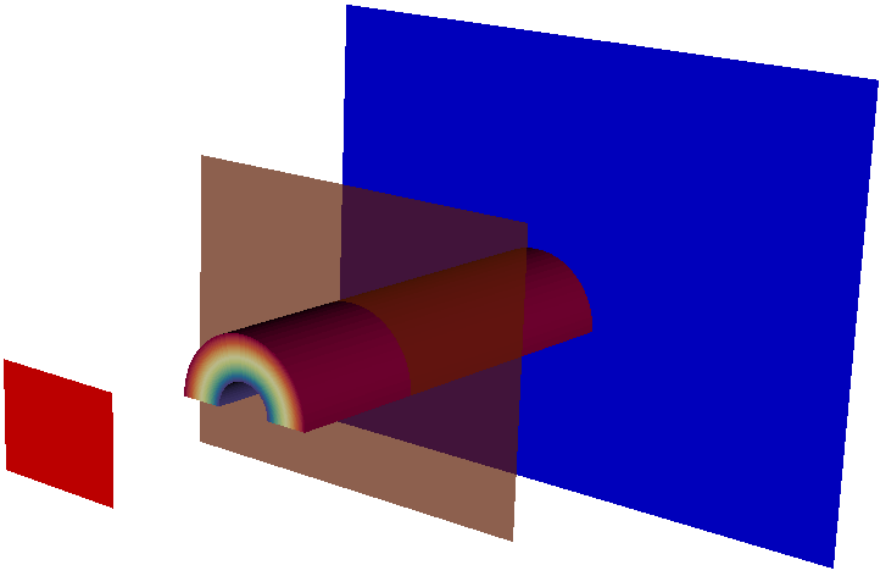


Fig. 4.265: The imaging planes used by the X Ray Image Query visualized on top of the simulation data. The near plane is in red, the view plane in transparent orange, and the far plane in blue.

Including this in the output gives a sense of where the camera is looking, and is also useful for checking if parts of the mesh being ray traced are outside the near and far clipping planes.

To visualize these meshes with VisIt, see *Visualizing the Imaging Planes*. To extract this mesh data with Python, see *Accessing Everything Else*. See the example below, which is taken from the example in *Overview of Output*, but this time with only the imaging plane meshes fully realized:

```
state:
  time: 4.8
  cycle: 48
  xray_view:
    ...
  xray_query:
    ...
  xray_data:
    ...
  domain_id: 0
coordsets:
  image_coords:
    ...
  spatial_coords:
    ...
  spatial_energy_reduced_coords:
    ...
  spectra_coords:
    ...
  near_plane_coords:
    type: "explicit"
    values:
      x: [-11.1966131618919, 11.1966131618919, 11.1966131618919, -11.1966131618919]
      y: [10.8974596211551, 10.8974596211551, -5.89745962115514, -5.89745962115514]
      z: [-40.0, -40.0, -40.0, -40.0]
  view_plane_coords:
    type: "explicit"
    values:
      x: [6.66666686534882, -6.66666686534882, -6.66666686534882, 6.66666686534882]
      y: [-2.5, -2.5, 7.5, 7.5]
      z: [10.0, 10.0, 10.0, 10.0]
  far_plane_coords:
    type: "explicit"
    values:
      x: [24.5299468925895, -24.5299468925895, -24.5299468925895, 24.5299468925895]
      y: [-15.8974596211551, -15.8974596211551, 20.8974596211551, 20.8974596211551]
      z: [60.0, 60.0, 60.0, 60.0]
  ray_corners_coords:
    ...
  ray_coords:
    ...
topologies:
  image_topo:
    ...
  spatial_topo:
    ...
  spatial_energy_reduced_topo:
    ...
  spectra_topo:
    ...
  near_plane_topo:
```

(continues on next page)

(continued from previous page)

```
    type: "unstructured"
    coordset: "near_plane_coords"
    elements:
      shape: "quad"
      connectivity: [0, 1, 2, 3]
  view_plane_topo:
    type: "unstructured"
    coordset: "view_plane_coords"
    elements:
      shape: "quad"
      connectivity: [0, 1, 2, 3]
  far_plane_topo:
    type: "unstructured"
    coordset: "far_plane_coords"
    elements:
      shape: "quad"
      connectivity: [0, 1, 2, 3]
  ray_corners_topo:
    ...
  ray_topo:
    ...
fields:
  intensities:
    ...
  path_length:
    ...
  intensities_spatial:
    ...
  path_length_spatial:
    ...
  intensities_spatial_energy_reduced:
    ...
  path_length_spatial_energy_reduced:
    ...
  intensities_spectra:
    ...
  path_length_spectra:
    ...
  near_plane_field:
    topology: "near_plane_topo"
    association: "element"
    volume_dependent: "false"
    values: 0.0
  view_plane_field:
    topology: "view_plane_topo"
    association: "element"
    volume_dependent: "false"
    values: 0.0
  far_plane_field:
    topology: "far_plane_topo"
    association: "element"
    volume_dependent: "false"
    values: 0.0
  ray_corners_field:
    ...
  ray_field:
    ...
```

Just like the *Basic Mesh Output*, each of the three meshes has three constituent pieces. For the sake of brevity, we will only discuss the view plane, but the following information also holds true for the near and far planes. First off is the `view_plane_coords` coordinate set, which, as may be expected, contains only four points, representing the four corners of the rectangle. Next is the `view_plane_topo`, which tells Conduit to treat the four points in the `view_plane_coords` as a quad. Finally, we have the `view_plane_field`, which has one value, "0.0". This value doesn't mean anything; it is just used to tell Blueprint that the entire quad should be colored the same color.

To visualize these meshes with VisIt, see *Visualizing the Imaging Planes*. To extract this mesh data with Python, see *Accessing Everything Else*.

Rays Meshes

Having the imaging planes is helpful, but sometimes it can be more useful to have a sense of the view frustum itself. Users may desire a clearer picture of the simulated x ray detector: where is it in space, exactly what is it looking at, and what is it not seeing? Enter the rays meshes, or the meshes that contain the rays used to generate the output images/data.

Why are there two? The first is the ray corners mesh. This is a Blueprint mesh containing four lines that pass through the corners of the *Imaging Planes*. Now the viewing frustum is visible:

The ray corners mesh is useful because no matter the chosen dimensions of the output image, the ray corners mesh always will only contain 4 lines. Therefore it is cheap to render in a tool like VisIt, and it gives a general sense of what is going on. But for those who wish to see all of the rays used in the ray trace, the following will be useful.

The second rays mesh provided is the ray mesh, which provides all the rays used in the ray trace, represented as lines in Blueprint. A note of caution: depending on how many rays are used in the ray trace, this mesh could be expensive to render, hence the inclusion of the ray corners mesh.

Depending on the chosen dimensions of the output image, this mesh can contain thousands of lines. See the following image, which is the same query as the previous image, but this time with 400x300 pixels.

This render is far less useful. Even the imaging planes have been swallowed up, and the input mesh is completely hidden. There are a couple quick solutions to this problem. **The first solution** is to temporarily run the query with less rays (i.e. lower the image dimensions) until the desired understanding of what the simulated x ray detector is looking at has been achieved, then switch back to the large number of pixels/rays. This can be done quickly, as the ray trace is the performance bottleneck for the x ray image query. Here are examples:

These renders are less overwhelming, they can be generated quickly, and they get across a good amount of information. But there is another option that does not require losing information.

The second solution is adjusting the opacity of the rays using VisIt. Here is a view of a different run of the query, this time with the simulated x ray detector to the side of the input mesh.

Even with only 40x30 rays, it is already hard to see the input mesh underneath the rays. With VisIt, it is very easy to adjust the opacity of the rays and make them semitransparent. Here is the same view but with the opacity adjusted for greater visibility.

Here is the same view but with 400x300 rays.

And here is the same view with 400x300 rays but with the ray opacity lowered.

Hopefully it is clear at this point that there are multiple ways of looking at the rays that are used in the ray trace.

To extract this mesh data with Python, see *Accessing Everything Else*. To visualize these meshes with VisIt, see *Visualizing the Rays Meshes*. Now we will take a look at another example inspired by the example in *Overview of Output*, but this time with only the rays meshes fully realized:

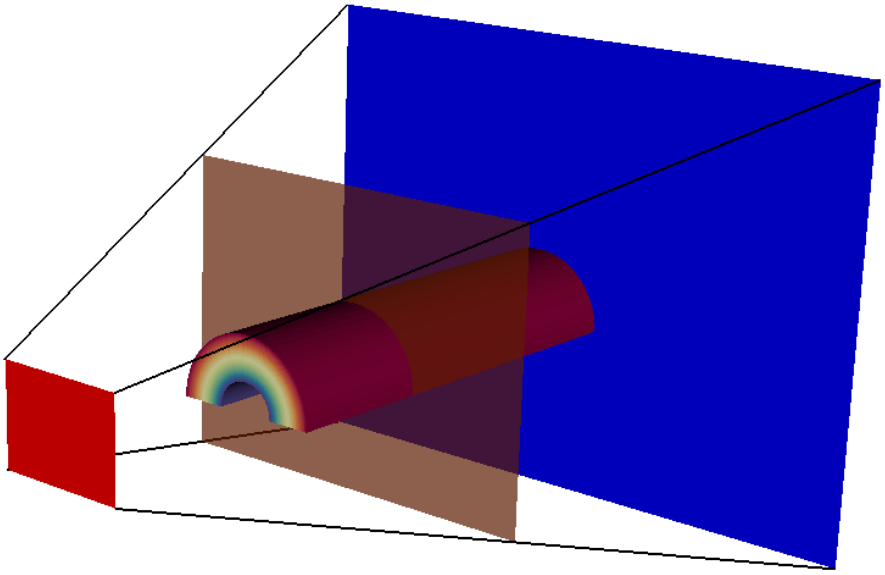


Fig. 4.266: A plot of 5 meshes: the actual mesh that the query used to generate results, the 3 imaging planes, and the ray corners mesh.

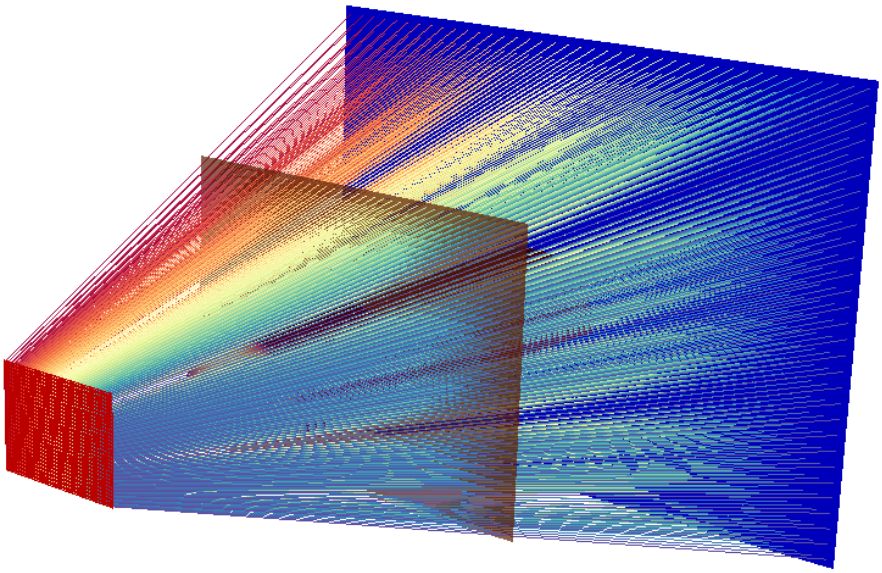


Fig. 4.267: There are 40x30 rays in this image, corresponding to an x ray image output of 40x30 pixels.

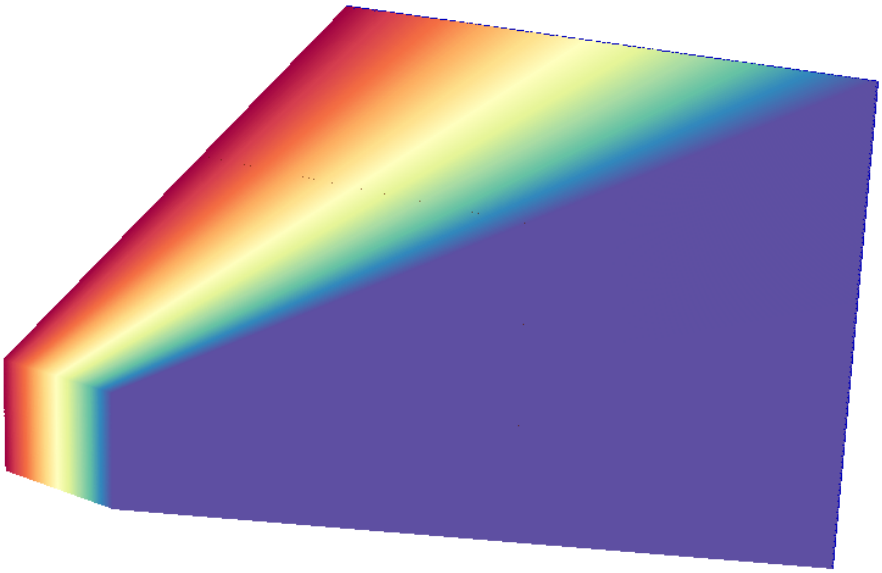


Fig. 4.268: There are 400x300 rays in this image, corresponding to an x ray image output of 400x300 pixels.

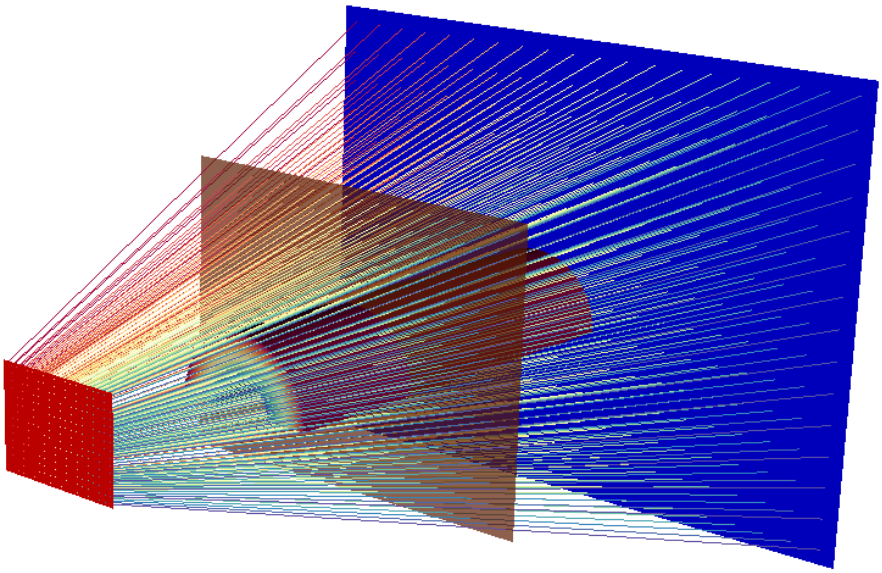


Fig. 4.269: There are 20x15 rays in this image, corresponding to an x ray image output of 20x15 pixels.

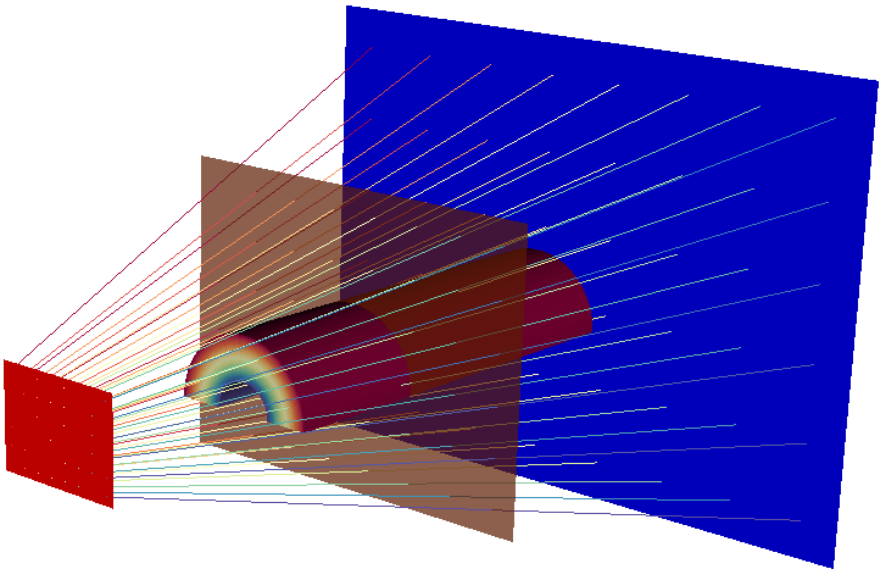


Fig. 4.270: There are 8x6 rays in this image, corresponding to an x ray image output of 8x6 pixels.

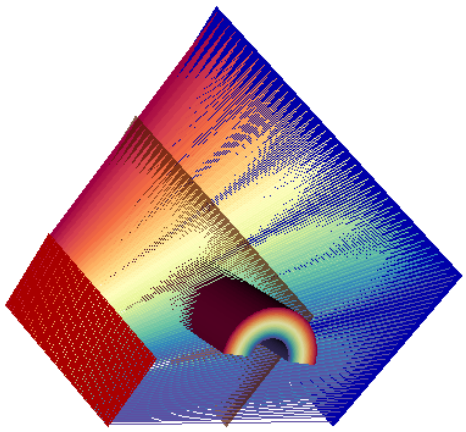


Fig. 4.271: There are 40x30 rays in this image, corresponding to an x ray image output of 40x30 pixels. This is a view of a different run of the query from the images shown thus far.

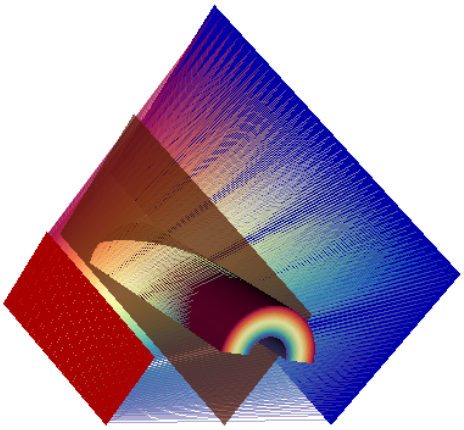


Fig. 4.272: The 40x30 rays have had their opacity lowered for greater visibility.

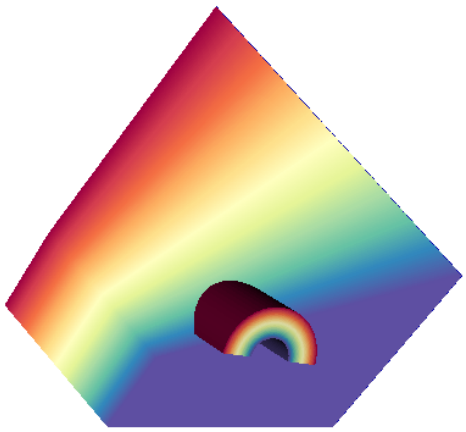


Fig. 4.273: There are 400x300 rays in this image, corresponding to an x ray image output of 40x30 pixels. The rays totally obscure the geometry.

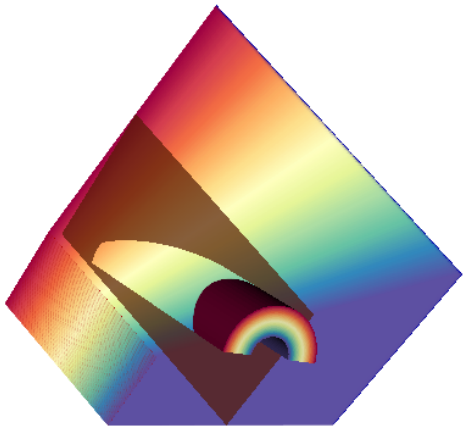


Fig. 4.274: The 400x300 rays have had their opacity lowered for greater visibility.

```
state:
  time: 4.8
  cycle: 48
  xray_view:
    ...
  xray_query:
    ...
  xray_data:
    ...
  domain_id: 0
coordsets:
  image_coords:
    ...
  spatial_coords:
    ...
  spatial_energy_reduced_coords:
    ...
  spectra_coords:
    ...
  near_plane_coords:
    ...
  view_plane_coords:
    ...
  far_plane_coords:
    ...
  ray_corners_coords:
    type: "explicit"
    values:
      x: [-11.1966131618919, 24.5299468925895, 11.1966131618919, ..., -11.
↪1966131618919, 24.5299468925895]
      y: [10.8974596211551, -15.8974596211551, 10.8974596211551, ..., -5.
↪89745962115514, 20.8974596211551]
      z: [-40.0, 60.0, -40.0, ..., -40.0, 60.0]
  ray_coords:
    type: "explicit"
    values:
      x: [11.1686216289872, 11.1686216289872, 11.1686216289872, ..., 24.4686220253581,
↪ 24.4686220253581]
      y: [10.8694680890846, 10.8134850249436, 10.7575019608025, ..., 20.7134850249436,
↪ 20.8361347557513]
      z: [-40.0, -40.0, -40.0, ..., 60.0, 60.0]
topologies:
  image_topo:
    ...
  spatial_topo:
    ...
  spatial_energy_reduced_topo:
    ...
  spectra_topo:
    ...
  near_plane_topo:
    ...
  view_plane_topo:
    ...
  far_plane_topo:
    ...
  ray_corners_topo:
```

(continues on next page)

(continued from previous page)

```
    type: "unstructured"
    coordset: "ray_corners_coords"
    elements:
      shape: "line"
      connectivity: [0, 1, 2, ..., 6, 7]
  ray_topo:
    type: "unstructured"
    coordset: "ray_coords"
    elements:
      shape: "line"
      connectivity: [0, 120000, 1, ..., 119999, 239999]
fields:
  intensities:
    ...
  path_length:
    ...
  intensities_spatial:
    ...
  path_length_spatial:
    ...
  intensities_spatial_energy_reduced:
    ...
  path_length_spatial_energy_reduced:
    ...
  intensities_spectra:
    ...
  path_length_spectra:
    ...
  near_plane_field:
    ...
  view_plane_field:
    ...
  far_plane_field:
    ...
  ray_corners_field:
    topology: "ray_corners_topo"
    association: "element"
    volume_dependent: "false"
    values: [0.0, 0.0, 0.0, 0.0]
  ray_field:
    topology: "ray_topo"
    association: "element"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, ..., 119998.0, 119999.0]
```

The Blueprint mesh setup may be familiar by now after reading the other sections, particularly the *Basic Mesh Output* section, so we will only mention here that for each ray mesh, there are the usual three components, a coordinate set, a topology, and a field. The topology tells Blueprint that the shapes in question are lines, which is how we represent the rays.

The final topic of note in this section ties in to the following questions: Why are the rays all different colors? What do the colors mean? The answer is that the colors mean nothing, and the color choices are entirely arbitrary. These colors come from the field values under `fields/ray_field`, which run from 0 to n, where n is the number of rays. We found that if all the rays were the same color, the resulting render was much harder to visually parse. Of course, rendering the rays as one color is still an option. With VisIt, one need only draw a Mesh Plot of the `mesh_ray_topo` as opposed to a Pseudocolor Plot of the `mesh_ray_topo/ray_field`.

To extract this mesh data with Python, see *Accessing Everything Else*. To visualize these meshes with VisIt, see *Visualizing the Rays Meshes*.

Spatial Extents Meshes

The spatial extents mesh and the spatial energy reduced mesh are two additional pieces that we include with the Conduit Output.

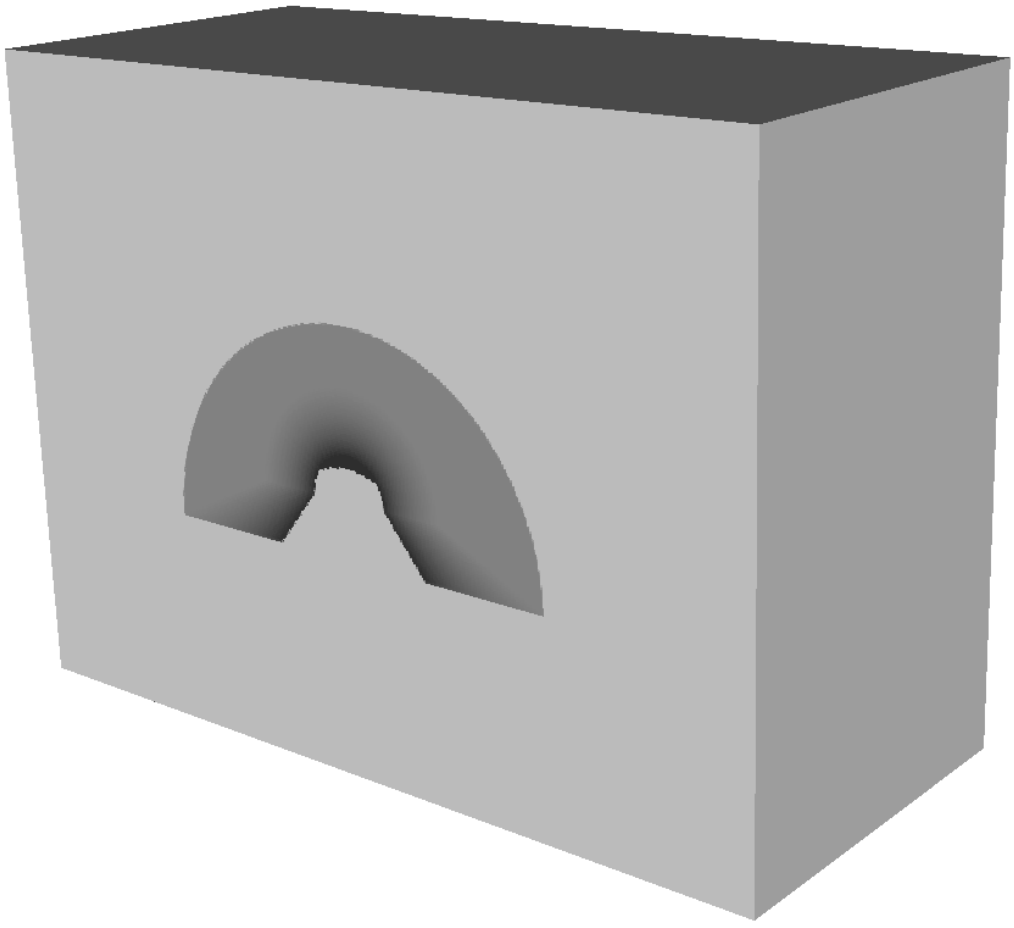


Fig. 4.275: The Spatial Extents Mesh visualized using VisIt.

The first of these two is the Spatial Extents Mesh, which bears great similarity to that of the *Basic Mesh Output*. The *Basic Mesh Output* gives users a picture, in a sense, that was taken by the simulated x ray detector. That picture lives in image space, where the x and y dimensions are given in pixels, and the z dimension represents the number of energy group bins.

The spatial extents mesh is the same picture that was taken by the simulated x ray detector, but living in physical

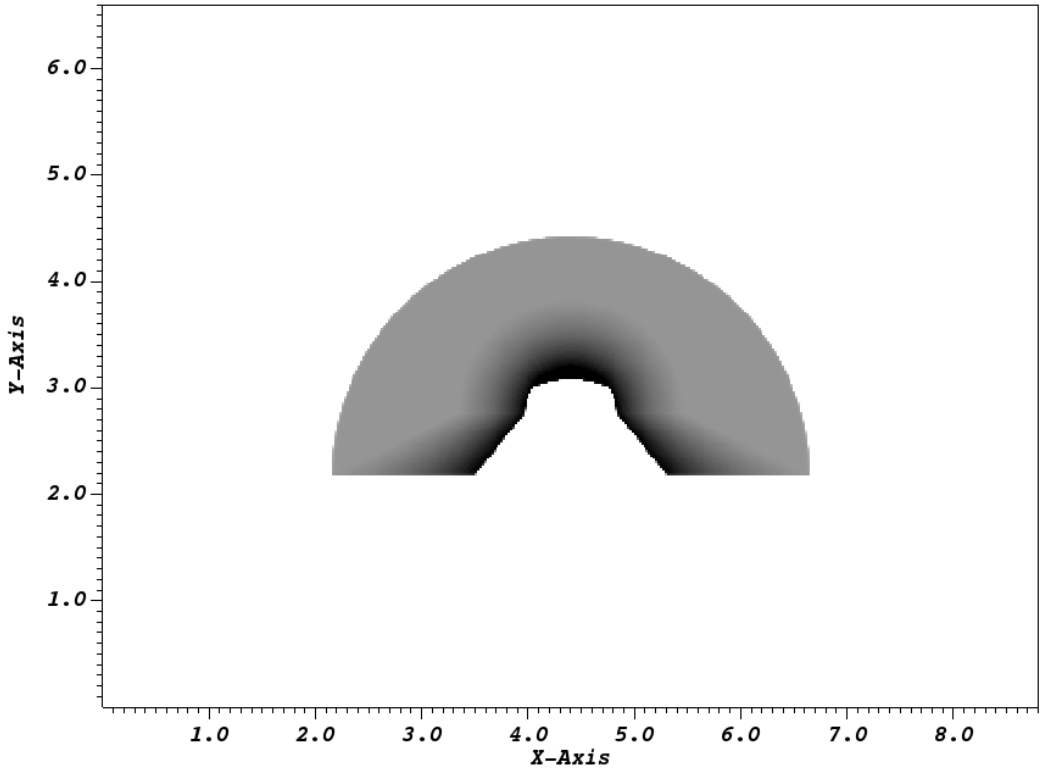


Fig. 4.276: The Spatial Energy Reduced Mesh visualized using VisIt.

space. Instead of the x and y dimensions representing pixels, the x and y dimensions here represent spatial values. In the example below, these dimensions are in centimeters. The x and y values run from 0 to the detector width and height values, respectively, that appear in the *Other Metadata* section of the Blueprint output. The z dimension represents actual energy group bins. These are values that were passed in via the query arguments (see *Standard Arguments* for more information). In the Blueprint example below, the z dimension represents Kiloelectron Volts.

Another way to think about the spatial extents mesh is if the basic mesh output was resized and then pasted on top of the near plane mesh (*Imaging Planes*), you would get the spatial extents mesh (ignoring the z dimension). The rationale for including this mesh is twofold:

1. It provides yet another view of the data. Perhaps seeing the output with spatial coordinates in x and y is more useful than seeing it with pixel coordinates. If parallel projection is used (*Complete Camera Specification*), the spatial view of the output is far more useful.
2. This mesh acts as a container for various interesting pieces of data that users may want to pass through the query. This is the destination for the `spatial_units` and `energy_units` (*Units*), which show up under `coordsets/spatial_coords/units`. This is also where the energy group bounds (*Standard Arguments*) appear in the output, under `coordsets/spatial_coords/values/z`.

If the energy group bounds were not provided by the user, or the provided bounds do not match the actual number of bins used in the ray trace, then there will be a message explaining what went wrong under `coordsets/spatial_coords/info`, and the z values will go from 0 to n where n is the number of bins.

The other mesh that is included, the Spatial Energy Reduced Mesh, is a simplification of the Spatial Extents Mesh. We collapse the information in the Spatial Extents Mesh into 2D by taking, for each x and y element (or pixel), the field value (either intensities or path lengths) to be the sum of the field values along the z axis scaled by the corresponding energy bin widths, if they are provided by the user.

To extract this mesh data with Python, see *Accessing the Spatial Extents Meshes Data*. To visualize these meshes with VisIt, see *Visualizing the Spatial Extents Meshes*. The following is the example from *Overview of Output*, but with only the spatial extents meshes fully realized:

```
state:
  time: 4.8
  cycle: 48
  xray_view:
    ...
  xray_query:
    ...
  xray_data:
    ...
  domain_id: 0
coordsets:
  image_coords:
    ...
  spatial_coords:
    type: "rectilinear"
    values:
      x: [-0.0, -0.0559830658094596, -0.111966131618919, ..., -22.3372432579744, -22.
↪3932263237838]
      y: [-0.0, -0.0559830641410342, -0.111966128282068, ..., -16.7389361781692, -16.
↪7949192423103]
      z: [3.7, 4.2]
    units:
      x: "cm"
      y: "cm"
      z: "kev"
    labels:
      x: "width"
```

(continues on next page)

(continued from previous page)

```
      y: "height"
      z: "energy_group"
  spatial_energy_reduced_coords:
    type: "rectilinear"
    values:
      x: [-0.0, -0.0559830658094596, -0.111966131618919, ..., -22.3372432579744, -22.
↪3932263237838]
      y: [-0.0, -0.0559830641410342, -0.111966128282068, ..., -16.7389361781692, -16.
↪7949192423103]
    units:
      x: "cm"
      y: "cm"
    labels:
      x: "width"
      y: "height"
  spectra_coords:
    ...
  near_plane_coords:
    ...
  view_plane_coords:
    ...
  far_plane_coords:
    ...
  ray_corners_coords:
    ...
  ray_coords:
    ...
topologies:
  image_topo:
    ...
  spatial_topo:
    coordset: "spatial_coords"
    type: "rectilinear"
  spatial_energy_reduced_topo:
    coordset: "spatial_energy_reduced_coords"
    type: "rectilinear"
  spectra_topo:
    ...
  near_plane_topo:
    ...
  view_plane_topo:
    ...
  far_plane_topo:
    ...
  ray_corners_topo:
    ...
  ray_topo:
    ...
fields:
  intensities:
    ...
  path_length:
    ...
  intensities_spatial:
    topology: "spatial_topo"
    association: "element"
    units: "intensity units"
```

(continues on next page)

(continued from previous page)

```
    values: [0.281004697084427, 0.281836241483688, 0.282898783683777, ..., 0.0, 0.0]
    strides: [1, 400, 120000]
  path_length_spatial:
    topology: "spatial_topo"
    association: "element"
    units: "path length metadata"
    values: [2.46405696868896, 2.45119333267212, 2.43822622299194, ..., 0.0, 0.0]
    strides: [1, 400, 120000]
  intensities_spatial_energy_reduced:
    topology: "spatial_energy_reduced_topo"
    association: "element"
    values: [0.70251174271, 0.7045906037, 0.7072469592, ..., 0.0, 0.0]
  path_length_spatial_energy_reduced:
    topology: "spatial_energy_reduced_topo"
    association: "element"
    values: [6.16014242172, 6.12798333168, 6.09556555748, ..., 0.0, 0.0]
  intensities_spectra:
    ...
  path_length_spectra:
    ...
  near_plane_field:
    ...
  view_plane_field:
    ...
  far_plane_field:
    ...
  ray_corners_field:
    ...
  ray_field:
    ...
```

As can be seen from the example, this view of the output is very similar to the *Basic Mesh Output*. It has all the same components, a coordinate set `spatial_coords`, a topology `spatial_topo`, and fields `intensities_spatial` and `path_length_spatial`. The topology and fields are exact duplicates of those found in the *Basic Mesh Output*. The Spatial Energy Reduced Mesh is similar, but notable in the sense that it is missing the z dimension.

The impetus for including the spatial extents mesh was originally to include spatial coordinates as part of the metadata, but later on it was decided that the spatial coordinates should be promoted to be a proper Blueprint coordset. We then duplicated the existing topology and fields from the *Basic Mesh Output* so that the spatial extents coordset could be part of a valid Blueprint mesh, and could thus be visualized using VisIt.

To extract this mesh data with Python, see *Accessing the Spatial Extents Meshes Data*. To visualize these meshes with VisIt, see *Visualizing the Spatial Extents Meshes*.

1D Spectra Curves

To provide yet another view of the intensities and path lengths data, we include two curves, represented as blueprint meshes.

Similar to the Spatial Energy Reduced Mesh (*Spatial Extents Meshes*), the provided mesh is a dimension collapse of the Spatial Extents Mesh. However, instead of collapsing the z dimension (energy group bounds) by taking a sum, we collapse the x and y dimensions (spatial extents). Thus we are left with a 1D curve, where for each energy group bin, there is one field value that is the result of summing the fields values (intensities or path lengths scaled by the spatial extents of each pixel) for each z-plane. There is one curve for the intensities and one curve for the path lengths.

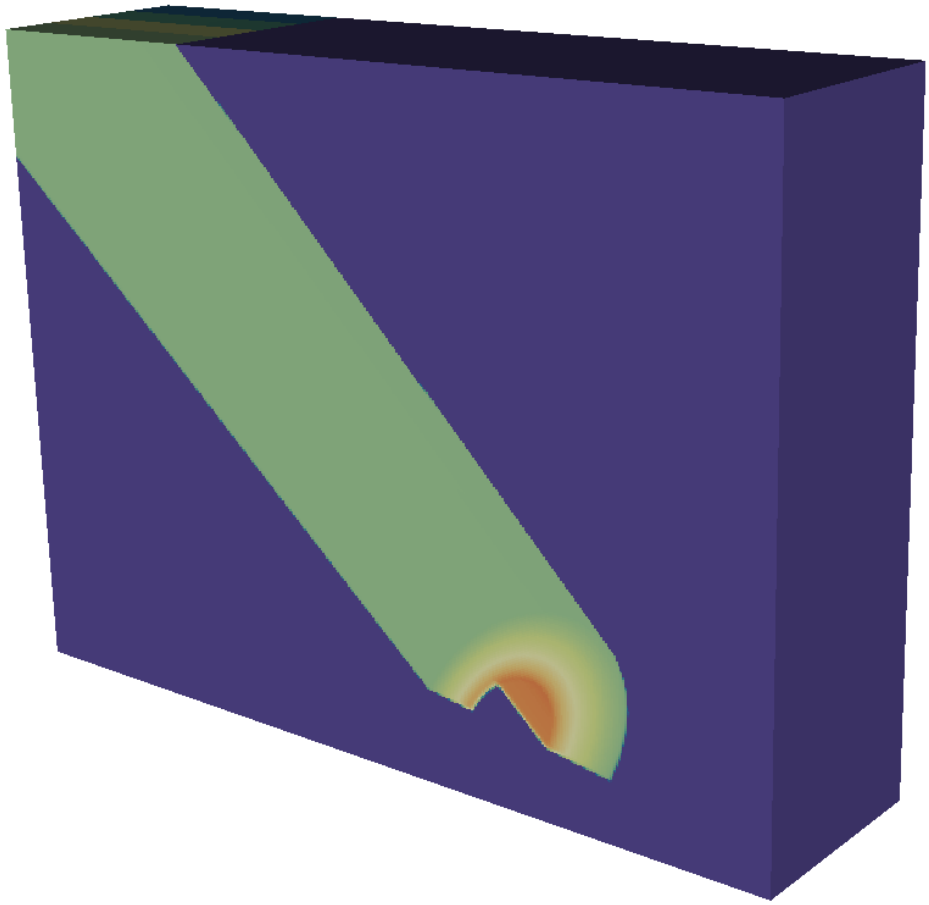


Fig. 4.277: The spatial extents mesh looks very similar to the basic mesh output. It is in 3D and the z dimension represents the energy group bounds, which in this example run from 0 to 12.

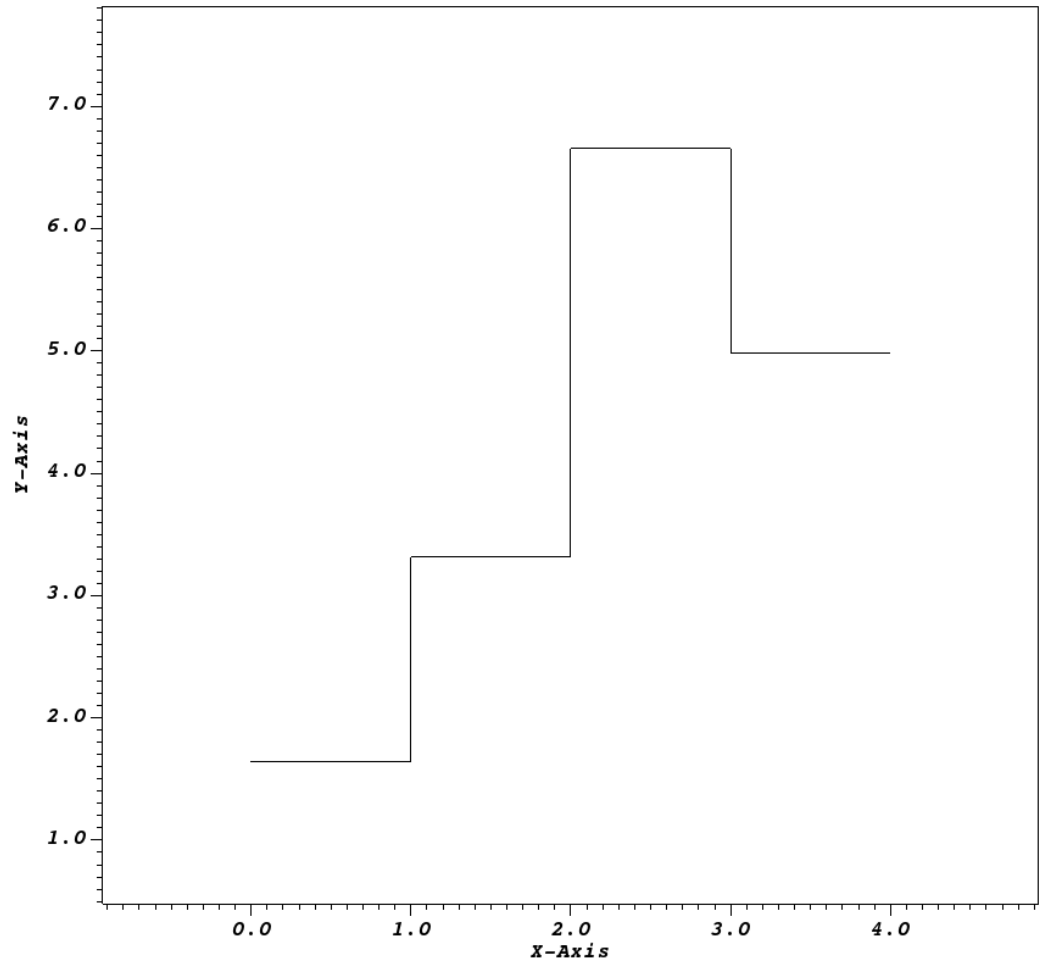


Fig. 4.278: One of the 1D Spectra Curves visualized using VisIt.

To extract this mesh data with Python, see *Accessing the 1D Spectra Curves Data*. To visualize this mesh with VisIt, see *Visualizing the 1D Spectra Curves*. The following is the example from *Overview of Output*, but with the Blueprint mesh representing the 1D Spectra Curves fully realized:

```
state:
  time: 4.8
  cycle: 48
  xray_view:
    ...
  xray_query:
    ...
  xray_data:
    ...
  domain_id: 0
coordsets:
  image_coords:
    ...
  spatial_coords:
    ...
  spatial_energy_reduced_coords:
    ...
  spectra_coords:
    type: "rectilinear"
    values:
      x: [0.0, 1.0, 2.0, 3.0, 4.0]
    units:
      x: "kev"
    labels:
      x: "energy_group"
  near_plane_coords:
    ...
  view_plane_coords:
    ...
  far_plane_coords:
    ...
  ray_corners_coords:
    ...
  ray_coords:
    ...
topologies:
  image_topo:
    ...
  spatial_topo:
    ...
  spatial_energy_reduced_topo:
    ...
  spectra_topo:
    coordset: "spectra_coords"
    type: "rectilinear"
  near_plane_topo:
    ...
  view_plane_topo:
    ...
  far_plane_topo:
    ...
  ray_corners_topo:
    ...
  ray_topo:
```

(continues on next page)

(continued from previous page)

```
    ...
fields:
  intensities:
    ...
  path_length:
    ...
  intensities_spatial:
    ...
  path_length_spatial:
    ...
  intensities_spatial_energy_reduced:
    ...
  path_length_spatial_energy_reduced:
    ...
  intensities_spectra:
    topology: "spectra_topo"
    association: "element"
    values: [1.64416097681804, 3.31540252150611, 6.65558651188286, 4.98593527287638]
  path_length_spectra:
    topology: "spectra_topo"
    association: "element"
    values: [356.40441526888, 712.808830537761, 1425.61766107552, 1069.21324547146]
  near_plane_field:
    ...
  view_plane_field:
    ...
  far_plane_field:
    ...
  ray_corners_field:
    ...
  ray_field:
    ...
```

Again, we have the typical 3 components of a Blueprint mesh. This is no different than the other Blueprint meshes, despite the fact that this will be represented differently under the hood in VisIt to make it appear as a curve when plotted.

To extract this mesh data with Python, see *Accessing the 1D Spectra Curves Data*. To visualize this mesh with VisIt, see *Visualizing the 1D Spectra Curves*.

Quick Results

One of the advantages of using *Conduit Output* is the ability to view quick results that give an overview of the output data. In this section, we will briefly discuss three of those quick results. Each of these have been discussed individually in other sections but not all together.

First is the **Spatial Energy Reduced Mesh** (discussed in greater detail here: *Spatial Extents Meshes*). This mesh is a 2-dimensional representation of the intensities and path lengths. We have collapsed the energy group bins to arrive at this result. The point of including this is to give a broad, at-a-glance view of the data. If, for example, many energy group bins contain uninteresting data, but a few have important structure, it can be difficult to get at that information using the *Basic Mesh Output* or the 3-dimensional Spatial Extents Mesh. Because the Spatial Energy Reduced Mesh fields are the result of taking the sum of the intensities and path lengths fields across all the energy group bins, that structure will be visible at a glance in a render of this mesh, as opposed to needing to be hunted for using VisIt's slicing and threshold tools, for example.

To render this, see *Visualizing the Spatial Extents Meshes*.

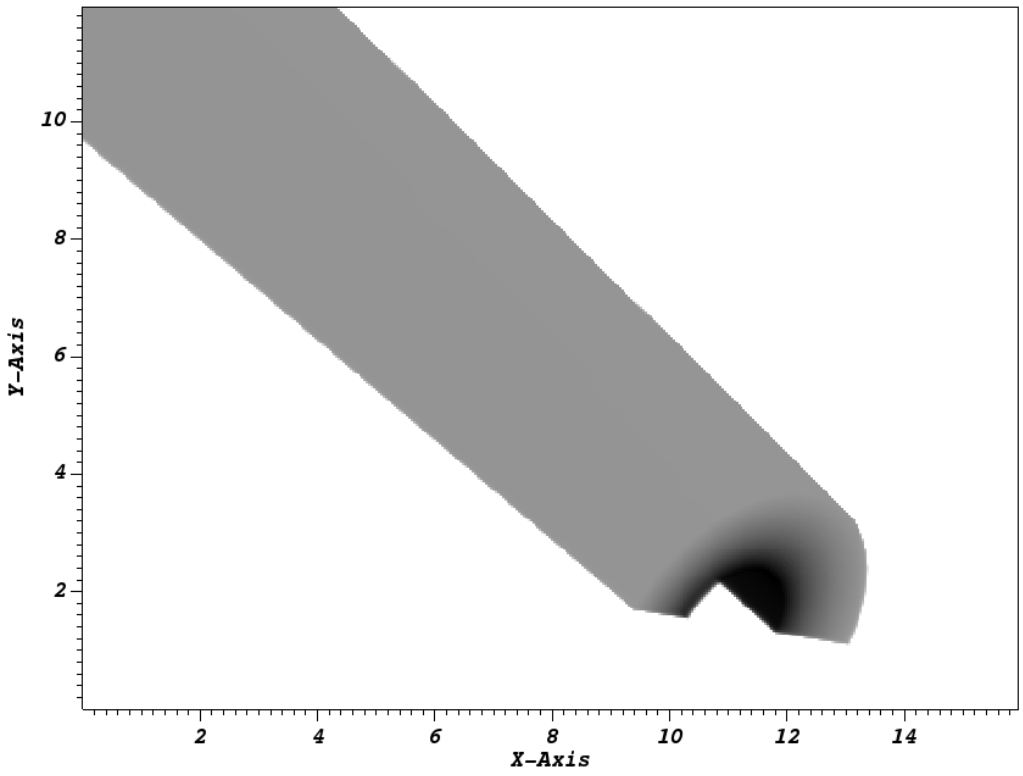


Fig. 4.279: A render of the spatial energy reduced mesh intensities, viewing our typical half cylinder example from the side.

Next up are the **Spectra Curves** (discussed in greater detail here: *1D Spectra Curves*). This mesh is a 1-dimensional representation of the intensities and path lengths. Instead of collapsing the energy group bins, we have collapsed the x and y spatial dimensions to arrive at this result. Thus we get a curve that associates energy levels with intensities or path lengths. It may be helpful to view this curve with a logarithmic scale. Now it is possible to see exactly how intensities or path length data varies across energy levels.

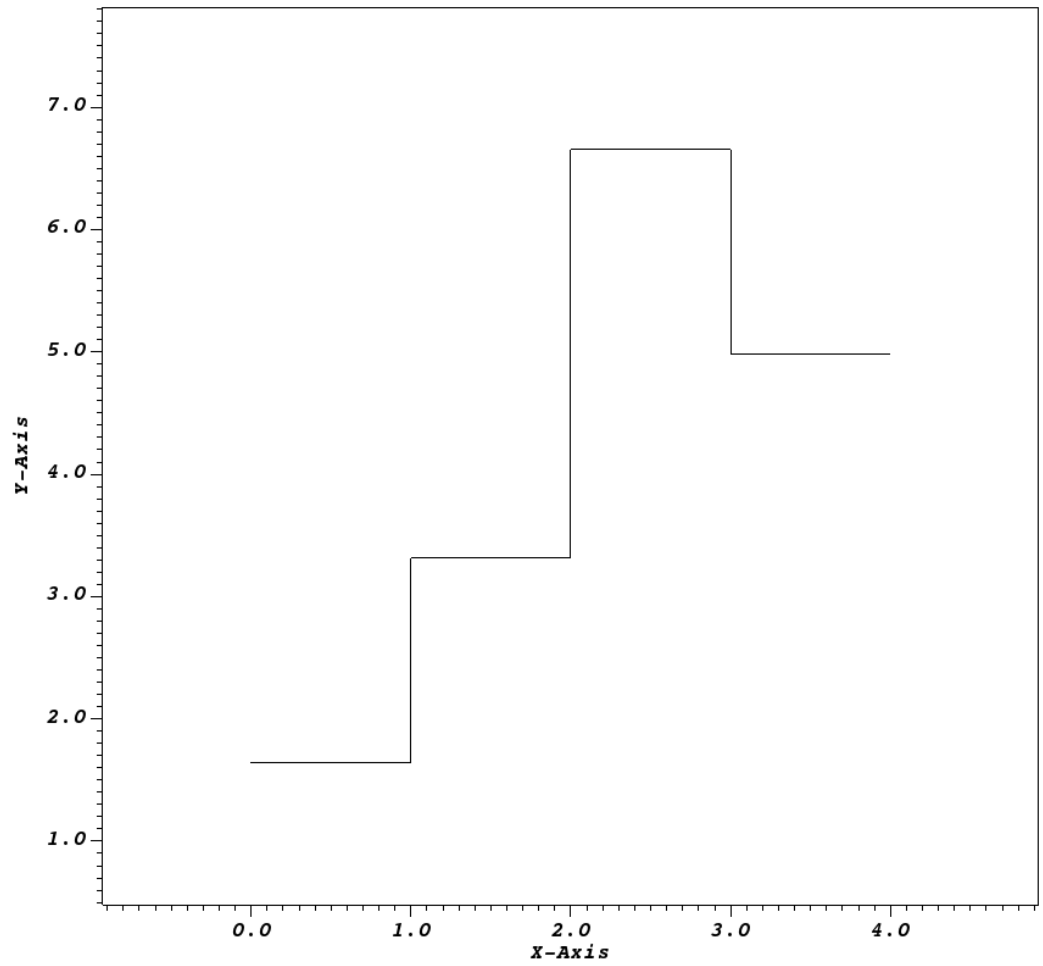


Fig. 4.280: A render of the intensities spectra curve. The X dimension represents energy and the Y dimension represents intensities.

To render this, see *Visualizing the 1D Spectra Curves*.

The final quick view of the data that we will cover in this section are the **intensities and path lengths maximums and minimums** included as part of the *Metadata*. For context, we have opted to calculate the maximum and minimum intensity and path length values and output that information under *Other Metadata*. These four values are not necessarily a "view" of the data, but they do give a shallow sense of what to expect. If all four are zero, for example, that means that all output images are blank. See *Troubleshooting* for more information about that case. Otherwise, these four values can yield a quick sanity check, as hopefully maximum and minimum intensity and path length values are within reason. See *Accessing the Metadata* for information on extracting these values from the Metadata.

Pitfalls

Despite all of these features being added to the X Ray Image Query to facilitate usability, there are still cases where confusion can arise. One such case is where the spatial extents mesh can appear to be upside down. Consider the following:

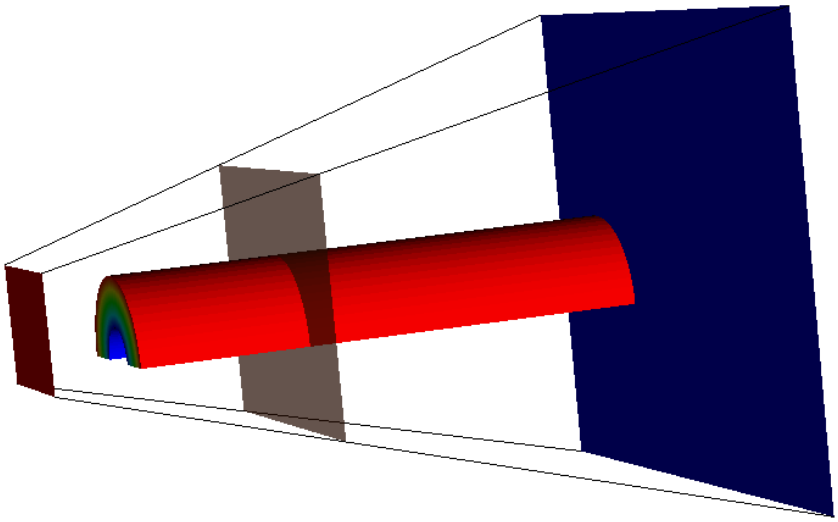


Fig. 4.281: An input mesh, imaging planes, and ray corners, viewed from the side.

If we adjust the query so that the near plane is further away (say maybe from -15 to -35), we will see this:

The near plane has passed out of the view frustum. This is because the view frustum is determined by the `view_angle` argument (see *Complete Camera Specification*). In this case, the query is using the default value of 30 degrees, and because the near plane is far enough back, it is outside the frustum.

So what does this mean for the other query results? It means that while we'd expect our Spatial Extents Mesh (*Spatial Extents Meshes*) to look like this:

It will actually look like this:

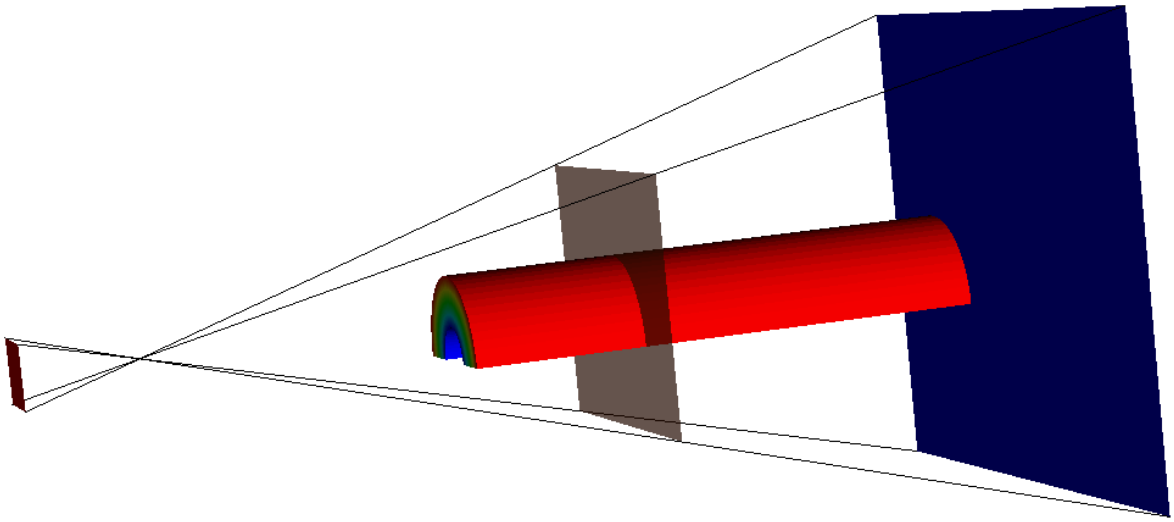


Fig. 4.282: The same set of plots as before, except this time the near plane has been moved back.

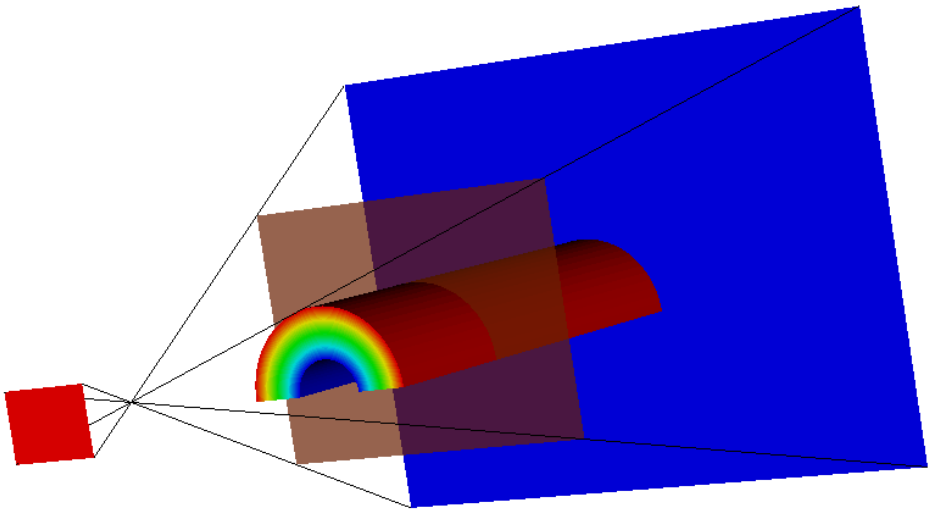


Fig. 4.283: Another view of this situation.

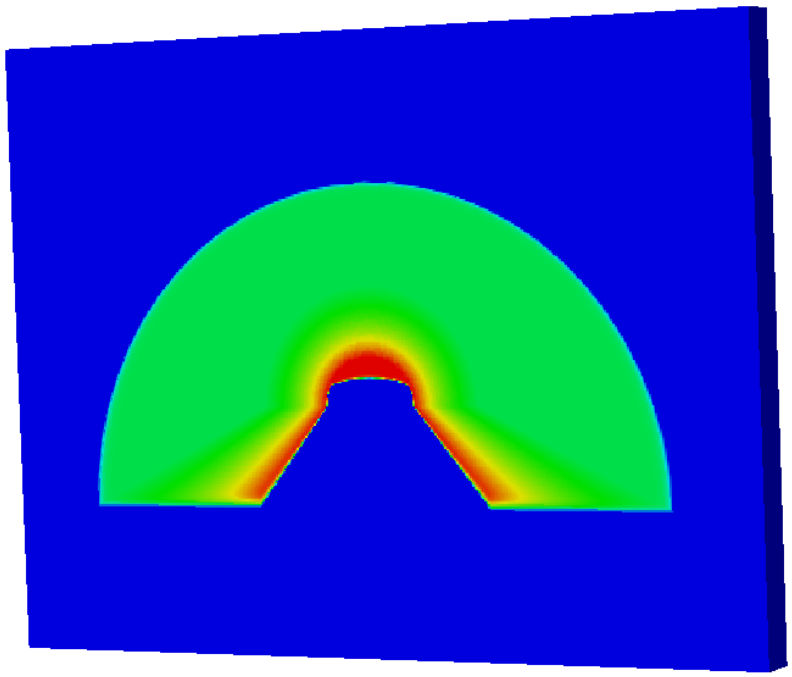


Fig. 4.284: The spatial extents mesh as we'd expect to see from running the query.

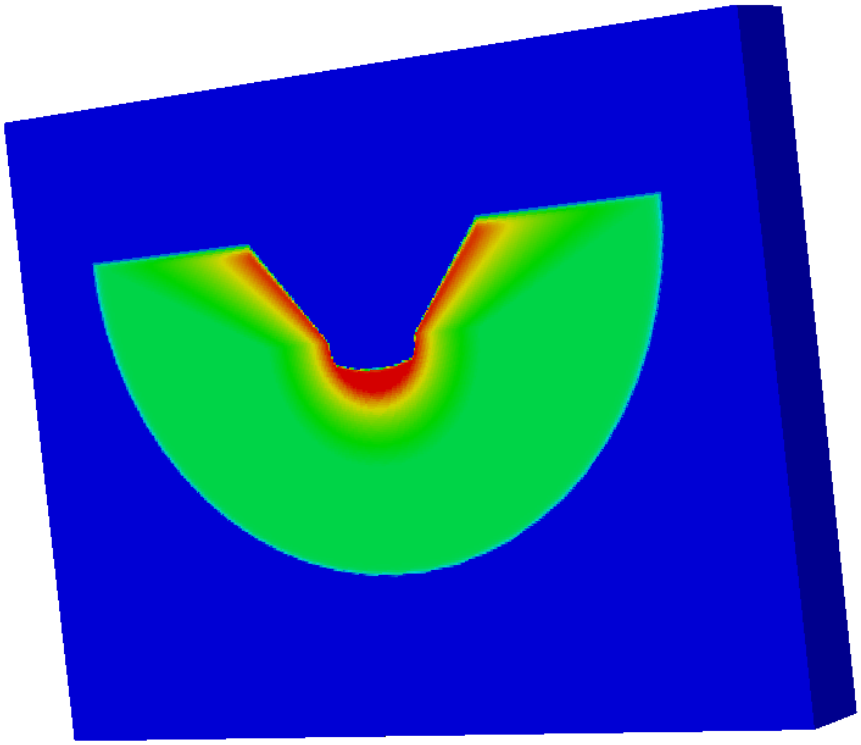


Fig. 4.285: The upside-down spatial extents mesh that we actually get from running the query.

Why is the mesh upside-down? The spatial extents mesh is upside-down because the simulated x ray detector is upside down. Previously, in the *Spatial Extents Meshes* section we described the spatial extents mesh as though we had taken the *Basic Mesh Output*, resized it, and pasted it on top of the near plane. That is exactly what is happening here. The spatial extents mesh is upside down because the near plane is upside down.

Here are the same images as above, but this time, in each one, the upper right corner of each imaging plane is marked in green:

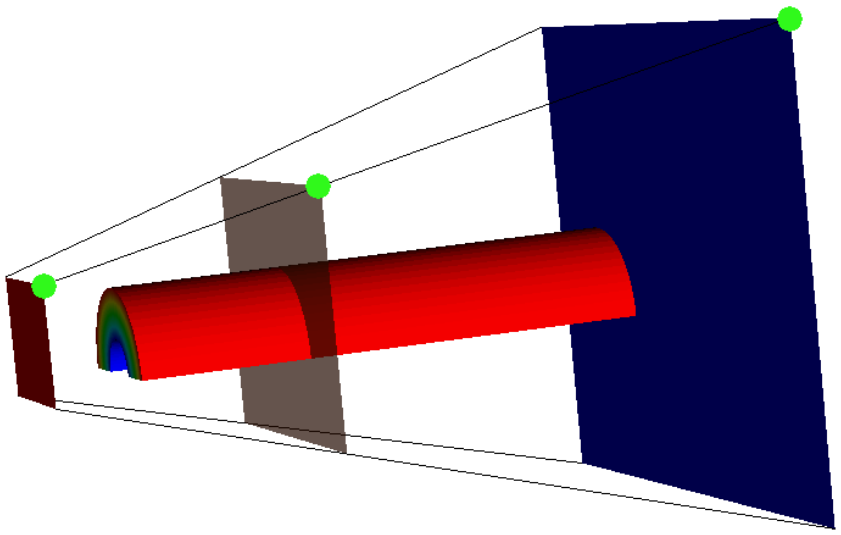


Fig. 4.286: An input mesh, imaging planes, and ray corners, viewed from the side. Note the upper right corner of each imaging plane is marked in green.

If we adjust the query so that the near plane is further away (say maybe from -15 to -35), we will see this:

Following the ray corners, we see that the upper right corner for the near plane is actually on the bottom left, because the whole near plane has been reflected to accommodate the fact that it is behind the frustum. This explains why the spatial extents mesh appears upside down; it is actually reflected across the x and y axes.

This special case will trigger a warning message if VisIt is run with `-debug 1`.

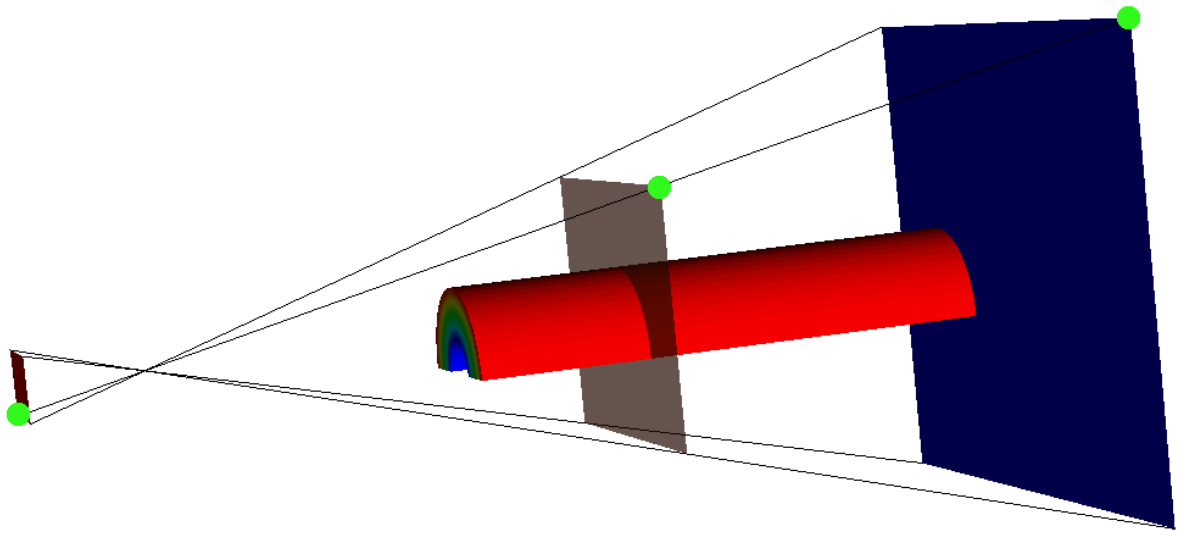


Fig. 4.287: The same set of plots as before, except this time the near plane has been moved back. Note the upper right corner of each imaging plane is marked in green. For the near plane (red), the upper right corner is not where we would expect.

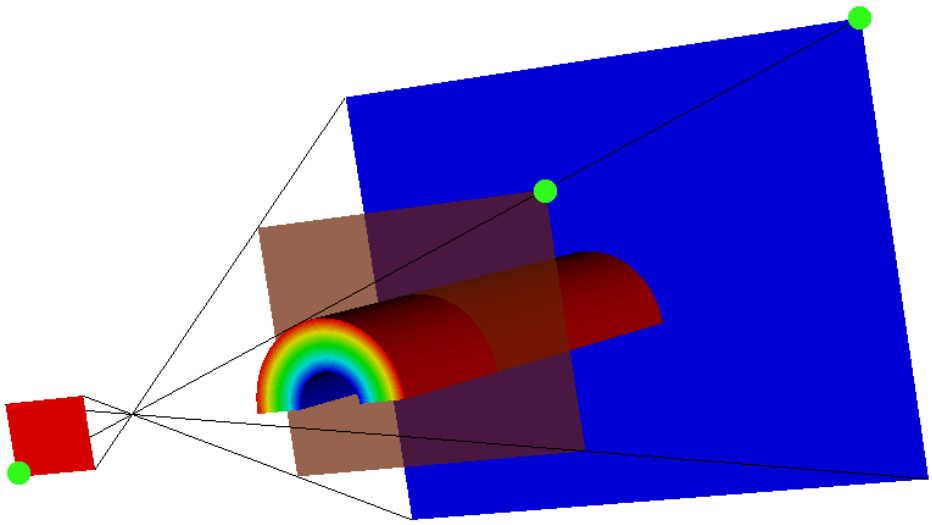


Fig. 4.288: Another view of this situation. Note the upper right corner of each imaging plane is marked in green. The upper right corner for the near plane (red) is on the bottom left because the near plane is reflected across the x and y axes.

Visualizing with VisIt

One of the advantages of using one of the *Conduit Output* types is that it is easy to both look at the raw data and generate x ray images. This section will cover generating x ray images using VisIt as well as visualizing the other components of the *Conduit Output*.

The later Python code examples assume that the following has already been run:

```
# The file containing the mesh I wish to ray trace
OpenDatabase("testdata/silo_hdf5_test_data/curv3d.silo")

# The query requires a plot to be visible
AddPlot("Pseudocolor", "d")
DrawPlots()

# Call the query
params = dict()
params["image_size"] = (400, 300)
# One of the Blueprint output types
params["output_type"] = "hdf5"
params["focus"] = (0., 2.5, 10.)
params["perspective"] = 1
params["near_plane"] = -25.
params["far_plane"] = 25.
params["vars"] = ("d", "p")
# Dummy values to demonstrate functionality
params["energy_group_bounds"] = [2.7, 6.2]
params["parallel_scale"] = 10.
Query("XRay Image", params)

# Open the file that was output from the query.
# In this case it is called "output.root"
OpenDatabase("output.root")
```

Once the query has been run, to visualize each constituent part of the output, follow these steps in Python:

Visualizing the Basic Mesh Output

First we will cover visualizing the *Basic Mesh Output*.

```
# Make sure we have a clean slate for ensuing visualizations.
DeleteAllPlots()

# Add a pseudocolor plot of the intensities
AddPlot("Pseudocolor", "mesh_image_topo/intensities")

# Alternatively add a plot of the path length instead
# AddPlot("Pseudocolor", "mesh_image_topo/path_length")

DrawPlots()
```

To make the output look like an x ray image, it is simple to change the color table.

```
# Make sure the plot you want to change the color of is active
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.colorTableName = "xray"
SetPlotOptions(PseudocolorAtts)
```

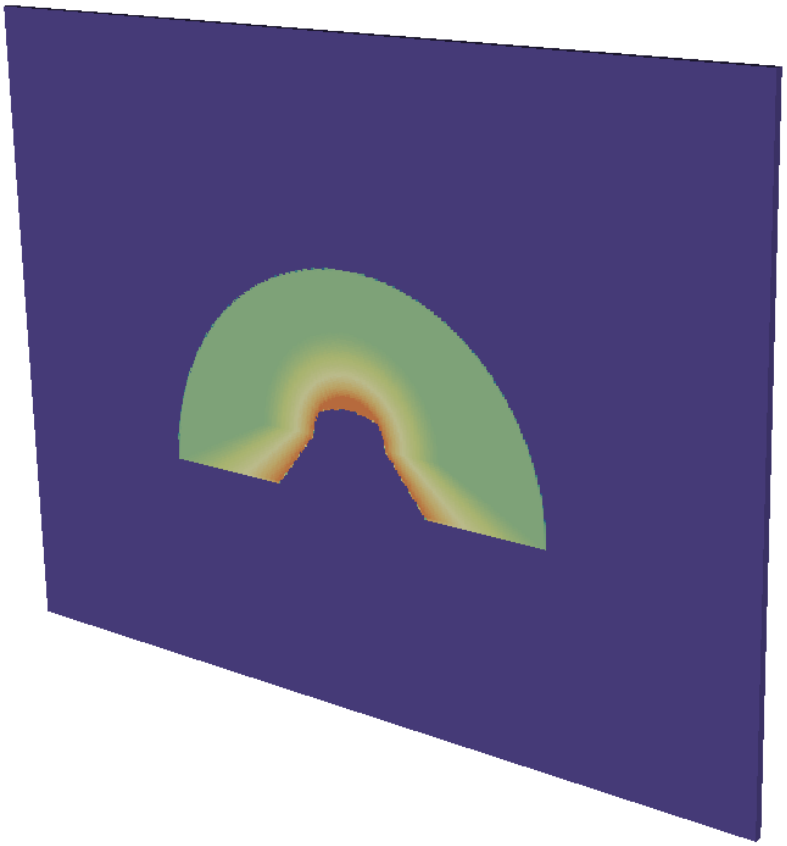


Fig. 4.289: A visualization of the basic mesh output.

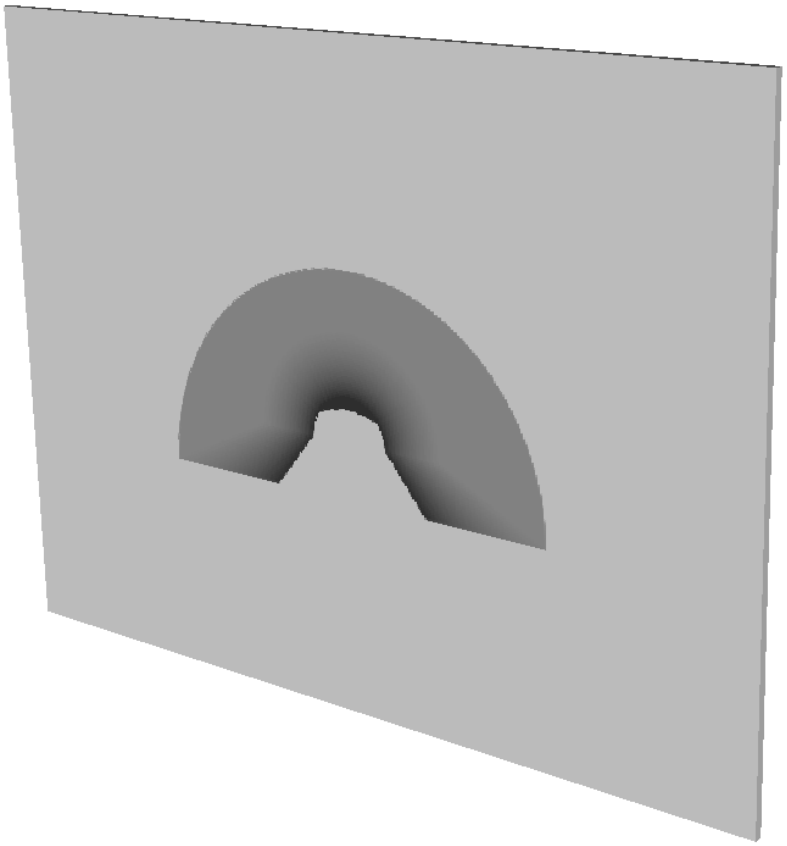


Fig. 4.290: A visualization of the basic mesh output using the x ray color table.

Visualizing the Imaging Planes

To simply render the *Imaging Planes* on top of your simulation data we will do the following:

```
# Make sure we have a clean slate for ensuing visualizations.
DeleteAllPlots()

# First we wish to make sure that the input mesh is visible
ActivateDatabase("testdata/silo_hdf5_test_data/curv3d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()

# Then we want to go back to the output file and visualize the imaging planes
ActivateDatabase("output.root")
AddPlot("Pseudocolor", "mesh_near_plane_topo/near_plane_field")
AddPlot("Pseudocolor", "mesh_view_plane_topo/view_plane_field")
AddPlot("Pseudocolor", "mesh_far_plane_topo/far_plane_field")
DrawPlots()
```

This will color the imaging planes all the same color. To make them distinct colors like in all the examples throughout this documentation, we can do the following:

```
# Make the plot of the near plane active
SetActivePlots(1)
PseudocolorAtts = PseudocolorAttributes()
# We invert the color table so that it is a different color from the far plane
PseudocolorAtts.invertColorTable = 1
SetPlotOptions(PseudocolorAtts)

# Make the plot of the view plane active
SetActivePlots(2)
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.colorTableName = "Oranges"
PseudocolorAtts.invertColorTable = 1
PseudocolorAtts.opacityType = PseudocolorAtts.Constant  # ColorTable, FullyOpaque,␣
↪Constant, Ramp, VariableRange
# We lower the opacity so that the view plane does not obstruct our view of anything.
PseudocolorAtts.opacity = 0.7
SetPlotOptions(PseudocolorAtts)

# leave the far plane as is
```

Visualizing the Rays Meshes

For the sake of visual clarity, as we visualize the *Rays Meshes*, we will build on the imaging planes visualization from above. To visualize the ray corners, it is a simple matter of doing the following:

```
# This plots the ray corners mesh
AddPlot("Mesh", "mesh_ray_corners_topo")

# Alternatively, we could plot the dummy field that is included, but
# plotting just the mesh will make sure the plot is in black, which
# looks better with the colors we are using to paint the imaging planes.
# AddPlot("Pseudocolor", "mesh_ray_corners_topo/ray_corners_field")
```

(continues on next page)

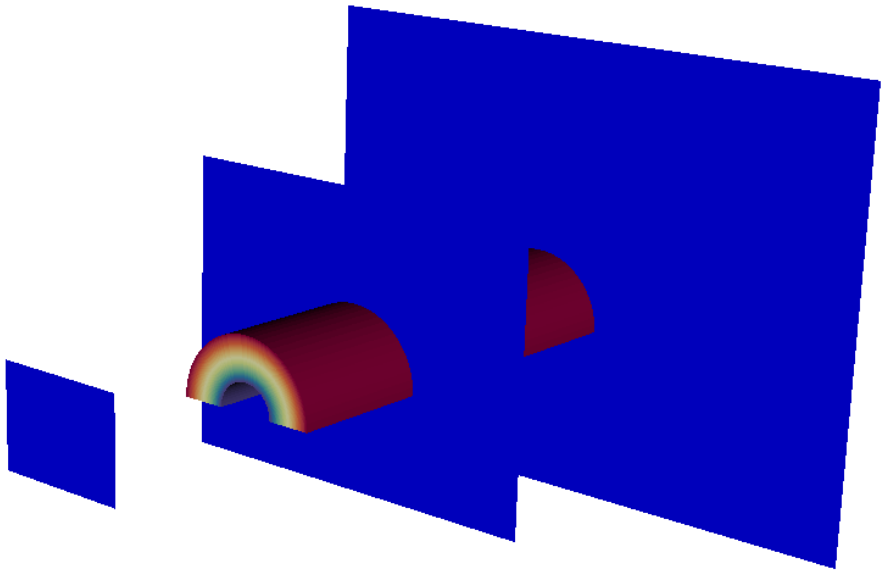


Fig. 4.291: A visualization of the input mesh along with the imaging planes.

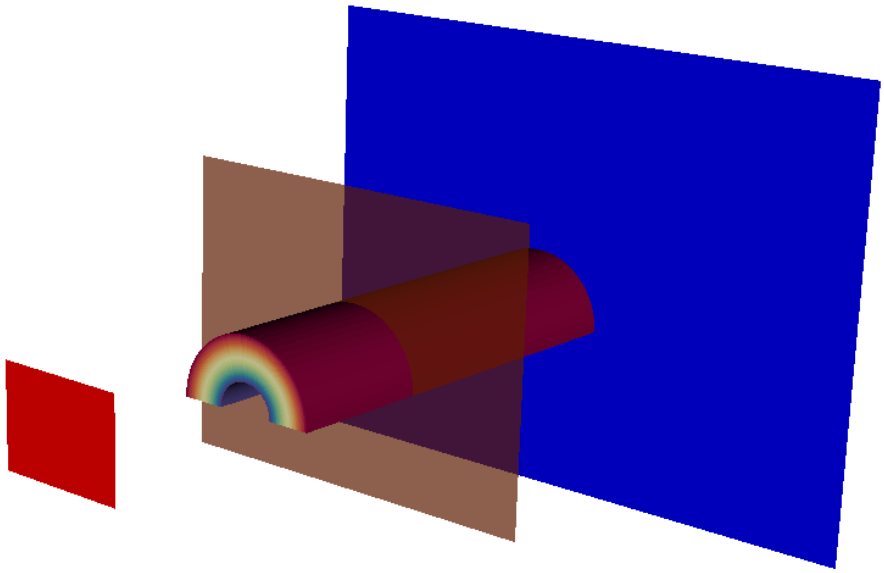


Fig. 4.292: A visualization of the input mesh along with the imaging planes, where they have had their colors adjusted.

(continued from previous page)

```
DrawPlots()

# The next few lines of code make the rays appear thicker for visual clarity.
MeshAtts = MeshAttributes()
MeshAtts.lineWidth = 1
SetPlotOptions(MeshAtts)
```

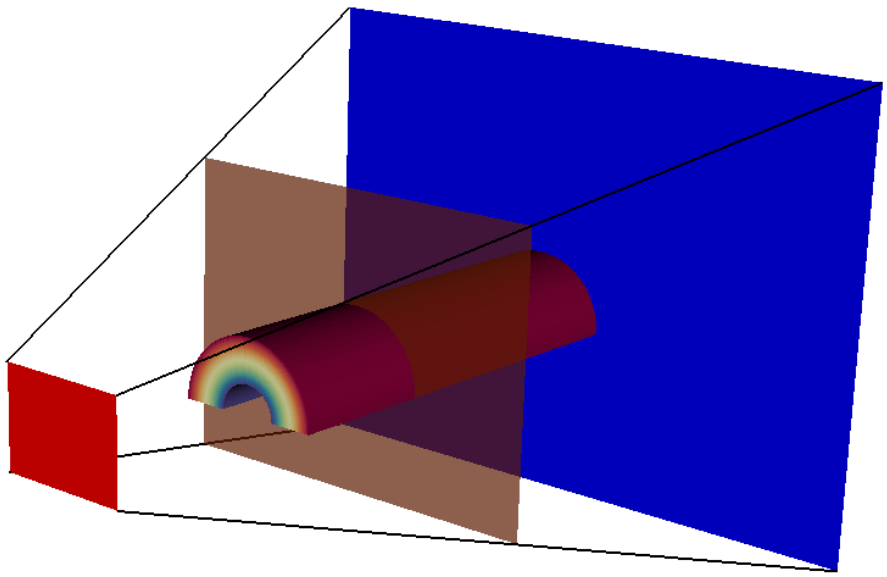


Fig. 4.293: A visualization of the input mesh, the imaging planes, and the ray corners.

Now we will visualize all of the rays.

```
AddPlot("Pseudocolor", "mesh_ray_topo/ray_field")
DrawPlots()
```

As discussed in the *Rays Meshes* section, this picture is not very helpful, so we will reduce the opacity for greater

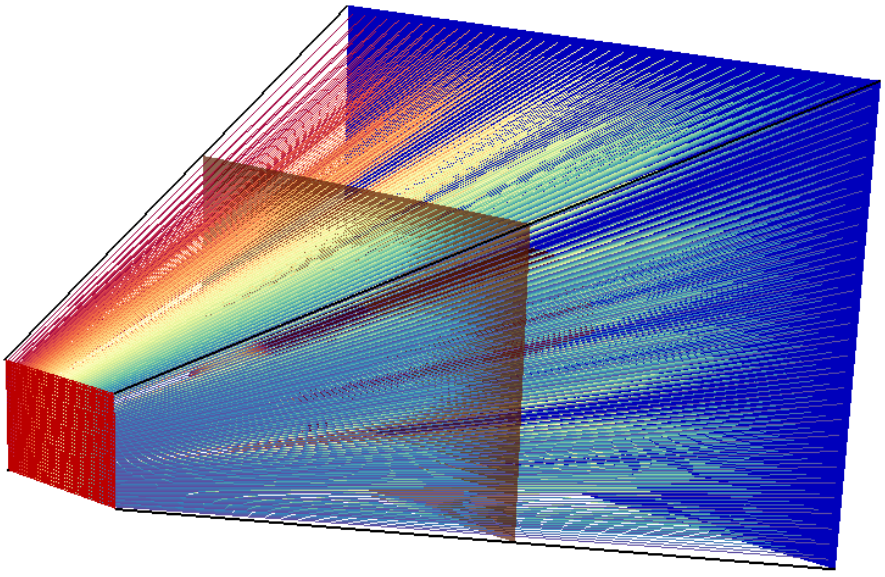


Fig. 4.294: A visualization of the input mesh, the imaging planes, the ray corners, and the rays.

visual clarity:

```
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.opacityType = PseudocolorAtts.Constant  # ColorTable, FullyOpaque,␣
↪Constant, Ramp, VariableRange
PseudocolorAtts.opacity = 0.5
SetPlotOptions(PseudocolorAtts)
```

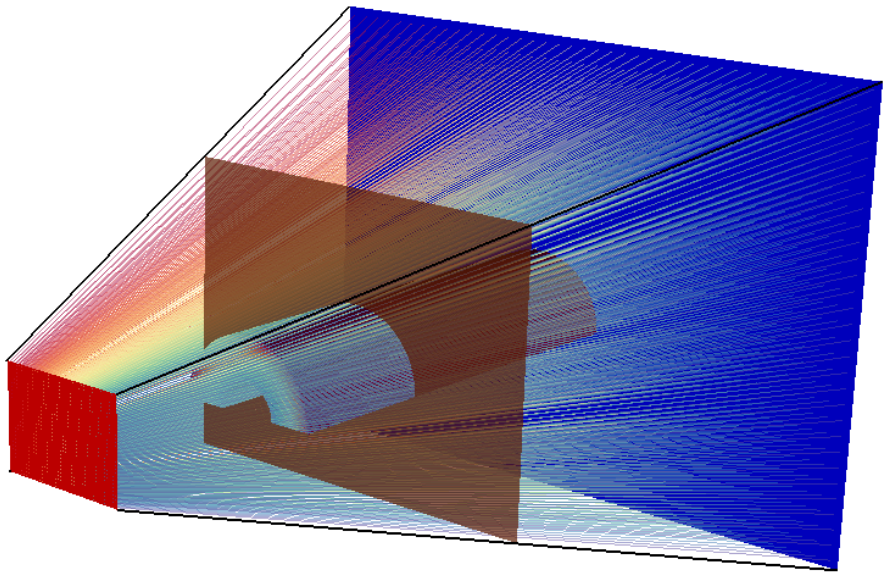


Fig. 4.295: A visualization of the input mesh, the imaging planes, the ray corners, and the rays, with their opacity adjusted.

See the *Rays Meshes* section for more tips for making sense of the rays.

Visualizing the Spatial Extents Meshes

Visualizing the *Spatial Extents Meshes* should be very similar to *Visualizing the Basic Mesh Output*.

First we render the spatial extents mesh:

```
# Make sure we have a clean slate for ensuing visualizations.
DeleteAllPlots()

# Add a pseudocolor plot of the intensities
AddPlot("Pseudocolor", "mesh_spatial_topo/intensities_spatial")

# Alternatively add a plot of the path length instead
# AddPlot("Pseudocolor", "mesh_spatial_topo/path_length_spatial")

DrawPlots()
```

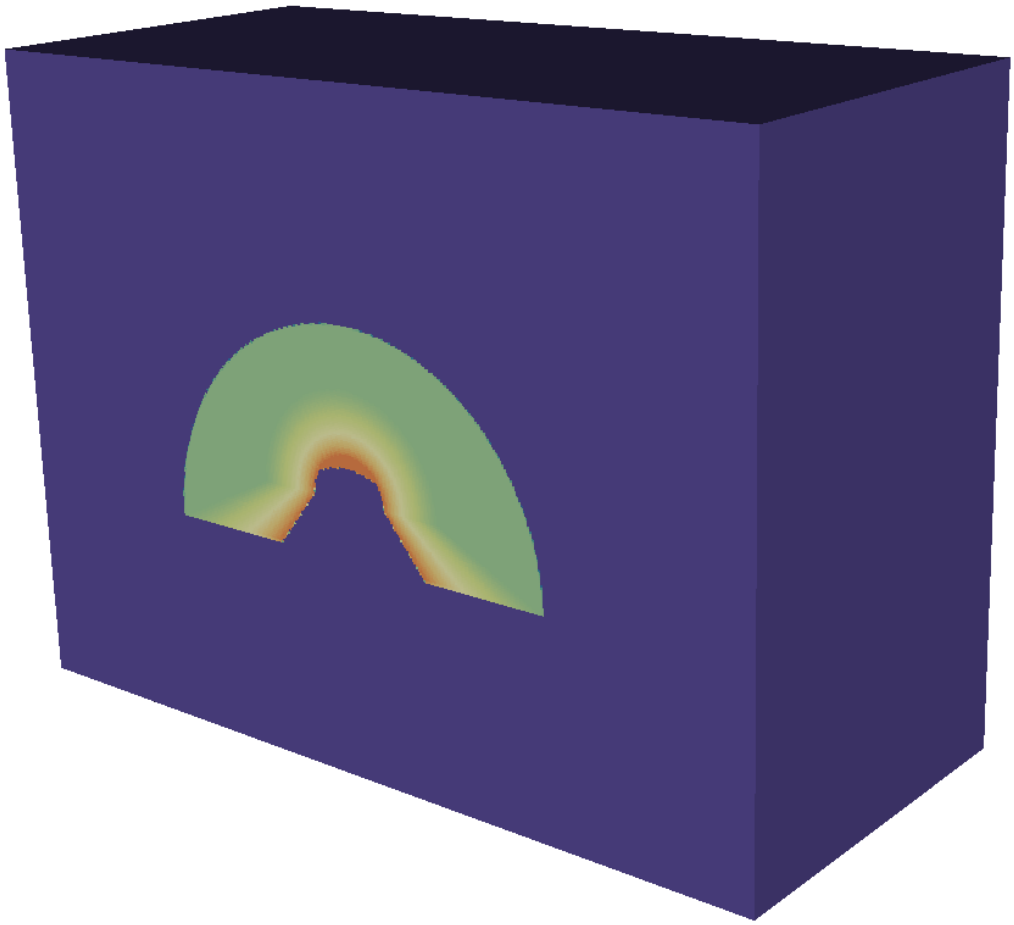


Fig. 4.296: A visualization of the spatial extents mesh.

To make the output look like an x ray image, it is simple to change the color table.

```
# Make sure the plot you want to change the color of is active
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.colorTableName = "xray"
SetPlotOptions(PseudocolorAtts)
```

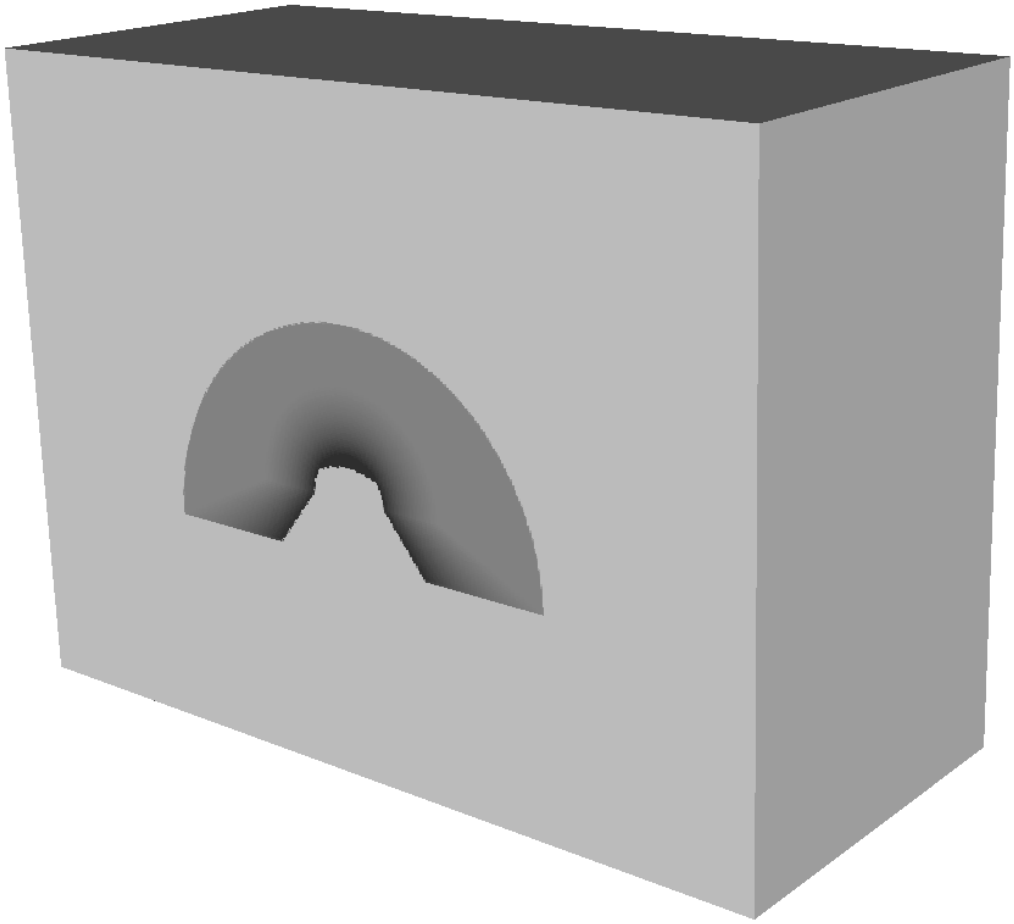


Fig. 4.297: A visualization of the spatial extents mesh using the x ray color table.

And then we render the spatial energy reduced mesh:

```
# Make sure we have a clean slate for ensuing visualizations.
DeleteAllPlots()

# Add a pseudocolor plot of the intensities
AddPlot("Pseudocolor", "mesh_spatial_energy_reduced_topo/intensities_spatial_energy_
↪reduced")
```

(continues on next page)

(continued from previous page)

```
# Alternatively add a plot of the path length instead
# AddPlot("Pseudocolor", "mesh_spatial_energy_reduced_topo/intensities_spatial_energy_
↪reduced")

DrawPlots()

# Change to x ray color table

# Make sure the plot you want to change the color of is active
PseudocolorAtts = PseudocolorAttributes()
PseudocolorAtts.colorTableName = "xray"
SetPlotOptions(PseudocolorAtts)
```

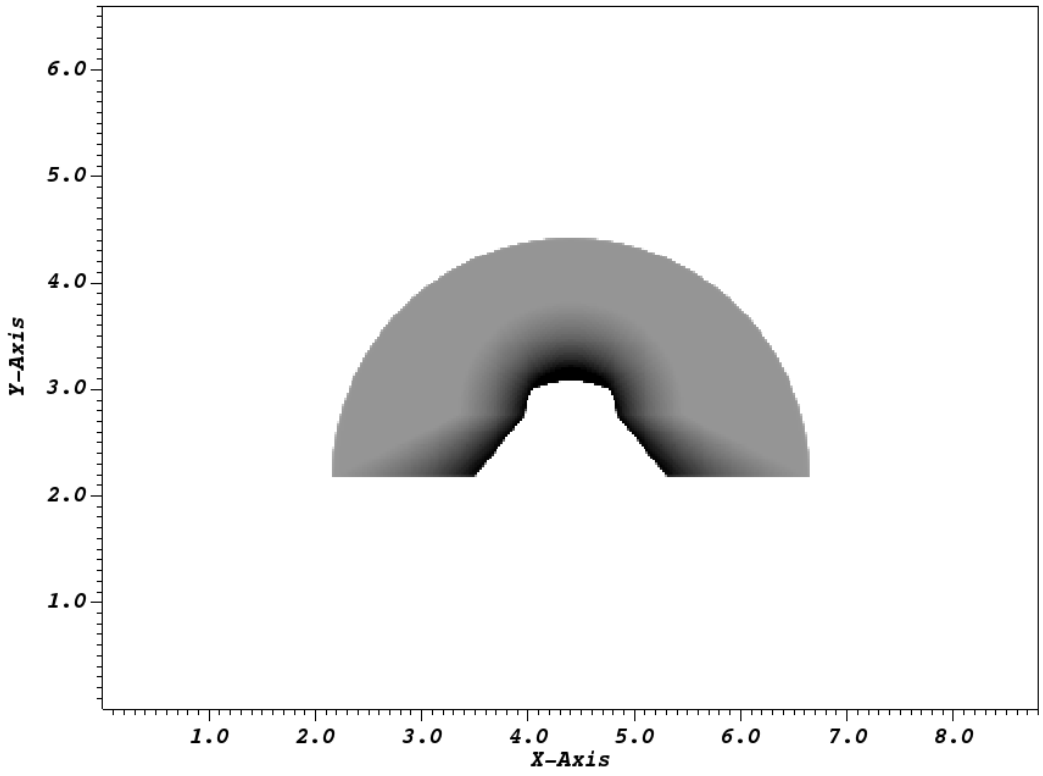


Fig. 4.298: A visualization of the spatial energy reduced mesh using the x ray color table.

Visualizing the 1D Spectra Curves

Visualizing the *1D Spectra Curves* is slightly different than visualizing the other Blueprint meshes. Because the Blueprint mesh is 1-dimensional, VisIt will interpret it as a curve. So, instead of adding a Pseudocolor plot, we will add a Curve plot instead.

```
# Make sure we have a clean slate for ensuing visualizations.
DeleteAllPlots()

# Add a curve plot of the intensities
AddPlot("Curve", "mesh_spectra_topo/intensities_spectra")

# Alternatively add a plot of the path length instead
# AddPlot("Curve", "mesh_spectra_topo/path_length_spectra")

DrawPlots()

# Remove the labels to clean up the image
SetActivePlots(0)
CurveAtts = CurveAttributes()
CurveAtts.showLabels = 0
SetPlotOptions(CurveAtts)
```

Introspecting with Python

We have covered visualizing every component of the *Conduit Output* in the *Visualizing with VisIt* section; now we will demonstrate how to access the raw data using Python.

Getting a General Overview of the Output

See *Overview of Output* for a visual of what the resulting Conduit tree looks like. First, we will get everything set up.

```
# make sure we import conduit
import conduit

# this node will be the destination for our output
xrayout = conduit.Node()

# actually perform the load
conduit.relay.io.blueprint.load_mesh(xrayout, "output.root")
```

Now we are ready to begin extracting data. To produce a Conduit tree like that of the example in *Overview of Output*, Conduit provides some simple tools:

```
# To print a condensed overview of the output
print(xrayout["domain_000000"])

# This is the same as
# print(xrayout["domain_000000"].to_summary_string())

# These will only print subsets of children, and for each child
# only a subset of leaf array values so as to not overwhelm the screen.

# The following will print the entirety of the output...
```

(continues on next page)

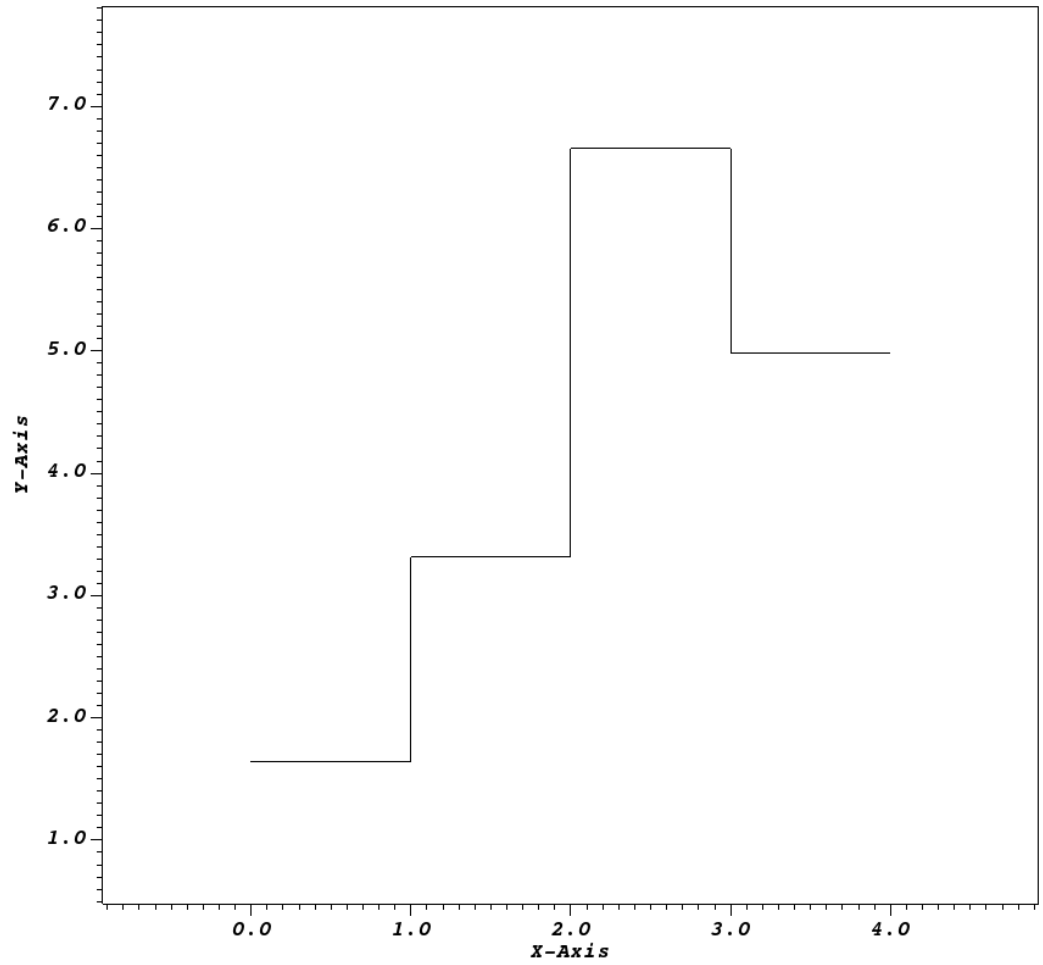


Fig. 4.299: A visualization of the 1D Spectra Curve mesh.

(continued from previous page)

```
# including every coordinate and field value,
# so use with caution.
print(xrayout["domain_000000"].to_yaml())
```

These simple features can be used not just on the root of the Conduit tree, but everywhere. We will see these used repeatedly in ensuing examples.

One other useful tool for interrogating a Conduit tree is the `childnames()` function. We can use `xrayout.childnames()` to see the names of all the top-level children as a list. In this case, calling `xrayout.childnames()` would produce `['state', 'coordsets', 'topologies', 'fields']`. We can call `childnames()` on any of the children of `xrayout` (`xrayout["state"].childnames()`, for example) to further investigate the layers of the tree.

Additionally, it is possible to iterate through the children of a Conduit node using this:

```
for child in xrayout.children():
   print(child.name(), child.node())
```

In general, children are not always named. For our purposes with the X Ray Image Query, they always will be. A node can behave like a python dictionary or a python list; for the latter, index access is possible.

Accessing the Basic Mesh Output Data

To get a sense of what the *Basic Mesh Output* looks like, we can run the following:

```
print("image_coords")
print(xrayout["domain_000000/coordsets/image_coords"])

print("image_topo")
print(xrayout["domain_000000/topologies/image_topo"])

print("intensities")
print(xrayout["domain_000000/fields/intensities"])
print("path_length")
print(xrayout["domain_000000/fields/path_length"])
```

This produces...

```
image_coords

type: "rectilinear"
values:
  x: [0, 1, 2, ..., 399, 400]
  y: [0, 1, 2, ..., 299, 300]
  z: [0, 1]
labels:
  x: "width"
  y: "height"
  z: "energy_group"
units:
  x: "pixels"
  y: "pixels"
  z: "bins"

image_topo
```

(continues on next page)

(continued from previous page)

```
coordset: "image_coords"
type: "rectilinear"

intensities

topology: "image_topo"
association: "element"
units: "intensity units"
values: [0.0, 0.0, 0.0, ..., 0.0, 0.0]
strides: [1, 400, 120000]

path_length

topology: "image_topo"
association: "element"
units: "path length metadata"
values: [0.0, 0.0, 0.0, ..., 0.0, 0.0]
strides: [1, 400, 120000]
```

Note that the long arrays are condensed for the sake of readability. If we wanted to see the entirety of the arrays, we could run `print(myconduitnode.to_yaml())` instead of `print(myconduitnode)`.

To actually extract the *Basic Mesh Output* data and not just see it, we can run the following:

```
# Extract the actual x values, label, and units
xvals = xrayout["domain_000000/coordsets/image_coords/values/x"]
xlabel = xrayout["domain_000000/coordsets/image_coords/labels/x"]
xunits = xrayout["domain_000000/coordsets/image_coords/units/x"]
# Extracting the same for y and z is similar

# Extract units and values for the intensity output
intensity_units = xrayout["domain_000000/fields/intensities/units"]
intensity_values = xrayout["domain_000000/fields/intensities/values"]
# Extracting the same for path_length is similar
```

These variables can be printed, manipulated, iterated over, etc.

Accessing the Metadata

Again, to get an overview of the *Metadata*, it is simple to print the `state` branch:

```
# get an overview of the metadata
print(xrayout["domain_000000/state"])

# see all the metadata
print(xrayout["domain_000000/state"].to_yaml())
```

The following code extracts each of the values. First is top level *Metadata*:

```
time = xrayout["domain_000000/state/time"]
cycle = xrayout["domain_000000/state/cycle"]
```

Next up is *View Parameters*:

```
normalx = xrayout["domain_000000/state/xray_view/normal/x"]
normaly = xrayout["domain_000000/state/xray_view/normal/y"]
normalz = xrayout["domain_000000/state/xray_view/normal/z"]

focusx = xrayout["domain_000000/state/xray_view/focus/x"]
focusy = xrayout["domain_000000/state/xray_view/focus/y"]
focusz = xrayout["domain_000000/state/xray_view/focus/z"]

view_upx = xrayout["domain_000000/state/xray_view/view_up/x"]
view_upy = xrayout["domain_000000/state/xray_view/view_up/y"]
view_upz = xrayout["domain_000000/state/xray_view/view_up/z"]

view_angle = xrayout["domain_000000/state/xray_view/view_angle"]
parallel_scale = xrayout["domain_000000/state/xray_view/parallel_scale"]
view_width = xrayout["domain_000000/state/xray_view/view_width"]
non_square_pixels = xrayout["domain_000000/state/xray_view/non_square_pixels"]
near_plane = xrayout["domain_000000/state/xray_view/near_plane"]
far_plane = xrayout["domain_000000/state/xray_view/far_plane"]

image_panx = xrayout["domain_000000/state/xray_view/image_pan/x"]
image_pany = xrayout["domain_000000/state/xray_view/image_pan/y"]

image_zoom = xrayout["domain_000000/state/xray_view/image_zoom"]
perspective = xrayout["domain_000000/state/xray_view/perspective"]
perspective_str = xrayout["domain_000000/state/xray_view/perspective_str"]
```

Then *Query Parameters*:

```
divide_emis_by_absorb = xrayout["domain_000000/state/xray_query/divide_emis_by_absorb
↪"]
divide_emis_by_absorb_str = xrayout["domain_000000/state/xray_query/divide_emis_by_
↪absorb_str"]
num_x_pixels = xrayout["domain_000000/state/xray_query/num_x_pixels"]
num_y_pixels = xrayout["domain_000000/state/xray_query/num_y_pixels"]
num_bins = xrayout["domain_000000/state/xray_query/num_bins"]
abs_var_name = xrayout["domain_000000/state/xray_query/abs_var_name"]
emis_var_name = xrayout["domain_000000/state/xray_query/emis_var_name"]
abs_units = xrayout["domain_000000/state/xray_query/abs_units"]
emis_units = xrayout["domain_000000/state/xray_query/emis_units"]
```

And finally, *Other Metadata*:

```
detector_width = xrayout["domain_000000/state/xray_data/detector_width"]
detector_height = xrayout["domain_000000/state/xray_data/detector_height"]
intensity_max = xrayout["domain_000000/state/xray_data/intensity_max"]
intensity_min = xrayout["domain_000000/state/xray_data/intensity_min"]
path_length_max = xrayout["domain_000000/state/xray_data/path_length_max"]
path_length_min = xrayout["domain_000000/state/xray_data/path_length_min"]
image_topo_order_of_domain_variables = xrayout["domain_000000/state/xray_data/image_
↪topo_order_of_domain_variables"]
```

Accessing the Spatial Extents Meshes Data

Because the *Spatial Extents Meshes* share a lot in common with the *Basic Mesh Output*, we will only cover here how to extract some of the unique values.

```
# Extract the actual x, y, and z values
spatial_xvals = xrayout["domain_000000/coordsets/spatial_coords/values/x"]
spatial_yvals = xrayout["domain_000000/coordsets/spatial_coords/values/y"]
energy_group_bounds = xrayout["domain_000000/coordsets/spatial_coords/values/z"]

# Extract the x label
spatial_xlabel = xrayout["domain_000000/coordsets/spatial_coords/labels/x"]
# Extracting the same for y and z is similar

# Extract the spatial and energy units
spatial_xunits = xrayout["domain_000000/coordsets/spatial_coords/units/x"]
spatial_yunits = xrayout["domain_000000/coordsets/spatial_coords/units/y"]
energy_units = xrayout["domain_000000/coordsets/spatial_coords/units/z"]
```

Accessing the 1D Spectra Curves Data

The *1D Spectra Curves* are similar in structure to the other standard Blueprint meshes.

```
# Extract the energy group bounds
energy_group_bounds = xrayout["domain_000000/coordsets/spectra_coords/values/x"]

# Extract the label
spectra_label = xrayout["domain_000000/coordsets/spectra_coords/labels/x"]

# Extract the units
spatial_xunits = xrayout["domain_000000/coordsets/spectra_coords/units/x"]

# Extract the field values
intensities_spectra_curve_values = xrayout["domain_000000/fields/intensities_spectra/
↪values"]
# Extracting the same for path_length is similar
```

Accessing Everything Else

All of the other data stored in the Conduit output can be accessed in the same way. To get a general sense of what is stored in particular branches of the tree, it is a simple matter of running `print(myconduitnode)` to quickly get an overview.

Troubleshooting

Now that we have explored the Conduit Blueprint output in detail, we can use it to troubleshoot unexpected or strange query results.

Is my image blank?

This question can be answered without even examining the image (or in the case of the Blueprint output, a render of the *Basic Mesh Output*). It is as simple as checking if the minimum and maximum values for the intensities and path length are zero. See *Other Metadata* for more information. These values can be pulled out of the Conduit output with ease, using the following:

```
import conduit
mesh = conduit.Node()

# In this case, "output.root" is the name of the Blueprint file
# that was output from the query.
conduit.relay.io.blueprint.load_mesh(mesh, "output.root")

# We extract the values from the node.
intensity_max = mesh["domain_000000/state/xray_data/intensity_max"]
intensity_min = mesh["domain_000000/state/xray_data/intensity_min"]
path_length_max = mesh["domain_000000/state/xray_data/path_length_max"]
path_length_min = mesh["domain_000000/state/xray_data/path_length_min"]

print("intensity_max = " + str(intensity_max))
print("intensity_min = " + str(intensity_min))
print("path_length_max = " + str(path_length_max))
print("path_length_min = " + str(path_length_min))
```

Yielding:

```
intensity_max = 0.49144697189331055
intensity_min = 0.0
path_length_max = 129.8570098876953
path_length_min = 0.0
```

If the maximums were also equal to zero, then the image would be blank. Hence, it is possible to quickly programmatically check if the image is blank, without any need for taking the time to look at the image. See *Introspecting with Python* for more information about extracting data from the query output.

Why is my image blank?

Is the camera facing the right way? Are the near and far clipping planes in good positions?

This line of questioning can be quickly answered by *Visualizing the Imaging Planes* and *Visualizing the Rays Meshes*.

```
# Make sure the mesh used in the query is already rendered.

# In this case, "output.root" is the name of the Blueprint file
# that was output from the query.
OpenDatabase("output.root")

# Add pseudocolor plots of each of the imaging planes.
AddPlot("Pseudocolor", "mesh_far_plane_topo/far_plane_field")
AddPlot("Pseudocolor", "mesh_view_plane_topo/view_plane_field")
AddPlot("Pseudocolor", "mesh_near_plane_topo/near_plane_field")

# Add a mesh plot of the ray corners.
AddPlot("Mesh", "mesh_ray_corners_topo")

DrawPlots()
```

Running this code using VisIt should result in renders like those shown in *Imaging Planes and Rays Meshes*. To make the planes different colors, use VisIt's color table controls, or see *Visualizing the Imaging Planes*.

The simulated x ray detector is situated at the near plane, looking in the direction of the view plane, and seeing nothing after the far plane. Once the imaging planes and ray corners have been visualized, it is clear to see where the camera

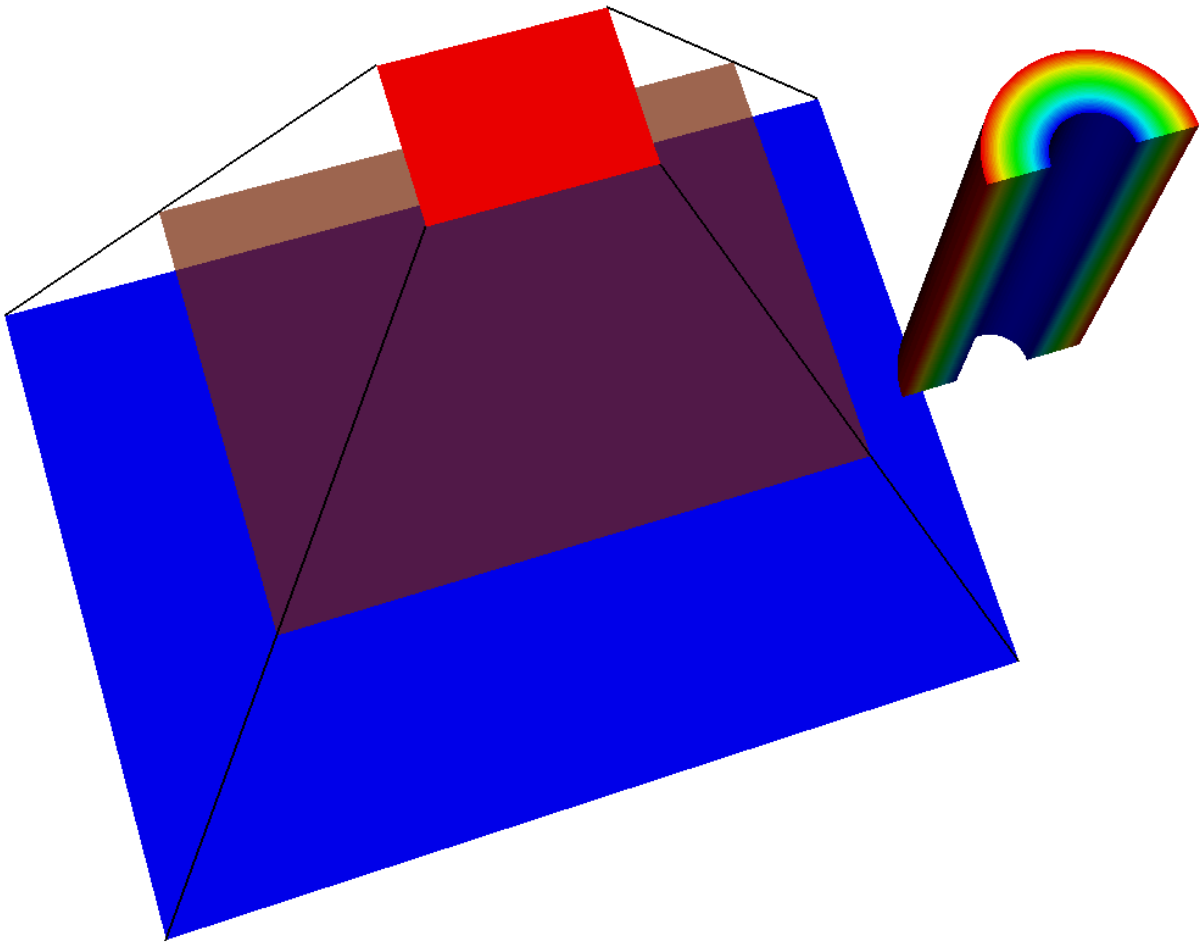


Fig. 4.300: An example of what could be going wrong. The simulated detector is not positioned correctly.

is looking, and if the near and far clipping planes are appropriately placed. See the text on visualizing the rays and imaging planes in *Visualizing the Imaging Planes* and *Visualizing the Rays Meshes*.

Where are the rays intersecting my geometry?

Answering this question is similarly simple. We will want to visualize the *Rays Meshes* on top of our input mesh.

```
# Make sure the mesh used in the query is already rendered.

# In this case, "output.root" is the name of the Blueprint file
# that was output from the query.
OpenDatabase("output.root")

# Add pseudocolor plots of each of the imaging planes.
# These help to add clarity to our final render.
AddPlot("Pseudocolor", "mesh_far_plane_topo/far_plane_field")
AddPlot("Pseudocolor", "mesh_view_plane_topo/view_plane_field")
AddPlot("Pseudocolor", "mesh_near_plane_topo/near_plane_field")

# Add a mesh plot of the rays.
AddPlot("Pseudocolor", "mesh_ray_topo/ray_field")

DrawPlots()
```

Running this code using VisIt should result in renders like those shown in *Rays Meshes*. Use the tips and tricks shown in that section to gain greater clarity for answering this question. See the text on visualizing the rays and imaging planes in *Visualizing the Imaging Planes* and *Visualizing the Rays Meshes*.

What information is the query using to create the output?

See *Accessing the Metadata* for information on how to extract and view the *Metadata*, which contains all of the view parameters and query parameters that the query uses to create the output.

The fields in the Conduit Output are 1D. How can I reshape them to be 3D?

The following code examples will take resultant data (the intensity values, in this case) from the X Ray Image Query and reshape it to be 3D.

```
import conduit

def fetch_coordset_dims(data):
    cset_vals = data.fetch_existing("coordsets/image_coords/values")
    # fetch rectilinear coordset dims
    nx = cset_vals.fetch_existing("x").dtype().number_of_elements()
    ny = cset_vals.fetch_existing("y").dtype().number_of_elements()
    nz = cset_vals.fetch_existing("z").dtype().number_of_elements()
    # # if you would like to see the labels for these axes, this is
    # # how you can access them:
    # cset_lbls = data.fetch_existing("coordsets/image_coords/labels")
    # print(cset_lbls.fetch_existing("x"))
    # print(cset_lbls.fetch_existing("y"))
    # print(cset_lbls.fetch_existing("z"))
    return nx, ny, nz
```

(continues on next page)

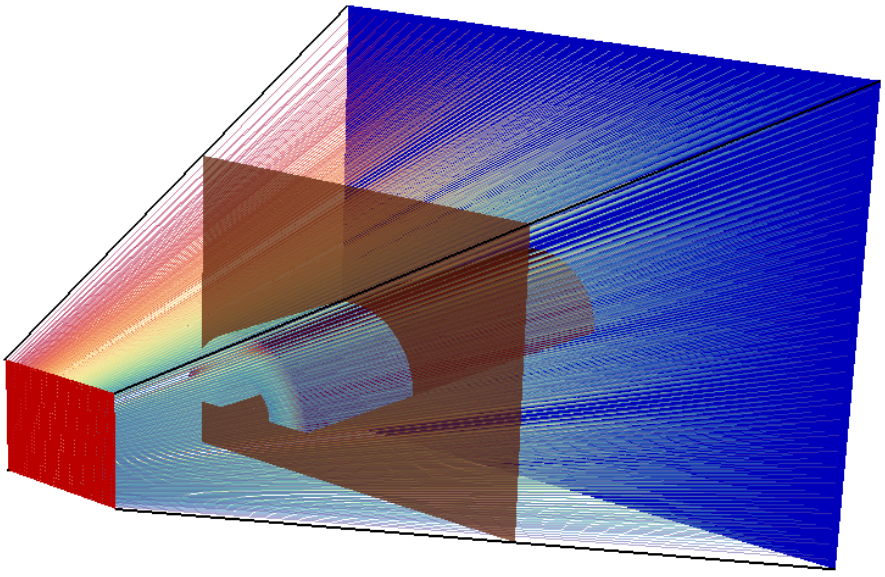


Fig. 4.301: An example render made using the above code.

(continued from previous page)

```
def fetch_topology_dims(data):
    # rectilinear topology have one less element in each
    # logical dim than the coordset
    cnx, cny, cnz = fetch_coordset_dims(data)
    return cnx - 1, cny - 1, cnz - 1

def fetch_reshaped_field_values(data, field_name):
    nx, ny, nz = fetch_topology_dims(data)
    vals = data.fetch_existing("fields/" + field_name + "/values").value()
    return vals.reshape(nz, ny, nx)
```

We know that we should reshape in this order because x varies the fastest, then y, and then z. To verify this, we can examine the `image_topo_order_of_domain_variables` metadata (more information on that here: *Other Metadata*) which records in which order the axes vary. For the X Ray Image Query, this will always be x, then y, then z, so `xrayout["domain_000000/state/xray_data/image_topo_order_of_domain_variables"] == "xyz"` always.

Now that we have set everything up, let's call the functions we created:

```
# read xray blueprint output
n = conduit.Node()

# the output file is in this case called "output.root"
conduit.relay.io.blueprint.load_mesh(n, "output.root")

# the node with our data is domain 0 -- or the first child
data = n[0] # the same could be done with data = 'n["domain_000000"]'

# the same could be done for "path_length"
intensity_vals = fetch_reshaped_field_values(data, "intensities")

print("1D intensities")
print(data["fields/intensities/values"])

print("3D intensity shape")
print(intensity_vals.shape)

print("3D intensities")
print(intensity_vals)
```

This produces the following:

```
1D intensities
[0.         0.         0.         0.         0.         0.20180537
 0.20180535 0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.20180537
 0.20180535 0.         0.         0.         0.         0.        ]

3D intensity shape
(2, 3, 4)

3D intensities
[[[0.         0.         0.         0.        ]
  [0.         0.20180537 0.20180535 0.        ]
  [0.         0.         0.         0.        ]]
```

(continues on next page)

(continued from previous page)

```
[[0.         0.         0.         0.        ]
 [0.         0.20180537 0.20180535 0.        ]
 [0.         0.         0.         0.        ]]]
```

The number of values is so small because I picked an image size of 4x3 pixels and 2 energy groups to demonstrate this.

4.8.4 Pick

VisIt provides a way to interactively pick values from the visualized data using the visualization window's Zone Pick and Node Pick modes. When a visualization window is in one of those pick modes, each mouse click in the visualization window causes VisIt to find the location and values of selected variables at the pick point. When VisIt is in Zone pick mode, it finds the variable values for the zones that you click on. When VisIt is in node pick mode, similar information is returned but instead of returning information about the zone that you clicked on, VisIt returns information about the node closest to the point that you clicked. Pick is an essential tool for performing data analysis because it can extract exact information from the database about a point in the visualization.

Pick mode

You can put the visualization window into one of VisIt's pick modes by selecting **Zone Pick** or **Node Pick** from the **Popup menu's Mode** submenu. After the visualization window is in pick mode, each mouse click causes VisIt to determine the values of selected variables for the zone that contains the picked point or the node closest to the picked point. Each picked point is marked with an alphabetic label which starts at A, cycles through the alphabet and repeats. The pick marker is added to the visualization window to indicate where pick points have been added in the past. To clear pick points from the visualization window, select the **Pick points** option from the **Clear** menu in the **Main Window's Window** menu. The dimension of the plots in the visualization does not matter when using pick mode. Both 2D and 3D plots can be picked for values. However, when using pick mode with 3D plots, only the surface of the plots can be picked for values. If you want to obtain interior values then you should use one of the Pick queries or apply operators that expose the interiors of 3D plots before using pick. An example of the visualization window with pick points is shown in Figure 4.302 and an example of node pick and zone pick markers is shown in Figure 4.303.

Pick Window

Each time a new pick point is added to the visualization window by clicking on a plot, VisIt extracts information about the pick point from the plot's database and displays it in the **Pick Window** (Figure 4.304) and the **Output Window**. If the **Pick Window** does not automatically open after picking, you can open the **Pick Window** by selecting the **Pick** option from the **Main Window's Controls** menu.

The **Pick Window** mainly consists of a group of tabs, each of which displays the values from a pick point. The tab label A, B, C, etc. corresponds to the pick point label in the visualization window. Since there is a fixed number of tabs in the **Pick Window**, tabs are recycled as the number of pick points increases. When a pick point is added, the next available tab, which is usually the tab to the right of the last unused tab, is populated with the pick information. If the rightmost tab already contains pick information, the leftmost tab is recycled and the process repeats. To see a complete list of picked points, open the **Output Window**.

The information displayed in each tab consists of the database name and timestep, the coordinates of the pick point, the zone/cell that contains the pick point, the nodes that make up the cell containing the pick point, and the picked variables. The rest of the **Pick Window** is devoted to setting options that format the pick output.

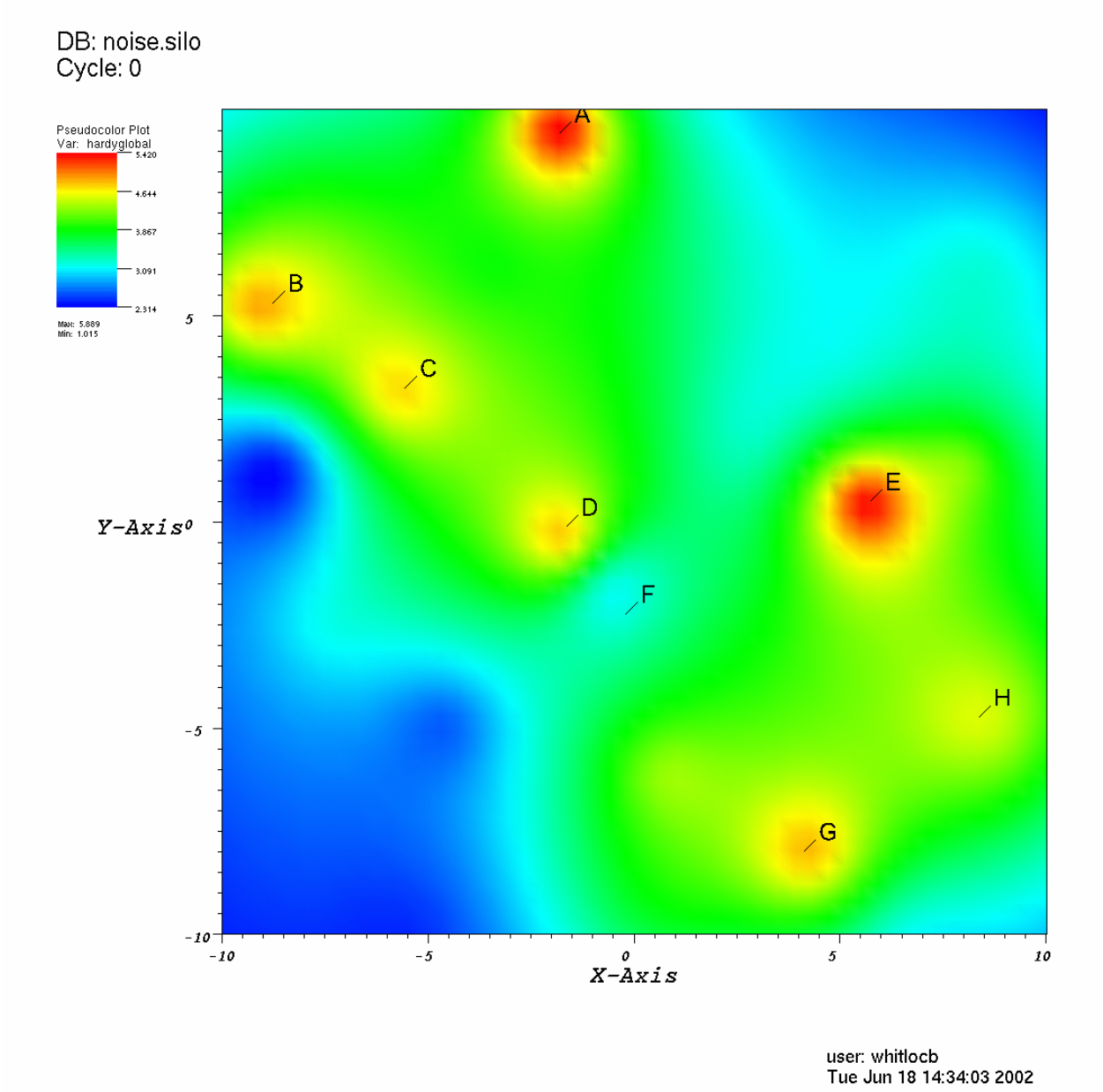


Fig. 4.302: Visualization window with pick points

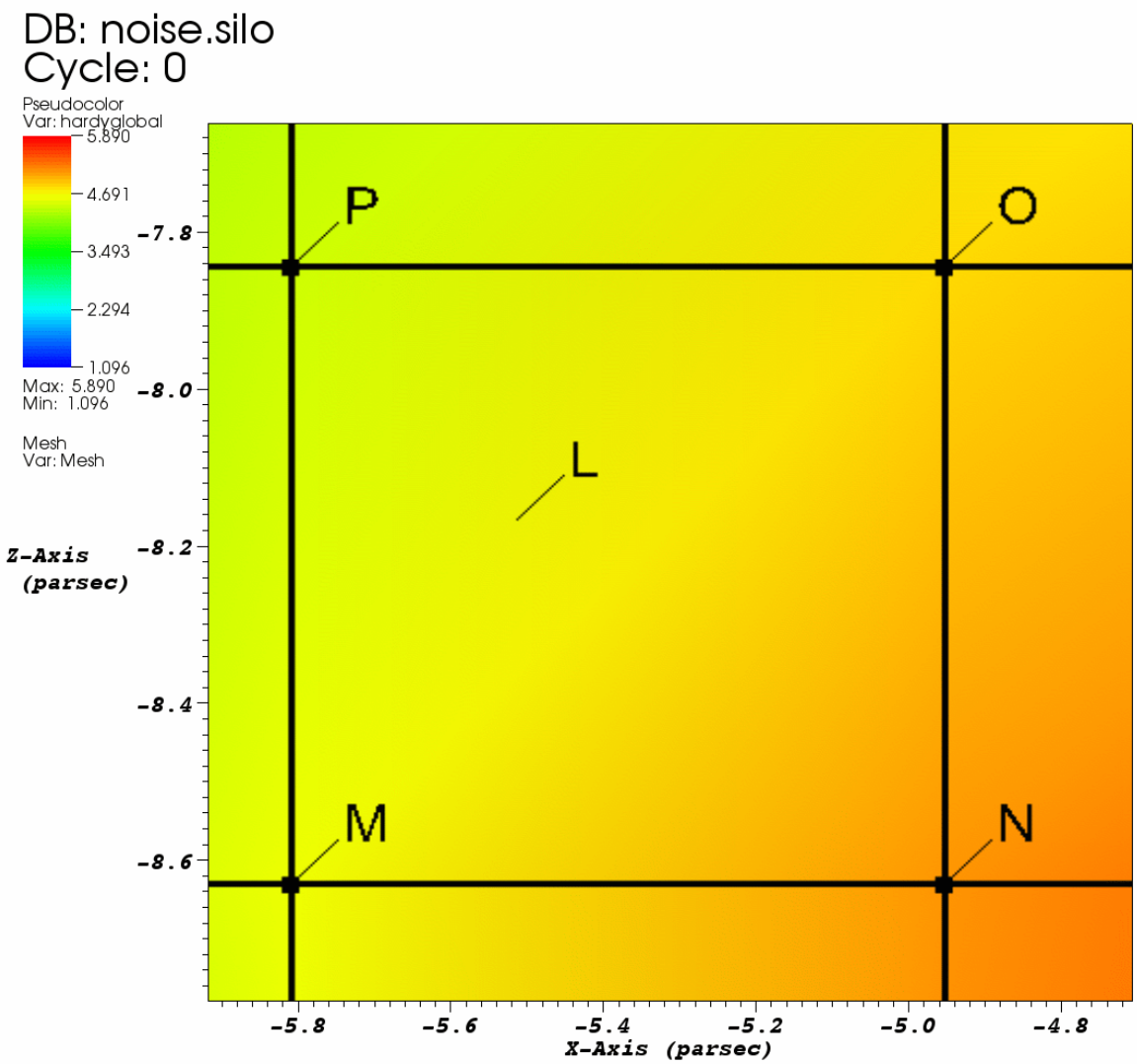


Fig. 4.303: Zone pick marker **L** and node pick markers **M**, **N**, **O**, **P**

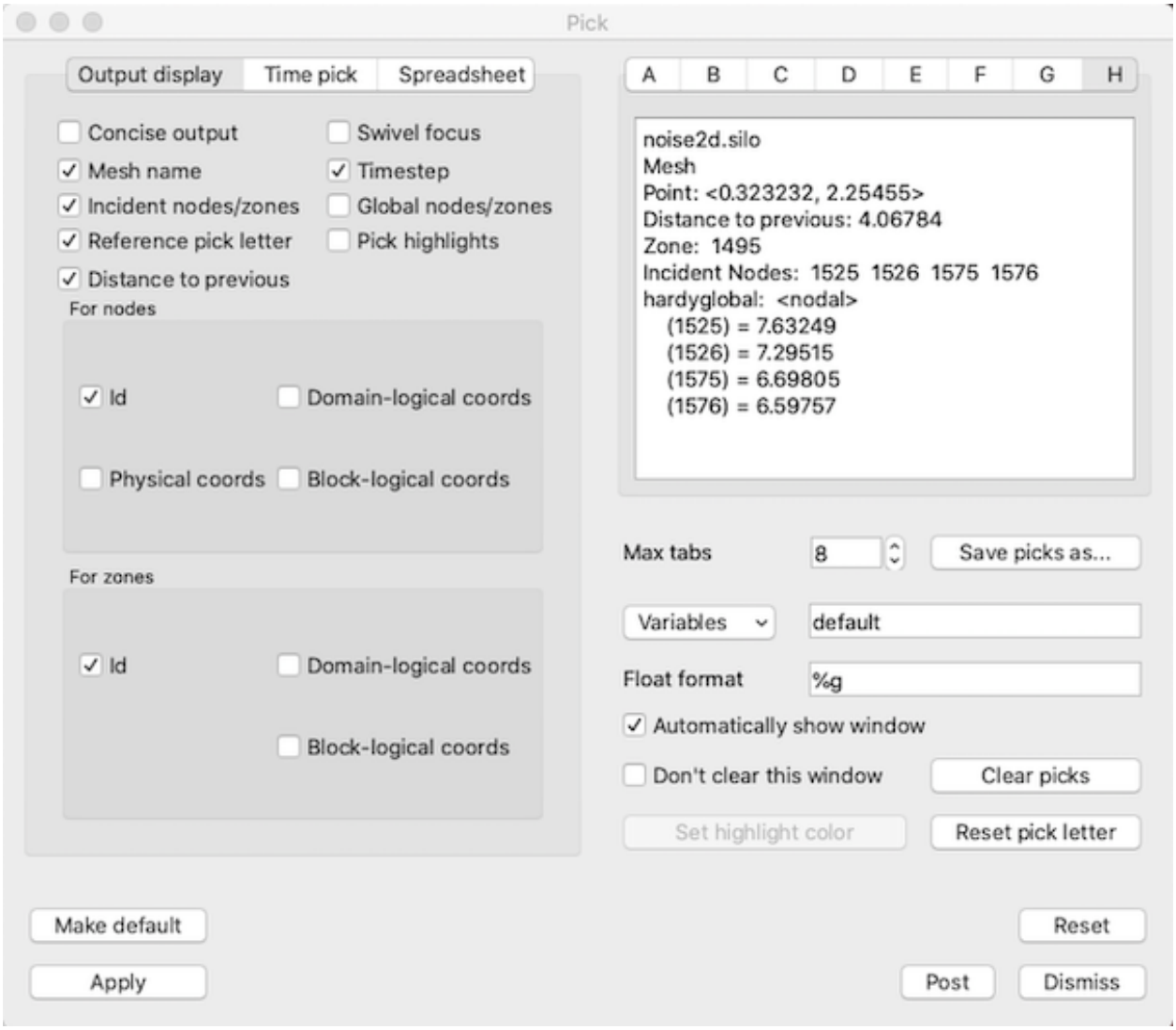


Fig. 4.304: Pick Window

Setting the pick variable

The **Pick Window** contains a **Variables** text field that allows you to specify pick variables Most of the time, the value in the text field is the word "default" which tells VisIt to use the plotted variables as the pick variables. You can replace the default pick variable by typing one or more valid variable names, separated by spaces, into the **Variables** text field. You can also select additional pick variables by selecting a new variable name from the **Variables** variable button to the left of the **Variables** text field. When more than one variable is picked, multiple variables appear in the pick information displayed in the information tabs.

Concise pick output

Pick returns a lot of information when you pick on a plot. The **Pick Window** usually displays the pick output one item per line, which can end up taking a lot of vertical space. If you want to condense the information into a smaller area, click the **Concise output** check box. Sometimes, not all of the information is relevant for your analysis so VisIt provides options to hide certain items in the pick output. If you don't want VisIt to display the name of the picked mesh, turn off the **Show Mesh Name** check box. If you don't want VisIt to show the time state, turn of the **Show timestep** check box.

Turning off incident nodes and cells in pick output

When VisIt performs a pick, the default behavior is to show a lot of information about the cell or node that was picked. This information usually includes the nodes or cells that were incident to the node or cell that was picked. The incident nodes and cells are included to give more information about the neighborhood occupied by the cell or node. If you want to turn off incident nodes and cells in the pick output, click off the **Display incident nodes/zones** check box.

Displaying global node and cell numbers

Many large meshes are decomposed into smaller meshes called domains that, when added together, make up the whole mesh. Each domain typically has its own range of cell numbers that begin at 0 or 1, depending on the mesh's cell origin. Any global cell numbering scheme that may have been in place before the original mesh was decomposed into domains is often lost. However, some meshes have auxiliary information that allows VisIt to use the original global node and cell numbers for the domains. If you want the pick output to contain global node and cell numbers if they are available, click on the **Display global nodes/zones** check box.

Turning off pick markers for new pick points

Some queries that perform picks create pick markers by default, as do VisIt's regular pick modes. If you want to prevent pick queries from creating pick markers, click off the **Pick Window's Display reference pick letter** check box.

Displaying distance to previous

The **Pick Window** contains a **Distance to previous** checkbox in the **Output display** tab. If checked, the **Pick** output will contain the Euclidean distance bewteen the current and previous **Pick** points. Note that, when picking zones, the picked point corresponds to the exact point that your mouse lands on the surface of the mesh, which exists within a zone. However, when picking nodes, the picked point is the closest node to where your mouse lands. Therefore, if you're calculating the distance between two node picks, the results will represent the exact distance between the coordinates of those chosen nodes. When calculating the distance between zone picks, the results will represent the

distance between your picked points within the zones. You can also calculate the distance between a picked node and a picked zone by performing one after the other.

Returning node information

In addition to printing the values of the pick variables, pick can also display information about the nodes or cells over which the pick variables are defined. By default, VisIt only returns the integer node indices of the nodes contained by the picked cell. You can make VisIt return the node coordinates in other formats by checking the **Id** check box in the **Display for Nodes** area. The node coordinates can be displayed 4 different ways: Node indices, physical coordinates, domain-logical coordinates, or block-logical coordinates. Click the check boxes in the **Display for Nodes** area that correspond to the types of node information that you want to examine.

Returning zone information

The **Pick Window** has controls in its **Display for Zones** area that allow you to specify how you want VisIt to display zone information. Click the check boxes that correspond to the types of information that you want to examine.

Automatically showing the Pick Window

When you pick on a plot, VisIt automatically opens the **Pick Window** to display the results of the pick operation. You can prevent VisIt from automatically showing the **Pick Window** after a pick operation by turning off the **Automatically show window** check box in the **Pick Window**. If the **Pick Window** does not automatically appear after picking then you can turn on the **Automatically show window** check box.

Picking over time

Querying over time is normally done using the controls in the **Query Window** but you can also pick over time to generate curves that show the behavior of a picked zone or node over time. To pick over time, you must click the **Create time curve with next pick** check box in the **Pick Window**. Once that check box is turned on, each pick operation will result in a new Curve plot that shows the behavior of the most recently picked zone or node over time.

Note on performance: You'll notice that you can either choose to follow the picked *coordinates* or the picked *element* through time. While each of these options generates very different results, it's worth keeping in mind that following the picked *element* will be substantially faster when working with datasets with large numbers of time steps.

4.8.5 Lineout

One-dimensional curves, created using data from 2D or 3D plots, are popular for analyzing data because they are simple to compare. VisIt's visualization windows can be put into a mode that allows you to draw lines, along which data are extracted, in the visualization window. The extracted data are turned into a Curve plot in another visualization window. If no other visualization window exists, VisIt creates one and adds the Curve plot to it. Curve plots are often more useful than 2D Pseudocolor plots because they allow the data along a line to be seen spatially as a 1D curve instead of relying on differences in color to convey information. Furthermore, the curve data can be exported to curve files that allow the data to be imported into other Lawrence Livermore National Laboratory curve analysis software such as *Ultra*.

Lineout mode

You can put the visualization window into lineout mode by selecting the **Lineout** icon (Figure 4.305) in the visualization window's Toolbar or from the **Popup menu's Mode** submenu. Note that lineout mode is only available with 2D plots in this version though you can create 3D lineouts using the Lineout query in the **Query Window**. After the visualization window is in lineout mode, you can draw reference lines in the window. Each reference line causes VisIt to extract data from the database along the prescribed path and draw the data as a Curve plot in another visualization window. Each reference line is drawn in a color that matches the initial color of the Curve plot so the reference lines, which may not have labels, can be easily associated with their corresponding Curve plots. To clear the reference lines from the visualization window, select the **Clear reference lines** option from **Popup menu's Clear** submenu. An example of the visualization window with reference lines and Curve plots is shown in Figure 4.306.

Fig. 4.305: Lineout mode toolbar icon

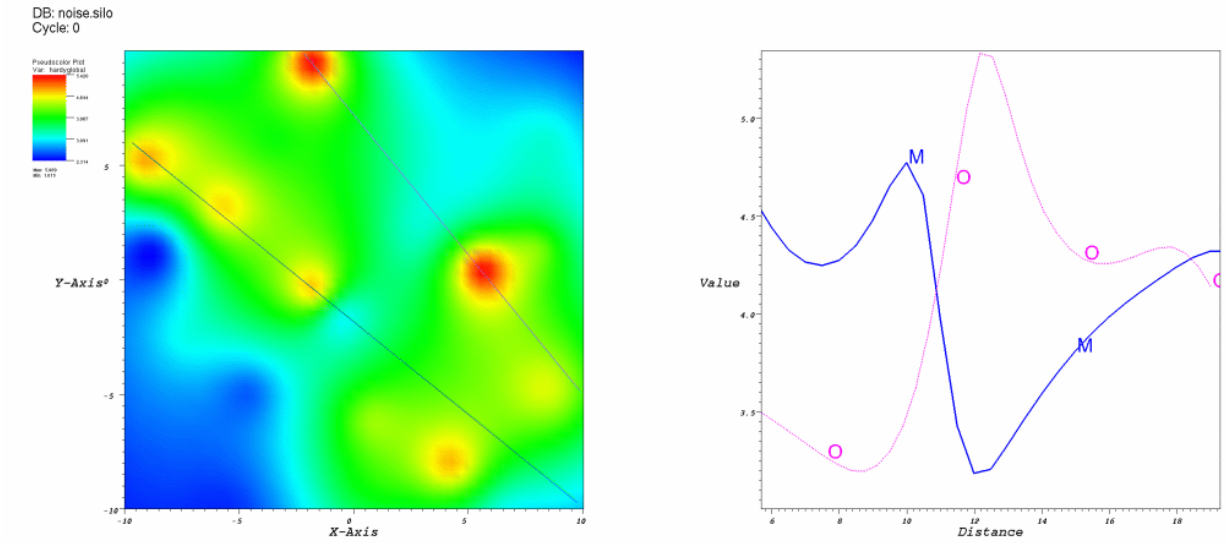


Fig. 4.306: Visualization windows with reference line and Curve plots

Curve plot

Curve plots are created by drawing reference lines. The visualization window must be in lineout mode before reference lines can be created. You can create a reference line by positioning the mouse over the first point of interest, clicking the left mouse button and then moving the mouse, while pressing the left mouse button, and releasing the mouse over the second endpoint. Releasing the mouse button creates a reference line along the path that was drawn with the mouse. When you draw a reference line, you cause a Curve plot of the data along the reference line to appear in another visualization window. If another visualization window is not available, VisIt opens a new one before creating the Curve plot. The Curve plot in the second window can be modified by setting the active window to the visualization window that contains the Curve plots.

See *Curve Plot* for information on changing the Curve plot's appearance.

Saving curves

Once a curve has been generated, it can be saved to a curve file. A curve file is an ASCII text file that contains the X-Y pairs that make up the curve and it is useful for exporting curve data to other curve analysis programs. To save a curve, make sure you first set the active window to the visualization window that contains the curve. Next, save the window using the *curve* file format. All of the curves in the visualization window are saved to the specified curve file.

Lineout Operator

The Curve plot uses the Lineout operator to extract data from a database along a linear path. The Lineout operator is not generally available since curves are created only through reference lines and not the **Plot menu**. Still, once a curve has been created using the Lineout operator, certain attributes of the Lineout can be modified. Note that when you modify the Lineout attributes, it is best to turn off the **Apply operators to all plots** check box in the **Main Window** so that all curves do not get the same set of Lineout operator attributes.

There are two factors that affect how the interpolation along the line is performed. These are the centering of the variable and the lineout sampling method. There are two types of centering and two types of sampling. The following sections will go into detail for the four cases.

Zonal variables are constant within a cell and a lineout would be expected to be a step function as the line moves from cell to cell.

All the images associated with the examples can be generated with the script lineout.py.

Zonal centering with sampling

In the case of sampling, the step function will become more and more apparent as the number of sample points increases.

In the example below there are only 12 samples points and the step function is only somewhat apparent, since the number of sample points within a cell ranges between one and three.

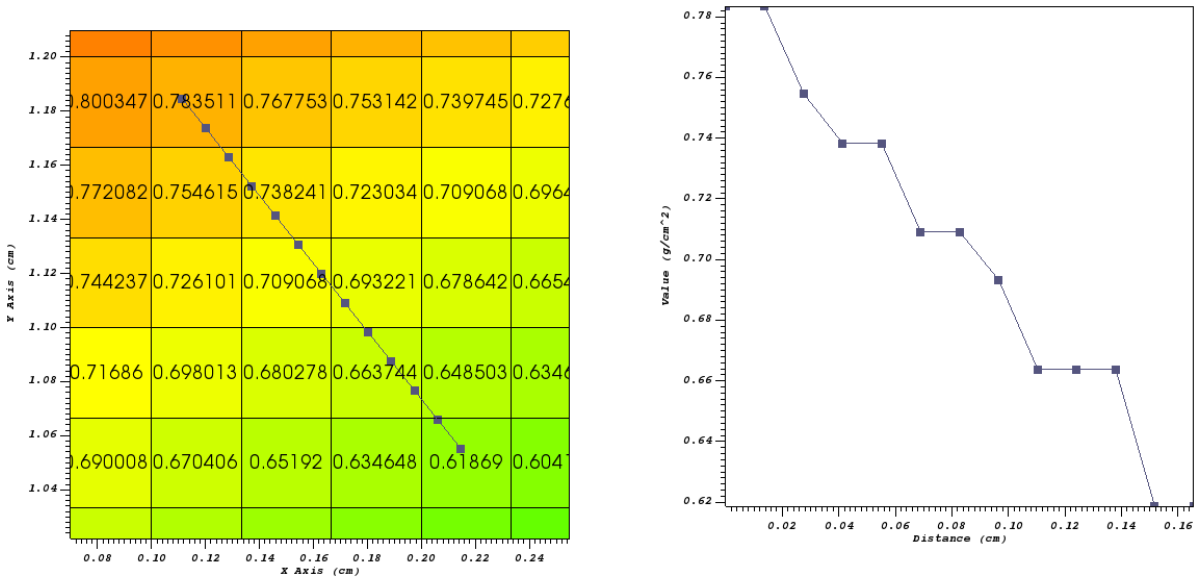


Fig. 4.307: A zonal variable with relatively few sample points.

In the example below there are 60 sample points and the step function is quite apparent.

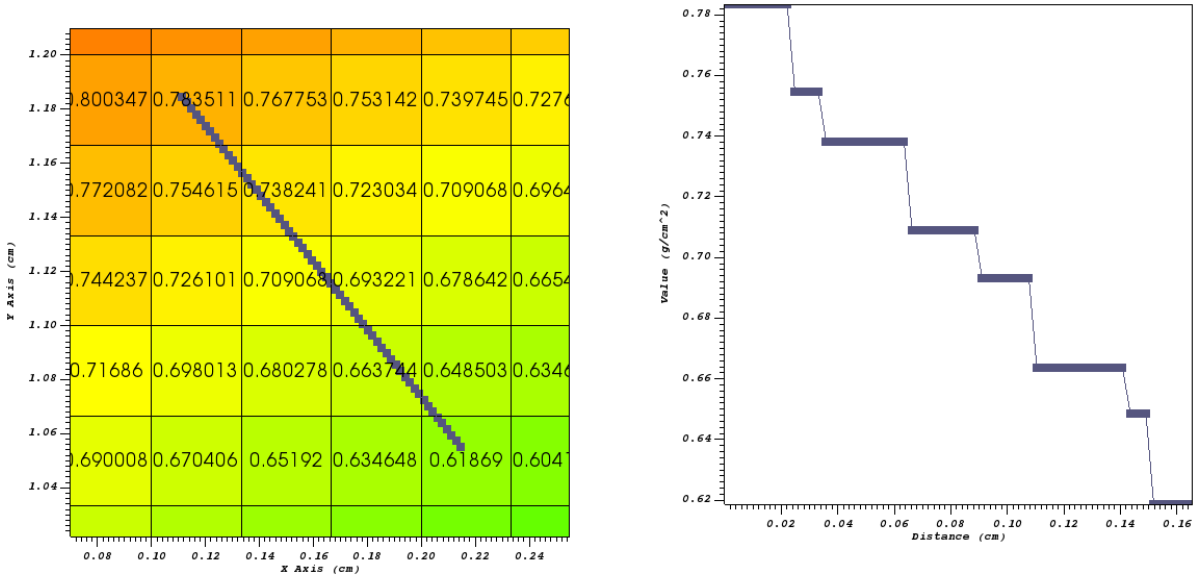


Fig. 4.308: A zonal variable with a large number of sample points.

Zonal centering without sampling

In the case of non-sampling, the sample points are chosen where the line intersects cell boundaries, which are lines in 2D and faces in 3D. The first point of the line has the zonal value of the cell it is within and the remaining points have the value of the cell the line is about to enter. In this case the step function nature of the variable is completely lost.

In the example below the sample points are placed based on where the line intersects the edges of the cells. The step function nature of the variable is completely lost and the line looks smoother than the sampled case.

Nodal variables vary linearly within a cell. Using sampling produces high quality results as long as the number of sample points is chosen such that all the cells along the line contain at least one sample point. Using non sampling tends to produce poor results based on its interpolation method (described below) and may result in jagged lines, even for smoothly varying functions.

Nodal centering with sampling

In the example below the 12 samples points do a good job of capturing the data along the line since all the cells are sampled at least once.

Increasing the number of sample points in this case doesn't change the shape of the curve.

Nodal centering without sampling

In the example below the sample points are placed based on where the line intersects the edges of the cells. The first point of the line has the average of the nodes of the cell that the point is within and the remaining points have the value of the average of the nodes of the cell the line is about to enter. This can lead to a jagged line even for a smoothly varying function.

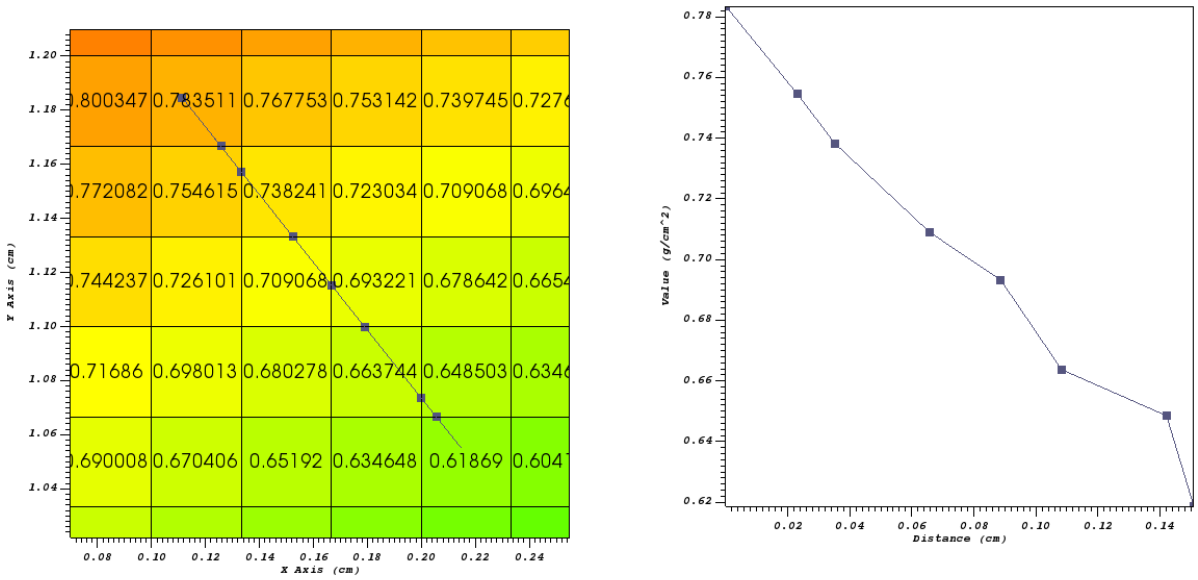


Fig. 4.309: A zonal variable without sampling.

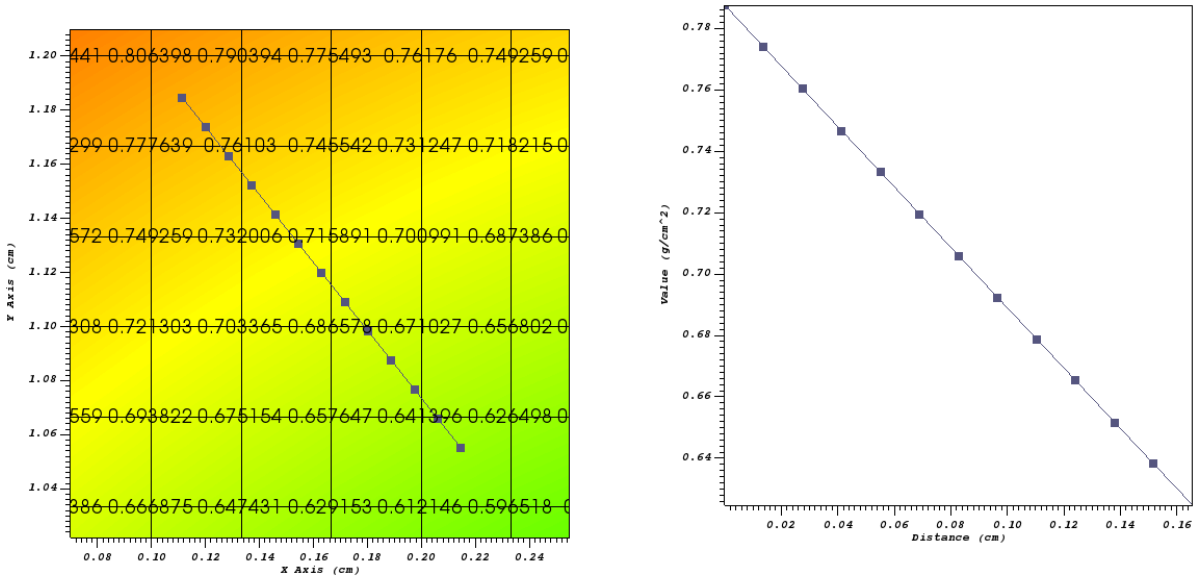


Fig. 4.310: A nodal variable with relatively few sample points.

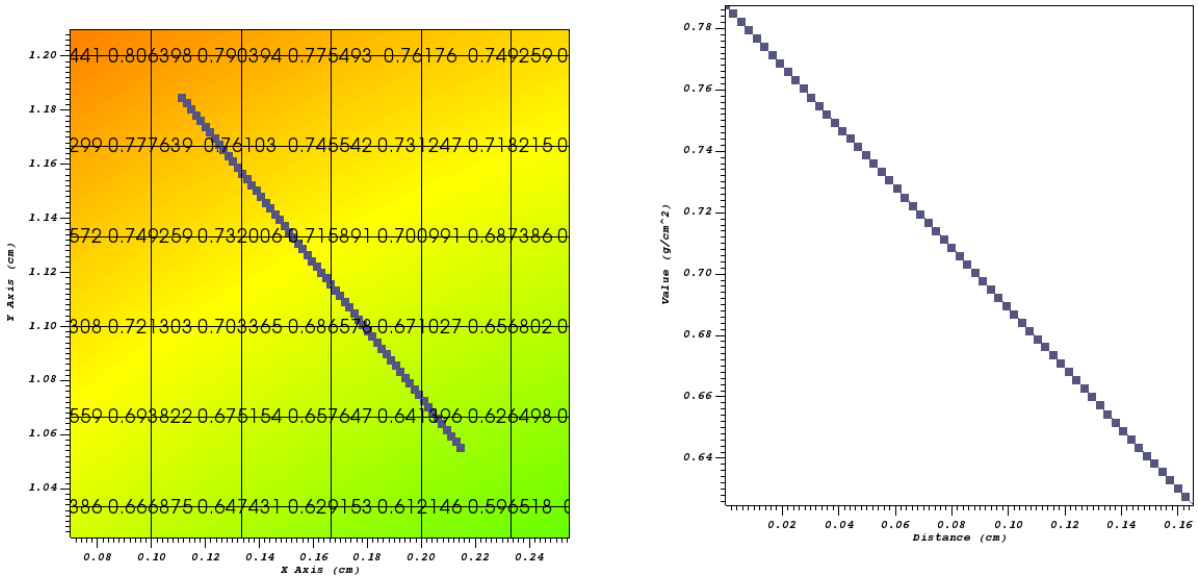


Fig. 4.311: A nodal variable with many sample points.

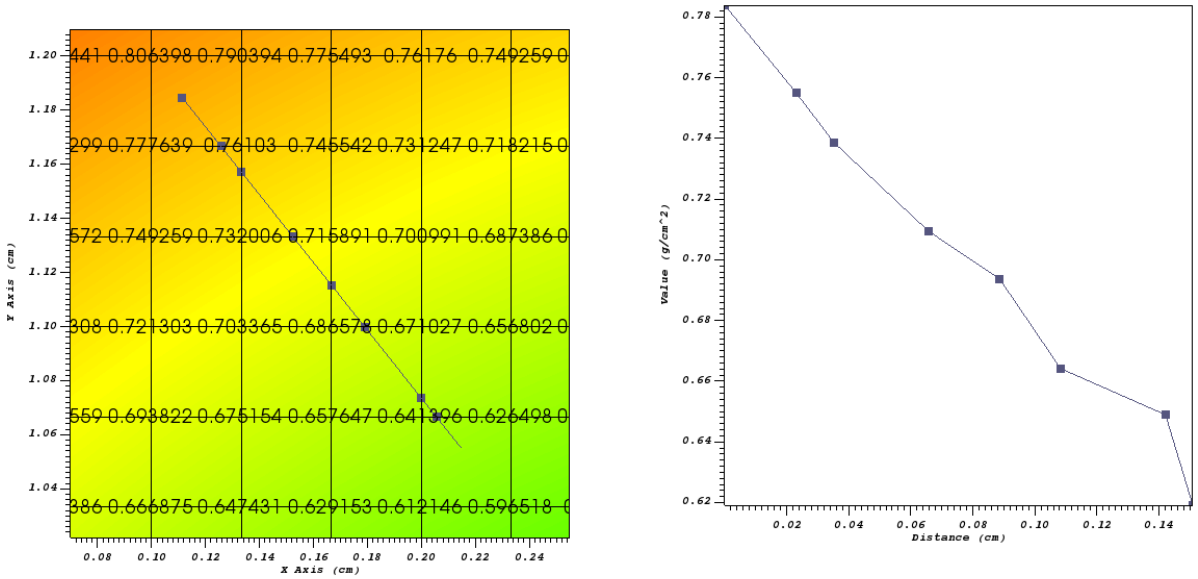


Fig. 4.312: A nodal variable without sampling.

Further exploring the Linout operator

The following script was used to generate 6 images above and can be used to further understand the behavior of the Lineout operator.

```
import math
import time

def create_images(sampling, n_samples, var):
    if (sampling == 1):
        save_name = "rect2d_%s_%d_lineout_sampled" % (var, n_samples)
        curve1_name = "rect2d_%s_%d_lineout_sampled.curve" % (var, n_samples)
        curve2_name = "rect2d_%s_%d_refline_sampled.curve" % (var, n_samples)
        image_name = "rect2d_%s_%d_pc_sampled" % (var, n_samples)
    else:
        save_name = "rect2d_%s_lineout_nonsampled" % var
        curve1_name = "rect2d_%s_lineout_nonsampled.curve" % var
        curve2_name = "rect2d_%s_refline_nonsampled.curve" % var
        image_name = "rect2d_%s_pc_nonsampled" % var

    #
    # Open the database to make the lineouts from.
    #
    OpenDatabase("rect2d.silo")

    #
    # Turn off extraneous annotations.
    #
    annot = AnnotationAttributes()
    annot.userInfoFlag = 0
    annot.databaseInfoFlag = 0
    annot.timeInfoFlag = 0
    annot.legendInfoFlag = 0
    SetAnnotationAttributes(annot)

    #
    # Create the lineout and do the lineout.
    #
    AddPlot("Mesh", "quadmesh2d")
    AddPlot("Pseudocolor", var)
    AddPlot("Label", var)
    labelAtts = LabelAttributes()
    labelAtts.numberOfLabels = 400
    SetPlotOptions(labelAtts)
    DrawPlots()
    view2D = View2DAttributes()
    view2D.windowCoords = (0.070, 0.255, 1.022, 1.210)
    view2D.viewportCoords = (0.15, 0.95, 0.1, 0.95)
    SetView2D(view2D)
    Lineout(start_point=(0.11137, 1.18468), end_point=(0.21461, 1.05520), use_
↪sampling=sampling, num_samples=n_samples)

    #
    # Go to the lineout window, save the image, save the curve and create
    # a reference line with the sample points from the saved curve.
    #
    SetActiveWindow(2)
```

(continues on next page)

(continued from previous page)

```
    SetAnnotationAttributes(annot)
    curveAtts = CurveAttributes()
    curveAtts.showPoints = 1
    curveAtts.pointSize = 8
    curveAtts.showLegend = 0
    curveAtts.showLabels = 0
    curveAtts.curveColorSource = curveAtts.Custom
    curveAtts.curveColor = (85, 85, 127, 255)
    SetPlotOptions(curveAtts)
    saveAtts = SaveWindowAttributes()
    saveAtts.fileName = save_name
    saveAtts.family = 0
    saveAtts.format = saveAtts.CURVE
    SetSaveWindowAttributes(saveAtts)
    SaveWindow()
    saveAtts.width = 600
    saveAtts.height = 600
    saveAtts.screenCapture = 0
    saveAtts.resConstraint = saveAtts.NoConstraint
    saveAtts.format = saveAtts.PNG
    SetSaveWindowAttributes(saveAtts)
    SaveWindow()

    #
    # Create a reference line with the sampled point from the saved curve
    # to overlay on the pseudocolor plot.
    #
    time.sleep(1)

    file1 = open(curve1_name, "r")
    file2 = open(curve2_name, "w")

    x1 = 0.11137
    y1 = 1.18468
    x2 = 0.21461
    y2 = 1.05520
    dx = x2 - x1
    dy = y2 - y1
    len = math.sqrt(dx * dx + dy * dy)
    dx = dx / len
    dy = dy / len
    slope = dy / dx

    line = file1.readline()
    line = file1.readline()
    file2.write("# refline\n")
    while line:
        vals = line.split()
        dist = float(vals[0])
        val = float(vals[1])
        x = x1 + (dist / len) * (x2 - x1)
        y = y1 + (dist / len) * (y2 - y1)
        file2.write("%g %g\n" % (x, y))
        line = file1.readline()

    file1.close()
    file2.close()
```

(continues on next page)

(continued from previous page)

```
    time.sleep(1)

    #
    # Add the reference line to the pseudocolor plot.
    #
    SetActiveWindow(1)
    OpenDatabase(curve2_name)
    AddPlot("Curve", "refline")
    DrawPlots()
    SetPlotOptions(curveAtts)
    saveAtts.fileName = image_name
    SetSaveWindowAttributes(saveAtts)
    SaveWindow()

    #
    # Clean up.
    #
    DeleteAllPlots()
    SetActiveWindow(2)
    DeleteAllPlots()
    SetActiveWindow(1)
    CloseDatabase("rect2d.silo")
    CloseDatabase(curve2_name)

OpenComputeEngine("localhost", ("-np", "1"))

DefineScalarExpression("d2", "recenter(<d>, \"nodal\")")

create_images(1, 12, "d")
create_images(1, 60, "d")
create_images(0, 12, "d")
create_images(1, 12, "d2")
create_images(1, 60, "d2")
create_images(0, 12, "d2")

quit()
```

Setting lineout endpoints

You can modify the line endpoints by typing new coordinates into the **Point 1** or **Point 2** text fields of the **Lineout attributes** window (Figure 4.313). Each endpoint is a 3D coordinate that is specified by three space-separated floating point numbers. If you are performing a Lineout operation on 2D data, you can set the value for the Z coordinate to zero.

Setting the number of lineout samples

The sampling is controlled with the **Use Sampling** toggle button and the **Samples** text field. The **Use Sampling** toggle button controls whether sampling is used and **Samples** is used to set the number of sample points when sampling.

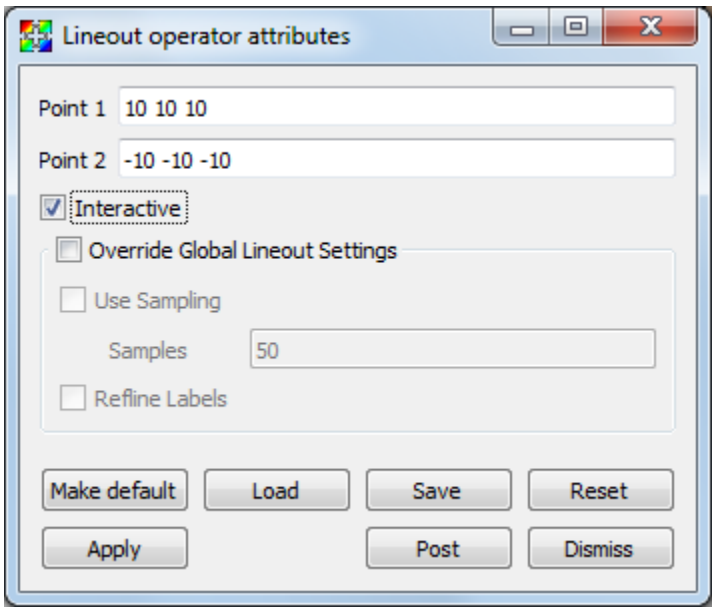


Fig. 4.313: Lineout attributes window

Interactive mode

When the **Interactive** check box is checked, changes to the Lineout operator can be made by using the **Line tool** available from the originating plot's visualization window Toolbar or Popup menu. *Interactive mode does not apply to lineouts created via the Curve plot's variable menu.*

To utilize the line tool to modify a Lineout curve, make the visualization window with the originating plot the active window. Choose the Line tool. It should be initialized with the endpoints of the reference line. Moving the tool will change the lineout. (*Note: Due to a current bug, the tool must be activated, deactivated, then activated a second time in order to be properly initialized with the Lineout's endpoint values.*) See *Interactive Tools* for more information on tool utilization.

Reference line labels

You can make the reference lines in the window that caused Curve plots to be generated to have labels by checking the Lineout operator's **Refline Labels** check box.

Lineout query

Performing a Lineout query requires an existing non-hidden plot in the active window. Choose **Lineout** from the **Query** window (available from the GUI's Controls dropdown menu). Set start and end points (similar to Setting lineout endpoints). Lineout query is the only Lineout method that allows you to create curves for multiple variables. Simply select the desired variables from the **Variables** dropdown menu. *Default* means the variable as plotted in the currently active plot. A lineout curve will be generated for each variable, plotted along the same reference line. Each curve will have its own color. The **Use Sampling** and **Sample Points** option is the same as before.

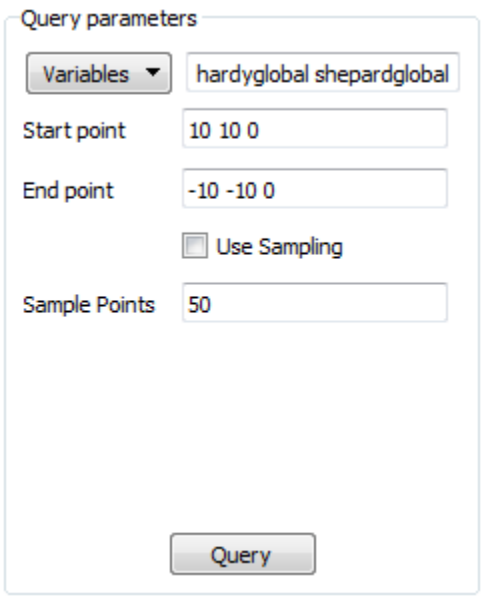


Fig. 4.314: Lineout query's parameters window

Lineout via Curve plot variable menu

With this method, Lineout is considered one of the *Operators that Generate New Variables*. That means you can use it without first generating a plot of the data from which you wish to extract the lineout. To create a Lineout in this manner, open your database, select Curve plot, then choose *operators/Lineout/<var-name>* from the Curve plot's variable menu as shown in Figure 4.315.

It is highly recommended that you modify the Lineout's endpoints before clicking draw, as the defaults will probably not be appropriate for your data.

Global lineout options

The **Lineout Options Window**, available by selecting **Lineout** from the **Controls** menu in the **Main Window** contains *global* lineout options. They are *global* in the sense that they will apply to *all* future lineouts. The **Lineout Options Window** has controls for choosing the destination window of the lineout curve plots, as well as settings for how changes to the originating plot affect the lineout curve plot. Modifying these options will only apply to future lineouts, not lineouts already created.

Lineout destination window

By default, VisIt will place all lineout curves in the same window. It will use the first unused open window or create one if one does not yet exist. You can override this behavior for future lineouts by unchecking the **Use 1st unused window** checkbox, and typing a window number into the **Window #** text box.

Freeze In Time

If the plot that originated the Lineout curve was from a time-varying database, the curve can be advanced in time using the animation controls for the window containing the lineout curve. If you would rather the lineout be frozen at the

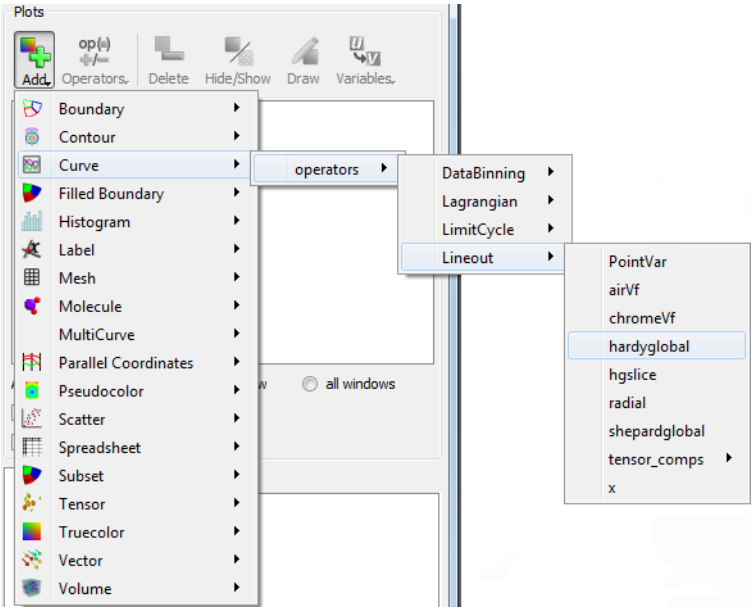


Fig. 4.315: Choosing lineout from the Curve plot's variable menu

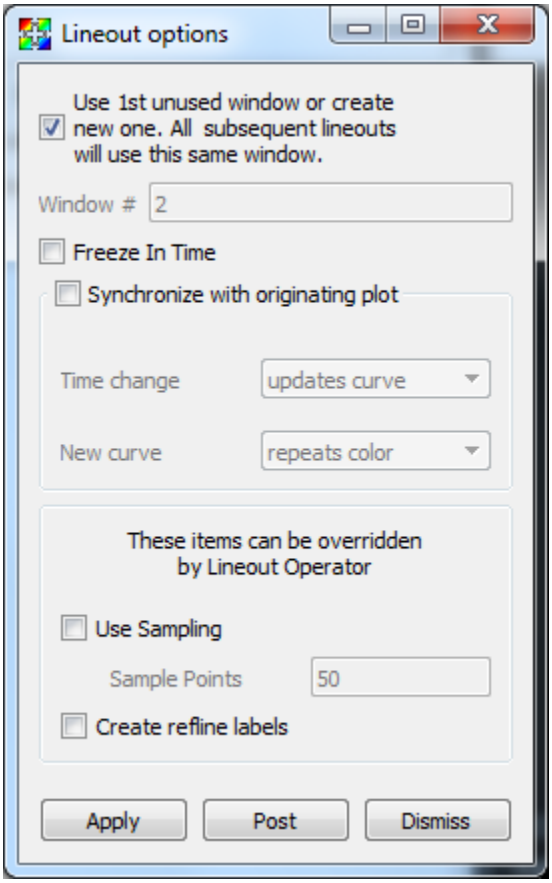


Fig. 4.316: Lineout Options Window

timestep from which it was taken, check the **Freeze in Time** option. This will also disable the ability to synchronize the lineout curve with its originating plot.

Synchronous lineout

Normally when you perform a lineout operation, the Curve plot that results from the lineout operation is in no way connected to the plots in the window that originated the Curve plot. If you want variable or time state changes made to the originating plots to also affect the Curve plots that were created via lineout, click the **Synchronize with originating plot** check box in the **Lineout Options Window** (see Figure 4.316).

With this option selected, any change to the variable in the plot that originated the lineout, will update the lineout to reflect the new variable's data. When you change time states for the plot that originated the lineout, the lineout will update to reflect the data at the new time state.

To make VisIt create a new Curve plot for the lineout instead of updating when you change time states in the originating plot, change the **Time change** behavior in the **Lineout Options Window** from **updates curve** to **creates new curve**. VisIt will then put a new curve in the lineout destination window each time you advance to a new time state, resulting in many Curve plots (see Figure 4.317). By default, VisIt will make all of the related Curve plots be the same color. You can override this behavior by selecting **creates new color** instead of **repeats color** from the **New curve** combo box.

Synchronization does not apply to lineout curves created via the Curve plot variable menu, as this type of lineout does not have an originating plot.

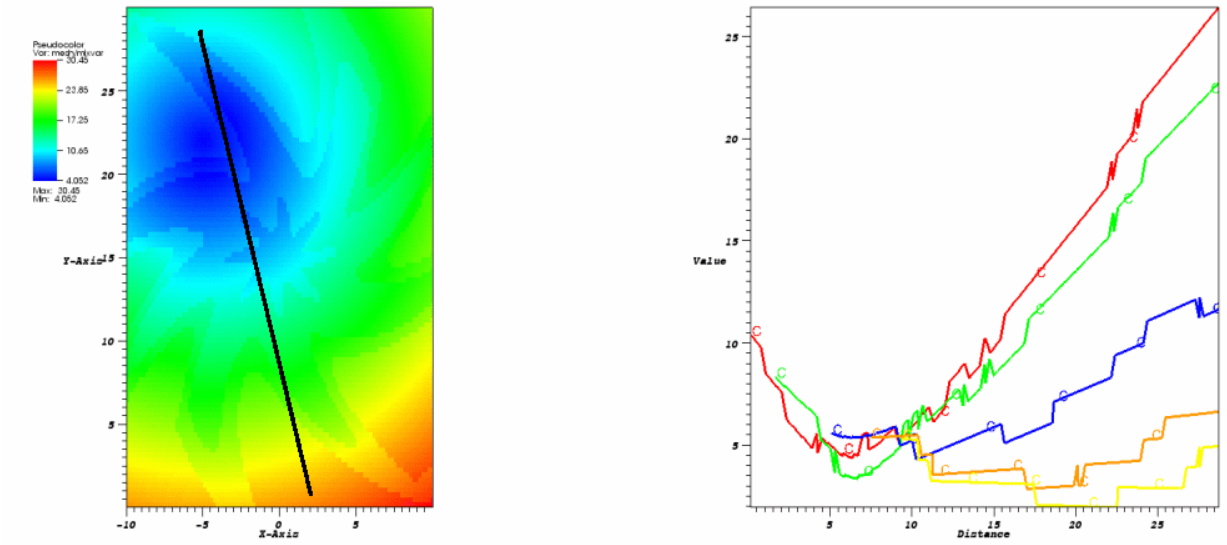


Fig. 4.317: Dynamic lineout can be used to create curves for multiple time states

Sampling and Refline labels

These options are the same as described for individual lineouts. Use these options when you want your choices to apply to *all* lineouts.

4.8.6 LineSampler

One dimensional curves, created using data from 2D or 3D plots, are popular for analyzing data because they are simple to compare. The LineSampler operator is similar to the Lineout tool in this regard except that it will create multiple curves instead of just a single curve.

As an operator, the user can define a series of "arrays" (e.g. planes) that consists of one or more "channels" over which the data is sampled. For each array the orientation of the plane can be defined. Whereas for each channel its orientation within the plane and the sampling type and spacing can be defined. For instance, the sampling can be a series of lines through the data, or a series of points integrated over time.

Once defined, the operator will produce the appropriate geometry and sampling regardless of the window dimension. That is for a 3D window each channel will be drawn showing its 3D position. Similarly for a 2D window for the Y=0 or the Phi=0 plane. For 1D a window the results will be displayed as a curve plot in the same way a Lineout curve plot would be drawn.

The genesis of the Line Sampler operator is from plasma physics and the synthetic diagnostics performed in fusion simulations in order to compare against experimental results. As such, the nomenclature is based on this usage.

LineSampler operator

Before using the LineSampler operator in main plot window set the "Apply to" to "active window" and uncheck "Apply operators to all plots." This setting is critical for setting up all of the plots that will use the LineSampler operator. Once all of the plots are set up, then, and only then should the settings be changed.

Create a 3D view of the data showing the channels - First, create a translucent Pseudocolor plot of the data for reference. Next, clone the plot and add the LineSampler Operator to the plot. Open the LineSampler operator attribute window.

Main tab

Select the "Main" tab, Figure 4.318 to set up the attributes accordingly.

1. Mesh geometry, which can be Cartesian, Cylindrical or Toroidal, determines how the sampling will be done. The most notable difference between Cylindrical and Toroidal is coordinate ordering: (r, phi, z) vs (r, z, phi).
2. The user can set up an array configuration specifying the geometry or manually read in a channel configuration file. The "Geometry" and "List" tabs reflect those respective choices.
3. The boundary of the sampling can be based on the data file or using a manually specified wall file. The wall file reflects the boundary in a 2D slice of the data in the Y=0 or the Phi=0 plane. Any sampling outside of the boundary will be disregarded, that is the sample ray will be clipped.

   Note currently the wall file format is specific to D3D fusion files. The coordinates are displayed in the text box.
4. When using the LineSampler operator, in main plot window one will need to set the "Apply to" to "all windows" and check "Apply operators to all plots" so that all of the parameters are propagated to all of the windows. However, the user may want to have a different "Instance" of the operator in order to compare different sampling configurations. By setting the "Instance" to 'A' for the first, and 'B' for the second the attributes will be propagated appropriately to the plots in other windows when using the "Apply to all windows" option.

Geometry Tab

The Geometry tab, Figure 4.319 is active only if the "Array configuration" is set to "Geometry." This tab allows the user to manually define one or more arrays with one or more channels.

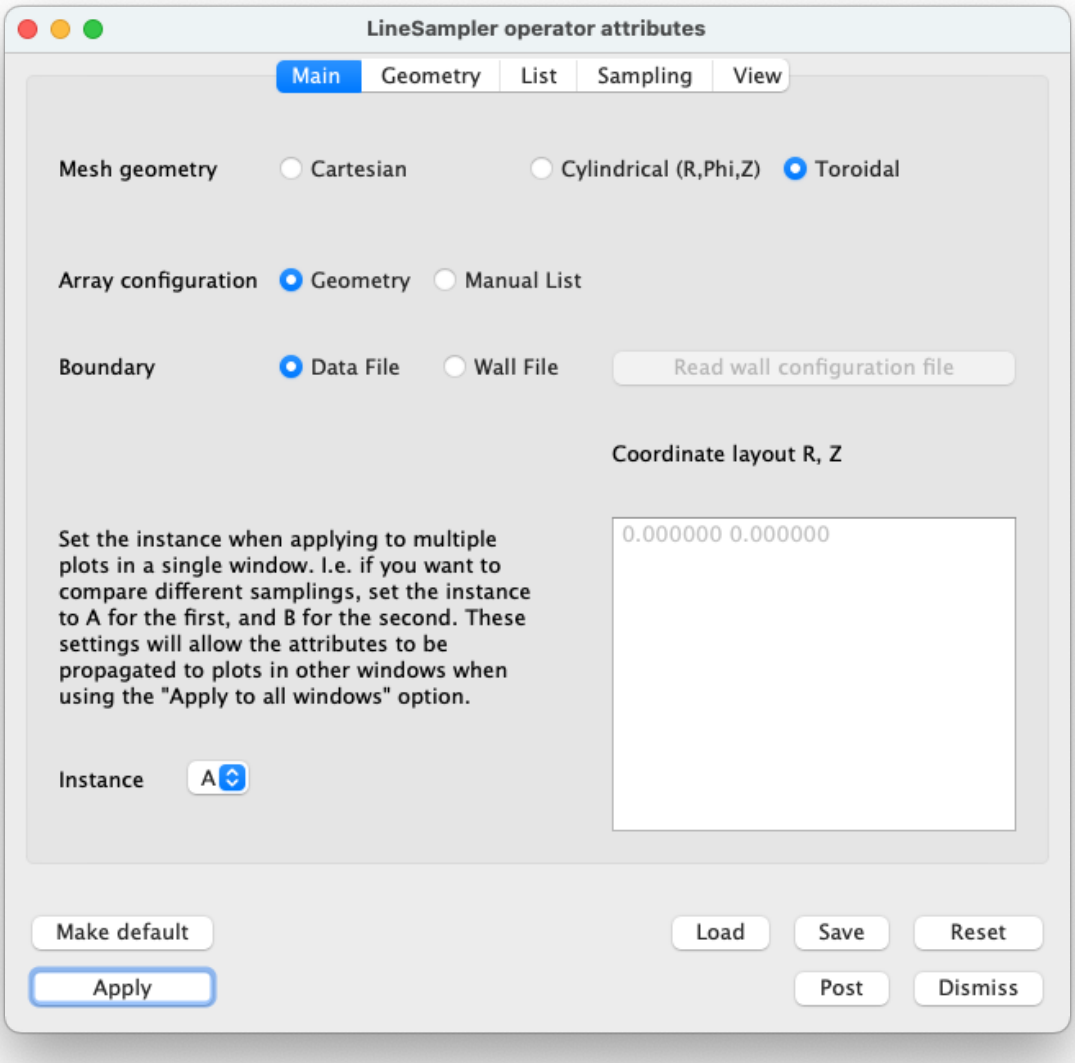


Fig. 4.318: LineSampler Main Tab

1. The number of arrays to be created. Each array consists of multiple sample channels all in a 2D plane. Multiple arrays can be defined in multiple planes.
2. Depending on the "Mesh Geometry" the Y distance or toroidal angle (in degrees) between arrays.
3. The projection of the channels, Divergent, Parallel, or a Grid. The user can select one of three projections for the channels. For each projection one can define the number of channels and their relative spacing.

   Divergent tab - For an array divergent channels one selects:

   a. The number of divergent channels.

   b. The relative angle between each channel.

   Parallel tab - For an array of parallel channels one selects:

   a. The number of parallel channels.

   b. The relative distance between each channel.

   Grid tab - For a grid of parallel channels one selects:

   a. The number of channels per column.

   b. The offset between each channel.

   c. The number of rows.

   d. The offset between rows.

For the remaining geometery consult the reference image showing the attributes.

4. Location of the origin for a divergent array. For a parallel and grid array the channels will be centered around the origin.
5. The array is assumed to be in the Y=0 or the Phi=0 plane as such one needs to select the axis direction from the origin which will be X or Z (Cartesian) or R or Z (cylindrical or toroidal). The channel sampling will start at the origin and proceed in hte negative axis direction.
6. The array plane may be tilted and well as be rotated. Depending on the mesh geometry these are described as a series of offsets (Cartesian) or angles (cylindrical or toroidal) and tilts which defines a transform to the array plane.
7. The Y offset / Toroidal angle - add an offset to each array.
8. Flip toroidal angle - flips the toroidal angle by negating it.

List Tab

The List tab, Figure 4.320 is active only if the "Array configuration" is set to "List." This tab allows the user to read a channel configuration file which defines an array with one or more channels.

1. The number of arrays to be created. That is each channel configuration file is considered to be one array. Multiple arrays can be defined in multiple planes.
2. Depending on the "Mesh Geometry" the Y distance or toroidal angle between arrays.
3. Read channel configuration file - read a D3D fusion Soft Xray channel configuration file. Each point consists of the origin and an associated angle and is shown in the Channel list.

   Channel list - single click selects the channel, double click selects the channel for editing.

4. Add channel - add a new channel to the list
5. Delete channel - delete the selected channel

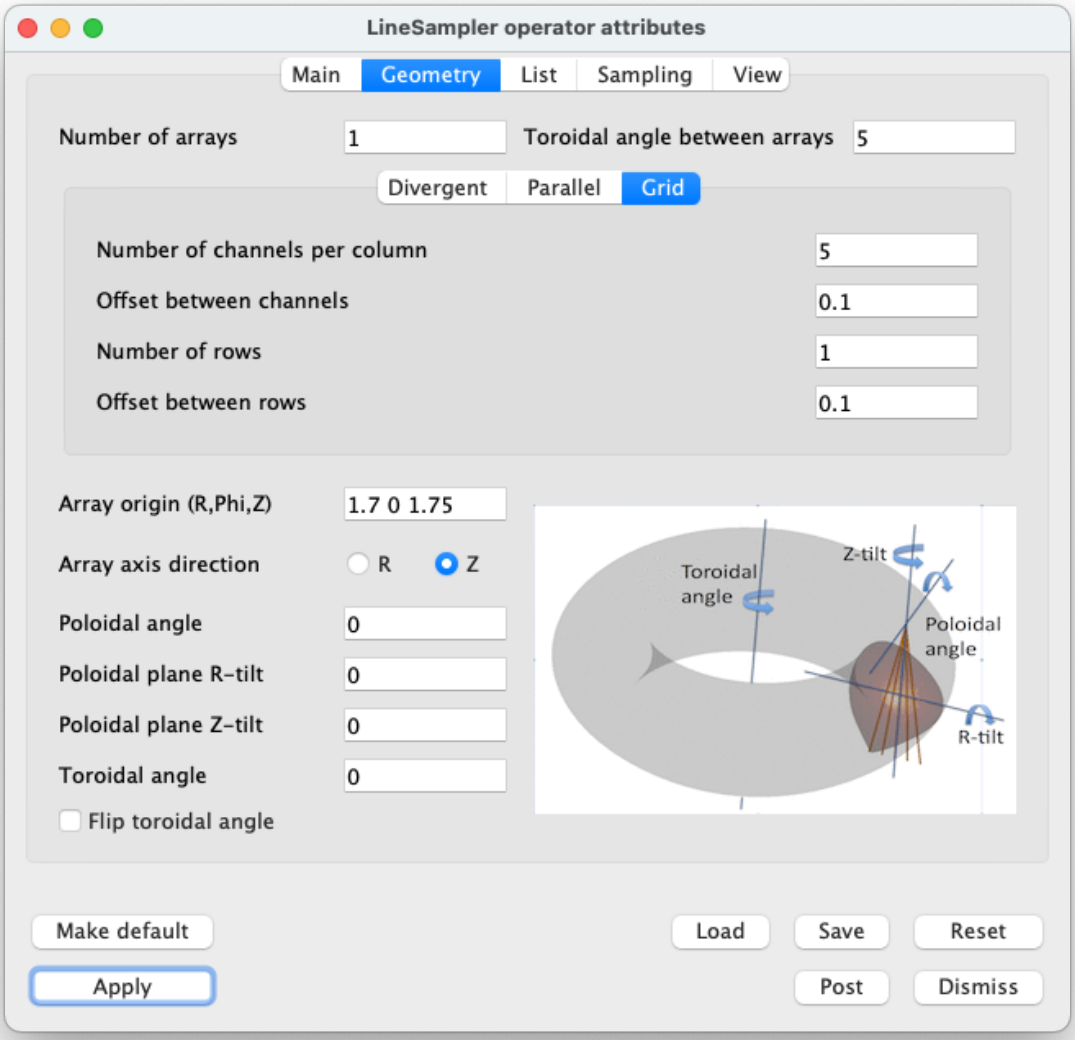


Fig. 4.319: LineSampler Geometry Tab

6. Delete all channels - delete all channels in the list
7. The Y offset / Toroidal angle - add an offset to each array.
8. Flip toroidal angle - flips the toroidal angle by negating it.

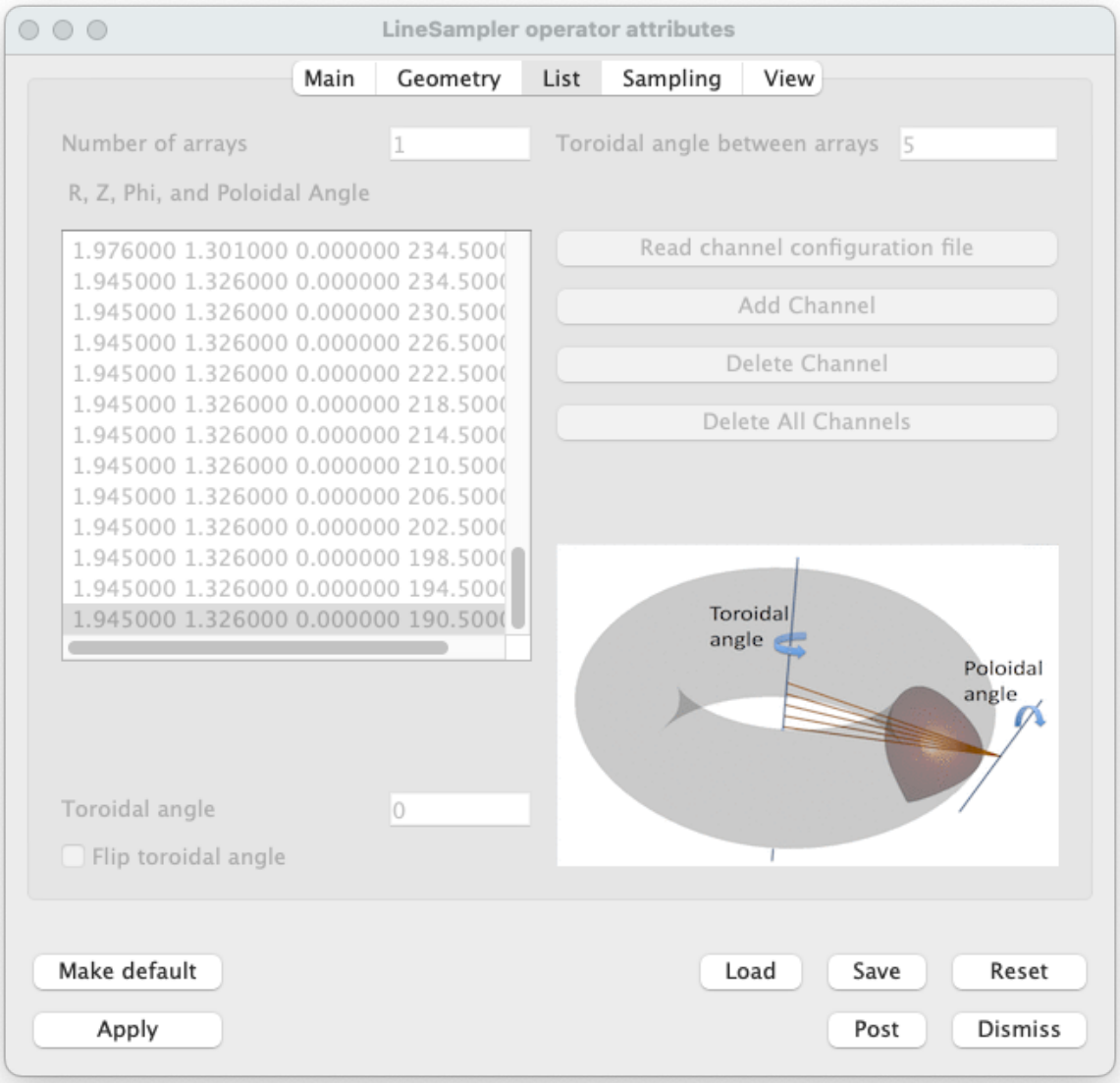


Fig. 4.320: LineSampler List Tab

Sampling Tab

The Sampling tab, Figure 4.321 sets up how each channel will be sampled.

Geometry

1. Channel geometry - Currently the sampling geometry is limited to a point or along a line. Future work includes cylindrical and cone sampling geometries.

2. Linear sampling distance - Sample along each channel using the distance specified.
3. Sample volume - For each sample multiple it by a volume.
4. Channel radius - The radius of a channel that is described by a cylinder.
5. Sample profile - The sample profile of a channel that is described by a cylinder. Either a TopHat or Gaussian profile. If a Gaussian profile is selected the standard deviation may be given.
6. Cone divergence - For a cone the divergence of the channel.

Integration

7. Channel integration - When sampling one can sample along the channel recording each individual sample or integrate (sum) all of the sample values together.
8. Toroidal integration - When sampling toroidally one can sample along the circumference recording each sample or integrate all of the sample values together.

Toroidal angle sampling

9. Sample - When sampling toroidally one can sample relative to the start point or on an absolute basis.
10. Toroidal sample angle - The start, stop, and stride for toroidal sampling.

Time sampling

11. When sampling one can sample just the current time step or across multiple times steps which becomes the X axis.
12. Time step - The start, stop, and stride for time sampling.

View Tab

The View tab, Figure 4.322 sets attributes based on the dimension of the plot.

1. When associating the LineSampler operator with a specific plot, the operator needs to know the plot's view dimension in order to display the sample data correctly. Normally one would have three windows, 1D, 2D, and 3D. The Line Sampler operator would be active for the three plots in each window and one would individually set this attribute for each.
2. When checked, assures that when the operator attributes are updated that the view dimension is not updated to all plots. Should always be set to true.

When displaying the resulting sampling as a 1D plot various viewing parameters can be set.

3. Scale each channel's Y value.
4. For each channel offset the Y value, so that possibly overlapping channels are offset.
5. For each array offset the X value, so that possibly overlapping arrays are offset.
6. When sampling over time set the X axis to be either the Step, Time, or Cycle.
7. The "View geometry" can be restricted to being Points, Lines, or Surfaces. Normally each channel is drawn as a line. By setting the "View geometry" to "Points," the the actual sample points will be drawn.

   Future work includes cylindrical and cone sampling geometries. For these cases setting the "View Geometry" to "Lines" the centerline of the cone or cylinder would be drawn or if "Points" the actual sample points would be drawn.

Once all of the attributes are set one can apply and draw the plots for the 3D view, Figure 4.323). This view shows all of the arrays and their channels as a 3D view.

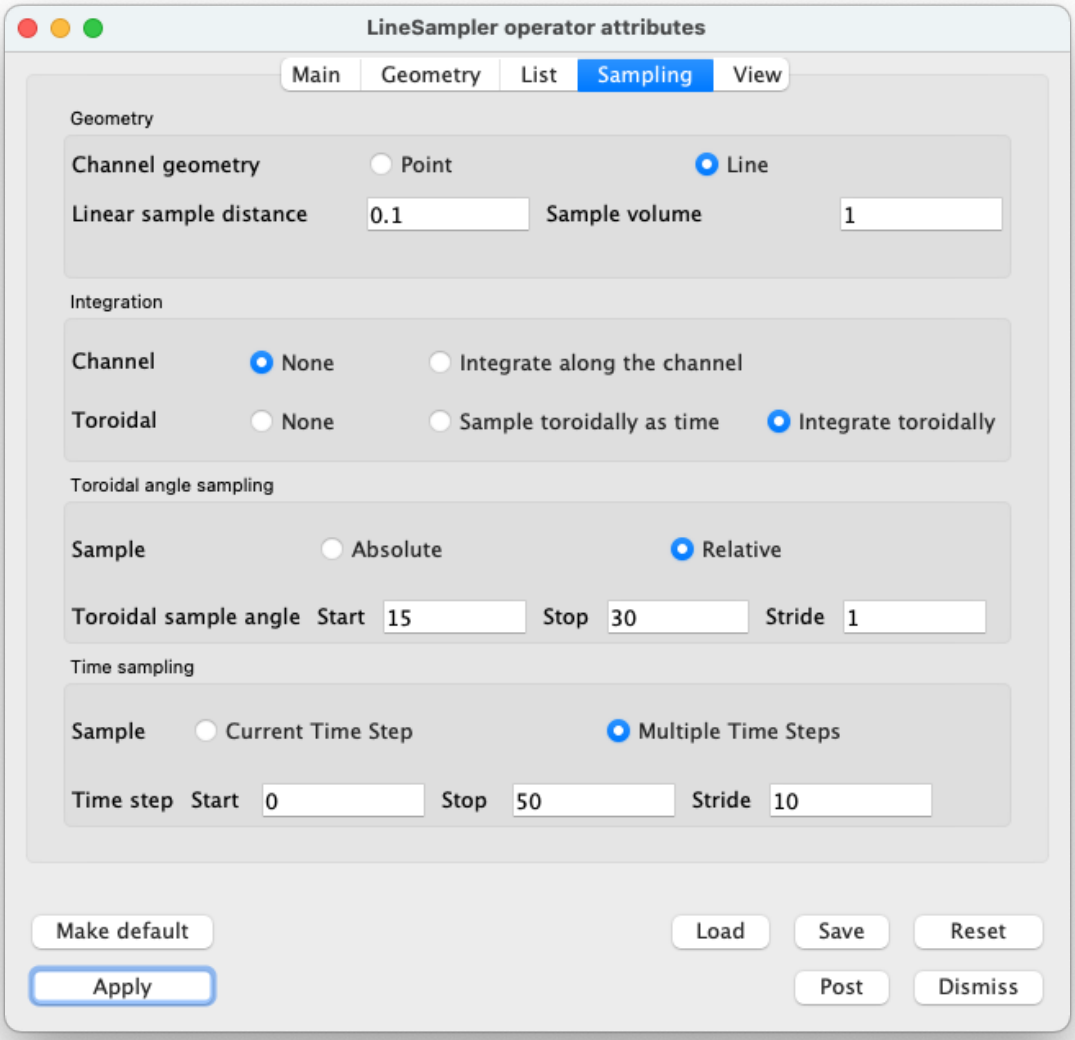


Fig. 4.321: LineSampler Sampling Tab

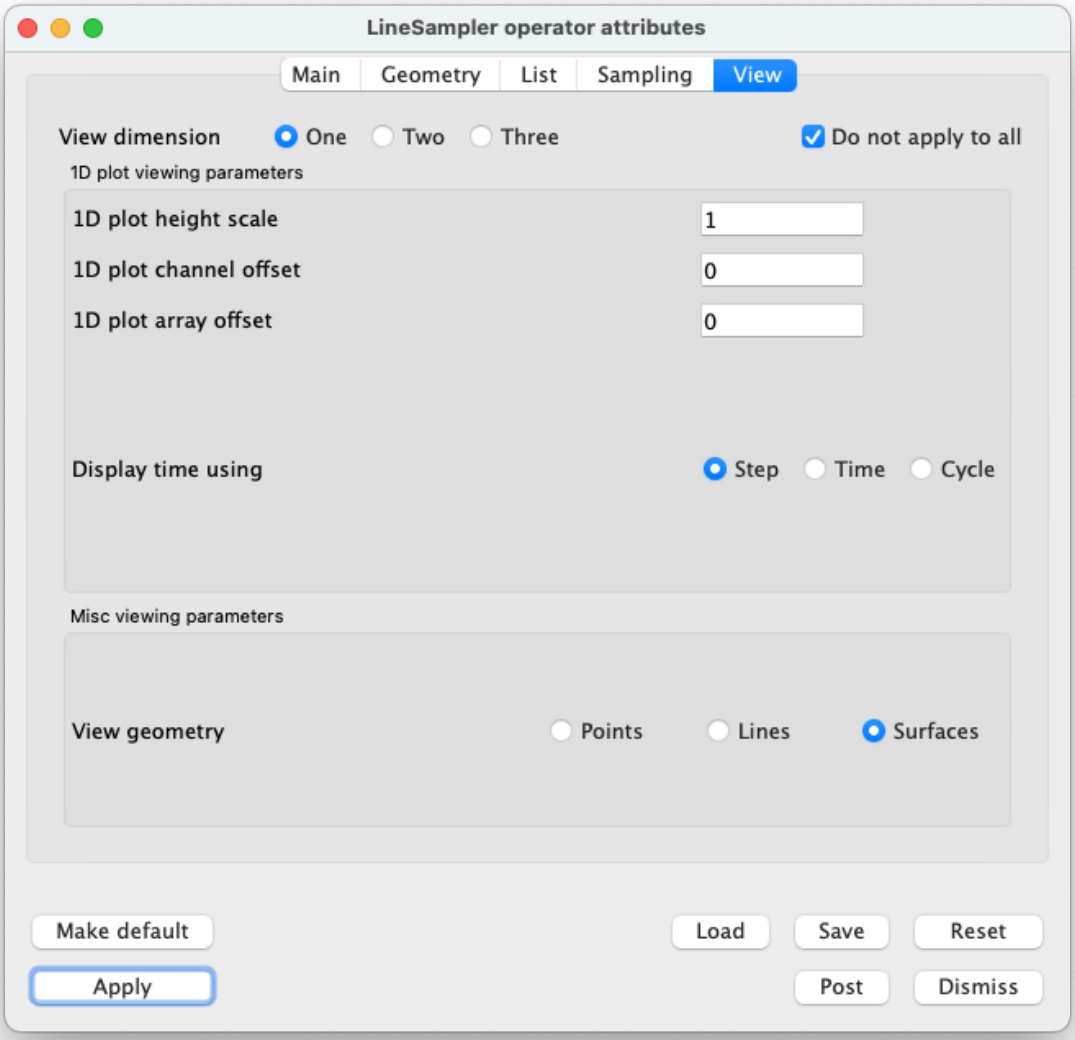


Fig. 4.322: LineSampler View Tab

Create a 2D view of the data showing the channels - Clone the 3D window and add a slice operator to the reference plot. Set the slice to be in the Y=0 or the Phi=0 plane and apply. This plane is considered to be the reference plane for the LineSampler operator and regardless of the Y or Phi of the "base" array it will be projected into this plane. The "base" array is the first array created, all other arrays are derived from it. As such, only it is shown in the 2D view.

For the plot with the LineSampler operator in the View tab set the View dimension to "Two." Apply and draw the plots, Figure 4.324). This view now shows the "base" array and each channel in it as a 2D view.

Create a 1D view of the channel values - Next clone the 2D window and delete the plot with the slice operator.

For the plot with the LineSampler operator in the View tab set the View dimension to "One." Apply and draw the plot Figure 4.325). This view now shows each channel from all of the arrays as series of 1D curves.

All this point all three plots with Linesampler operator are in sync. To keep them in sync in the main plot window set the "Apply to" to "all windows" and check "Apply operators to all plots." At this point if one changes any attribute in the LineSampler operator all of the plots will be updated. For instance change the number of channels and apply. All of the plots will be updated.

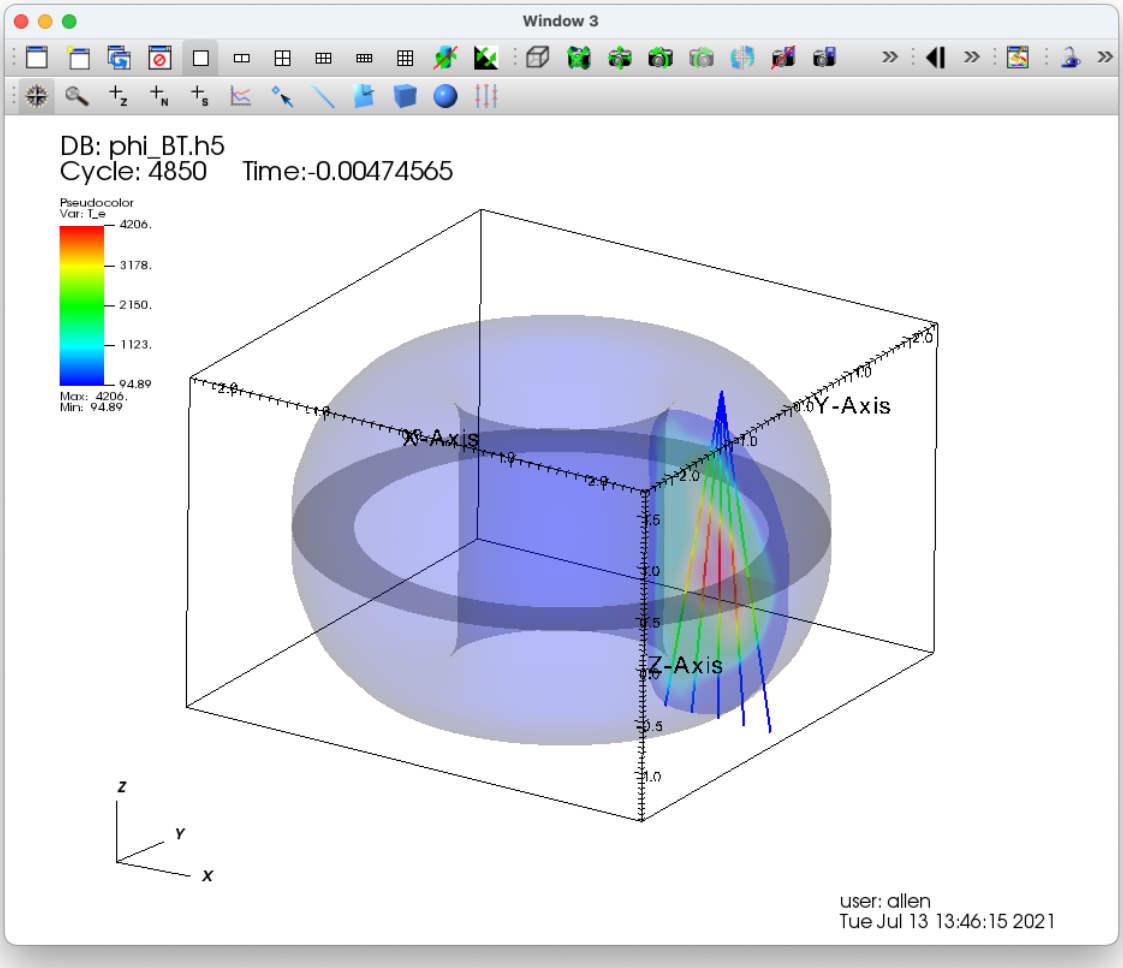


Fig. 4.323: LineSampler 3D view of toroidal data

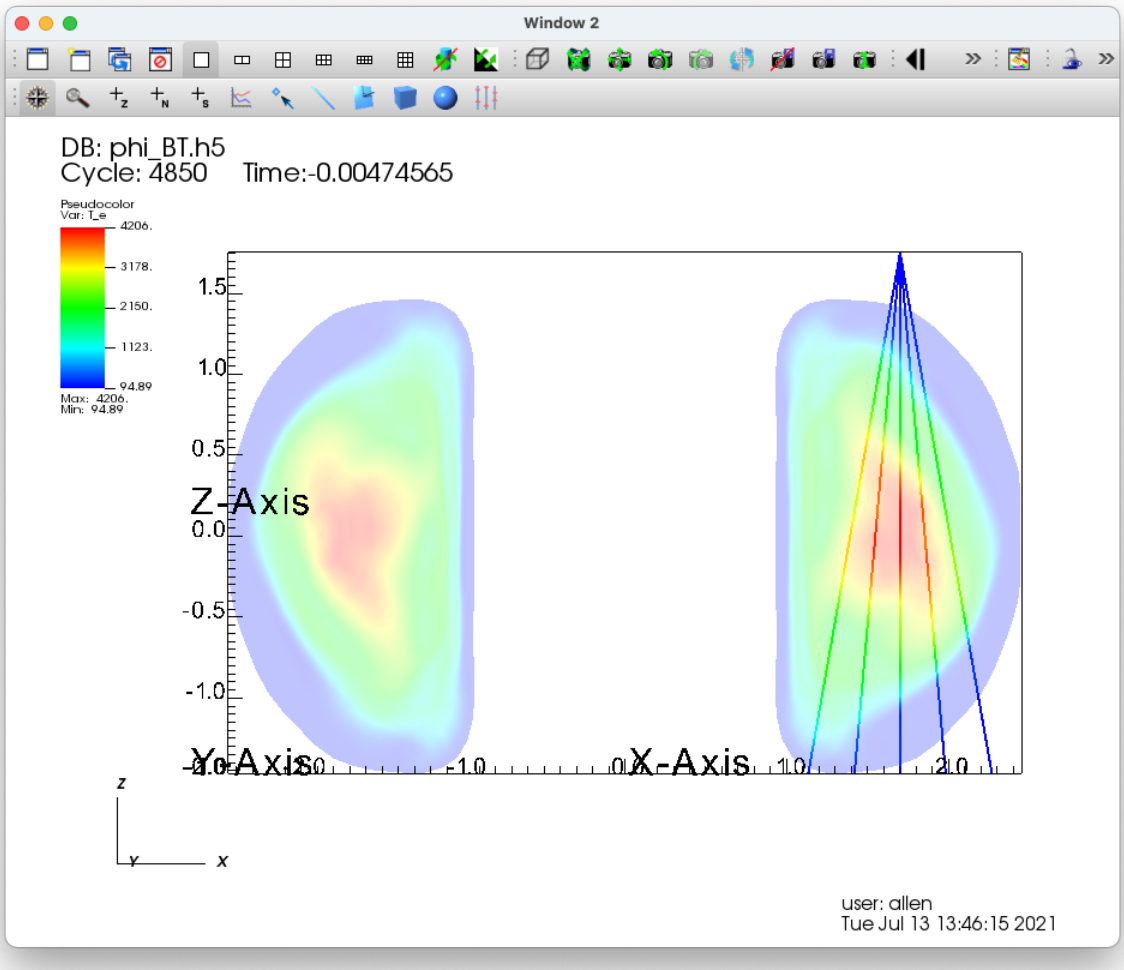


Fig. 4.324: LineSampler 2D view of toroidal data

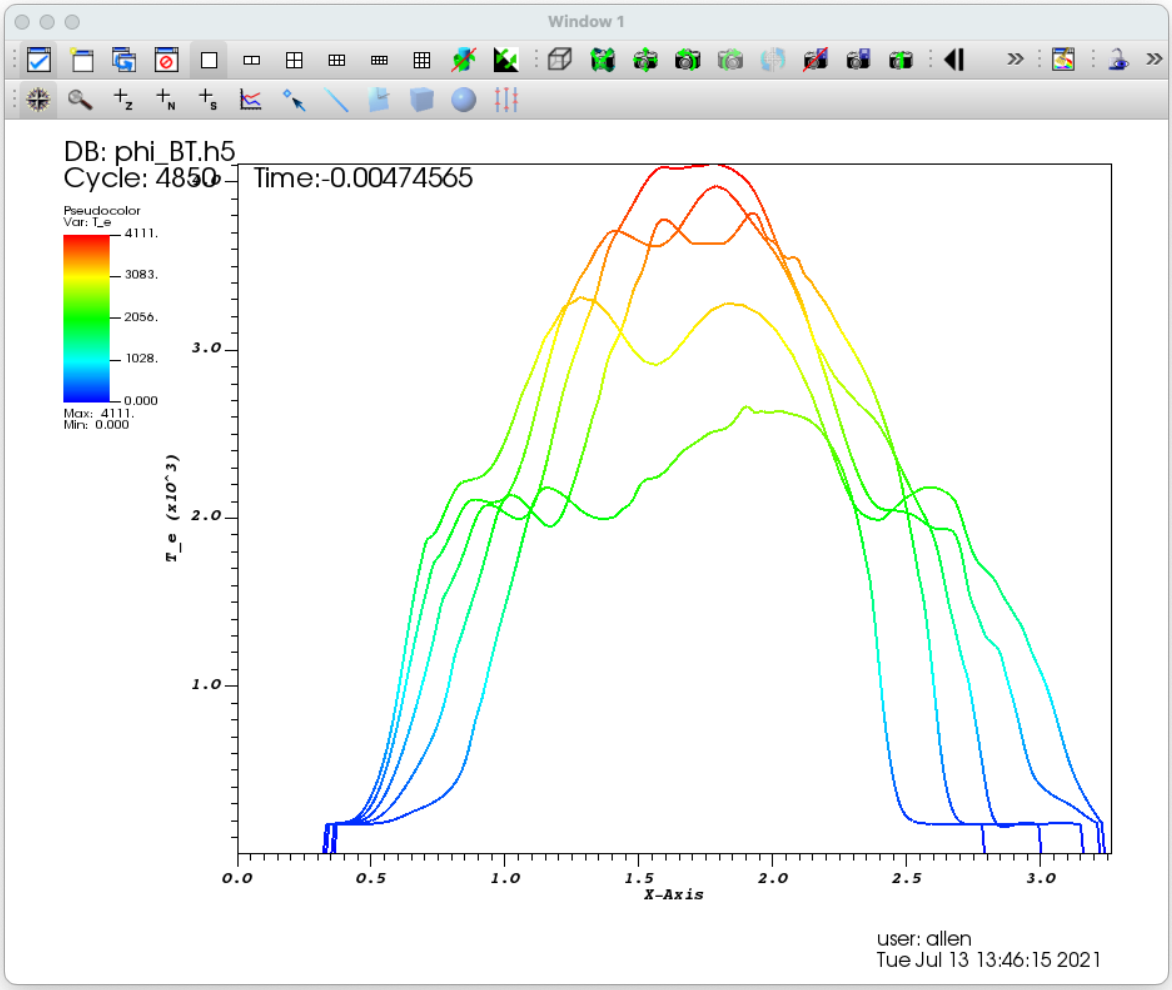


Fig. 4.325: LineSampler 1D view of toroidal data

4.8.7 Data-Level Comparisons Wizard

The data-level comparisons wizard facilitates creation of expressions that can be used when comparing fields on different meshes and/or in different databases. Such expressions are also known as *Cross-Mesh Field Evaluation (CMFE)* expressions because they effectively take a field defined on one mesh and *evaluate* it (e.g. map it) onto a new mesh. The data-level comparisons wizard is a very helpful alternative to entering CMFE expressions directly into the expression system manually.

These expressions involve the concepts of a *donor variable* and a *target mesh*. The donor variable is the variable to be mapped onto a new mesh. The target mesh is the mesh onto which the donor variable is to be mapped. In addition, the term *donor mesh* refers to the mesh upon which the donor variable is defined. Also, the target mesh is always interpreted as a mesh in the currently *active* database. Data-level comparison expressions (CMFEs) are always mapping data from *other* meshes, possibly in *other* databases onto a target mesh which is understood to be in the currently *active* database.

To start the wizard, go to Controls->Data-Level Comparisons... as shown in Figure 4.326.

This will open the the initial window where the user is asked to choose between a few basic varieties of CMFE expressions. These differ in the relative locations (e.g. which database) of the donor variable and target mesh.

1. Donor variable and target mesh are in the *same* database.
2. Donor variable and target mesh are from different time states of the *same* database.
3. Donor variable and target mesh are in wholly different databases.

Note: if you wish to create a CMFE that works properly across a time series with wholly different databases (3rd case above), the data-level comparisons wizard does not directly support that. However, you can use wizard to construct an *initial* CMFE expression and then edit it manually in the *Expression Window* to adjust it for a time series following the documentation on *donor variable syntax*.

If the user is unsure, selecting the last option is usually fine. There are some simplifications and maybe some small performance optimizations in the creation and evaluation of the expressions that can be made for the other cases. But, VisIt will operate fine even if those are not chosen. In the description that follows, we demonstrate only this selection but describe variations where necessary.

After selecting the variety of CMFE expression to create, the user is presented with the next wizard window to specify the target mesh and donor variables to be used in the expression.

The target mesh selection will present the user with a pull-down list of currently opened databases with the currently *active* database in the list selected. If another database is desired, the user may either select it from among the pull-down list of currently open databases or, if the database is not yet open, press the ellipsis (3 dots) button next to the database selection list to open a file browser and navigate to the desired database in the file system as shown in Figure 4.329

Once the database of the target mesh is specified, the target mesh within that database is specified with the **Target Mesh:** pull down list.

A similar sequence of steps is followed for specifying the donor variable. The example in Figure 4.330 demonstrates the selection of a specific donor variable from the donor database with the **Donor Variable:** pull down list.

Next, the user is presented with a window to specify the manner in which the CMFE expression is to be evaluated. The choices are either *connectivity-based* or *position-based*. A position-based CMFE is a more general evaluation at the likely expense of lower performance. When in doubt, it is best to use this option. Connectivity-based evaluation is applicable *only* when donor and target meshes are one-for-one *both* topologically and geometrically. In this case, VisIt can optimize the evaluation and avoid having to deal with cases where the donor and target meshes do not wholly overlap.

For a position-based CMFE, the user is required to also specify what VisIt should do for those positions on the target mesh that do not overlap with the mesh of the donor variable. The user can choose either a constant numerical value (e.g. a *fill value*) or can specify a variable already defined on the target mesh. It is possible for the user to make a choice

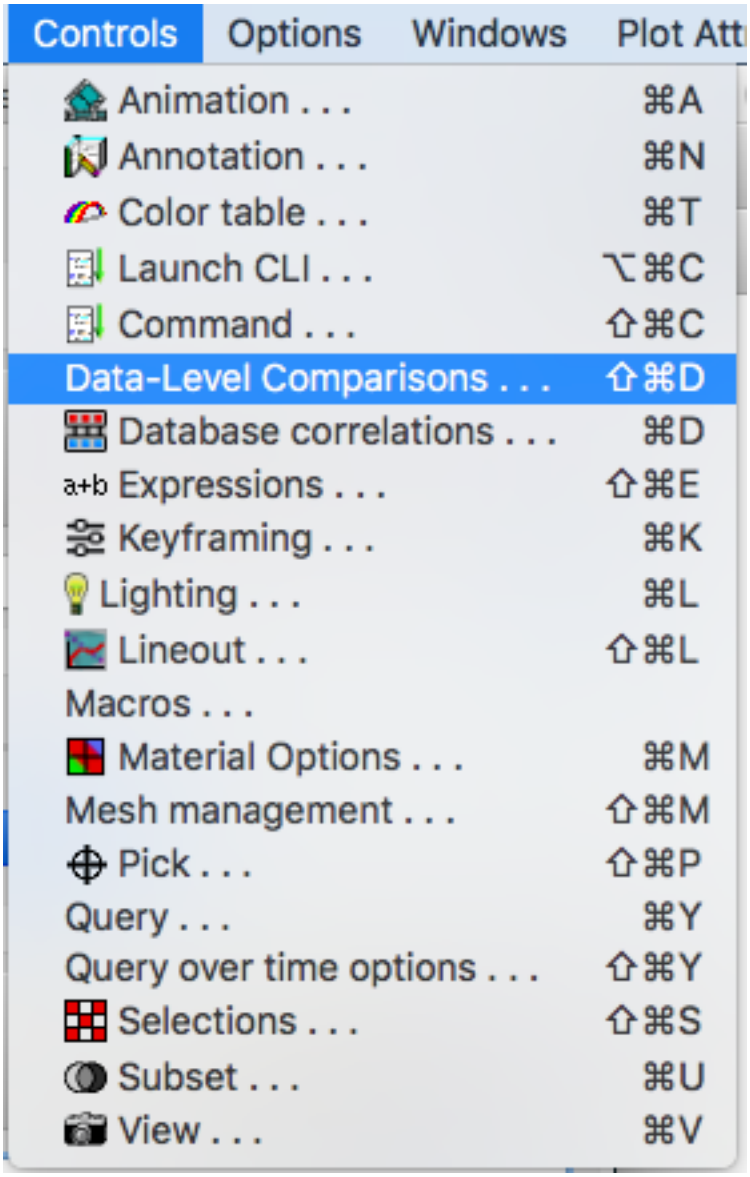


Fig. 4.326: Starting the Data-Level Comparisons Wizard

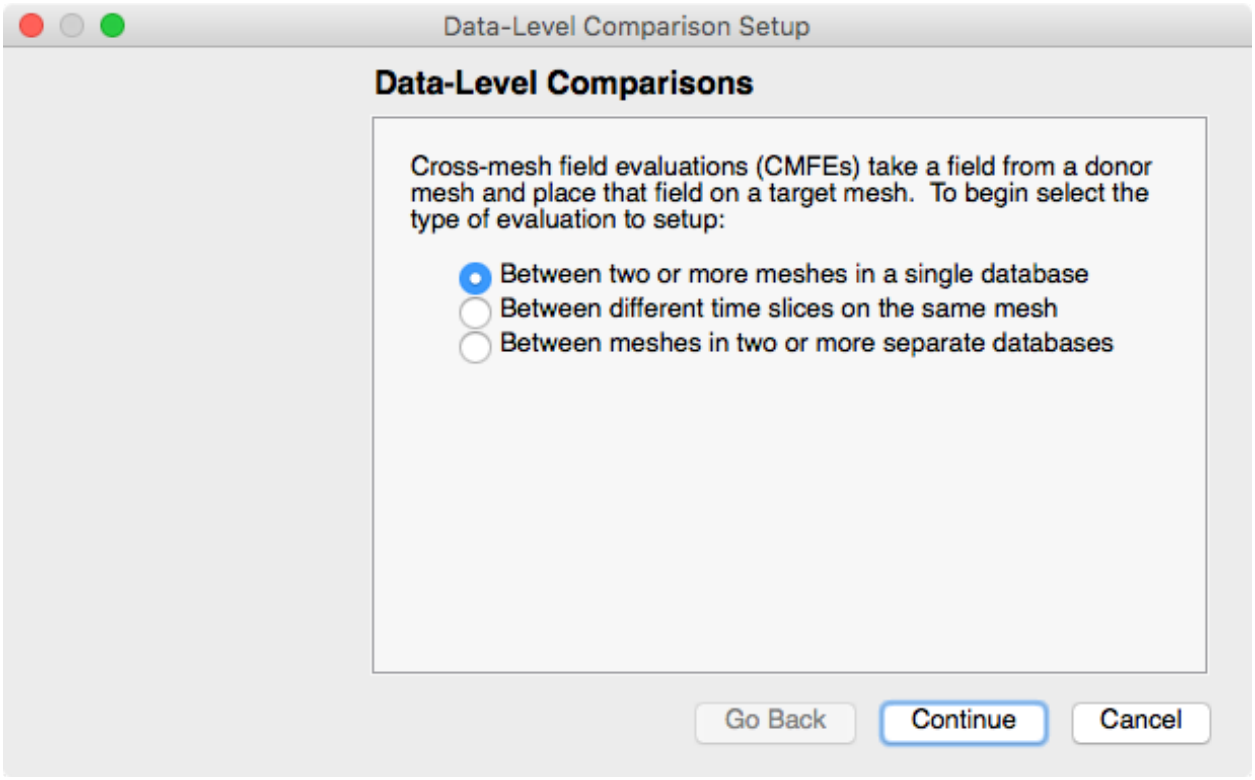


Fig. 4.327: Selecting among varieties of CMFE expressions

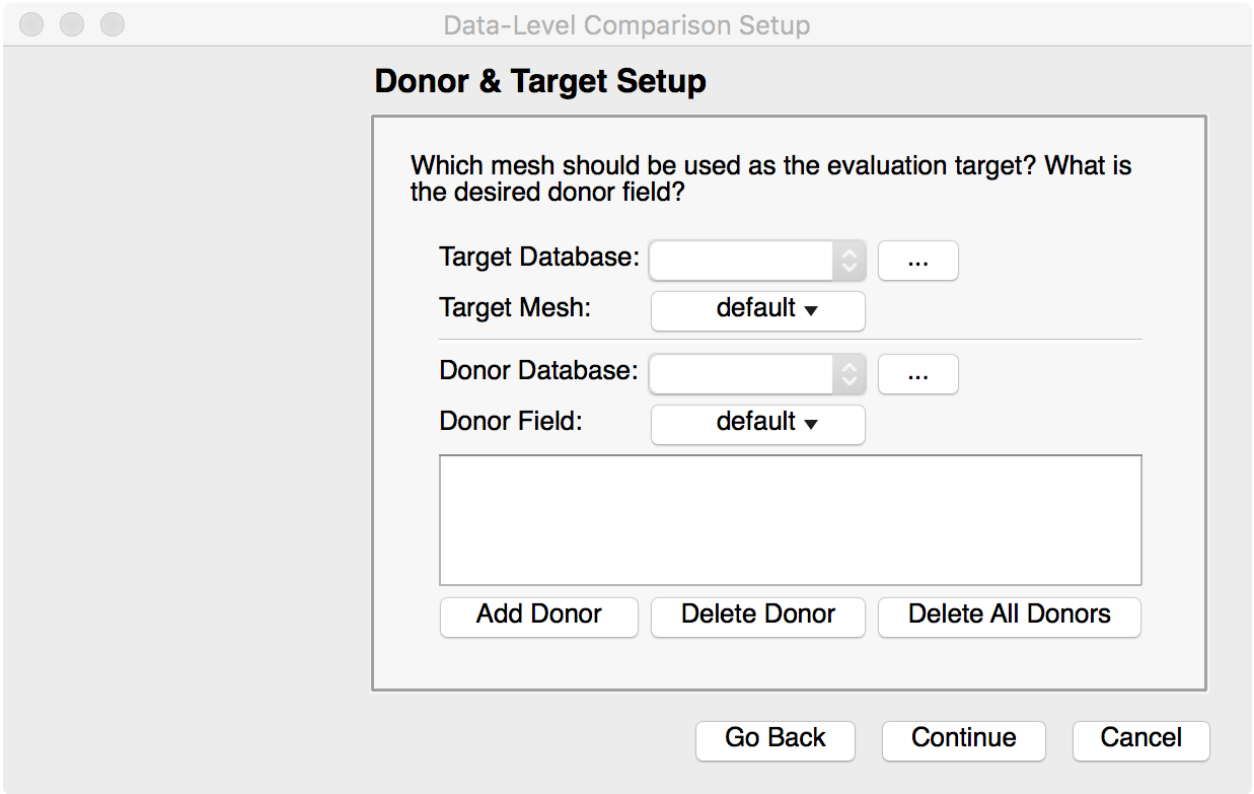


Fig. 4.328: Setting up the target mesh and donor variables

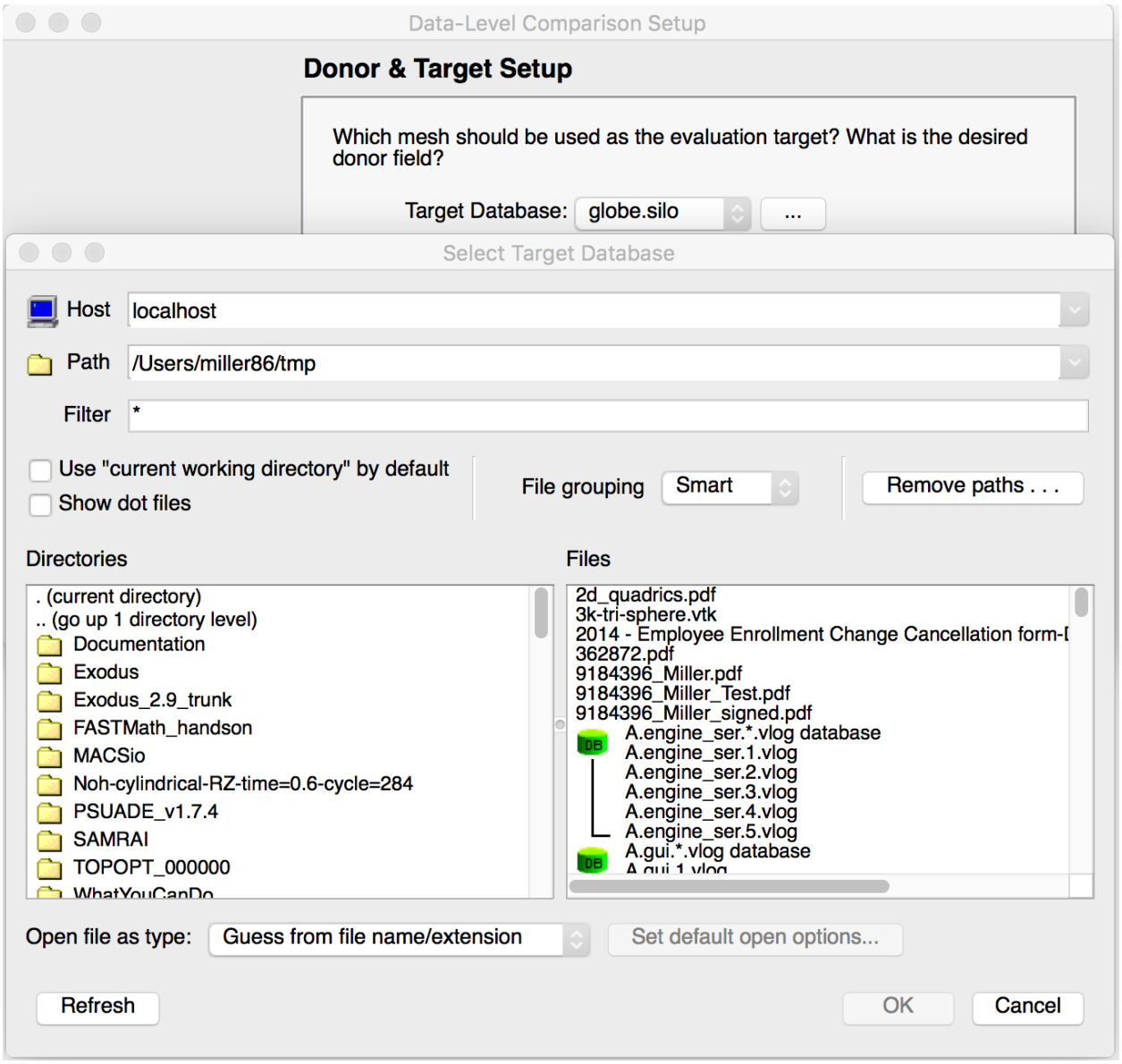


Fig. 4.329: Setting up the target mesh and donor variables

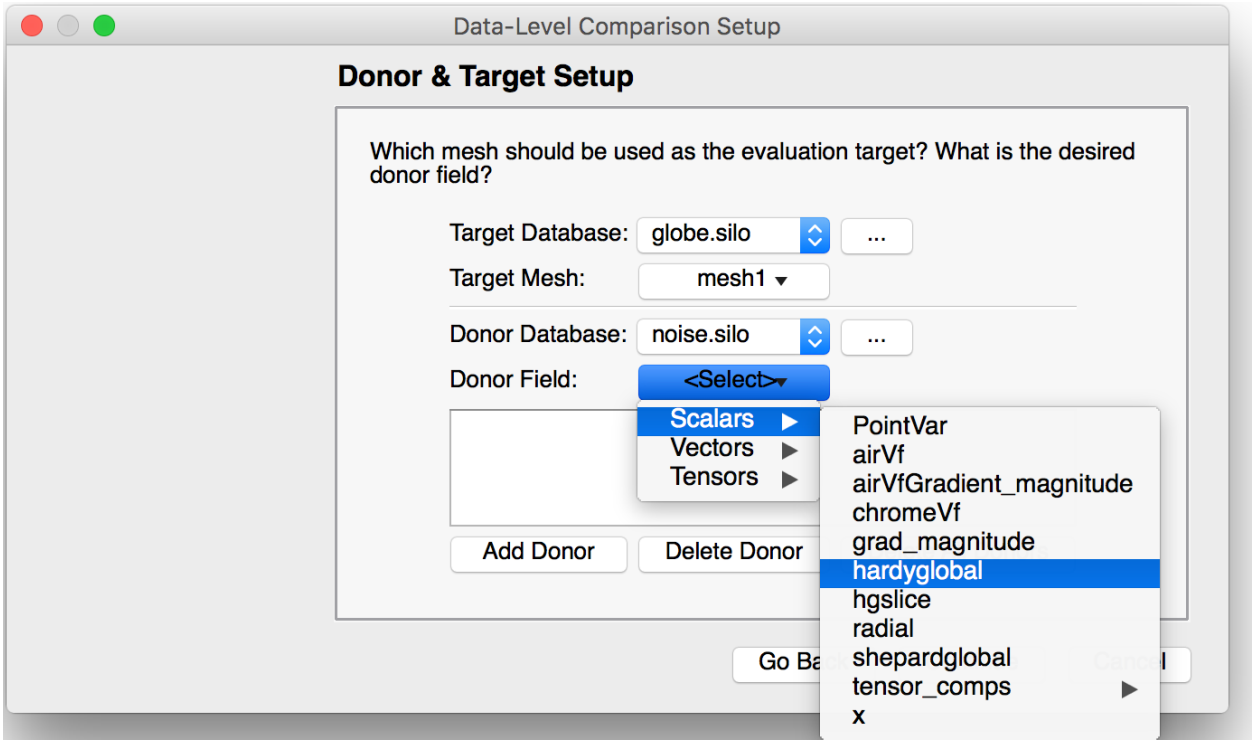


Fig. 4.330: Selecting a specific variable from a database

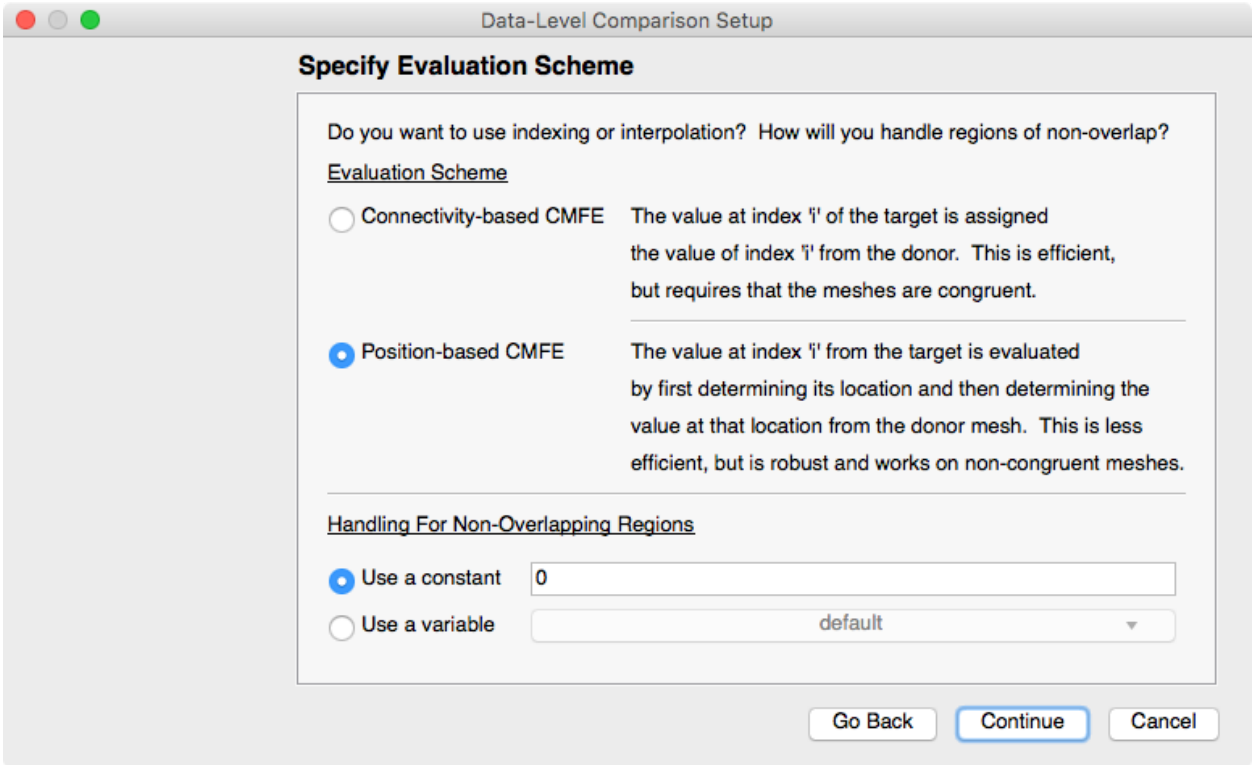


Fig. 4.331: Selecting the mode of evaluation

that either enhances or inhibits one's ability to distinguish between values in the result that come from the donor and values that come from the selected *fill* choice. A common practice is to choose a constant value that is an extremum of the donor variable's range. For example, if the donor variable has a maximum value of 25.7, then selecting this as the constant to use for non-overlapping regions in the CMFE has the benefit of not altering the variable's range but then also being indistinguishable from real data. Another practice is to choose a value that is easily distinguishable and later apply a threshold operator to remove those portions of the result.

The final step in the wizard is to give the result variable a name and then decide what to do with the result variable. In Figure 4.332, we have given the result variable the name *hardyglobal_onto_mesh1_from_globe*.

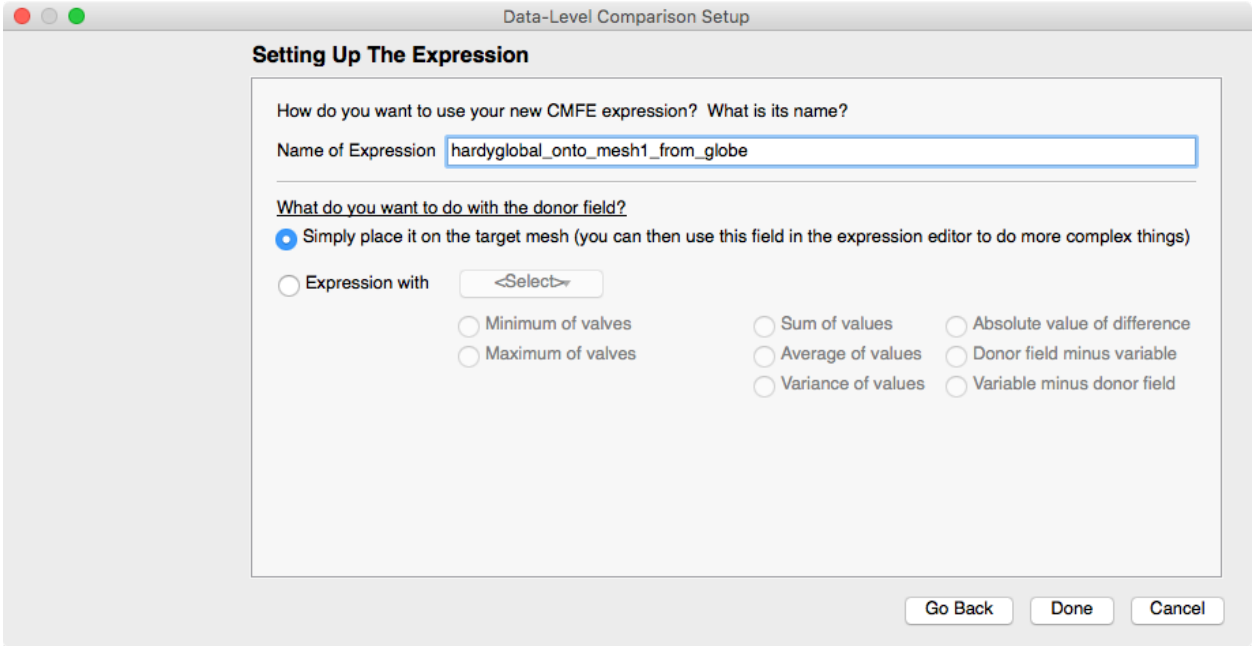


Fig. 4.332: Selecting result variable name and comparison method

Often, it is sufficient to have VisIt just compute the mapped variable and then allow the user to use the result variable in other expressions. However, for convenience, the wizard also offers a number of options common to the work of *comparing* the mapped variable to another variable. This last window in the wizard allows the user to select from among several common methods for comparing the mapped variable to another variable on the target mesh. By selecting the *Expression with* option, the user is then offered the ability to select a variable already defined on the target mesh from the pull down list. Then, the user can select from one of several common methods for comparing the two variables. For example, the *Absolute value of difference* choice will have the effect of creating a single expression that computes the difference in the donor and selected variables and then take its absolute value.

At any point during the steps in the wizard, the user can hit the *Go Back* button to go back and make different choices. The user completes the wizard by hitting the **Done** button. There is no way to *go back* after hitting the **Done** button. Upon completion of the wizard, a new expression is created according to user's selections. This new expression can be edited in the expression window, like any other expression as illustrated in Figure 4.333

In addition, this new expression can be used in other expressions. Finally, if for some reason the resulting expression is problematic, it can be deleted from the Expression system and the Data-Level Comparisons wizard can be run again to re-create it as desired.

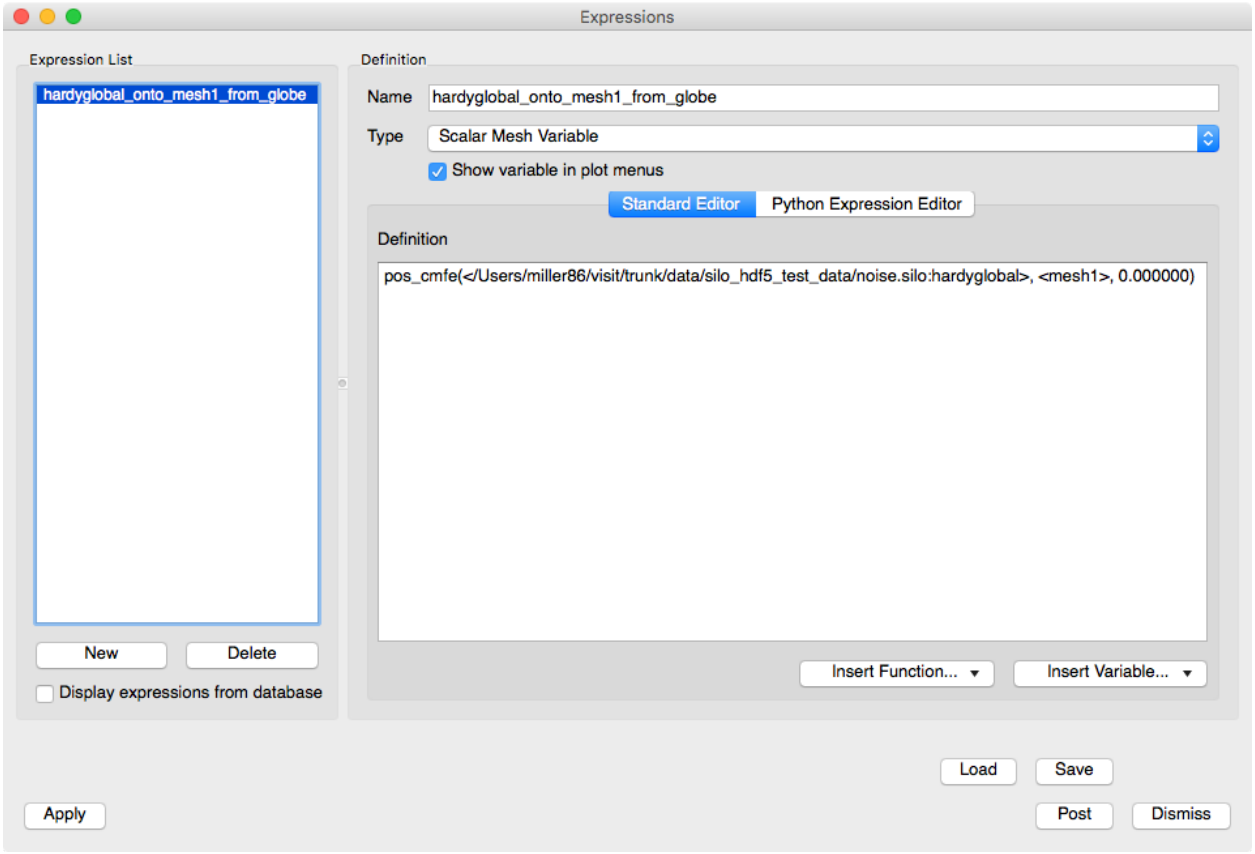


Fig. 4.333: New can be manipulated in the Expression window

4.9 Making it pretty

Now that you know how to visualize databases, it is time to learn how to make presentation quality visualizations. This chapter explains what options are available for making professional looking visualizations and introduces new windows that allow you to control annotations, colors, lighting, and the view.

4.9.1 Annotations

Annotations are objects in the visualization window that convey information about the plots. Annotations can be global objects that show information such as the database name, or they can be objects like plot legends that are directly tied to plots. Annotations are an essential component of a good visualization because they make it clear what is being visualized and make the visualization appear more polished.

VisIt supports several different annotation types that can be used to enhance visualizations. The first category of annotations includes general annotations like the database name, the user name, and plot legends. These annotations convey a good deal of information about what is being visualized, what values are in the plots, and who created the visualization. The second category of annotations include the plot axes and labels. This group of annotations comes in three groups: 2D, 3D and Array. The attributes for these groups can be set independently. Colors can greatly enhance the look of a visualization so VisIt provides controls to set the colors used for annotations and the visualization window that contains them. The third and final category includes annotation objects that can be added to the visualization window. You can add as many annotation objects as you want to a visualization window. The currently supported annotation objects are: 2D text, 3D text, time slider, 2D line, 3D line, and image annotations.

Annotation Window

The **Annotation Window** (Figure 4.334) contains controls for the various annotations that can appear in a visualization window. You can open the window choosing the **Annotation** option from the **Main Window's Controls menu**. The **Annotation Window** has a tabbed interface which groups the different categories of annotations together.

General Annotations

VisIt has a few general annotations that describe the visualization and are independent of the type of database in the visualization. General annotations encompass the user name, the database name, and plot legends. The general annotation controls are located in the **General** tab. Figure 4.335 shows common locations for some general annotations.

Turning plot legends off globally

Plot legends are special annotations that are added by plots. An example of a plot legend is the color bar and title that the Pseudocolor plot adds to the visualization window. Normally, plot legends are turned on or off by a check box in a plot attribute window but VisIt also provides a check box in the **General** tab that can turn off the plot legends for all the plots in the visualization window. You can use the **Legend** check box at the top of the **General** tab to turn plot legends off if they are present.

Displaying database information

When plots are displayed in the visualization window, the name of the database used in the plots is shown in the visualization window's upper left corner. You can turn the database information on or off using the **Database** check box in the **General** tab.

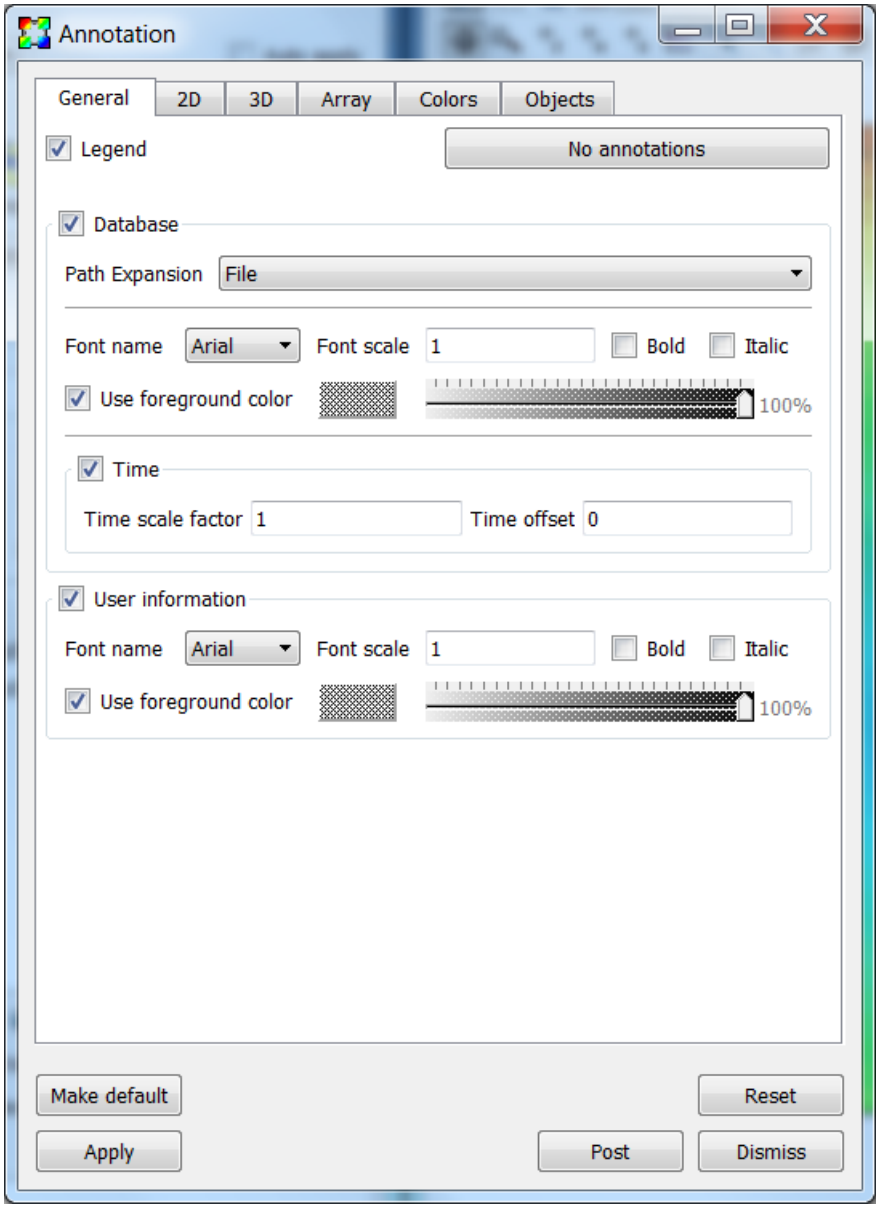


Fig. 4.334: The Annotation window

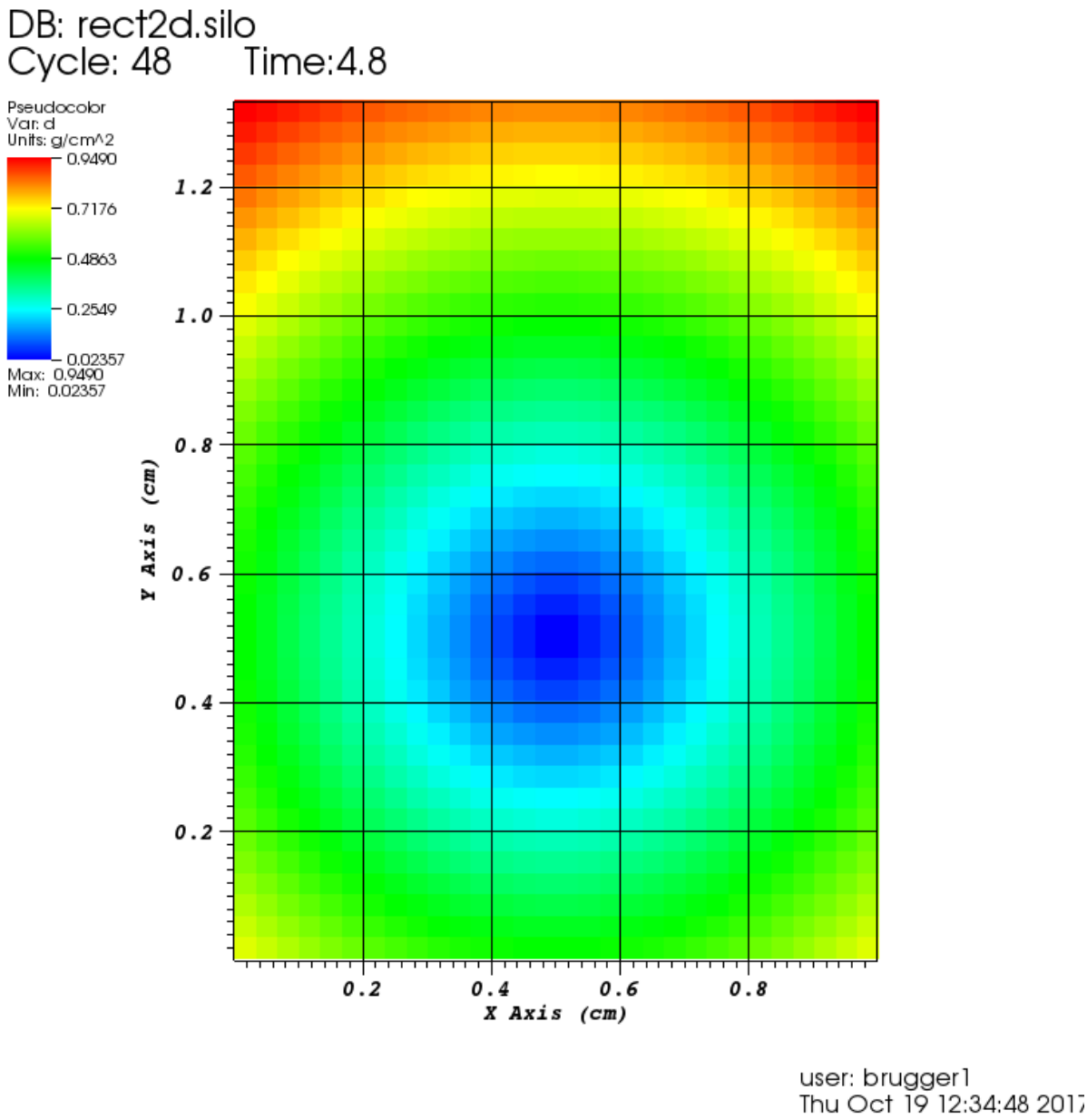


Fig. 4.335: 2D plot with annotations

The **Path Expansion** selection box controls the display of the filename text. **File** causes just the name of the file to be displayed. **Directory** causes the directory name of the file to be displayed. **Full** causes the full path of the file to be displayed. **Smart** uses simulation code specific conventions to display the file name in an optimal fashion. **Smart Directory** uses simulation code specific conventions to display the directory name in an optimal fashion.

The **Time** check box controls the display of the time associated with the current database. If **Time** is enabled then the **Time scale factor** and **Time offset** controls become active, allowing you to scale as well as apply an offset to the time associated with a database when displaying it.

Displaying user information

When you add plots to the visualization window, your username is shown in the lower right corner. The user information annotation is turned on or off using the **User information** check box. You may want to turn off user information when you are generating images for presentations.

2D Annotations

VisIt has a number of controls in the **Annotation Window** to control 2D annotations on the **2D** tab (Figure 4.336). The 2D annotation settings are primarily concerned with the appearance of the 2D axes that frame plots of 2D databases. Figure 4.335 shows a plot with various annotations.

The **Show axes** check box turns on and off the display of the 2D axes.

General 2D axis properties

Auto scale label values causes the labels to be multiplied by a factor of 10 to a multiple of 3 power such that the labels are in the range 0.001 to 999. It then displays the multiplier in the axis title. An example is shown in Figure 4.337. The X-Axis range is 0 to 100,000, which causes the labels to be in the range 0 to 100, with a (x10^3) added to the X-Axis and Y-Axis labels to indicate that the true range is actually 0 to 100x10^3 or 100,000.

The tick marks are small lines that are drawn along the edges of the 2D viewport. Tick marks can be drawn on a variety of axes by selecting a new option from the **Show tick marks** menu. Tick marks can also be drawn on the inside, outside, or both sides of the plot viewport by selecting a new option from the **Tick mark locations** menu.

Tick mark spacing is usually changed to best suite the plots in the visualization window but you can explicitly set the tick mark spacing by first unchecking the **Auto set ticks** check box and then typing new tick spacing values into the **Major minimum**, **Major maximum**, **Major spacing**, and **Minor spacing** text fields in the **X-Axis** and **Y-Axis** tabs.

Setting the X-Axis and Y-Axis properties

There are tabs for separately controlling the properties of the X and Y axes. The tab for setting the X-Axis properties is shown in Figure 4.338.

The axis titles are the names that are drawn along each axis, indicating the meaning of the values shown along the axis. Normally, the names used for the axis titles come from the database being plotted so the axis titles are relevant for the displayed plots. Many of VisIt's database readers plugins read file formats that have no support for storing axis titles so VisIt uses default values such as: "X-Axis", "Y-Axis". VisIt provides options that allow you to override the defaults or the axis titles that come from the file. You can control the display of the axis titles by enabling and disabling the **Title** check box. If you want to override the axis titles that VisIt uses for 2D visualizations, turn on the **Custom title** check box and type the new axis title into the adjacent text field.

In addition to overriding the names of the axis titles, you can also override the units that are displayed next to the axis titles. Units are displayed only when they are available in the file format and like axis titles, they are not always stored

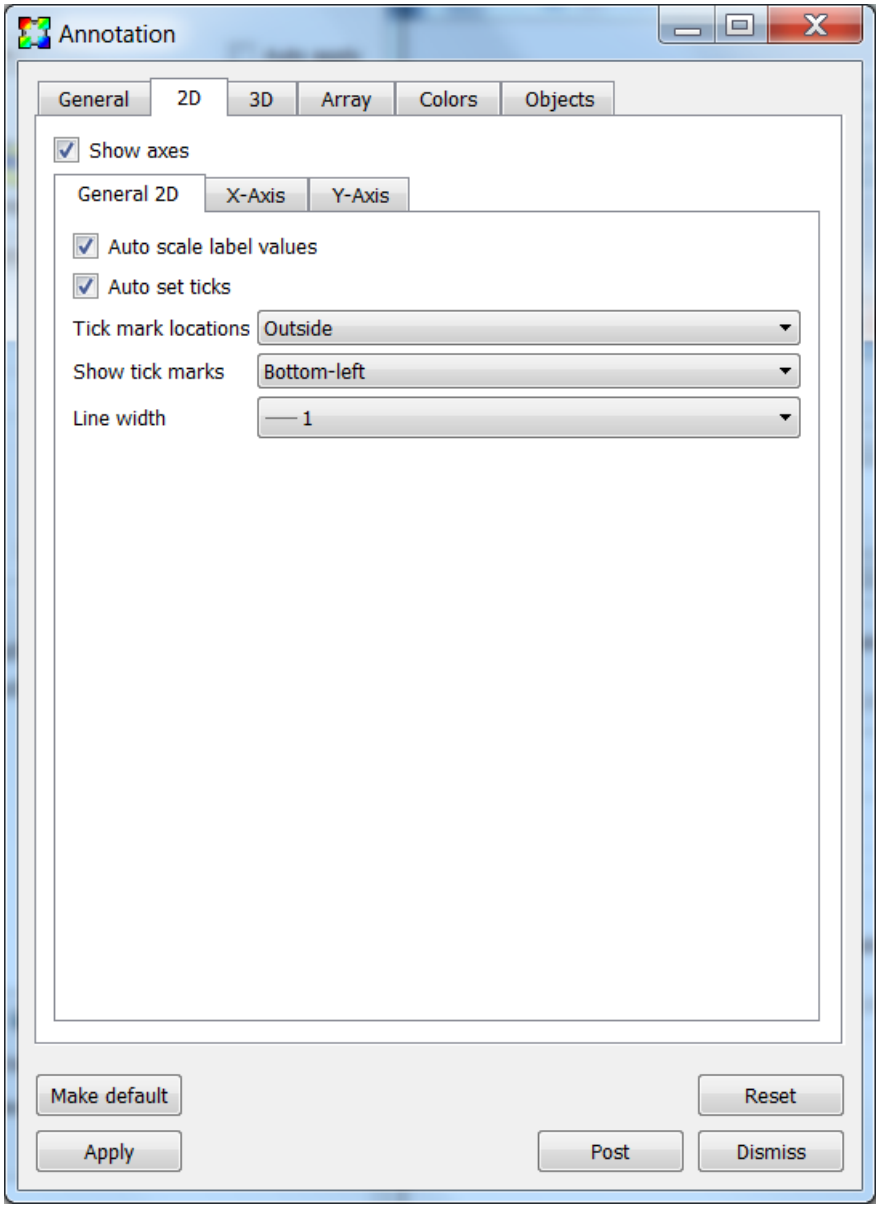


Fig. 4.336: The general 2D properties

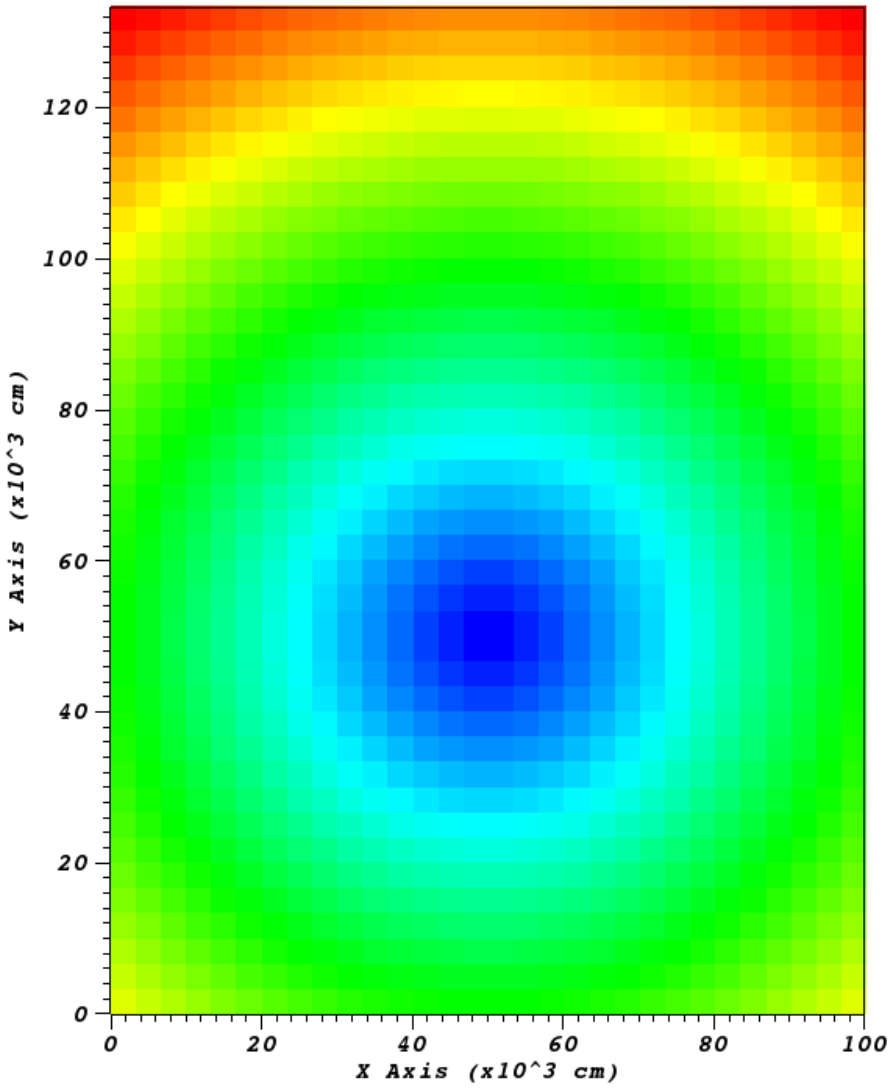


Fig. 4.337: 2D plot with axes labels being scaled by 10^3

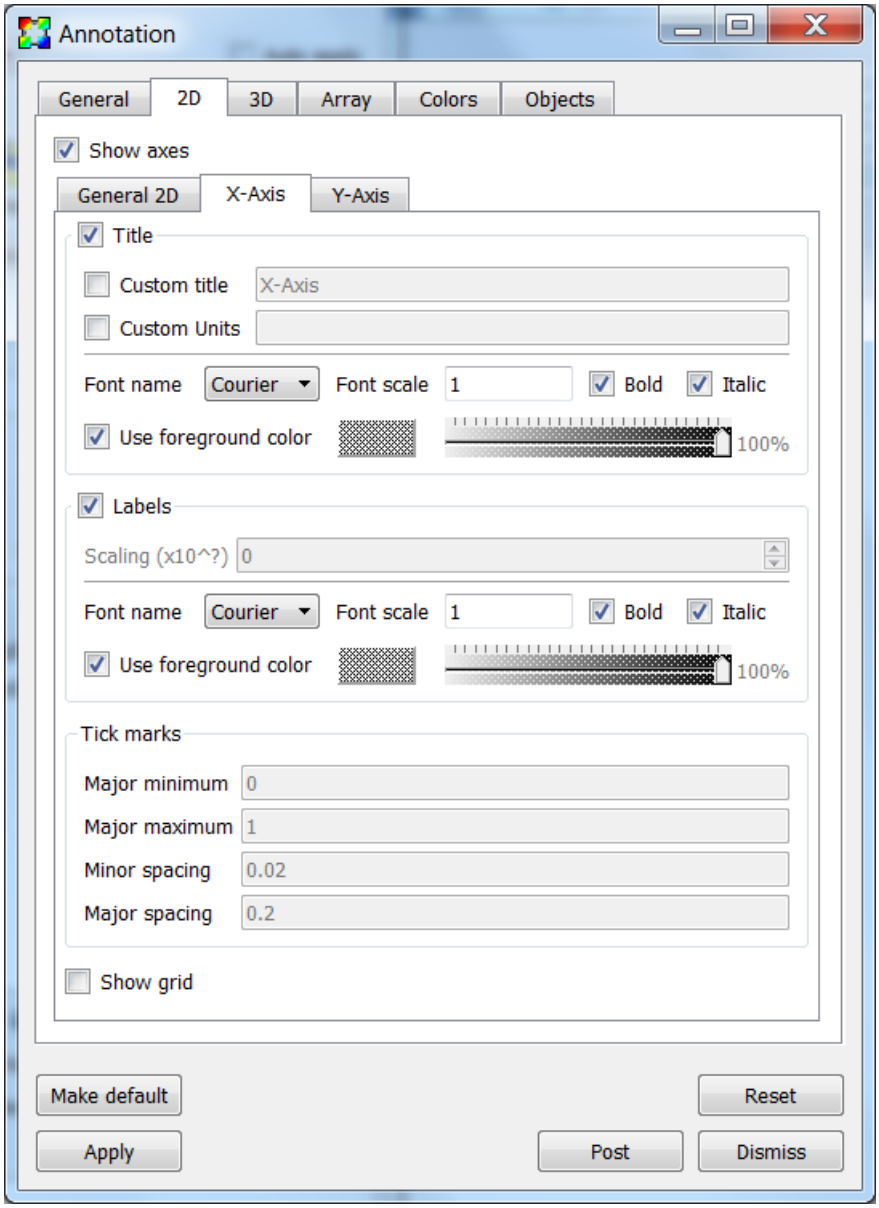


Fig. 4.338: The 2D axes properties

in the file being plotted. If you want to specify units for the axes, turn on the **Custom Units** check box and type new units into the adjacent text field.

The axis labels are the labels that appear along the 2D plot viewport. By default, the axis labels are enabled and set to appear. You can turn the labels off by unchecking the **Labels** check box. You can change the label scale factor by changing the **Scaling (x10^?)** text field.

Tick mark spacing is usually changed to best suite the plots in the visualization window but you can explicitly set the tick mark spacing by first unchecking the **Auto set ticks** check box on the **General 2D** tab and then typing new tick spacing values into the **Major minimum**, **Major maximum**, **Major spacing**, and **Minor spacing** text fields.

The 2D grid lines are a set of lines that make a grid over the 2D viewport. The grid lines are disabled by default but you can enable them by checking the **Show grid** check box. The grid lines correspond to the major tick marks.

3D Annotations

VisIt has a number of controls, located on the **3D** tab in the **Annotation Window** for controlling annotations that are used when the visualization window contains 3D plots. Like the 2D controls, these controls focus mainly on the axes that are drawn around plots. Figure 4.339 shows an example 3D plot with the 3D annotations. Figure 4.340 and Figure 4.341 shows the **Annotation Window's 3D tab**.

The **Show axes** check box turns on and off the display of the 3D axes.

The **Show triad** check box turns on and off the display of the triad annotation. The triad annotation consists of a small set of axes and is displayed in the lower left corner of the visualization window and help you get your bearings in 3D.

The **Show bounding box** check box turns on an off the display of the bounding box. The bounding box annotation displays the edges of a box that contains all the data.

General 3D axis properties

Auto scale label values causes the labels to be multiplied by a factor of 10 to a multiple of 3 power such that the labels are in the range 0.001 to 999. It then displays the multiplier in the axis title. A 2D example is shown in Figure 4.337. The X-Axis range is 0 to 100,000, which causes the labels to be in the range 0 to 100, with a (x10^3) added to the X-Axis and Y-Axis labels to indicate that the true range is actually 0 to 100x10^3 or 100,000.

The tick marks are small lines that are drawn along the edges of the bounding box surfaces. Tick marks can be drawn on a variety of axes by selecting a new option from the **Show tick marks** menu. Tick marks can also be drawn on the inside, outside, or both sides of the plot bounding box by selecting a new option from the **Tick mark locations** menu.

Tick mark spacing is usually changed to best suite the plots in the visualization window but you can explicitly set the tick mark spacing by first unchecking the **Auto set ticks** check box and then typing new tick spacing values into the **Major minimum**, **Major maximum**, **Major spacing**, and **Minor spacing** text fields in the **X-Axis**, **Y-Axis** and **Z-Axis** tabs.

Setting the X-Axis, Y-Axis and Z-Axis properties

There are tabs for separately controlling the properties of the X, Y and Z axes. The tab for setting the X-Axis properties is shown in Figure 4.341.

The axis titles are the names that are drawn along each axis, indicating the meaning of the values shown along the axis. Normally, the names used for the axis titles come from the database being plotted so the axis titles are relevant for the displayed plots. Many of VisIt's database readers plugins read file formats that have no support for storing axis titles so VisIt uses default values such as: "X-Axis", "Y-Axis" and "Z-Axis". VisIt provides options that allow you to override the defaults or the axis titles that come from the file. You can control the display of the axis titles by enabling

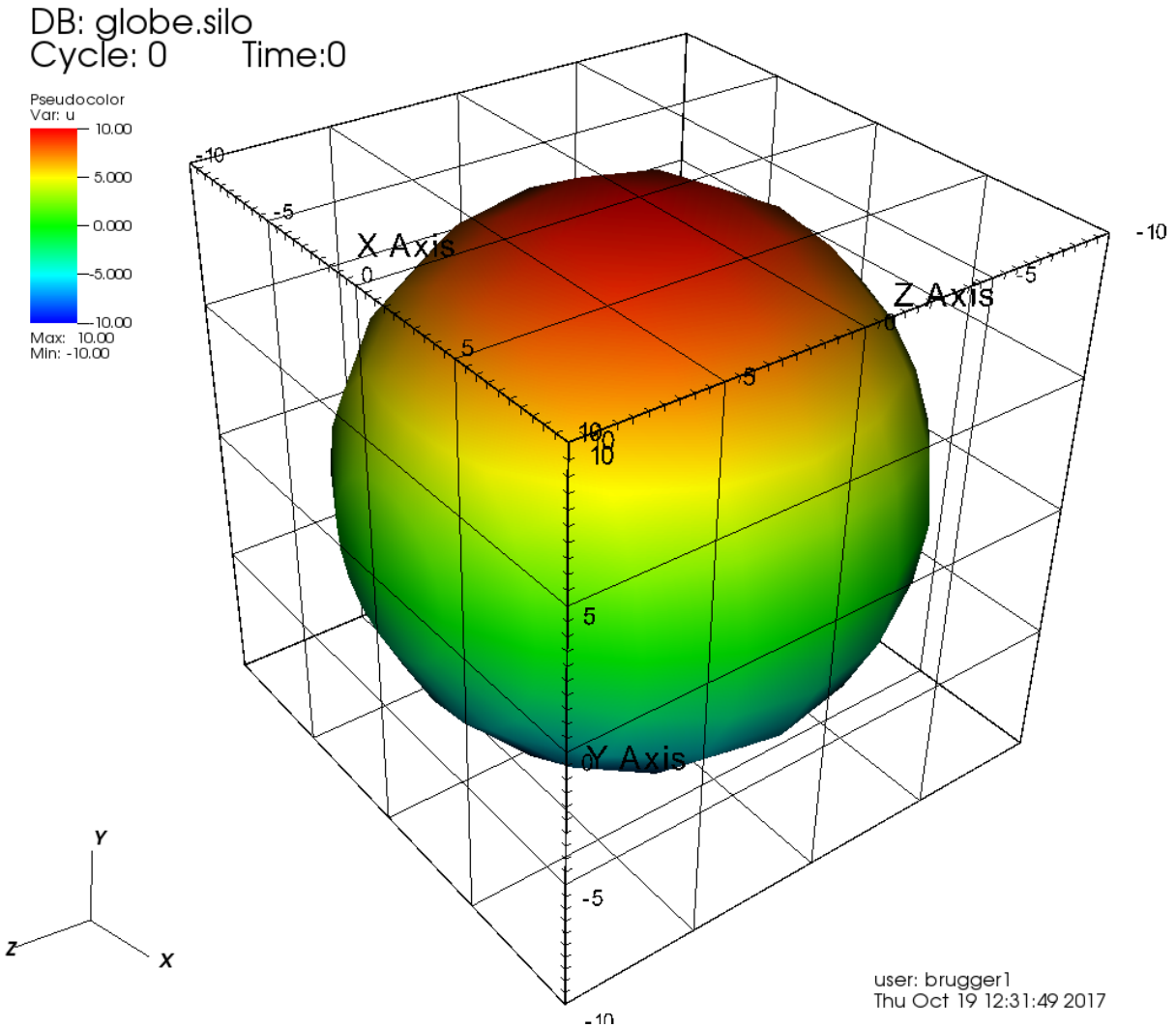


Fig. 4.339: 3D plot with annotations

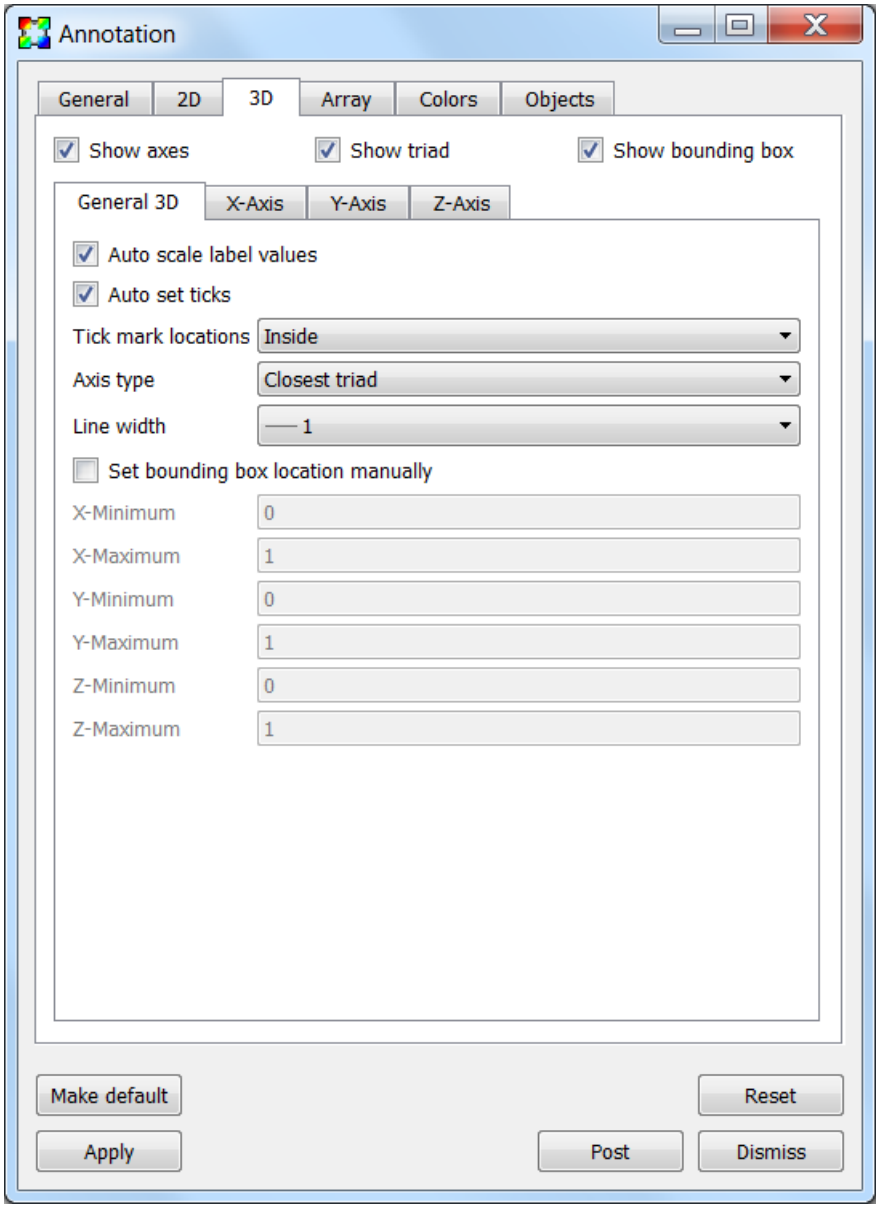


Fig. 4.340: The general 3D properties

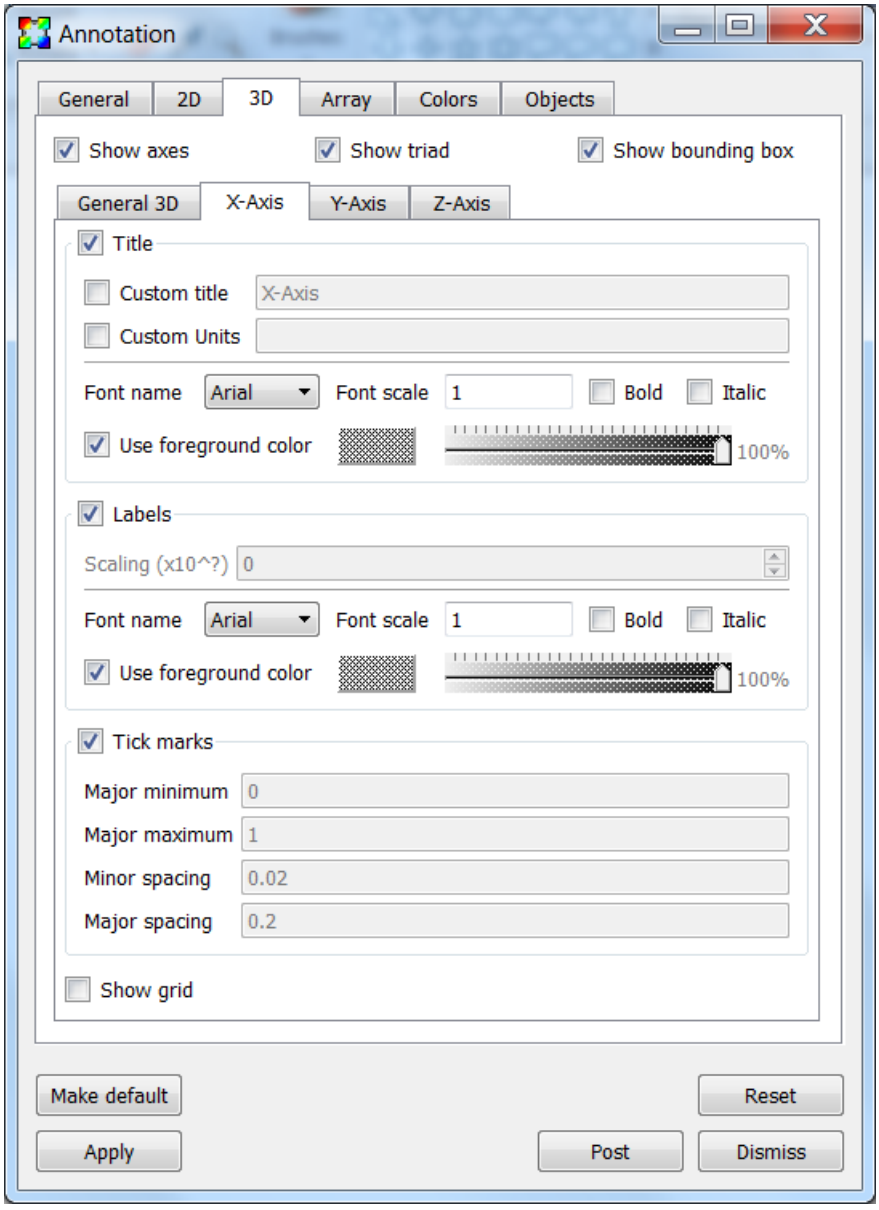


Fig. 4.341: The 3D axes properties

and disabling the **Title** check box. If you want to override the axis titles that VisIt uses for 3D visualizations, turn on the **Custom title** check box and type the new axis title into the adjacent text field.

In addition to overriding the names of the axis titles, you can also override the units that are displayed next to the axis titles. Units are displayed only when they are available in the file format and like axis titles, they are not always stored in the file being plotted. If you want to specify units for the axes, turn on the **Custom Units** check box and type new units into the adjacent text field.

The axis labels are the labels that appear along the edges of the bounding box. By default, the axis labels are enabled and set to appear. You can turn the labels off by unchecking the **Labels** check box. You can change the label scale factor by changing the **Scaling (x10^?)** text field.

Tick mark spacing is usually changed to best suite the plots in the visualization window but you can explicitly set the tick mark spacing by first unchecking the **Auto set ticks** check box on the **General 3D** tab then typing new tick spacing values into the **Major minimum**, **Major maximum**, **Major spacing**, and **Minor spacing** text fields.

The 3D grid lines are a set of lines that make a grid over the the bounding box. The grid lines are disabled by default but you can enable them by checking the **Show grid** check box. The grid lines correspond to the major tick marks.

Annotation Colors

Colors are very important in a visualization since they help to determine how easy it is to read annotations. VisIt provides a tab in the **Annotation Window**, shown in Figure 4.342, specifically devoted to choosing annotation colors. The **Colors** tab contains controls to set the background and foreground for the visualization window which, in turn, set the colors used for annotations. The **Colors** tab also provides controls for more advanced background colors called gradients which are colors that bleed into each other.

The **Background color** and **Foreground color** buttons allow you to set the background and foreground colors. To set the color, click the color button and select a color from the **Popup color menu** (see Figure 4.343). Releasing the mouse outside of the **Popup color menu** cancels color selection and the color is not changed. Once you select a new color and click the **Apply** button, the colors for the active visualization window change. Note that each visualization window can have different background and foreground colors.

The **Background style** setting allows you to select from four background styles. The default background style is **Solid** where the entire background is a single color. The second style is a **Gradient** background. In a gradient background, two colors are blended into each other in various ways. The resulting background offers differing degrees of contrast and can enhance the look of many visualizations. The third style is an **Image** background, where an image is tiled across the background. The fourth style is an **Image sphere**, where an image is projected onto a sphere. This can be used to paint the stars onto the background of an astrophysics simulation. To change the background style, click the **Background style** radio buttons.

VisIt provides controls for setting the colors and style used for gradient backgrounds. There are two color buttons: **Gradient color 1** and **Gradient color 2** that are used to change colors. To change the gradient colors, click on the color buttons and select a color from the **Popup color menu**. The gradient style is used to determine how colors blend into each other. To change the gradient style, make a selection from the **Gradient style** menu. The available options are Bottom to Top, Top to Bottom, Left to Right, Right to Left, and Radial. The first four options blend gradient color 1 to gradient color 2 in the manner prescribed by the style name. For example, Bottom to Top will have gradient color 1 at the bottom and gradient color 2 at the top. The radial gradient style puts gradient color 1 in the middle of the visualization window and blends gradient color 2 radially outward from the center. Examples of the gradient styles are shown in Figure 4.344.

The **Background image** text field allows you to specify the name of the file to use for the background image. The **Repetitions in X** and **Repetitions in Y** settings allow you to specify how many times to replicate the image in each of the X and Y image directions.

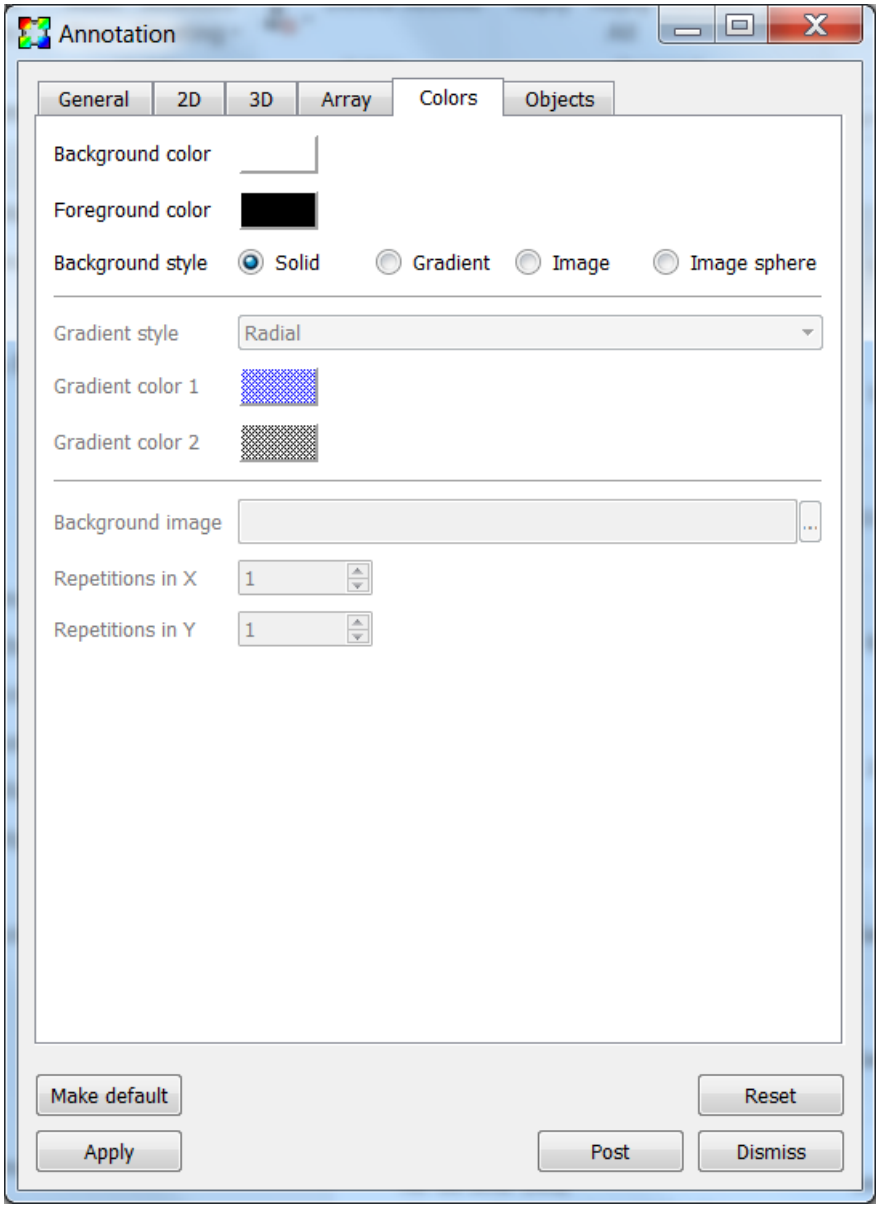


Fig. 4.342: The annotation colors tab

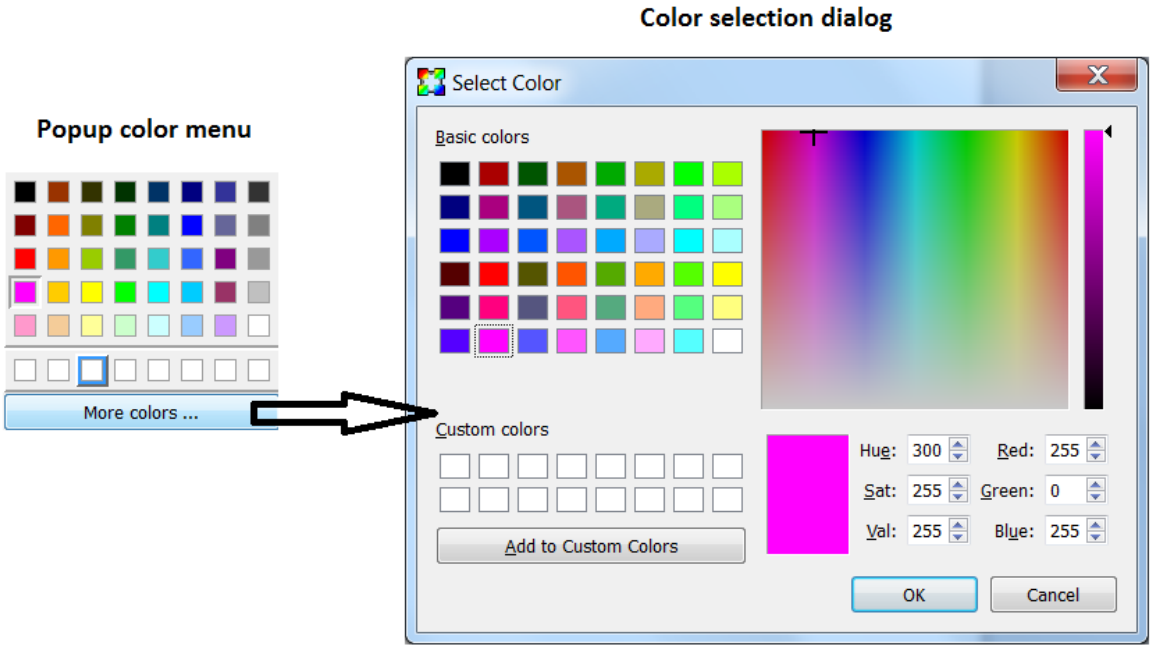


Fig. 4.343: The popup color menu and the color selection dialog

Fig. 4.344: The various gradient styles

Annotation Objects

So far, the annotations that have been described can only have a single instance. To provide more flexibility in the types and numbers of annotations, VisIt allows you to create annotation objects, which are objects that are added to the visualization window to convey information about the visualization. Currently, VisIt supports six types of user-creatable annotation objects (2D text, 3D text, time slider, 2D line, 3D line, image) and one automatically generated object tied to specific plots: Legend. All of those types of annotation objects will be described herein. The **Objects** tab, in the **Annotation Window** (Figure 4.345) is devoted to managing the list of annotation objects and setting their properties.

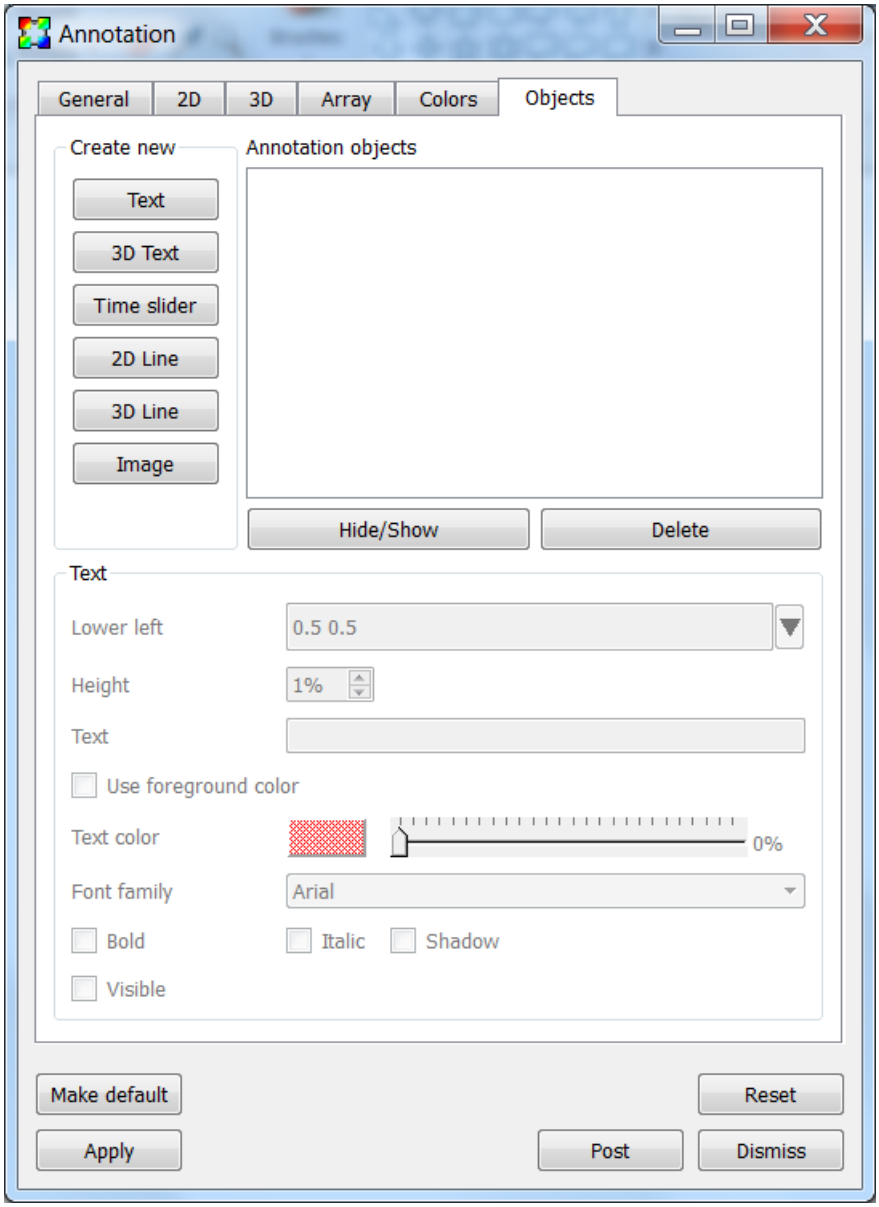


Fig. 4.345: The annotation objects tab

The **Objects** tab in the **Annotation Window** is divided up into three main areas. The top of the window is split vertically into two areas that let you create new annotation objects and manage the list of annotation objects. The bottom half of the **Objects** tab displays the controls for setting the attributes of the selected annotation object. Each

annotation object provides a separate user interface that is tailored for setting its particular attributes. When you select an annotation in the annotation object list, the appropriate annotation object interface is displayed.

Creating a new annotation object

The **Create new** area in the **Annotation Window's Objects** tab contains one button for each type of annotation object that VisIt can create. Each button has the name of the type of annotation object VisIt creates when you push it. After pushing one of the buttons, VisIt creates a new instance of the specified annotation object type, adds a new entry to the **Annotation objects** list, and displays the appropriate annotation object interface in the bottom half of the **Objects** tab to display the attributes for the new annotation object.

Selecting an annotation object

The **Objects** tab displays the annotation object interface for the selected annotation object. To set attributes for a different annotation object, or to hide or delete a different annotation object, you must first select a different annotation object in the **Annotation objects** list. Click on a different entry in the **Annotation objects** list to highlight a different annotation object. Once you have highlighted a new annotation object, VisIt displays the object's attributes in the lower half of the **Objects** tab.

Hiding an annotation object

To hide an annotation object, select it in the **Annotation objects** list and then click the **Hide/Show** button on the **Objects** tab. To show the hidden annotation object, click the **Hide/Show** button a second time. The interfaces for the currently provided annotation objects also have a **Visible** check box that can be used to hide or show the annotation object.

Deleting an annotation object

To delete an annotation object, select it in the **Annotation objects** list and then click the **Delete** button on the **Objects** tab. You can delete more than one object if you select multiple objects plots in the **Annotation objects** list before clicking the **Delete** button.

Text annotation objects

Text annotation objects, shown in Figure 4.346, are created by clicking the **Text** button in the **Create new** area on the **Objects** tab. Text annotation objects are simple 2D text objects that are drawn on top of plots in the visualization window and are useful for adding titles to a visualization.

The text annotation object properties, shown in Figure 4.347, can be used to set the position, size, text, colors, and font properties.

Text annotation objects are placed using 2D coordinates where the X, and Y values are in the range [0,1]. The point (0,0) corresponds to the lower left corner of the visualization window and the point (1,1) corresponds to the upper right of the visualization window. The 2D coordinate used to position the text annotation matches the text annotation's lower left corner. To position a text annotation object, enter a new 2D coordinate into the **Lower left** text field. You can also click the down arrow next to the **Lower left** text field to interactively choose a new lower left coordinate for the text annotation using the screen positioning control, which represents the visualization window. The screen positioning control, shown in Figure 4.348, lets you move a set of cross-hairs to any point on a square area that represents the visualization window. Once you release the left mouse button, the location of the cross-hairs is used as the new coordinate for the text annotation object's lower left corner.

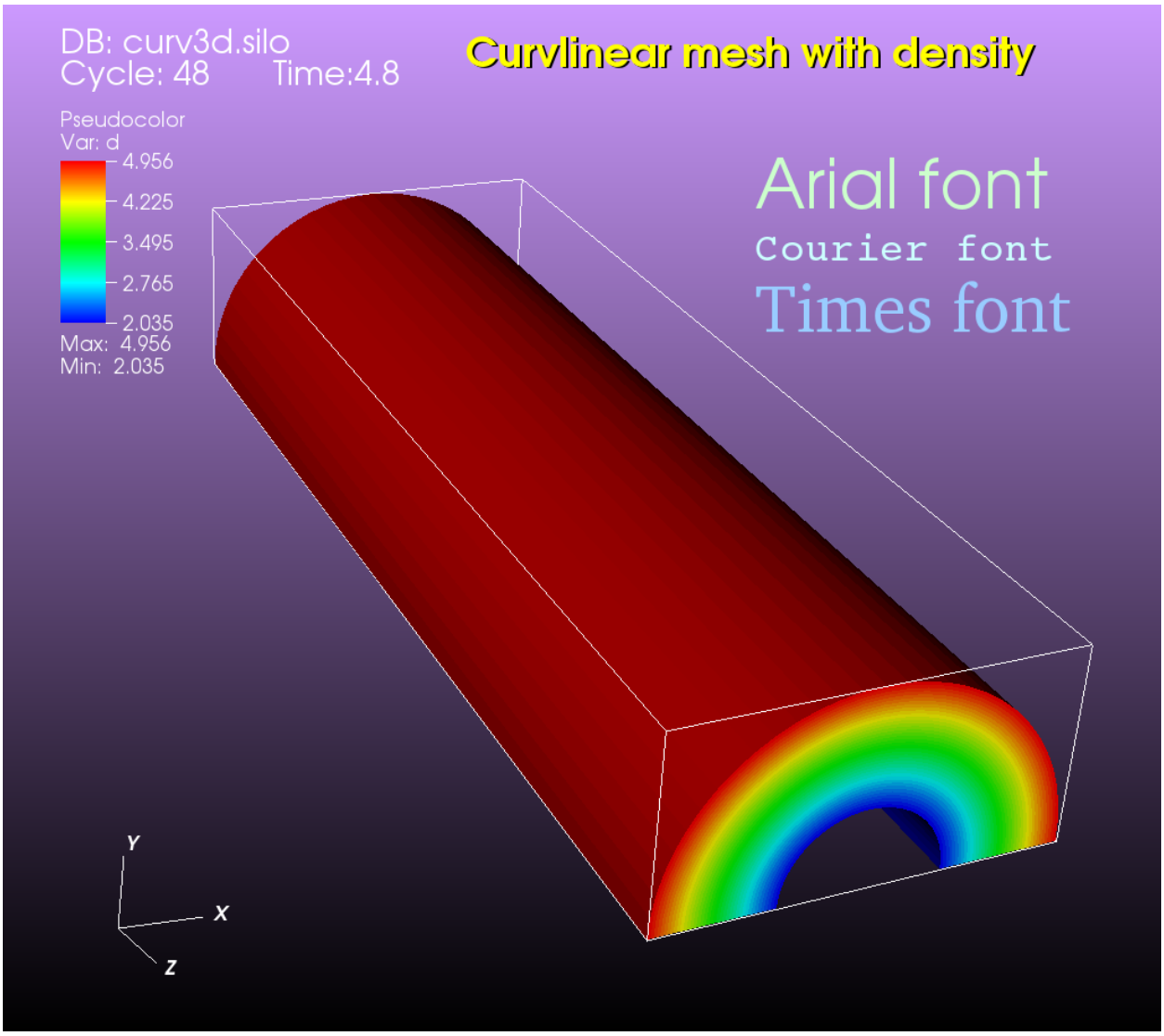


Fig. 4.346: Examples of text annotations

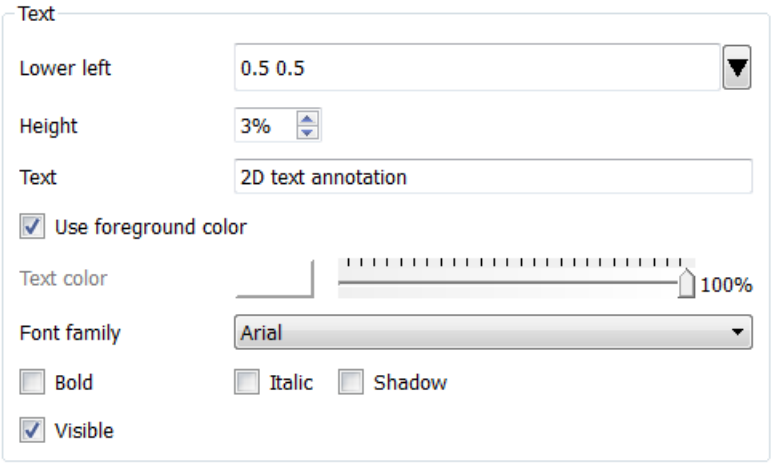


Fig. 4.347: The text annotation interface

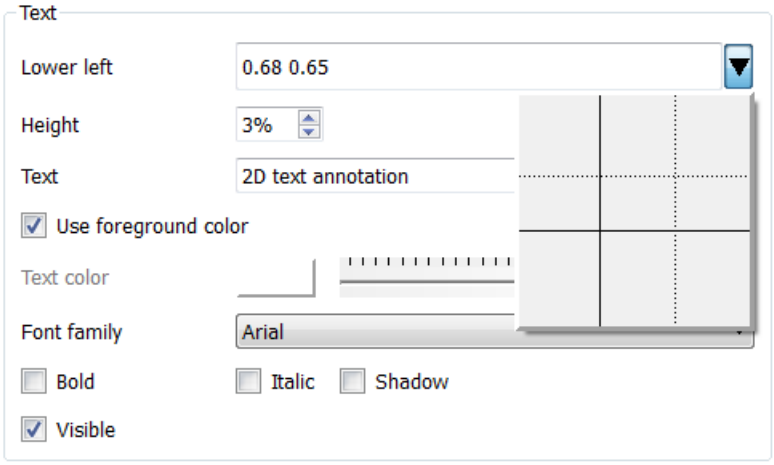


Fig. 4.348: Screen positioning control

The size of the text is set using the **Height** spin box. The height is the fraction of the visualization window height.

To set the text that a text annotation object displays, type a new string into the **Text** text field. You can make the text annotation object display any characters that you type in. Whatever text you enter for the text annotation object is used to identify the text annotation object in the **Annotation objects** list. In addition to the usual text properties, text annotation objects can also include a shadow.

Named database values in text annotations

A variety of named database values can be displayed in text annotations. These are introduced with a leading `$` character followed by the value's name. An optional `%` character following the value's name introduces a printf-style formatting string which can be used to control the value's printed format.

Warning: Presently, the `$` values that are displayed are taken always from database associated with the *first* plot in the plot list. If the plot list is changed such that the database associated with the *first* plot that was in effect at the time the annotation was created is changed, the rendered text for named database value annotations may change.

The list of named values currently supported along with their *default* formats are

Value name	Fmt	Meaning
time	%g	time value
cycle	%d	cycle number
index	%d	state number starting from zero
numstates	%d	total number of states in database
dbcomment	%s	database comment
lod	%z	levels of detail
vardim	%d	variable dimension
numvar	%d	number of variables
topdim	%d	topological dim. of assoc. mesh
spatialdim	%d	spatial dim. of assoc. mesh
varname	%s	variable name
varunits	%s	variable units
meshname	%s	name of mesh assoc. w/variable
filename	%s	name of database file
fulldbname	%s	full path name of database file
xunits	%s	x units
yunits	%s	y units
zunits	%s	z units
xlabel	%s	x axis label
ylabel	%s	y axis label
zlabel	%s	z axis label

In addition, the following `$<T>tafile<I>` named values permit arbitrary text annotation content to be taken from a `txt` file with name of the form `<T>tafile<I>.txt` where `<T>` is either `s` (for files of string values), `i` (for files of integer values) or `f` (for files of floating point values) and `<I>` is either `1`, `2` or `3` to provide 3 separate options for storing files of values used for different annotation purposes. Each line of such a file corresponds to a time step in a time series. If a `$$<T>tafile<I>` named annotation is used, VisIt will search for the associated file first in the same directory containing the database, then in the directory `/$TMPDIR/$USER` or (`/var/tmp/$USER`) and finally in *The Platform and the User's Home Directory*.

A common use case for `$<T>tafile<I>` named values is for animations to display the numerical values from a query over time and have those values update as the time step being displayed changes.

Value name	Fmt	Meaning
itafile1	%d	ints from itafile1.txt one line per timestep.
itafile2	%d	ints from itafile2.txt one line per timestep.
itafile3	%d	ints from itafile3.txt one line per timestep.
ftafile1	%g	floats from ftafile1.txt one line per timestep.
ftafile2	%g	floats from ftafile2.txt one line per timestep.
ftafile3	%g	floats from ftafile3.txt one line per timestep.
stafile1	%s	strings from stafile1.txt one line per timestep.
stafile2	%s	strings from stafile2.txt one line per timestep.
stafile3	%s	strings from stafile3.txt one line per timestep.

Warning: Only the first 255 characters on each line of a `$<T>tafile<I>` annotation are used.

Multiple named values can appear in a text annotation string and the same named value can also appear multiple times.

For example, to create a text annoation which displays `State index = XXX` where `XXX` is the number for the index, set the annotation string to `State index = $index`. To display the current cycle number always with 6 digits and leading zeros when necessary, use the string `$cycle%06d` where the optional `%` followed by a printf-style format string is specified. To display the first 3 characters of the variable name, use the string `$varname%.3s`.

The `$dbcomment` and `$<T>tafile<I>` named values are useful for complicated cases because they allow arbitrary text defined in the database comment or an external file to be used. For example, the *state space* of a given database could be rather complicated involving not only iterations of the main PDE solve loop but also mesh adaptivity iterations, material advection iterations, etc. In this case, if the data producer created appropriate content in the database comment or in a text file, the `$dbcomment` or `$<T>tafile<X>` named value is a way to render all relevant iteration identifiers as a text annotation.

3D text annotation objects

3D text annotation objects, shown in Figure 4.349, are created by clicking the **3D Text** button in the **Create new** area on the **Objects** tab. 3D text annotation objects are extruded text that are positioned in 3D and are part of the 3D scene, so they may become obscured by other objects in the scene and will move in space as the image is panned and zoomed.

The 3D text annotation object properties, shown in Figure 4.350, can be used to set the text, position, size, orientation and color properties.

To set the text that a 3D text annotation object displays, type a new string into the **Text** text field.

3D text annotation objects are placed in 3D coordinates in the same coordinate system used by the simulation data. To position a 3D text annotation object, enter a new 3D coordinate into the **Position** text field.

The size of the text can be specified in two different ways. The first is using a relative height, where the height is a fraction of the size of the simulation data. The second is a fixed size, where the size is specified in the coordinate system of the simulation data. If you were to specify a relative height and apply the Transform operator to scale the data in each direction by a factor of 10, the size of the text would not change. If you were to specify a fixed height, scaling the data by a factor of 10 would result in the text being one tenth the size. To specify a relative height, select the **Relative** radio button and set the size using the spin box next to it. The specify a fixed height, select the **Fixed** radio button and enter the new height in the text box next to it.

The orientation of the text can also be specified in two different ways. The first is relative to the screen coordinate system and the second is in the coordinate system of the simulation data. If the orientation is relative to the screen coordinate system, then rotating the image will not change the orientation of the text. If the orientation is relative to the coordinate system of the simulation data, then rotating the image will change the orientation of the text. To make

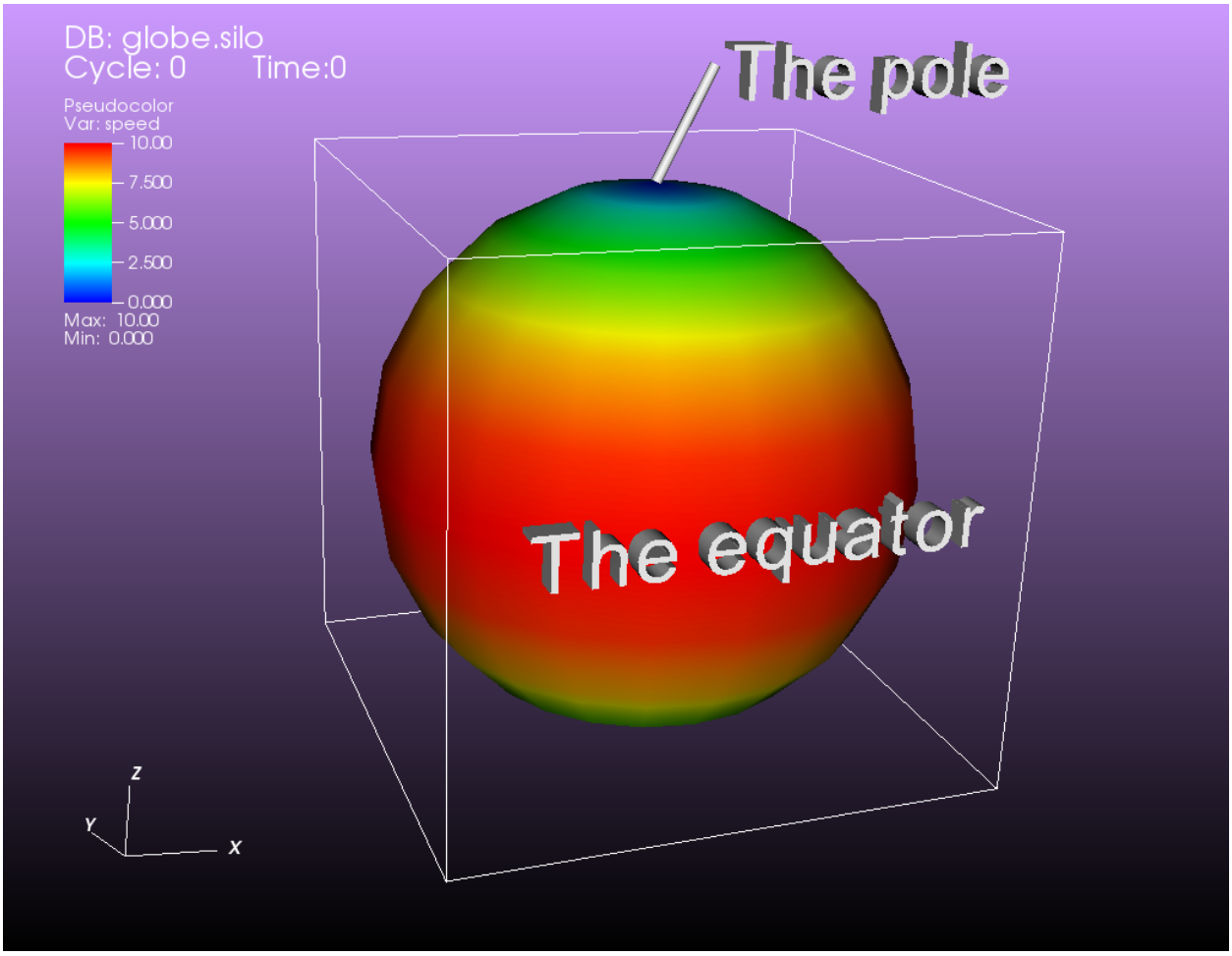


Fig. 4.349: Examples of 3d text annotations

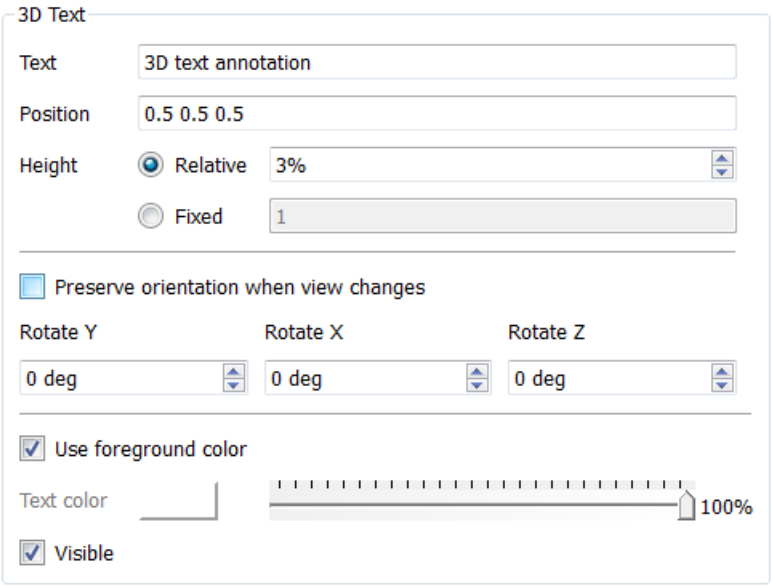


Fig. 4.350: The 3D text annotation interface

the orientation relative to the screen, select the **Preserve orientation when view changes** radio button. To make the orientation relative to the simulation coordinate system, uncheck the **Preserve orientation when view changes** radio button. To set the orientation, set the **Rotate Y**, **Rotate X** and **Rotate Z** spin boxes. The rotations are applied in the left to right order of the spin boxes in the interface.

Time slider annotation objects

Time slider annotation objects, shown in Figure 4.351, are created by clicking the Time slider button in the **Create new** area on the **Objects** tab. Time slider annotation objects consist of a graphic that shows the progress through an animation using animation and text that shows the current database time. Time slider annotation objects can be placed anywhere in the visualization window and you can set their size, text, colors, and appearance properties.

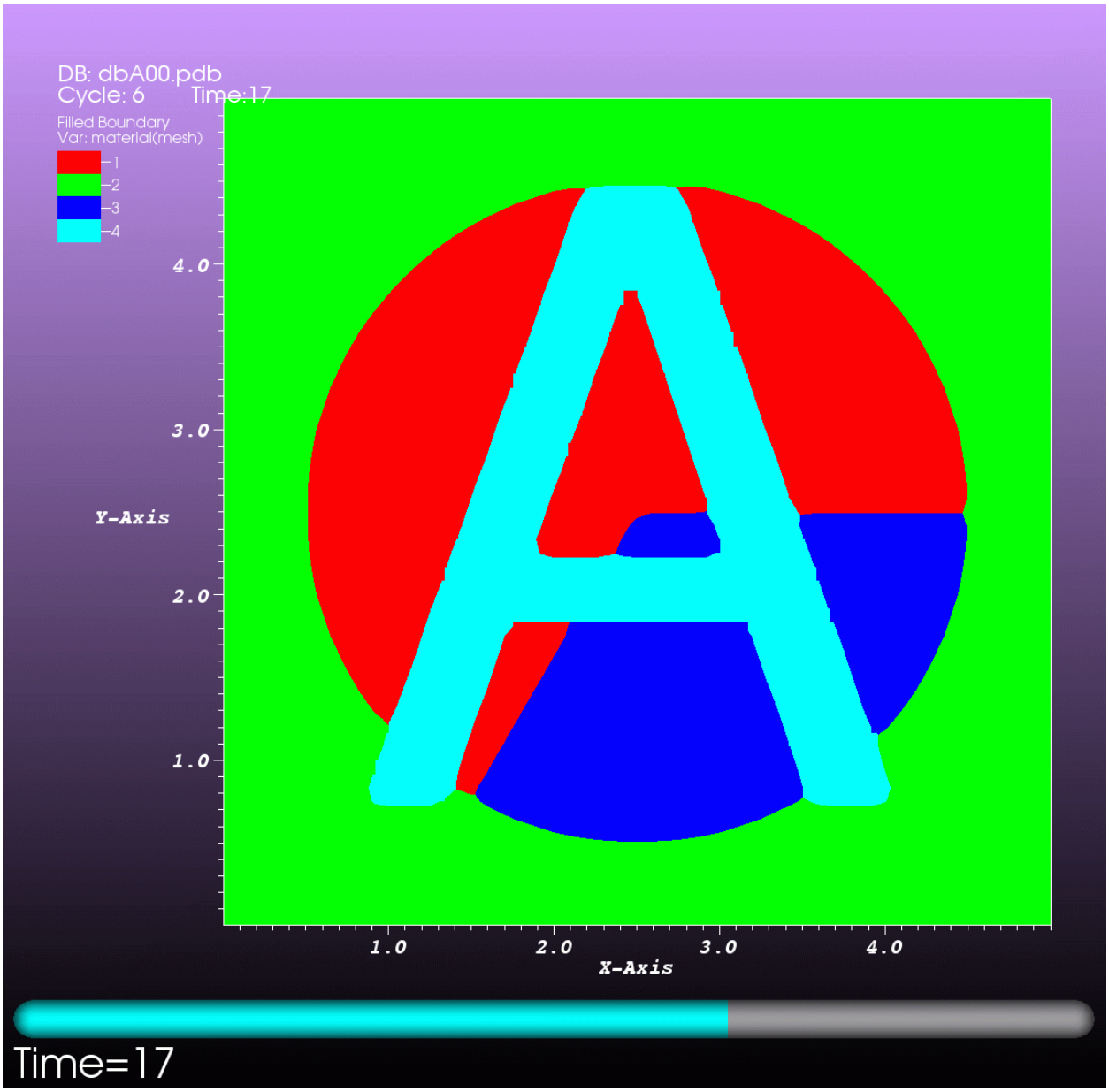


Fig. 4.351: An example of a time slider annotation object

Time slider annotation objects are placed using 2D coordinates where the X, and Y values are in the range [0,1]. The point (0,0) corresponds to the lower left corner of the visualization window and the point (1,1) corresponds to the upper right of the visualization window. The 2D coordinate used to position the text annotation matches the text annotation's lower left corner. To position a text annotation object, enter a new 2D coordinate into the **Lower left** text field. You can also click the down arrow next to the **Lower left** text field to interactively choose a new lower left coordinate for the text annotation using the screen positioning control, which represents the visualization window.

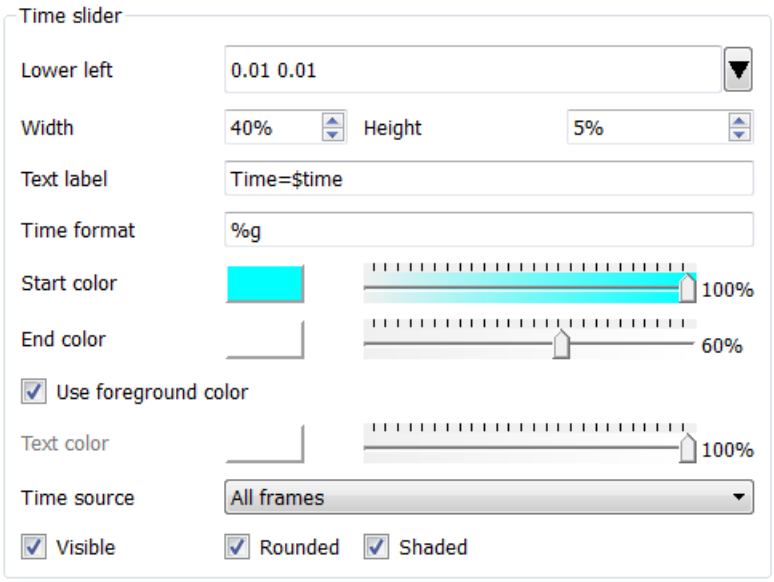


Fig. 4.352: The time slider interface

The size of a time slider annotation object is controlled by settings its height and width as a percentage of the visualization window height and width. Type new values into the **Width** and **Height** spin buttons to set a new width or height for the time slider annotation object.

You can set the text displayed by the time slider annotation object by typing a new text string into the **Text label** text field. Text is displayed below the time slider annotation object and it can contain any message that you want. The text can even include wildcards such as *$time*, which evaluates to the current time for the active database. If you use *$time* to make VisIt incorporate the time for the active database, you can also specify the format string used to display the time. The format string is a standard C-language format string (e.g. "%4.6g") and it determines the precision used to write out the numbers used in the time string. You will probably want to specify a format string that uses a fixed number of decimal places to ensure that the time string remains the same length during the animation, preventing distracting differences in the length of the string from taking the eye away from the visualization. Type a C-language format string into the **Time format** text field to change the time format string.

Time slider annotations have three color attributes: start color, end color, and text color. A time slider annotation object displays time like a progress bar in that the progress bar starts out small and then grows to the right until it takes up the whole length of the annotation. The color used to represent the progress can be set by clicking the **Start color** button and choosing a new color from the **Popup color** menu. As the time slider annotation object shows more progress, the color that is used to fill up the time that has not been reached yet (end color) is overtaken by the start color. To set the end color for the time slider annotation object, click the **End color** button and choose a new color from the **Popup color** menu. Normally, time slider annotation objects use the foreground color of the visualization window when drawing the annotation's text. If you want to make the annotation use a special color, turn off the **Use foreground color** check box and click the **Text color** button and choose a new color from the **Popup color** menu.

Time slider objects have two more attributes that affect their appearance. The first of those attributes is set by clicking on the **Rounded** check box. When a time slider annotation object is rounded, the ends of the annotation are curved. The last attribute is set by clicking on the **Shaded** check box. When a time slider annotation object is shaded, simple

lighting is applied to its geometry and the annotation will appear to be more 3-dimensional.

2D line annotation objects

2D line annotation objects, shown in Figure 4.354, are created by clicking the **2D Line** button in the **Create new** area on the **Objects** tab. 2D line annotation objects are simple line objects that are drawn on top of plots in the visualization window and are useful for pointing to features of interest in a visualization. 2D line annotation objects can be placed anywhere in the visualization window and you can set their locations, arrow properties, and color.

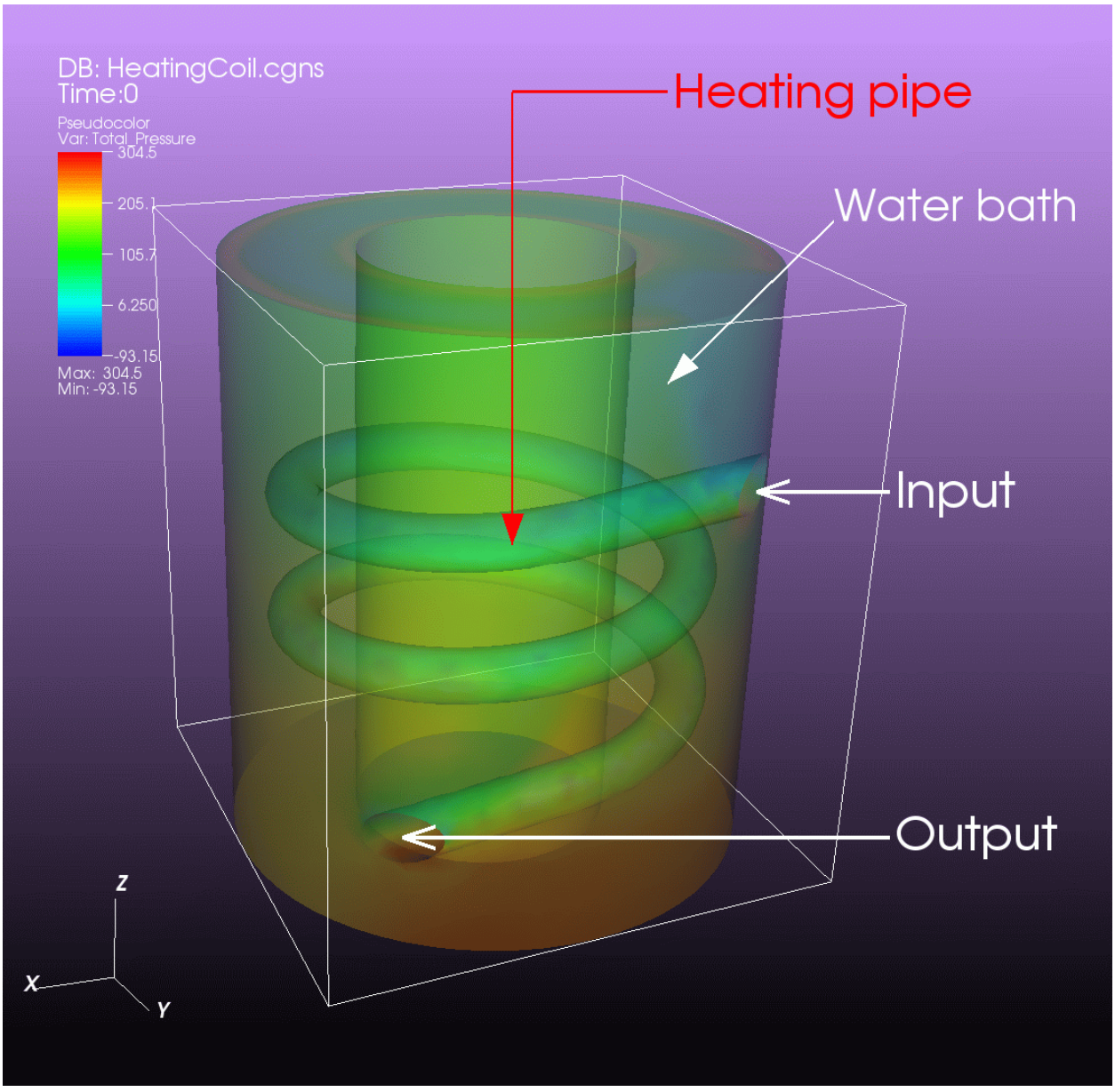


Fig. 4.353: Examples of 2D line annotations

2D line annotations are described mainly by two coordinates that specify the start and end points for the line. The start and end coordinates are specified as pairs of floating point numbers in the range [0,1] where the point (0,0) corresponds to the lower left corner of the visualization window and the point (1,1) corresponds to the upper right corner of the

visualization window. You can set the start or end points for the 2D line annotation by entering new start or end points into the **Start** or **End** text fields in the 2D line object interface. You can also click the down arrow to the right of the **Start** or **End** text fields to interactively choose new coordinates using the screen positioning control.

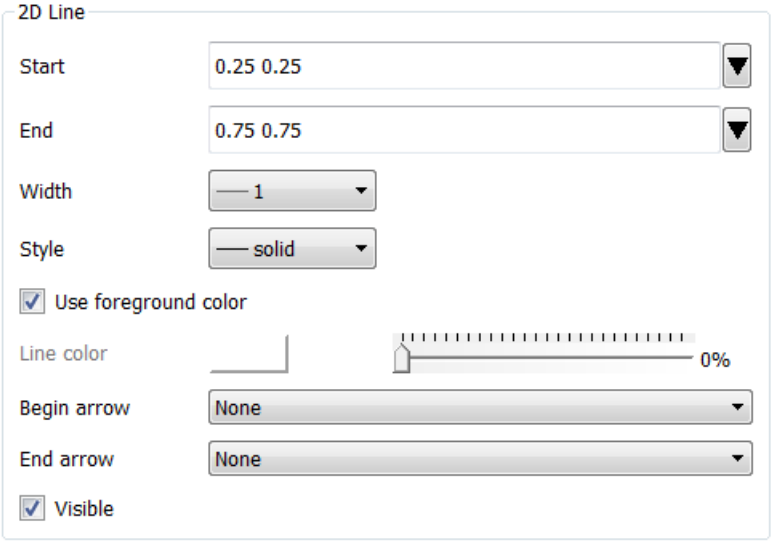


Fig. 4.354: The 2D line object interface

Once the 2D line annotation has been positioned there are other attributes that can be set to improve its appearance. First of all, if the 2D line annotation is being used to point at important features in a visualization, you might want to increase the 2D line annotation's width to make it stand out more. To change the width, select the new pixel width from the **Width** menu. It is also possible to set the line style. To change the style of the line, select the new line style from the **Style** menu. After changing the width and style, the color of the 2D line annotation should be chosen to stand out against the plots in the visualization. The color that you use should be chosen such that the line contrasts sharply with the plots over which it is drawn. To choose a new color for the line, click on the **Line color** button and choose a new color from the **Popup color** menu. You can also adjust the opacity of the line by using the opacity slider next to the **Line color** button.

The last properties that are commonly set for 2D line annotations determine whether the end points of the line have arrow heads. The 2D line annotation supports two different styles of arrow heads: filled and lines. To make your line have arrow heads at the start or the end, make new selections from the **Begin arrow** and **End arrow** menus.

3D line annotation objects

3D line annotation objects, shown in Figure 4.349, are created by clicking the **3D Line** button in the **Create new** area on the **Objects** tab. 3D line annotation objects are lines that are positioned in 3D and are part of the 3D scene, so they may become obscured by other objects in the scene and will move in space as the image is panned and zoomed.

The 3D line annotation object properties, shown in Figure 4.355, can be used to set the position, style and color properties.

3D text annotation objects are placed in 3D coordinates in the same coordinate system used by the simulation data. To position a 3D line annotation object, specify the start and end location of the line by entering the start location in the **Start** text field and the end location in the **End** text field.

There are two types of lines supported, one is a normal line and the other is a tube. The line type is selected through the **Line type** menu. When using a normal line, you can specify the normal line width and line style properties using the **Line Width** and **Line Style** menus. When using a tube you can specify the tube quality and radius. The tube is

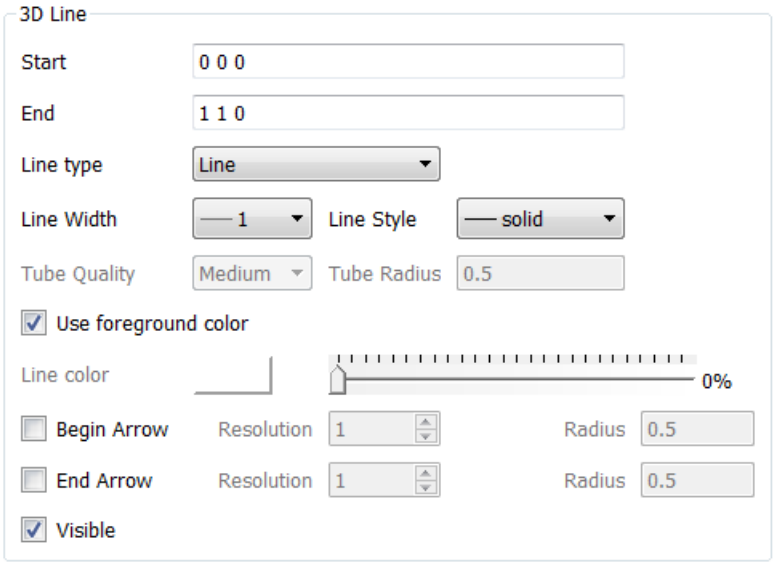


Fig. 4.355: The 3D line object interface

created from a series of flat surfaces around the center of the line to approximate a tube. The number of surfaces used is controlled by the tube quality. The tube radius is the radius of the tube in the coordinate system of the simulation data. These properties can be changed through the **Tube Quality** and **Tube Radius** menus.

It is also possible to add arrows to the beginning and end of the line. These can be enabled with the **Begin Arrow** and **End Arrow** toggle buttons. For each arrow, the user can also control the resolution and radius of the arrows. The arrows consist of cones places at the ends of the line and are constructed out of triangles that approximate a cone. The number of triangles used is controlled by the resolution. The radius is the radius of the cone in the same coordinate system as the simulation data. The resolution can be changed using the **Resolution** spin box and the radius is changed by typing a new value into the **Radius** text field.

Image annotation objects

Image annotation objects, shown in Figure 4.356, are created by clicking the **Image** button in the **Create new** area on the **Objects** tab. Image annotation objects display images from image files on disk in a visualization window. Images are drawn on top of plots in the visualization window and are useful for adding logos, pictures of experimental data, or other views of the same visualization. Image annotation objects can be placed anywhere in the visualization window and you can set their size, and optional transparency color.

The first step in incorporating an image annotation into a visualization is to choose the file that contains the image that will serve as the annotation. To choose an image file for the image annotation, type in the full path and filename to the file that you want to use into the **Image source** text field. You can also use the file browser to locate the image file if you click on the "..." button to the right of the **Image source** text field in the **Image annotation interface**, shown in Figure 4.357.

Warning: Currently, there is a limitation in the use of image annotations while operating in client/server mode. The *path* used must be the *same* on both the local (client) and remote (server) machines. Often, the only way to achieve this may be to have the image file in `/tmp` or `/var/tmp`. For parallel engines, this may necessitate a complicated set of manual steps to a) wait for the parallel job to start, b) identify the node running MPI rank 0 (the only MPI rank where annotations are processed) and c) copy the file to the `/tmp` directory on that node assuming access controls will even allow that.

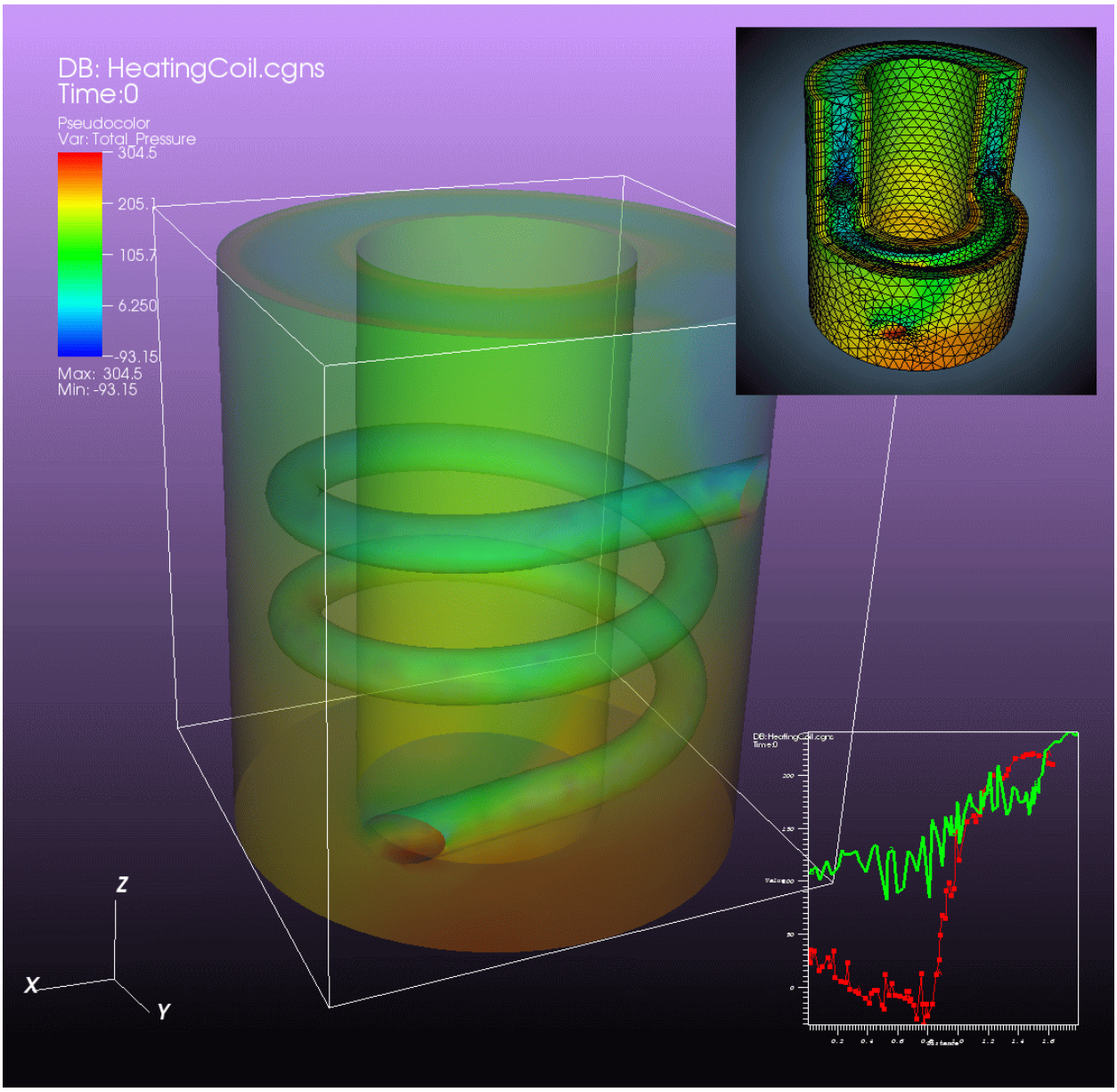


Fig. 4.356: An Example of a visualization with two overlaid image annotations

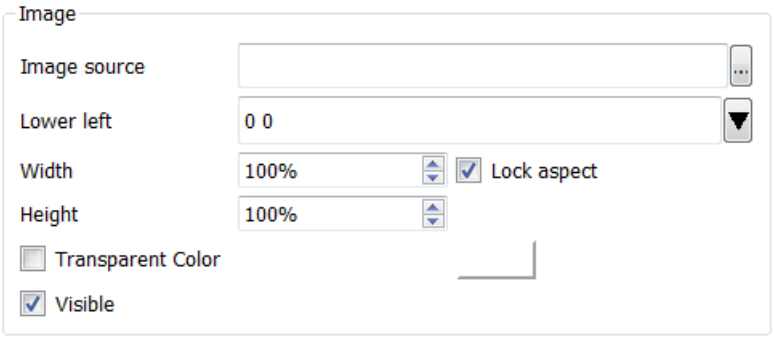


Fig. 4.357: The image object interface

After selecting an image file, you can position its lower left coordinate in the visualization window. The lower left corner of the visualization window is the origin (0,0) and the upper right corner of the visualization window is (1,1).

Once you position the image where you want it, you can optionally scale it relative to its original size. Unlike some other annotation objects, the image annotation does not scale automatically when the visualization window changes size. The image annotation will remain the same size - something to take into account when setting up movies that use the image annotation. To scale the image relative to its original size, enter new percentages into the **Width** and **Height** spin boxes. If you want to scale one dimension of the image and let the other dimension remain unchanged, turn off the **Lock aspect** check box.

Finally, if you are overlaying an image annotation whose image contains a constant background color or other area that you want to remove, you can pick a color that VisIt will make transparent. For example, Figure 4.356 shows an image of some Curve plots overlaid on top of the plots in the visualization window and the original background color in the annotation object was removed to make it transparent. If you want to make a color in an image transparent before VisIt displays it as an image annotation object, click on the **Transparent color** check box and then select a new color by clicking on the **Transparent color** button and picking a new color from the **Popup color menu**.

Legend annotation objects

Legends are automatically added when plots are created, and have names that include the plot type. Figure 4.358 shows multiple legends listed in the **Annotation Objects** tab. The Legend annotation object interface can be used to customize the legends position, size, tick labels and aspects of its appearance.

The **Position** tab, shown in Figure 4.359 has controls for position, size and orientation. VisIt generally controls the positions of legends, but if you want a specific legend to be placed elsewhere in the visualization window, uncheck the **Let VisIt manage legend position** checkbox and modify the **Legend position**. The 2D coordinate used to position the legend matches the legend's lower left corner. To change the position of a legend, enter a new 2D coordinate into the **Legend position** text field. You can also click the down arrow next to the **Legend position** text field to interactively choose a new lower left coordinate using the screen positioning control, which represents the visualization window. The screen positioning control, shown in Figure 4.348, lets you move a set of cross-hairs to any point on a square area that represents the visualization window. Once you release the left mouse button, the location of the cross-hairs is used as the new coordinate for the legend's lower left corner.

The **X scale** and **Y scale** spin boxes control the size of the legend, with values of '100%' being the default size. You can enter new values using the text field or use the + and - buttons to the right of the text field.

The **Orientation** of the legend is changed using the corresponding drop-down menu, with options: **Vertical, Text on Right**; **Vertical, Text on Left**; **Horizontal, Text on Top** and **Horizonatal, Text on Bottom**.

The **Tick marks** tab, shown in Figure 4.360, allows you to change the display of a legend's tick marks. You can turn off the drawing of tickmarks completely by choosing the **None** option in the **Draw** dropdown menu. Other **Draw** options are **Values** (the default), **Labels** and **Values and Labels**. Legends usually only have **Values** defined for tick marks, so

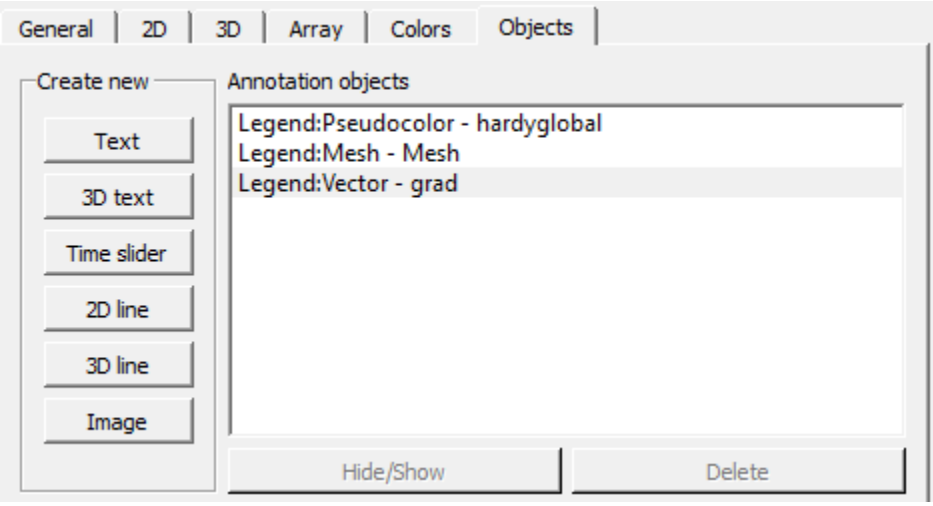


Fig. 4.358: Multiple legends in the **Annotation Objects** tab.

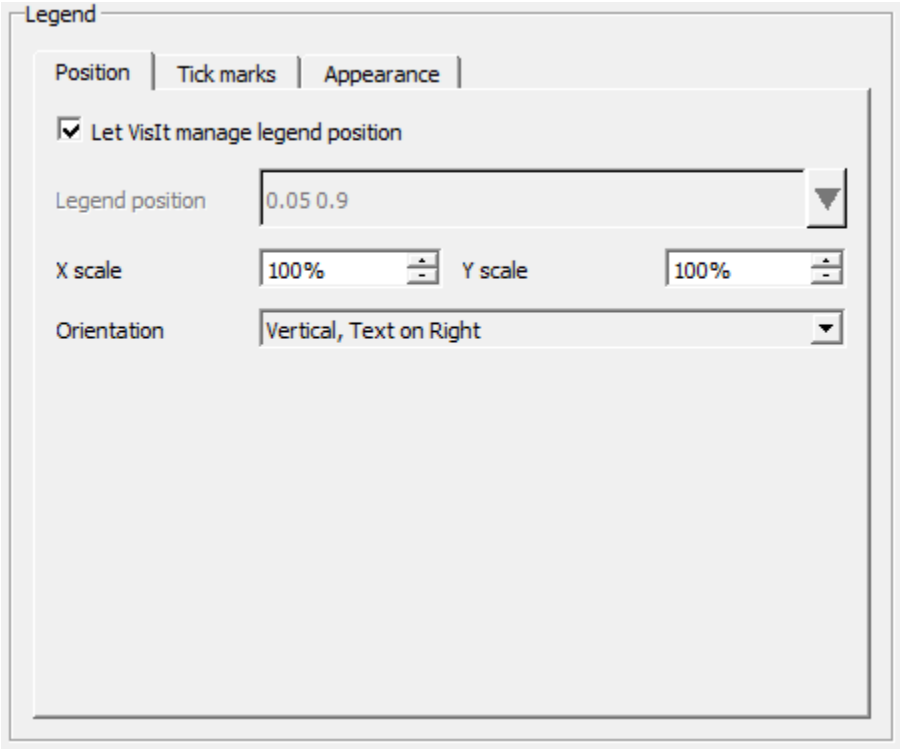


Fig. 4.359: The legend object interface for position

to display **Labels**, you would have to define them by unchecking the **Let VisIt determine tick labels** checkbox and modifying the **Specify tick values and labels** table. It starts out with the **Values** column filled in with defaults. You can modify those, add text to the **Labels column** and change the number of table items by using the **Add tick value** and **Delete selected value** buttons. When modifying, adding, or deleting values in the table, keep in mind they must fall within the Min/Max range of the current plot or they won't be displayed in the legend. Not all plot types allow adding or deleting values.

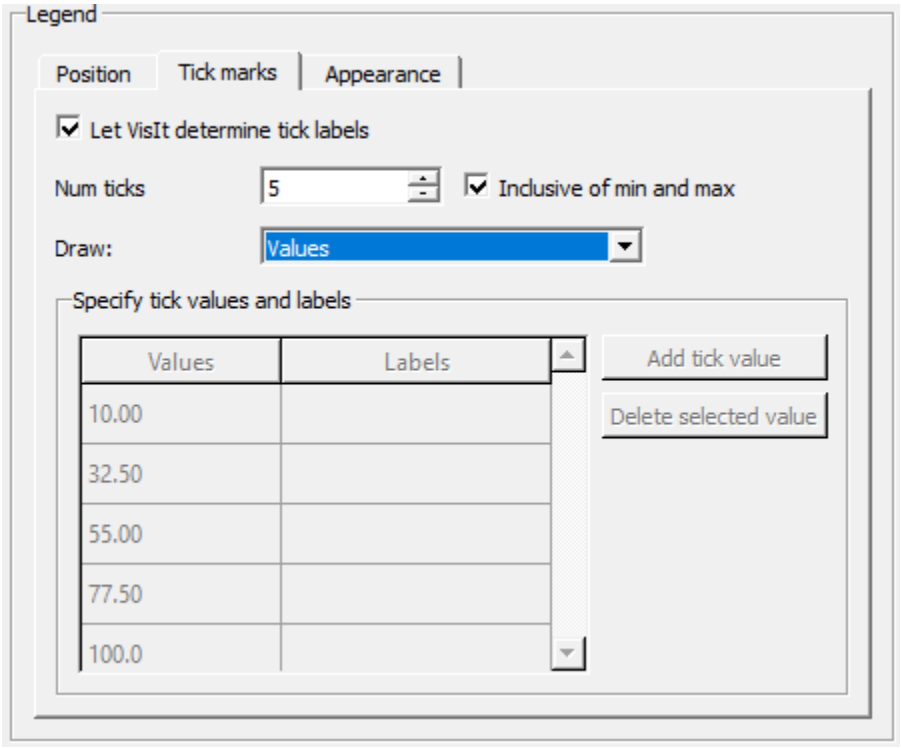


Fig. 4.360: The legend object interface for tick marks

Figure 4.361 shows an example of modifying and displaying **Labels** for a *Boundary and FilledBoundary Plots*. The **Values** have both a number and the name of the material, and the **Labels** were added so that only the material names would be displayed in the legend.

Figure 4.362 shows the controls for changing the **Appearance** of a legend.

The **Bounding box** checkbox controls whether or not a background bounding box is drawn behind the legend. When checked, widgets for controlling the color and opacity of the background bounding box will become active.

The title can be turned off via the **Draw title** checkbox. A custom title can be specified via the **Custom title** checkbox and text edit widgets.

The **Draw min/max** checkbox controls whether or not Min/Max text (where applicable) will be added to the legend.

By unchecking the **Use foreground color** checkbox, you can change the color of the legend's text and tickmarks via the **Text color** button.

Number format controls the fromat for tick mark and Min/Max values. You can modify the font style via the **Font height** text edit, **Font family** dropdown menu and **Bold** and **Italic** check boxes. (**Shadow** is currently disabled).

Figure 4.363 shows an example of a modified legend, where position, orientation, size, tick marks, background font height and font family have all been changed.

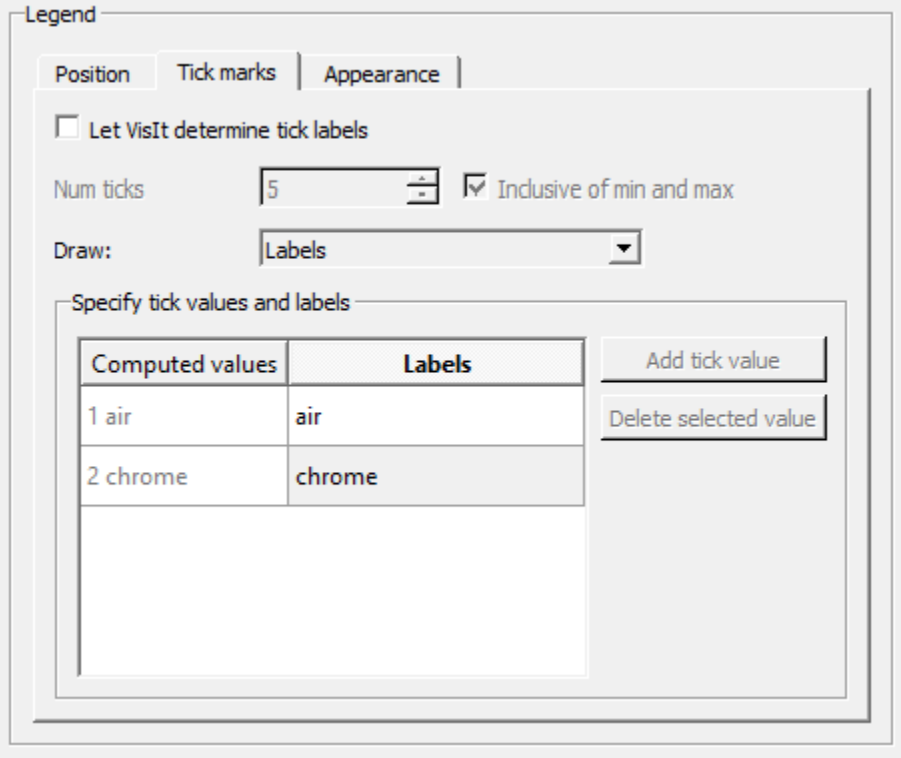


Fig. 4.361: An example of modifying the Labels on a filled boundary legend

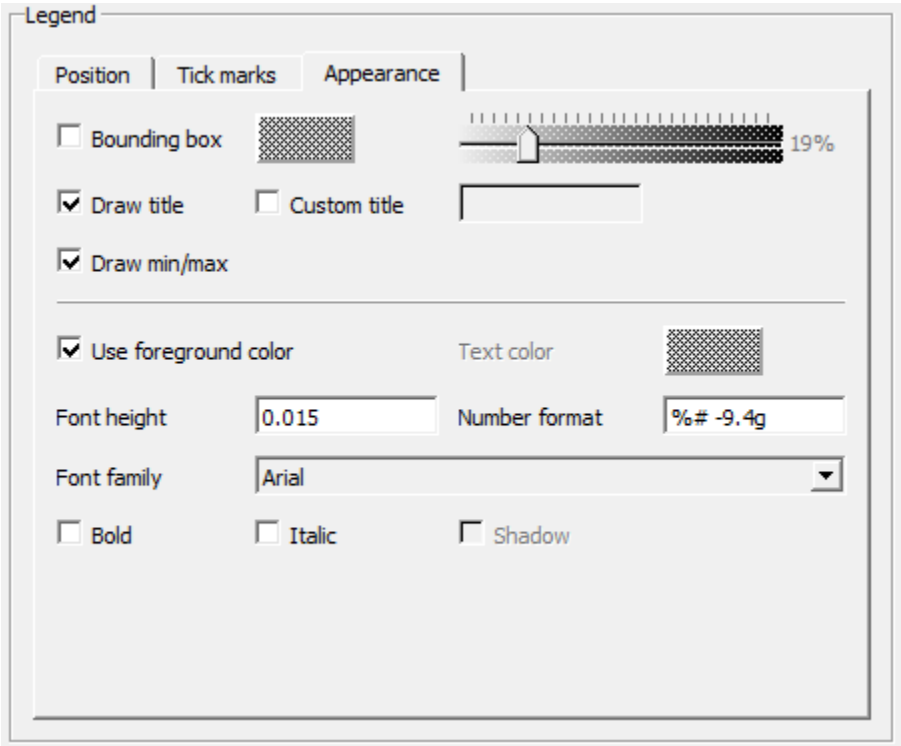


Fig. 4.362: The legend object interface for appearance

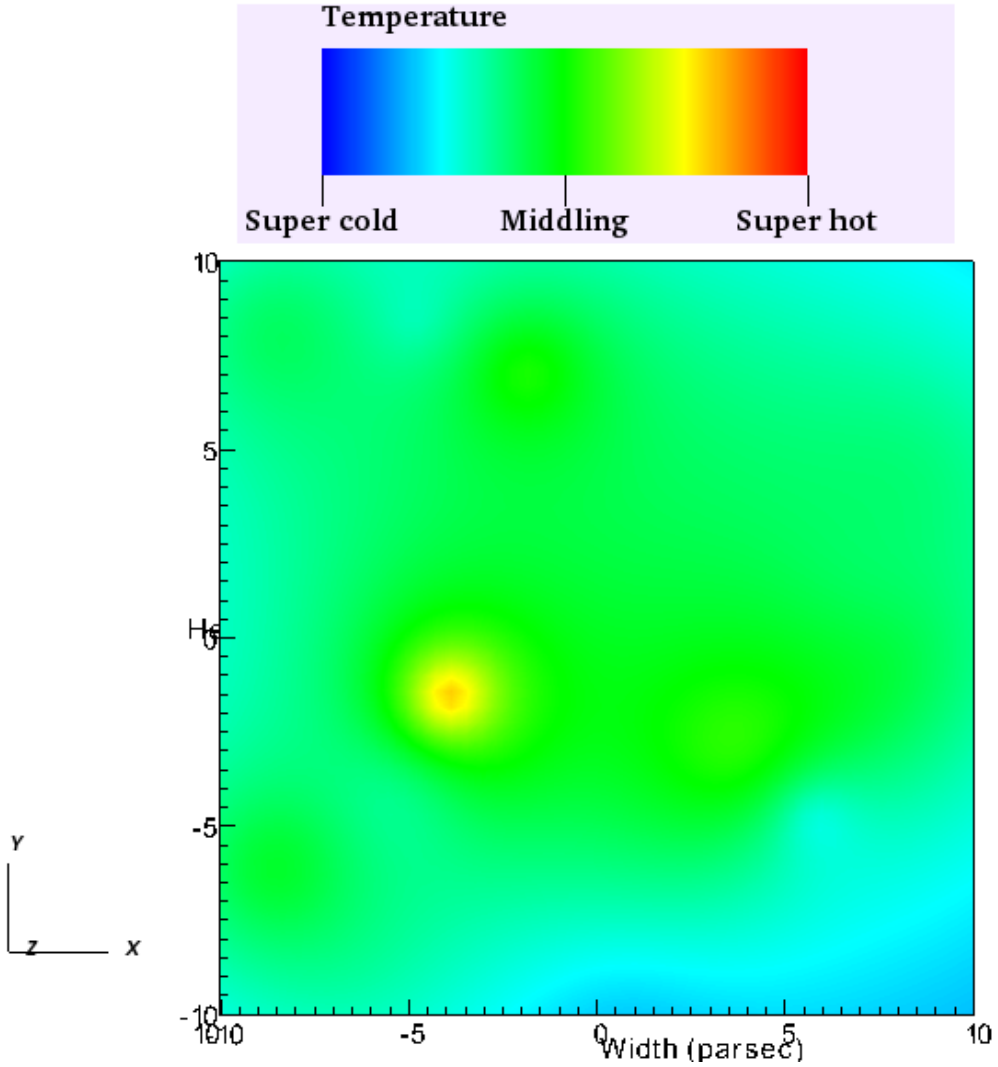


Fig. 4.363: Example of a modified legend

4.9.2 Color Tables

A color table is a set of colors that is used by certain plots to color variables. Color tables can be immensely important for understanding visualizations since changes in color can highlight interesting features. VisIt has several built-in color tables that can be used in visualizations. VisIt also provides a **Color Table Window** for organizing and designing color tables.

Color tables come in two types: continuous and discrete. Their definitions are dependent upon our notion of a color control point, which is defined as a point in one dimensional space (also between 0 and 1) that has color information (i.e. RGBA values). A continuous color table is defined as a set of (relatively few) color control points defined at certain intervals such that the gaps between adjacent points are filled by smoothly interpolating their colors. This makes continuous color tables look smooth since there are several colors that are blended to form the color table. Continuous color tables are used by several plots including the Pseudocolor, Tensor, and Vector plots. A plot that uses a continuous color table attempts to use all of the colors in the color table. Some plots that opt to only use a handful of colors from a continuous color table pick colors that are evenly distributed throughout the color table so that the plots end up with colors that still somewhat resemble the original colors from the continuous color table.

A discrete color table is a set of N colors that can be set individually. There are no other colors in a discrete color table other than the colors that you provide. Internally, a discrete color table is represented as a set of color control points. Their respective positions correspond to the order the colors appear in. Discrete color tables are usually used by plots like the Boundary, Contour, FilledBoundary, or Subset plots, which need only a small set of colors. Typically, these plots use a color from a discrete color table to color some object and then use the next color to color another object, and so on. When they reach the end of the color table and still need more colors, they start again at the beginning with the first color from the discrete color table.

Color Table Window

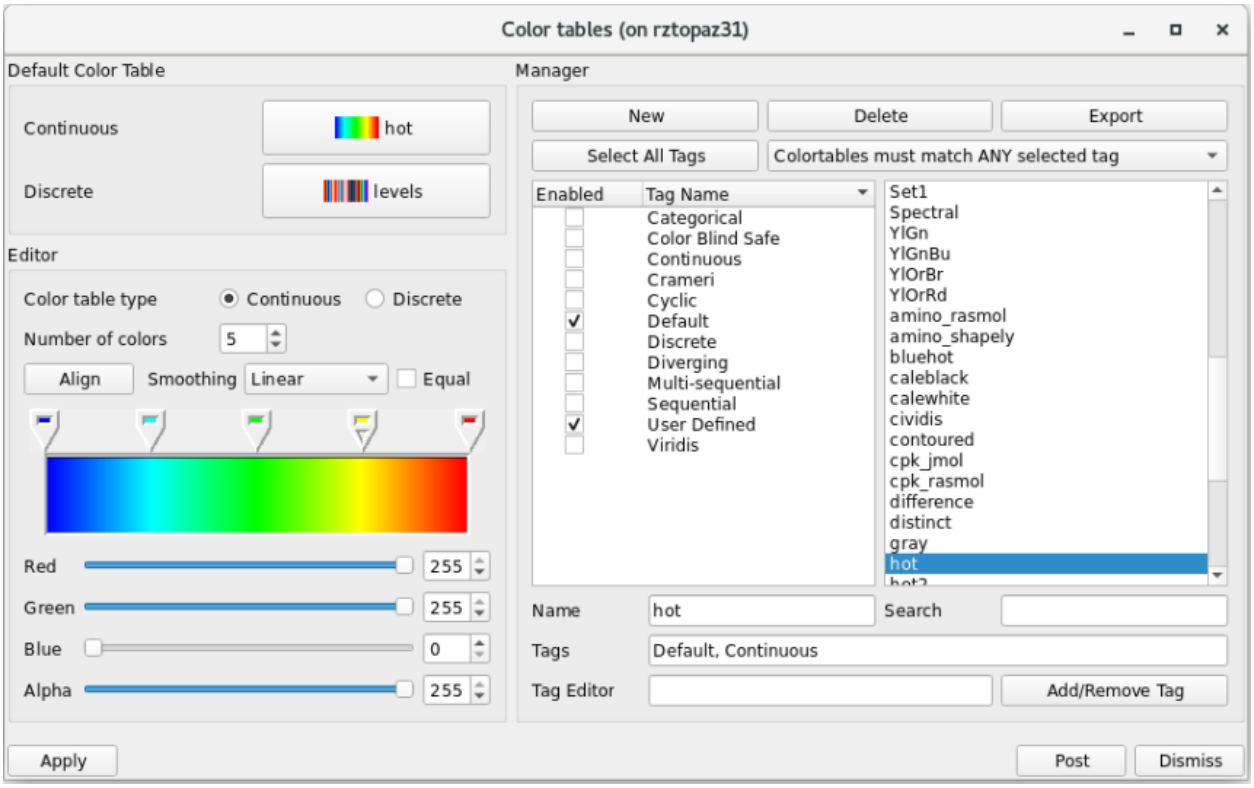


Fig. 4.364: The color table window

You can open VisIt's **Color table window**, shown in Figure 4.364, by selecting **Color table** from the **Main Window**'s **Controls** menu, or by pressing `ctrl + t`. The **Color table window** is separated into three areas: The top area allows you to set the default color tables. The right-hand side, or the manager portion of the window, allows you to create, delete, and export new color tables, as well as organize all the available color tables. See *Filtering With Tags* for more information. The left-hand side, or editor portion of the window, allows you to edit color tables by adding, removing, moving, or changing the color of control points. A color control point is a point with a color that influences how colors vary through the table.

Setting the default color table

VisIt has a concept of default color tables, which are the color tables used to color plots that do not specify a color table. There is both a default continuous color table (for plots that prefer to use continuous color tables) and a default discrete color table (for plots that prefer to use discrete color tables). The default color table can be different for each visualization window. To set the default continuous color table, select a new color table name from the **Continuous** menu in the **Default color table** area. To select a new default discrete color table, select a new color table name from the **Discrete** menu in the **Default color table** area. If you would like to see more or less color table options for both the **Continuous** and **Discrete** buttons, see *Filtering With Tags* for instructions on filtering which color tables are available to be used.

Creating a new color table

Creating a new color table is a simple process where you first type a new color table name into the **Name** text field and then click the **New** button. This creates a copy of the currently highlighted color table, which is the color table that is selected in the **Manager** area, and inserts it into the color table list with the specified name. After creating the new color table, you can modify the color control points and other attributes to fashion a new color table.

Deleting a color table

To delete a color table, click on a color table name in the color table list and then click the **Delete** button. You can only delete color tables that are not built-in to VisIt. When you delete a color table that is currently set as one of your default color tables, the default color table is set to the color table that comes first in the list. If a color table is in use when it is deleted, plots that used the deleted color table will use the default color table from that point on.

Exporting a color table

If you design a color table that you want to share with colleagues or otherwise use in the future, click the **Export** button in the **Manager** area to save an XML file containing the color table definition for the highlighted color table to your .visit directory. The name of a color table file will usually be composed of the name of the color table with a *".ct"* extension. Copying a color table file to a user's .visit directory will allow VisIt to find the color table the next time VisIt runs. Look for the color table file in the directory in which VisIt was installed if you use the Windows version of VisIt.

Editing a continuous color table

There are a handful of controls in the editor portion of the **Color table window**, shown in Figure 4.365, that are used to change the definition of a color table. To change a color table definition, you must alter its color control points. This means adding and removing color control points as well as changing their colors and locations.

You can change the number of color control points in a color table using the **Number of colors** spin box. When a new color control point is added, it appears to the right of the selected color control point and to the left of the next color

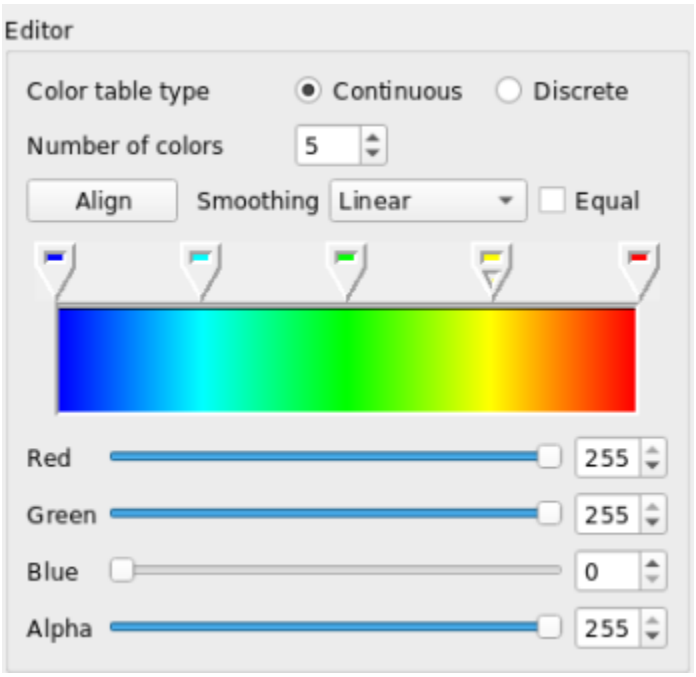


Fig. 4.365: The continuous color table editor

control point. Color control points are represented as a pointy box just above the color spectrum. The color control point that has a small triangular mark is the selected color control point. When a color control point is removed, the color control point that was created before the deleted color control point becomes the new selected color control point. Clicking the **Align** button makes all color control points have equal spacing.

Clicking on a color control point makes it active. You can also use the Space bar if the color spectrum has keyboard focus. Clicking and dragging on a color control point changes its position. Clicking the arrow keys on the keyboard also moves a color control point. To change a color control point's color, right click on it and choose a new color from the **Popup color** menu that appears under the mouse cursor. You can also change the color control point's color by making the color control point active and then using the **Red**, **Green** and **Blue** sliders.

The **Color table window** also has a couple of settings that can be set to influence a color table's appearance without having permanent effects on the color table. The **Smoothing** menu can be used to select between no smoothing, linear smoothing and cubic spline smoothing. The **Equal** check box can temporarily tell the color table to ignore the positions of its color control points and use equal spacing instead. The **Equal** check box is often used with no smoothing.

Editing a discrete color table

The **Color table window's Editor** area looks different when you edit a discrete color table. Instead of showing a spectrum of colors, the window shows a grid of colors that correspond to the colors in the discrete color table. The order of the color control points is left to right, top to bottom. To edit a discrete color table, first left click on the color that you want to edit and then use the **Red**, **Green**, and **Blue** sliders to change the color. You can also right click on a color to select it and open the **Popup color** menu to choose a new color.

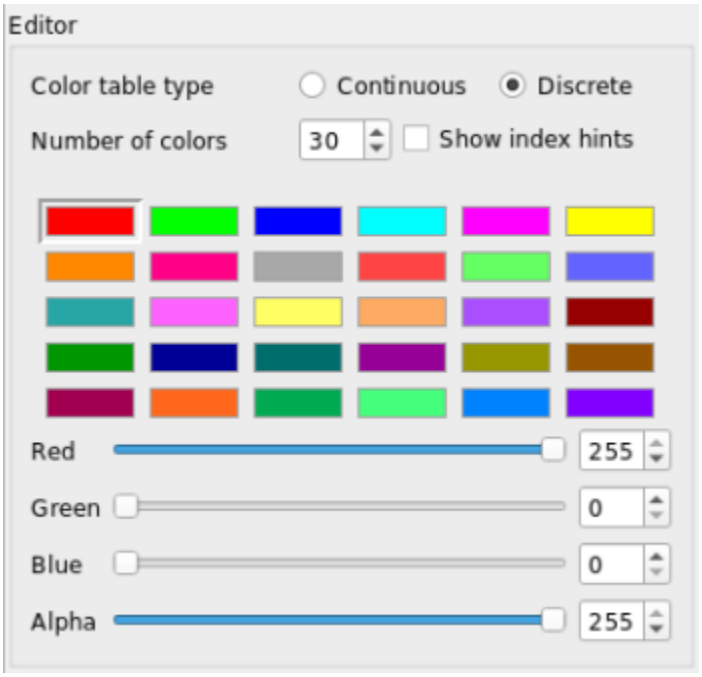


Fig. 4.366: The discrete color table editor

Editing color control point positions numerically

In both of the sections above, the color control points are *positioned* graphically using the GUI rather than specifying explicit numerical values.

When a variable is plotted, the variable's values are mapped to the range `[0...1]` to determine the colors to associate with the variable's values. A color table defines a 1:1 association of that range with a set of color control points. Each control point in a color table is assigned a *position* in the `[0...1]` range.

Sometimes, users want specific numerical values to map to specific colors. There is no way to achieve this through VisIt's color table GUI. The only solution is to edit a color table manually or, if there are a large number of color control points to edit, to create a script that produces the color table.

For example, a user wanted a smoothly graded coloring of a variable using the following logic and colors...

Variable Value Range	Hex Color
<0	cccccc
3	66ccff
10	66ff66
25	ffffcc
50	ffff00
100	ff9900
1000	ff0000
>=5000	9900cc

The above table has 8 colors. The input variable has range `[0...5000]`. The first step is to *normalize* the variable's value transitions to the `[0...1]` interval and convert the hexadecimal values to rgb colors using a color conversion tool. This information is in the table below.

Normalized Variable Value	RGB Color
<0.0 (0/5000)	204 204 204
0.0006 (3/5000)	102 204 255
0.002 (10/5000)	102 255 102
0.005 (25/5000)	255 255 204
0.01 (50/5000)	255 255 000
0.02 (100/5000)	255 153 000
0.2 (1000/5000)	255 000 000
>=1.0 (5000/5000)	153 000 204

To create this color table, start VisIt's GUI and go to *Controls* → *Color table ...* . There, enter a name for the color table in the **Name** text box. Lets say it is named `my8colors`. Clicking the **New** button adds the named table to the list of color tables, copying the settings of the default color table. For the example above, we want the **Number of colors** to be set to 8 and the **Color table type** to be **Continuous**. To create a file for this color table that can be edited with a text editor, it needs to be exported by clicking the **Export** button. This will create an XML file in *VUSER_HOME/my8colors.ct* with 8 color control point entries in it. At this point, the user should exit VisIt. With a text editor, the user can now edit the file `my8colors.ct`. Starting at the *top* of the file where the *first* color control point is defined (e.g. the one closest to the *zero* end of the `[0...1]` range), edit the *position* and *rgb color* of the first control point to match the values in the above table. Note that there is a 4th entry for each rgb color. This is for setting *transparency* of that color in the range `[0...255]` where `0` is fully transparent and `255` is fully opaque. If *transparency* effects are not needed, this 4th entry can be ignored and just always set equal to `255`.

When VisIt is restarted, it will load this color table file. The user can then set this color table as the one to be used in various plots.

One final issue to deal with in this example is how to handle the user's goal of having all *negative* values in the input variable map to the first color in the color table and all values greater or equal to 5000 to the last color. To do this, the user will have to define a new variable to plot using a *conditional expression* of the form `if(lt(var,0),0,if(ge(var,5000),5000,var))` where `var` is the variable and then use this new *expression variable* in place of `var` for the desired behavior.

Show/Hide XML color table file

```
<?xml version="1.0"?>
<Object name="ColorTable">
    <Field name="Version" type="string">3.0.1</Field>
    <Object name="ColorControlPointList">
        <Object name="ColorControlPoint">
            <Field name="colors" type="unsignedCharArray" length="4">204 204 204 255
↪</Field>
            <Field name="position" type="float">0.0</Field>
        </Object>
        <Object name="ColorControlPoint">
            <Field name="colors" type="unsignedCharArray" length="4">102 204 255 255
↪</Field>
            <Field name="position" type="float">0.0006</Field>
        </Object>
        <Object name="ColorControlPoint">
            <Field name="colors" type="unsignedCharArray" length="4">102 255 102 255
↪</Field>
            <Field name="position" type="float">0.002</Field>
        </Object>
        <Object name="ColorControlPoint">
            <Field name="colors" type="unsignedCharArray" length="4">255 255 204 255
↪</Field>
            <Field name="position" type="float">0.005</Field>
```

(continues on next page)

(continued from previous page)

```
        </Object>
        <Object name="ColorControlPoint">
            <Field name="colors" type="unsignedCharArray" length="4">255 255 0 255 </
↪Field>
            <Field name="position" type="float">0.01</Field>
        </Object>
        <Object name="ColorControlPoint">
            <Field name="colors" type="unsignedCharArray" length="4">255 153 0 255 </
↪Field>
            <Field name="position" type="float">0.02</Field>
        </Object>
        <Object name="ColorControlPoint">
            <Field name="colors" type="unsignedCharArray" length="4">255 0 0 255 </
↪Field>
            <Field name="position" type="float">0.2</Field>
        </Object>
        <Object name="ColorControlPoint">
            <Field name="colors" type="unsignedCharArray" length="4">153 0 204 255 </
↪Field>
            <Field name="position" type="float">1</Field>
        </Object>
        <Field name="tags" type="stringVector">"UserDefined"</Field>
    </Object>
</Object>
```

Numerically Controlled Banded Coloring

Sometimes it is convenient to create *banded* coloring of smoothly varying data. For example, it is an easy way to create something akin to a contour plot. In VisIt, it is possible to do this with either a **Continuous** or a **Discrete** color table. In general, it is much easier with a **Continuous** color table (with smoothing set to `None`). Trying to do the same thing with a **Discrete** color table requires an additional step to create a *companion conditional expression* which implements the non-uniform banding.

For example, given the smoothly varying variable, `u`, in the range `[-1...+1]` shown in normal (e.g. `hot`) **Pseudocolor** plot in Fig. 4.367.

we would like to produce a 4-color banded plot using the coloring logic below...

Values in Range	Map to this Color
-inf...-0.95	blue
-0.95...0	cyan
0...+0.95	green
+0.95...+inf	red

Banded Coloring with Continous Color Table

The input variable's range is `[-1...+1]` and this range needs to be mapped into the color-table range `[0...1]`. This results in color table *positions* for the four colors to be used in the mapping table above...

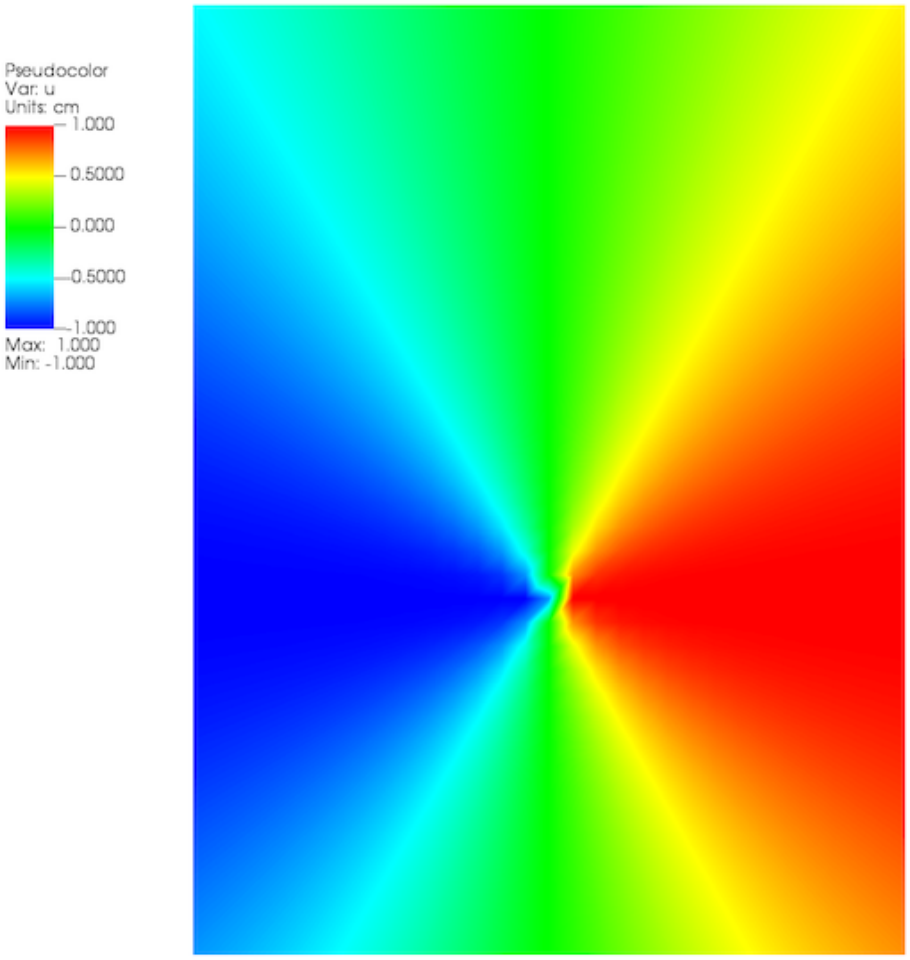


Fig. 4.367: Smoothly colored variable using `hot` color table.

Color Table Position	Color Table Color
-inf...0.0125	blue
0.0125...0.5000	cyan
0.5000...0.9875	green
0.9875...+inf	red

This results in the banded coloring in Fig. 4.368

To create the color table for this result, follow the instructions for *editing color control point positions numerically*. After editing the color table, re-import it back into VisIt. Make sure the color table is **Continuous** and that the *smoothing* is set to `None`. Apply this color table to the Pseudocolor plot used to plot the variable `u` and the same coloring as shown in the figure should be observed.

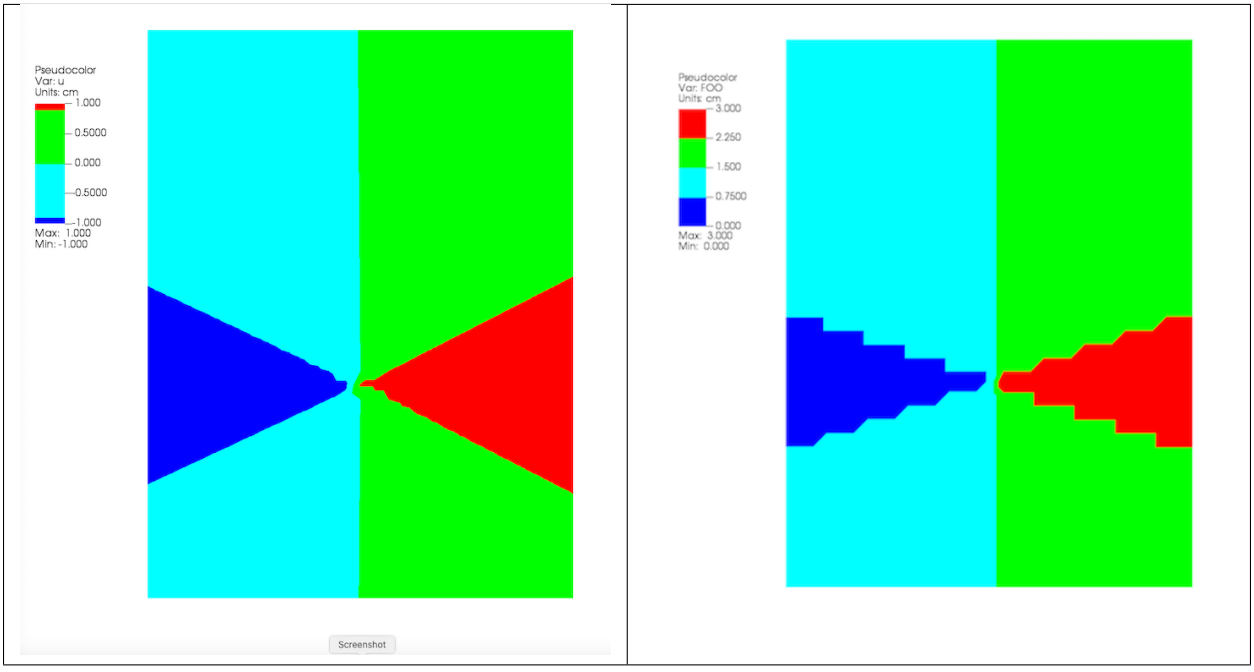
Banded Coloring with Discrete Color Table

A **Discrete** color table does indeed wind up *banding* smoothly varying data. However, the band boundaries are uniformly spaced in the variable's *range* and this may not always be desirable. Sometimes, it is desirable to have finely tuned banding around specific portions of the variable's range. This requires the coordination of a **Discrete** color table and an appropriately constructed *conditional expression*.

Using a 4-color **Discrete** color table alone, only the plot in Fig. 4.369 is produced.

This is because the colors in a **Discrete** color table are always uniformly spaced over the variable's value range. To produce the desired coloring we need to use a *conditional expression* that maps the input variable into 4 distinct values using the range logic from the table. In this case, the correct expression would be `if(lt(u,-0.95),0, if(lt(u,0),1, if(lt(u,0.95),2,3)))`. Then, plotting this expression using the 4-color **Discrete** color table, the desired coloring is produced as shown in

As can be seen in the side-by-side comparison below, there is a noticeable difference between the results produced by the two approaches demonstrated here to create a banded coloring.



The second approach (on the right) using the combination of a **Discrete** color table and a conditional expression shows

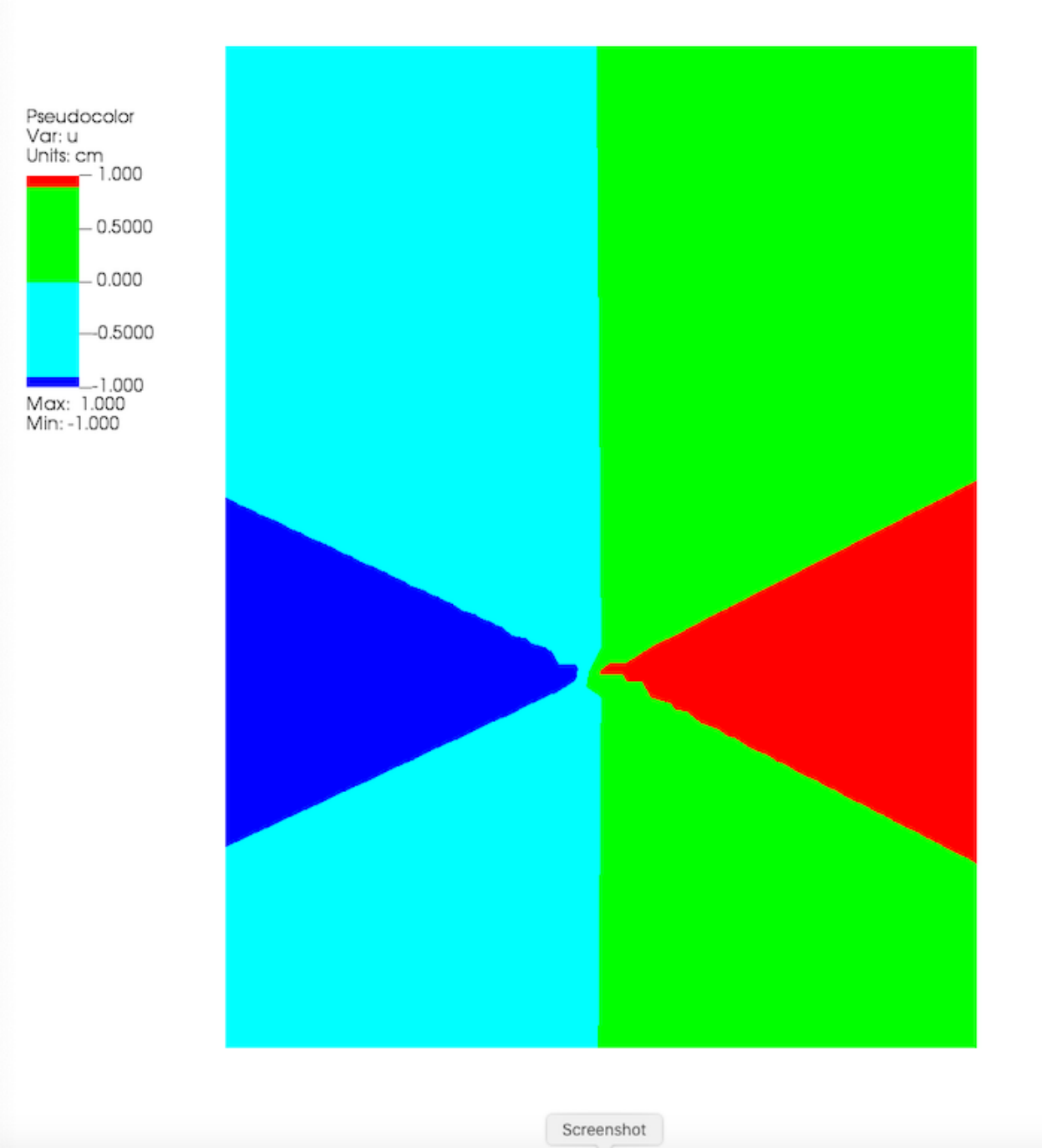


Fig. 4.368: The variable `u` plotted with a 4-color **Continuous** color table with `None` *smoothing*.

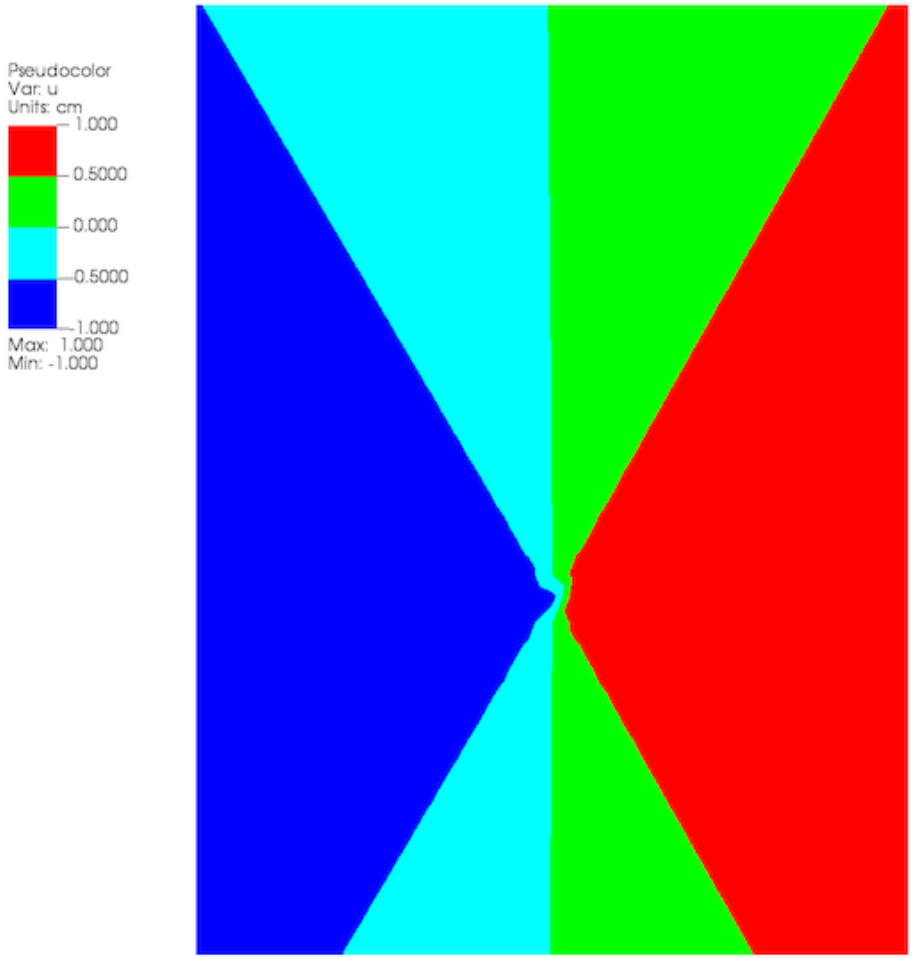


Fig. 4.369: A 4-color **Discrete** color table coloring alone

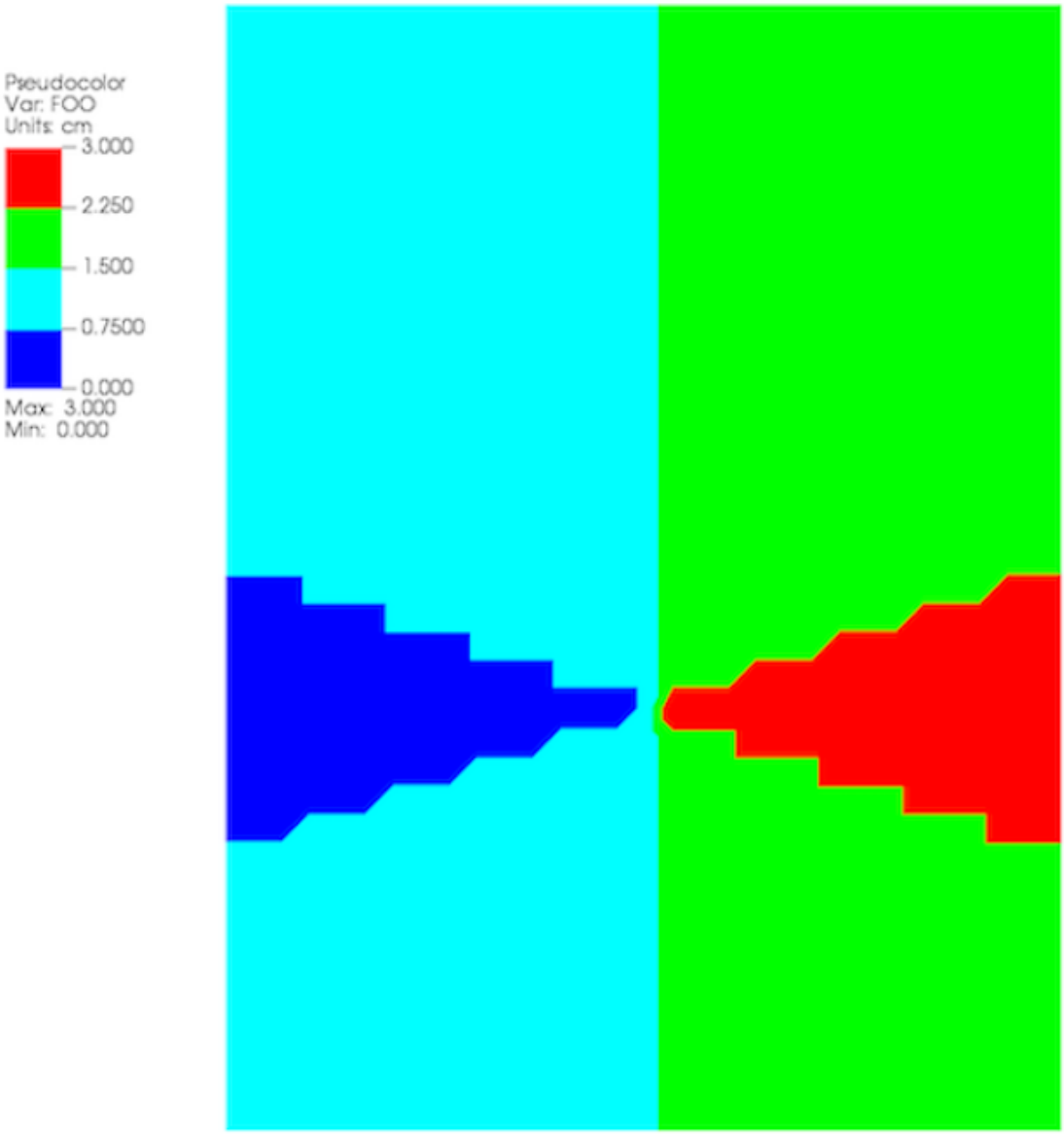


Fig. 4.370: A 4-color **Discrete** color table coloring combined with a conditional expression

significantly more jagged banding. This has to do with the ordering of operations of *interpolation* of the variable across mesh zones and then *mapping* of the variable values to colors.

The first approach performs *interpolation* followed by *mapping*. The second approach effectively performs the *mapping* first through its use of the conditional expression replacing the smoothly varying variable with a highly discontinuous variable after which *interpolation* is performed. In addition, the legend (on the right) does not do a good a job conveying the mapping of the variable's color to its values. This has to do with the use of the *intermediate* expression variable to help perform the mapping.

Converting color table types

It is possible to convert a continuous color table to a discrete color table and vice-versa using the **Continuous** and **Discrete** radio buttons in the editor portion of the **Color table window**. Changing the color table type from discrete to continuous does not change the color table's color control points; it only changes how they are used. If you select a discrete color table and click the **Continuous** radio button, the color table will be changed into a continuous color table and the **Editor** area will change to continuous mode and show the color table in a spectrum but no color control points will have changed. You can even turn the color table back into a discrete color table and the **Editor** area will show the color table in discrete mode, but the color control points will not have changed.

Built-In Color Tables

VisIt 3.4 supports the notion of *built-in* color tables. These are color tables that are either hard coded in VisIt or appear in the resources directory. They have special rules, namely, they cannot be edited, deleted, or exported. They can, however, be copied, using the **New** button as discussed up above.

Better Color Tables

VisIt includes sets of color tables that are *optimized* for various properties. These color tables, although they are less well known and used, often represent better choices than the *default* **hot** color table. Some are better for consumers of visual data who have *color vision deficiency (CVD)* (e.g. color blindness). Some represent emerging standards used in various corners of the visualization community. In some sense, these color table choices are more *inclusive* in that they convey the same information to a broader segment of the population and more *portable* in that they work across a number of different contexts; print, projection, monitor.

The Viridis color tables

The Viridis color tables assert the following properties...

- **Colorful:** spanning as wide a palette as possible so as to make differences easy to see.
- **Perceptually uniform:** meaning that values close to each other have similar-appearing colors and values far away from each other have more different-appearing colors, consistently across the range of values.
- **Robust to colorblindness:** so that the above properties hold true for people with common forms of colorblindness, as well as in grey scale printing.

These color tables all have the `Color Blind Safe`, `Viridis`, and `Default` tags.

The Crameri color tables

The Crameri color maps (also called the *Scientific Colour Maps*) assert the following properties.

- **Fairly representing data:** perceptually uniform and ordered to represent data both fairly, without visual distortion, and intuitively.
- **Universally readable:** The color combinations are readable both by color-vision deficient and color-blind people, and even when printed in black and white.
- **Citable & reproducible:** The color maps and their diagnostics are permanently archived and versioned to enable upgrades and acknowledge developers and contributors.

The Crameri color tables define five broad classes of color tables...

- Sequential (`seq`) - good for displaying continuous numerical data.
- Diverging (`div`) - good for displaying comparative data or data differencing.
- Mult-sequential (`2seq`) - good for displaying multi-modal data.
- Cyclic (`cyc`) - good for displaying periodic data.
- Categorical (`jumbled`) - good for displaying categorical data or false contouring of smooth data.
  - This is just a jumbling of the associated *sequential* color table to maximize perceptual differentiation between *neighboring* data values.

In addition, *discrete* versions consisting of 10 colors from their associated *main* color table are defined.

These color tables all have the `Color Blind Safe` and `Crameri` tags and are named `<name>-<class>-<count>` where `<name>` is the specific color table in the Crameri set of color tables, `<class>` is one of the five classes defined above and `<count>` is defined only for discrete color tables and indicates the number of colors in the table.

Filtering With Tags

To aid with choosing a color table, VisIt has a color table tagging scheme built into the Color Table Manager. Prior to VisIt 3.4, users had to select the tag filtering check box to enable tag filtering. In VisIt 3.4, tagging is always enabled. The manager portion of the color table window is pictured below.

How it works

Each color table has a number of tags associated with it. These are visible in the **Tags** Bar underneath the color table name. Users can select tags from the tag list to see only color tables that have those tags. So, for example, checking the box for the `Viridis` tag will cause the color table name box to show color tables that have the `Viridis` tag. However, this does not just affect the color table name box. All buttons allowing you to choose color tables will now only show color tables that match the current tag filtering selection. An exception to this is the default continuous and default discrete color table buttons, which will further restrict the set of chooseable color tables to continuous and discrete color tables respectively.

Users can also control how tags are combined: the dropdown button above the list of color table names gives users the option to have color tables match ALL of the selected tags or ANY of the selected tags. For example, if only the `Color Blind Safe` and `Continuous` tags are selected and the dropdown is set so color tables match ANY of the selected tags, then all color tables that have *either* the `Color Blind Safe` tag or the `Continuous` tag will appear. However, if you set the dropdown so color tables match ALL of the selected tags, then only color tables that have *both* the `Color Blind Safe` tag and the `Continuous` tag will appear.

To view the full list of color tables, users can use the **Select All Tags** Button while the drop down is set so color tables match ANY of the selected tags. Clicking this button while all tags are selected will deselect all tags.

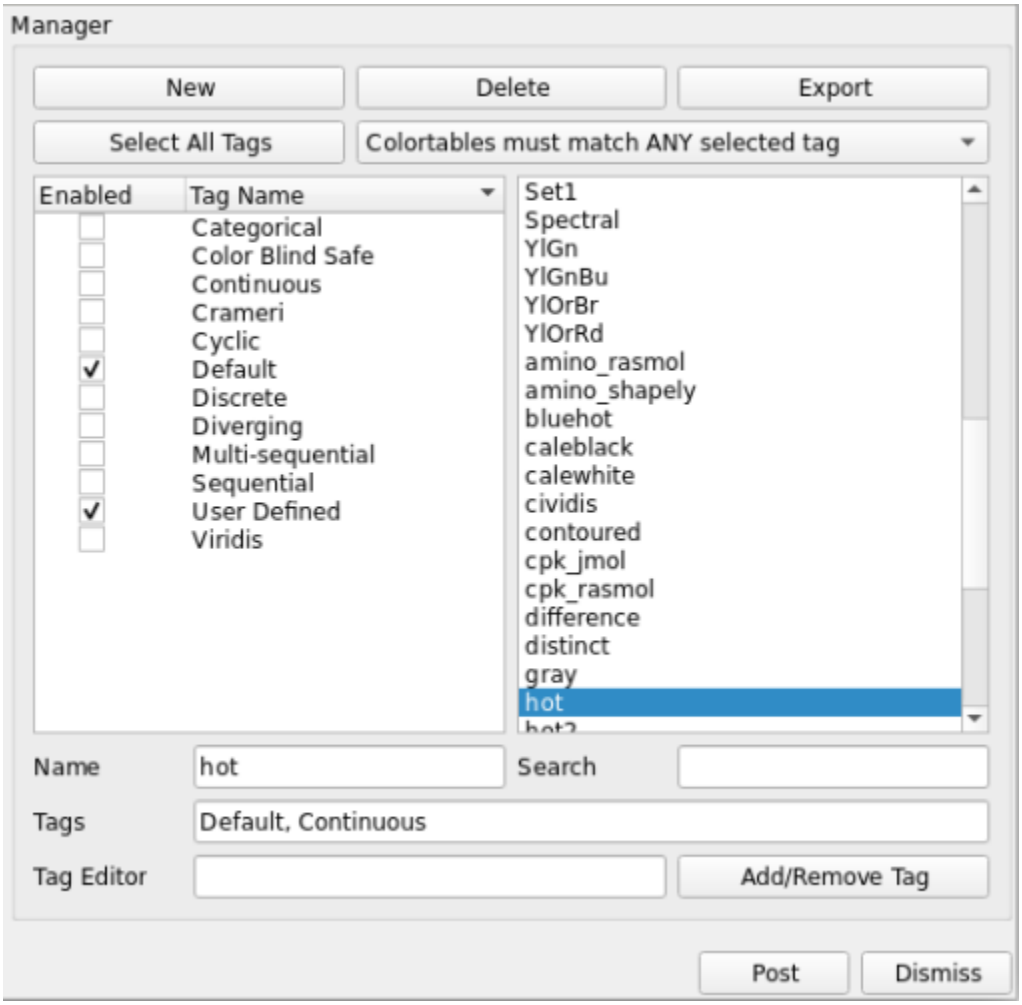


Fig. 4.371: The color table window manager is where tag filtering comes into play.

Specific Behavior

By default, only the `Default` and `User Defined` tags are selected (If there are no user defined color tables, the `User Defined` tag will not appear in the list). The Color Table Manager will retain the tag filtering selection even if the window is closed and reopened.

When a new color table is created, it will inherit the tags from the color table it is based on. In addition, it will get the `User Defined` tag associated with it. The same is true of color tables found in the user's .visit directory. When loaded into VisIt, they will automatically be assigned the `User Defined` tag if they do not have it already.

If VisIt encounters a color table that does not have any tags, that color table will be assigned the `No Tags` tag.

Exporting a color table will export its tags, so they are accessible the next time you use your color table in VisIt.

Default Tags

The following is a list of all the tags that appear in VisIt's current set of color tables.

- Default
- Color Blind Safe
- Crameri
- Viridis
- Sequential
- Diverging
- Multi-sequential
- Cyclic
- Categorical
- Continuous
- Discrete
- User Defined
- No Tags

Descriptions of most of these tags can be found elsewhere on this page.

Tag Editing

With VisIt 3.4 it is possible to edit tags directly via the Color Table Window. The **Tag Editor** is located beneath the **Tags** bar. In it, you may type the name of a tag that you wish to either add or remove from the currently selected color table. To add the tag to or remove the tag from a color table, click the **Add/Remove Tag** button next to the **Tag Editor**. If the tag name you have typed *is not* in the list of tags associated with the current color table, then it will be added upon clicking the **Add/Remove Tag** button. Likewise, if the tag name you have typed *is* in the list of tags associated with the current color table, then it will be removed upon clicking the **Add/Remove Tag** button.

However, built-in color tables behave differently. With built-in color tables, it *is* possible to add new tags, but it *is not* possible to remove tags that are present by default. So you can add tags to built-in color tables at will, but you can only remove tags that you or others have added.

Adding a new tag to a color table will immediately add it to the list of tags for filtering.

Prior to VisIt 3.4, editing tags in the GUI was not supported. To edit a color table's tags, users must directly edit the color table's `.ct` file. If there are tags defined for that color table, they will appear in a field (usually near the end of the file) called `tags`. Tags can be added to the list as desired. VisIt generates its tag list from the color table files it reads, so creating a new tag is as easy as writing the name of the tag in a color table file in the tags field.

Another new feature of VisIt 3.4 is tag name rules. Tags must now only be comprised of alphanumeric characters, whitespace, dashes, equal signs, and greater than and less than signs.

Searching for Color Tables

Another option for finding the right color table is to use the search feature. With this option, users can type a search term into the **Search** text box, and only color tables that have a name containing that term will appear in the color table name table.

Users need only edit the search bar contents to use it. To disable searching, simply delete all text in the search bar.

Searching also works in tandem with the tagging system. So, if tagging is enabled, only results from a search which have the specified tags will appear in the color table name table.

Color Tables and Saving Settings/Sessions

New to VisIt 3.4 are updates to how color tables interact with saving state (saving settings and sessions). Color tables themselves will now never be saved when saving state, since built-in color tables have no changes to save, and user defined color tables are exported to the .visit directory, so there is no color table information to save when saving state. However, there are two exceptions to this: the current tag filtering selection will be preserved if state is saved, and all modifications to tags will also be saved.

4.9.3 Lighting

Lighting is an important element when producing 3D visualizations because all areas of interest in the visualization should be lit so they can be easily seen. To this end, it is often necessary to have multiple light sources so all of the visualization's important areas are bright enough. VisIt can have up to 8 light colored light sources in order to improve the look of 3D visualizations. Each light source can be positioned and colored using VisIt's **Lighting Window**. It is also possible to have specular highlights in addition to multiple colored lights. For more information on specular highlights, which can make visualizations appear much more realistic, read about specular lighting in the *Preferences* chapter.

Lighting Window

You can open the **Lighting Window** (see Figure 4.372) by selecting the **Lighting** option from the **Main Window's Controls** menu. The **Lighting Window** has two modes of operation: edit and preview. When the window is in preview mode, light sources cannot be modified, but they are all visible and illuminate the **Lighting Window's** test sphere so the cumulative effect of the lights can be observed. When the window is in edit mode, light sources can be modified one at a time. You set light properties using the controls in the **Properties** panel and you can position lights interactively by moving them around in the lighting panel to the left of the **Properties** panel.

Switching between edit mode and preview mode

Changing the **Mode** between **Edit** and **Preview** switches the **Lighting Window** into the desired mode. When the **Lighting Window** is in edit mode, one light source at a time is shown in the lighting panel and the lights properties can be set by moving the light interactively or by settings its properties by using the controls in the **Properties** panel.

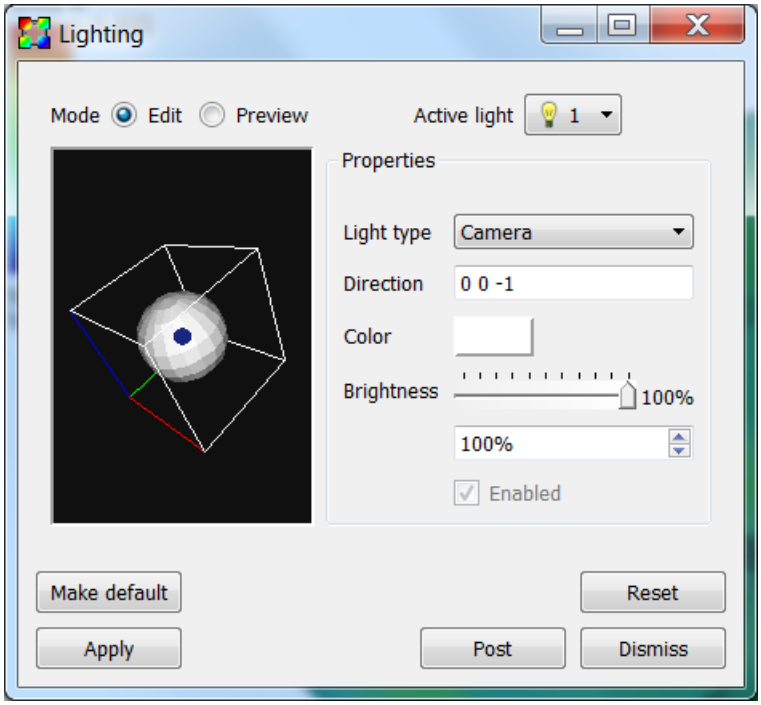


Fig. 4.372: The lighting Window

When the **Lighting Window** is in preview mode, all lights are shown in the lighting panel and none of them can be modified.

Choosing the active light

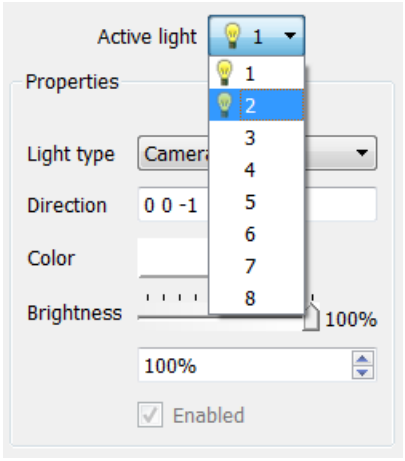


Fig. 4.373: The active light menu

The active light is the light whose properties are shown in the **Lighting Window**. Only the active light can be modified so you must switch active lights each time you want to make changes to a light. To change the active light, select a new light from the **Active light** menu (Figure 4.373). The **Active light** menu contains a list of eight possible lights of which only light 1 is active by default. When a light is active, it has a small light bulb icon next to it. Inactive lights have no light bulb icon. Once a new light has been selected from the **Active light** menu, its properties are displayed in

the **Lighting Window's Properties** panel.

Turning a light on

You can turn lights on and off using the **Enabled** check box that appears at the bottom of the **Lighting Window's Properties** panel. You can only modify lights when the **Lighting Window** is in edit mode.

Light type

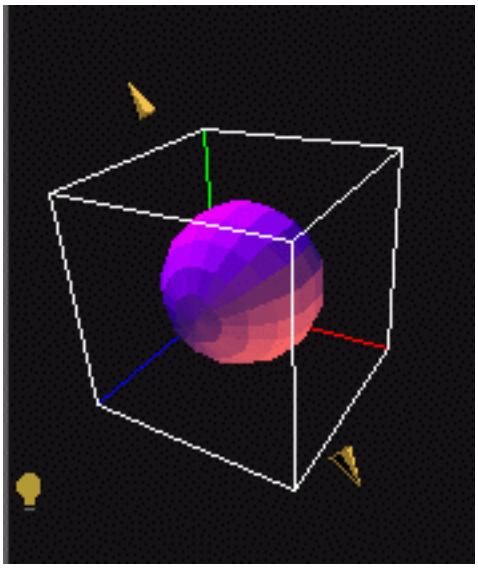


Fig. 4.374: The different kinds of lights

VisIt supports three types of lights. The first type is called an ambient light. An ambient light is a light that has no direction and contributes brightness to the entire visualization. When an ambient light is present, the lighting panel displays a small light bulb. The second type of light and the default light in VisIt is a camera light. A camera light stays fixed in space and always points the same direction regardless of how the objects in the visualization are positioned. Camera lights are represented in the lighting panel as small blue arrows. The third type of light in VisIt is the object light. An object light has a direction that is relative to the orientation of the object in the visualization. When the objects in the visualization are rotated, an object light keeps shining on the same area of the object. Object lights are represented in the lighting panel as small yellow cones. To change the light type for the active light, select a new light type from the **Light type** menu in the **Properties** panel.

Positioning a light

There are two ways to position a light. The first, and most intuitive, way is to interactively position the light by dragging it to the desired location in the lighting panel. Lights move in a sphere around the test sphere. Experiment with the motion until you are comfortable moving the light. The second way to move the light is to type a direction vector into the **Direction** text field. The coordinate system for specifying a direction vector is right-handed. Suppose you want to create a light that looks directly into the visualization. Since the Z-axis points directly out of the screen, the negative Z-axis points into the screen. This can be captured by entering a direction vector of: 0 0 -1. Note that ambient lights have no direction.

Light color and brightness

VisIt enables the user to vary lights both in color and in brightness.

Danger: When plotted surface colors are used to convey important information such as the magnitude of a variable or type of material, colored lighting has the potential to profoundly skew the surface colors and should be used with great care. Nonetheless, colored lighting can produce interesting effects that may be desirable for purposes aiming towards photorealism.

To change the light color, click on the light **Color** button and select a new color from the **Color** menu. Once a color is picked, you can also set the brightness for the light. The brightness is essentially a knob that allows you to dim the light. If the brightness is set completely to the right then the light will have exactly the color that was picked for it. If the brightness is not set to full intensity then the light will be dimmer. You can set the brightness by adjusting the **Brightness** slider in the **Lighting Window**.

4.9.4 Rendering Options

VisIt provides support for setting various global rendering options that improve quality and realism of the plots in the visualization. Specifically, VisIt provides controls that let you smooth the appearance of lines, add specular highlights, add shadows, and apply depth cueing to plots in your visualizations. The controls for setting these options are located in the **Rendering options Window** (see Figure 4.375) and they will be covered here while the other controls in that window will be covered in the **Preferences** chapter. To open the **Rendering options Window**, click on **Rendering** in the **Main Window's Preferences** menu.

Making Lines Look Smoother

Computer monitors contain an array of millions of tiny rectangular pixels that light up to form patterns which your eyes perceive as images. Lines tend to look blocky on computer monitors because they are drawn using a relatively small set of pixels. Lines can be made to look better by blending the edges of the line with the color of the background image. This is a form of antialiasing that VisIt can use to make plots which use lines, such as the Mesh plot, look better (see Figure 4.376). If you want to enable antialiasing, which is off by default, you check the **Antialiasing** check box located at the top of the **Basic** tab (see Figure 4.375). When antialiasing is enabled, all lines drawn in a visualization window are blended with the background image so that they look smoother.

Specular Lighting

VisIt supports specular lighting, which results in bright highlights on surfaces that reflect a lot of incident light from VisIt's light sources. Specular lighting is not handled in the **Lighting Window** because specular lighting is best described as a property of the material reflecting the light. The controls for specular lighting don't control any lights but instead control the amount of specular highlighting caused by the plots. Specular lighting is not enabled by default. To enable specular lighting, click the **Specular lighting** check box near the bottom of the **Basic** tab (see Figure 4.375).

Once specular lighting is enabled, you can change the strength and sharpness properties of the material reflecting the light. The strength, which you can set using the **Strength** slider, influences how glossy the plots are and how much light is reflected off of the plots. The sharpness, which is set using the **Sharpness** slider, controls the locality of the reflections. Higher sharpness values result in smaller specular highlights. Specular highlights are a crucial component of lighting models and including specular lighting in your visualizations enhances their appearance by making them more realistic. Compare and contrast the plots in Figure 4.377. The plot on the left side has no specular highlights and the plot on the right side has specular highlights.

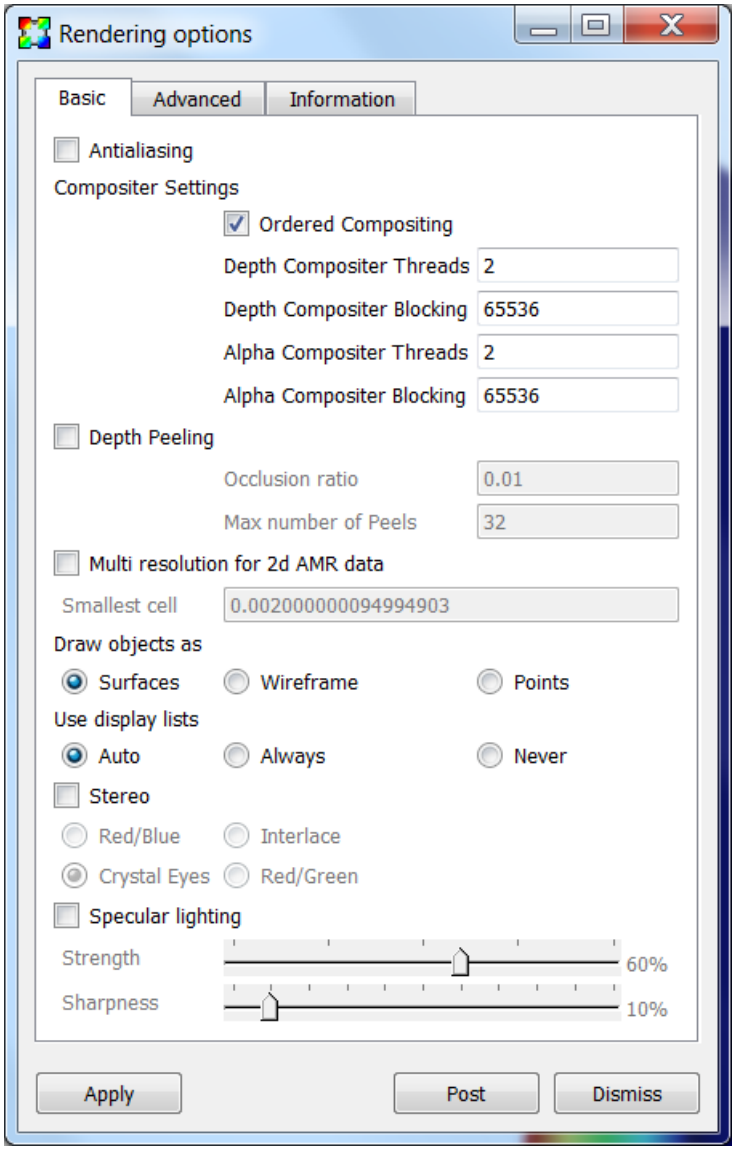


Fig. 4.375: The basic rendering options

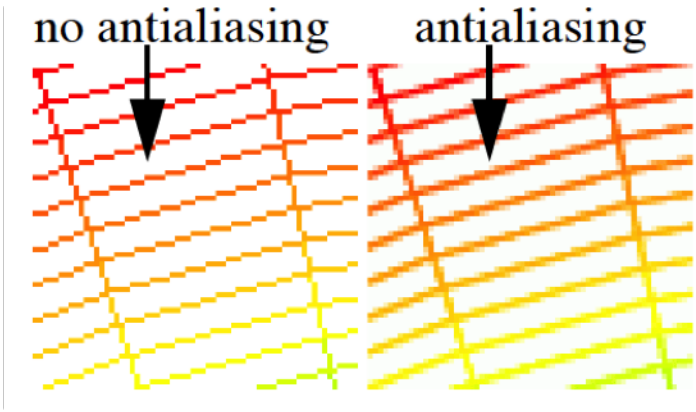


Fig. 4.376: An example of antialiasing

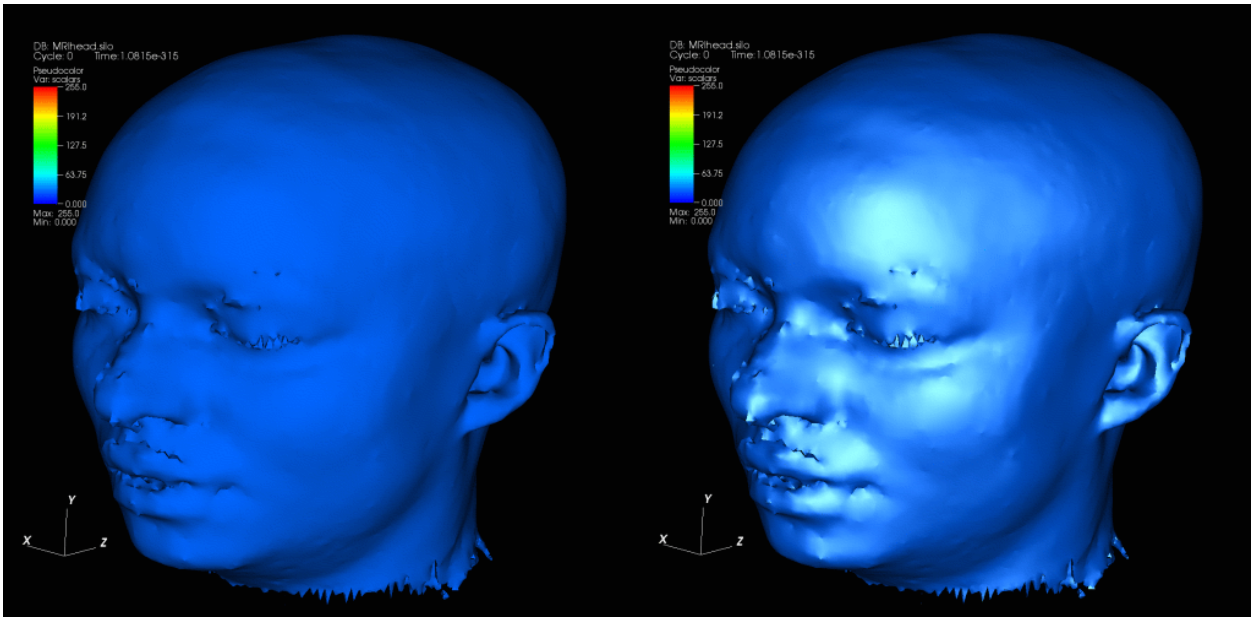


Fig. 4.377: The effects of specular lighting on plots

Shadows

VisIt supports shadows when scalable rendering is being used. Shadows can be useful for increasing the realism of your visualization. The controls to turn on shadows can be found near the bottom of the **Advanced** tab (see Figure 4.378). To turn on shadows, you must turn on scalable rendering by clicking on the **Always** radio button under the **Use scalable rendering** label. Once scalable rendering has been turned on, the shadows controls become enabled. The default shadow strength is 50%. If you desire a stronger or weaker shadow, adjust the **Strength** slider until you are satisfied with the amount of shadow that appears in the visualization. The same plot is shown with and without shadows in Figure 4.379.

Depth Cueing

VisIt supports depth cueing when scalable rendering is being used. Depth cueing can be useful for increasing the realism of your visualization. Depth cueing causes objects to be blended with the background with increasing distance from the camera. The controls to turn on depth cueing can be found near the bottom of the **Advanced** tab (see Figure 4.378). To turn on depth cueing, you must turn on scalable rendering by clicking on the **Always** radio button under the **Use scalable rendering** label. Once scalable rendering has been turned on, the depth cueing controls become enabled. By default, depth cueing is performed along the camera direction. The depth cueing can be done along a different direction by unchecking the **Cue automatically along camera depth** check box and then entering the coordinates defining the direction to perform the depth cueing in the **Manual start point** and **Manual end point** text fields. The coordinates are defined in the coordinate system of the simulation data. The same plot is shown with and without depth cueing in Figure 4.380.

4.9.5 View

The view is one of the most critical properties of a visualization since it determines what parts of the dataset are seen. The view is also one of the most difficult properties to set. It is not that the act of setting the view is difficult. In fact, it is quit the opposite. The problem with setting the view is finding a flattering view for a database that will continue to be a good view for the entire life of the visualization. Many plots will deform or expand over the course of an

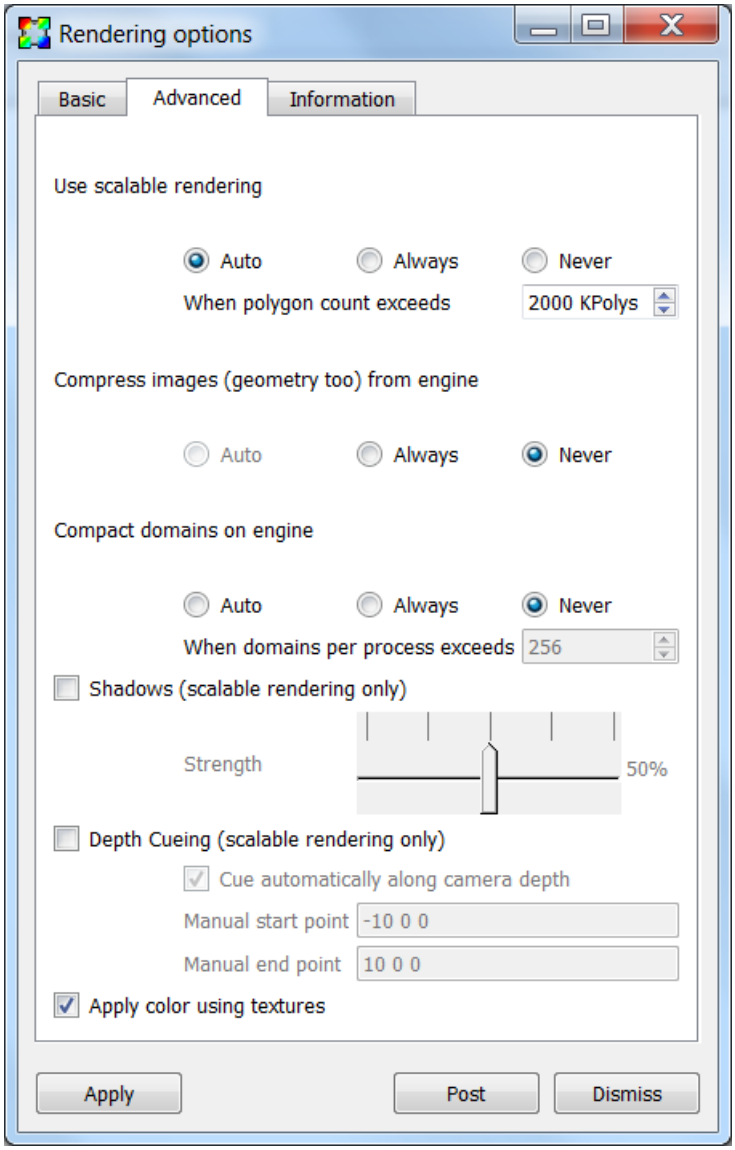


Fig. 4.378: The advanced rendering options

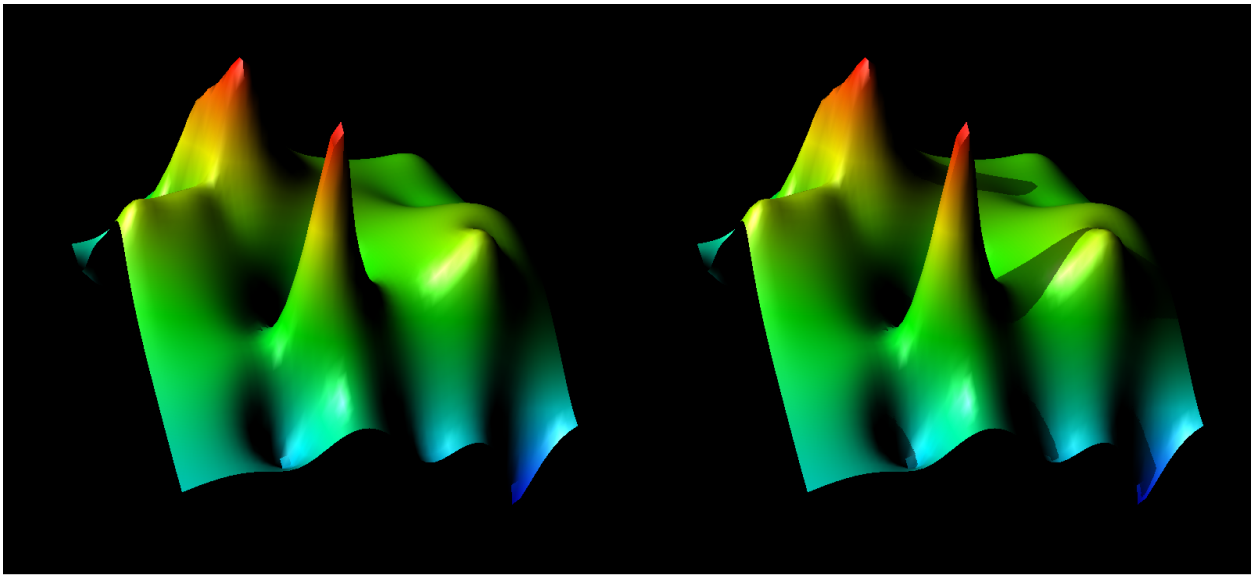


Fig. 4.379: The effects of shadows on plots

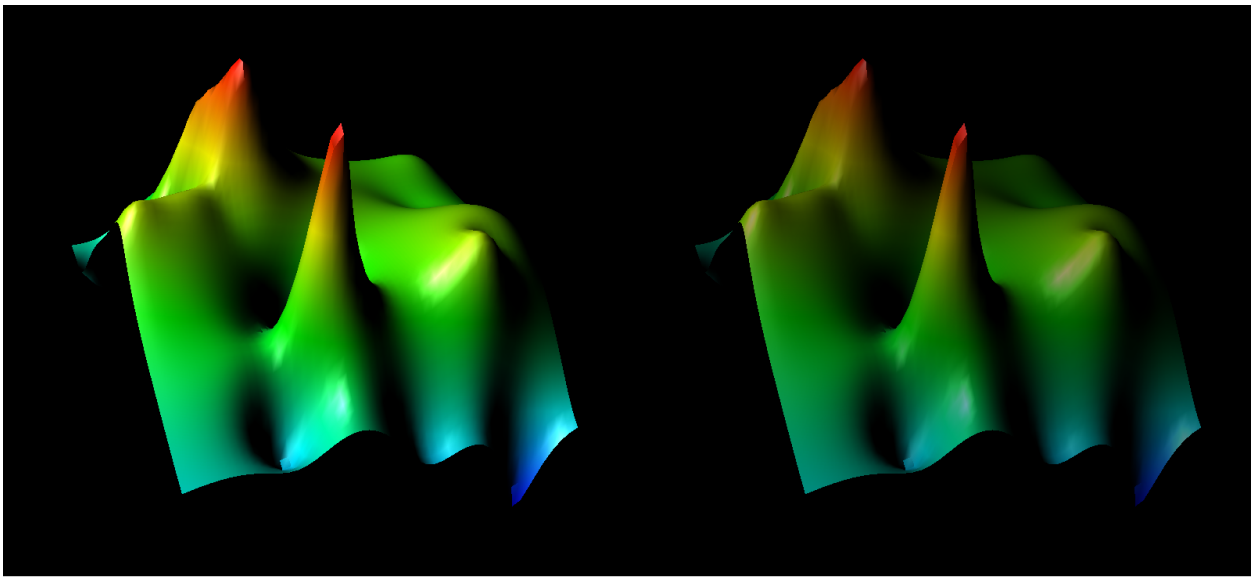


Fig. 4.380: The effects of depth cueing on plots

animation and you have to decide how to pick a good view. You can pick a view that is zoomed way out and then let your plots expand and deform until they make good use of the visualization window. You can also decide to keep changing the view throughout the animation. A common technique is to interpolate views or do some sort of fly-by animation when the plots in the animation are expanding or not behaving in a static manner. The fly-by animation is used to distract the audience from the fact that you need to change to a more suitable view.

The view in VisIt can be set in two different ways. The first and best way to set the view is to navigate to it interactively in the visualization window. This is the fastest and most direct way of setting the view. The problem with setting the view in this manner is that it is not very reproducible. It is often the case that users want to look at the same feature in their database using the same view. VisIt provides a **View Window** that they can use to set the view information exactly the same every time.

View Window

You can open the **View Window** by selecting **View** from the **Main Window's Controls** menu. The **View Window** is divided into five tabbed sections. The first tab sets the curve view, the second tab sets the 2D view, the third tab sets the 3D view, the fourth tab sets the axis array view, and the last tab sets advanced view options. The **View Window** also contains a **Command** text field at the bottom for entering view commands.

Setting the curve view

Visualization windows that contain Curve plots use a special type of view known as a curve view. A curve view consists of: viewport, domain, and range. The viewport is the area of the visualization window that will be occupied by the plots and is specified using X and Y values in the range [0,1]. The point (0,0) corresponds to the lower-left corner of the visualization window while the point (1,1) corresponds to the visualization window's upper-right corner. To change the viewport, type new numbers into the **Viewport** text field on the **Curve view** tab of the **View Window** (Figure 4.381). The minimum and maximum X values should come first, followed by the minimum and maximum Y values.

The domain and range refer to the limits on the X and Y axes. You can set the domain, which is the range of X values that will be displayed in the viewport, by typing new minimum and maximum values into the **Domain** text field. You should use domain values that use the same dimensions as the Curve plot that will be plotted in the visualization window. You can set the range, which is the range of Y values that will be displayed in the viewport, by typing new values into the **Range** text field. The domain and range values may also be log scaled and may be controlled independently. To log scale the domain, check the **Log** radio box to the right of the **Domain Scale** label. To log scale the range, check the **Log** radio box to the right of the **Range Scale** label.

Setting the 2D view

Setting the 2D view is conceptually simple. There are only two pieces of information that you need to supply. The first piece of information that you must enter is the viewport, which is an area of the visualization window in which you want the 2D plots to appear. Imagine that the lower left corner of the visualization window is the origin of a coordinate system and that the upper left and lower right corners both have values of 1. Every point in the visualization window can be characterized as a Cartesian coordinate where both values in the coordinate are in the range [0,1]. The viewport is specified by entering four numbers in the form x0 x1 y0 y1 where x0 is the leftmost X value, x1 is the rightmost X value, y0 is the lower Y value, and y1 is the upper Y value that will be used in the viewport. The window is an area in the space occupied by the 2D plots. You can start with a window that is the same size as the plot's spatial extents and then zoom in from there by making the window values smaller and smaller. The window values are also of the form x0 x1 y0 y1. To change the 2D view, type new values into the **Viewport** and **Window** text fields on the **View Window's 2D view** tab (Figure 4.382).

Some databases yield plots that are so long and skinny that they leave most of the visualization window blank when VisIt displays them. A common example is equation of state data, which often has at least 1 exponential dimension.

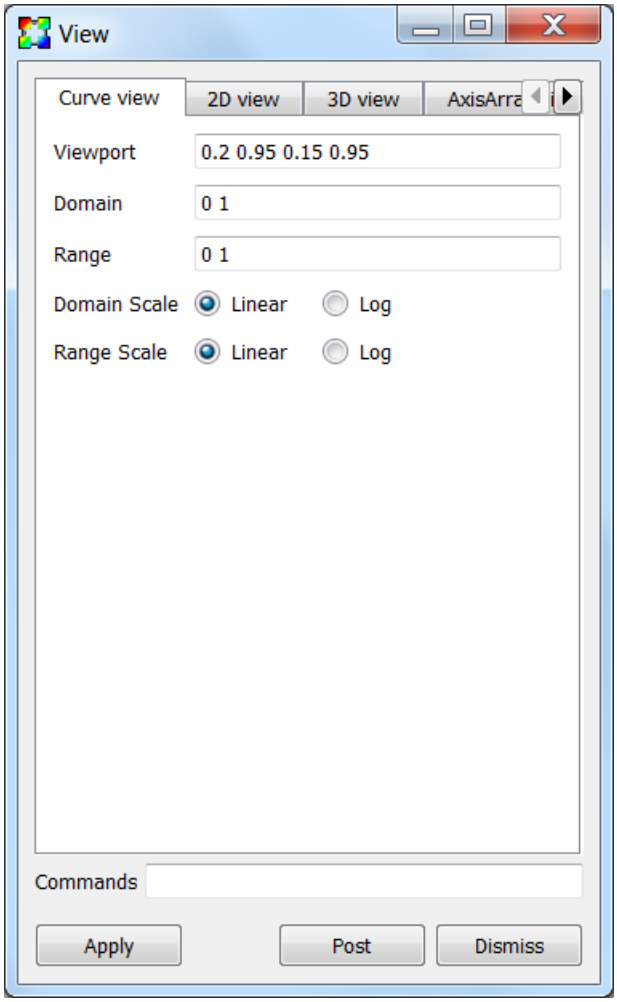


Fig. 4.381: The curve view options

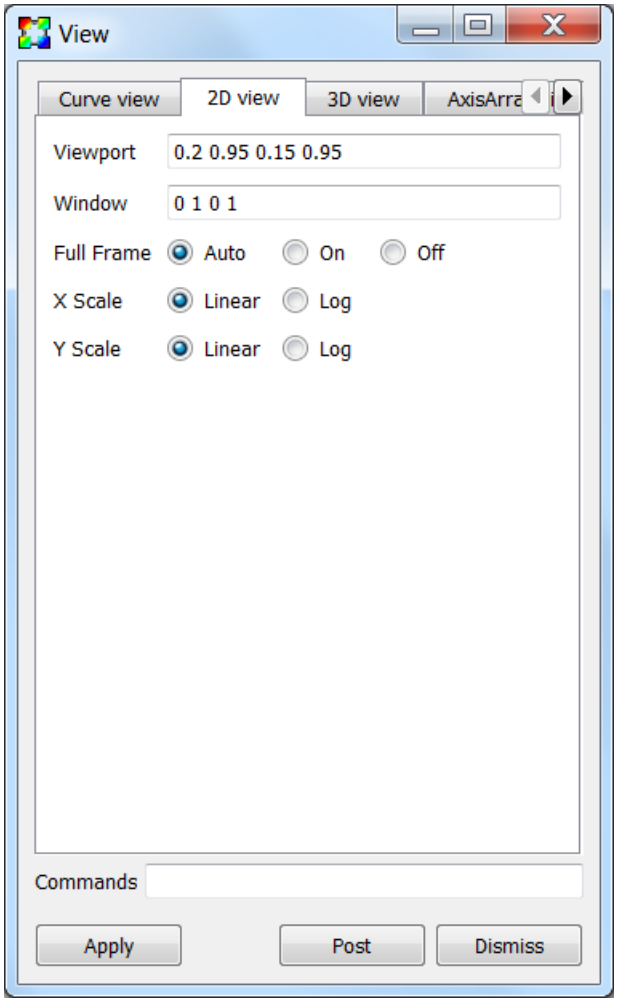


Fig. 4.382: The 2D view options

VisIt provides Fullframe mode to stretch long, skinny plots so they fill more of the visualization window so it is easier to see them. It is worth noting that Fullframe mode does not preserve a 1:1 aspect ratio for the displayed plots because they are stretched in each dimension so they fit better in the visualization window. To activate full frame mode, click on the **Auto** or **On** radio buttons to the left of the **Full Frame** label. When full frame mode is set to **Auto**, VisIt determines the aspect ratio of the X and Y dimensions for the plots being visualized and automatically scales the plots to fit the window when extents for one of the dimensions are much larger than the extents of the other dimension.

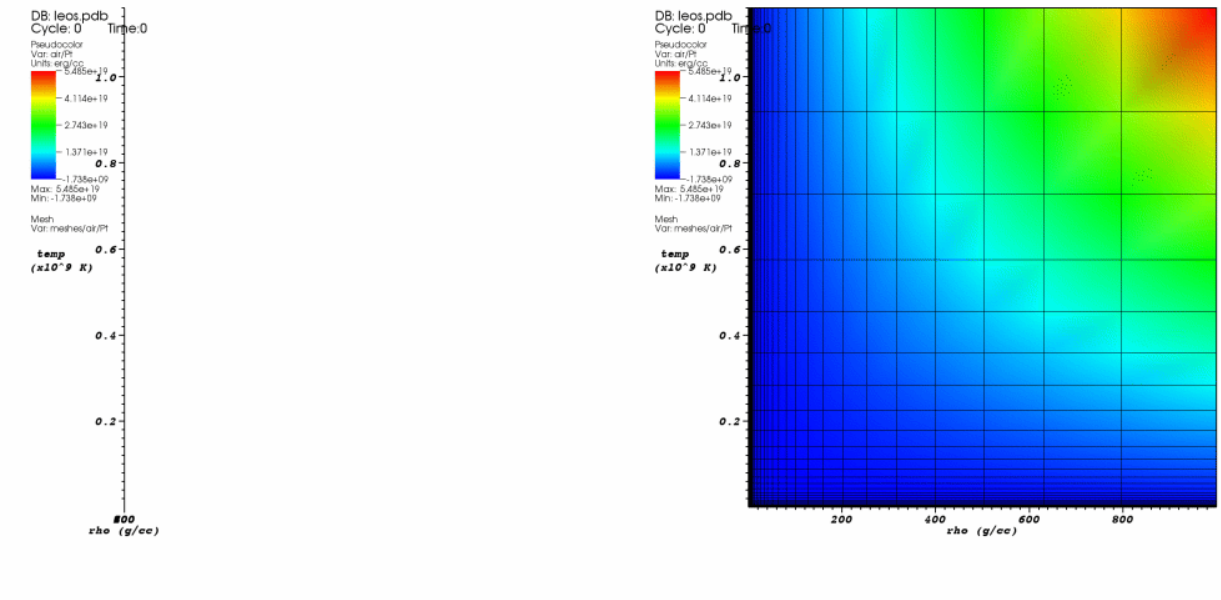


Fig. 4.383: The effect of full frame mode on an extremely skinny plot

Just like the with the curve view, the x and y values may be log scaled independently. To log scale the x values, check the **Log** radio box to the right of the **X Scale** label. To log scale the y values, check the **Log** radio box to the right of the **Y Scale** label.

Setting the 3D view

Setting the 3D view using controls in the **View Window's 3D view** tab (see Figure 4.384) demands an understanding of 3D views. A 3D view is essentially a location in space (view normal) looking at another location in space (focus) with a cone of vision (view angle). There are also clipping planes that lie along the view normal that clip the near and far objects from the view. Figure 4.385 depicts the various components of a 3D view.

To set the 3D view, fill in the following fields:

View normal Where you want to look from.

Focus What you want to look at.

Up axis Determines which way is up. A good default value for the up axis is 0 1 0. VisIt will often calculate a better value to use for the up axis so it is not too important to figure out the right value.

View Angle Determines how wide the field of view is. The view angle is specified in degrees and a value around 30 is usually sufficient.

Near clipping and Far clipping Values along the view normal that determine where the near and far clipping planes are to be placed. It is not easy to know that good values for these are so you will have to experiment.

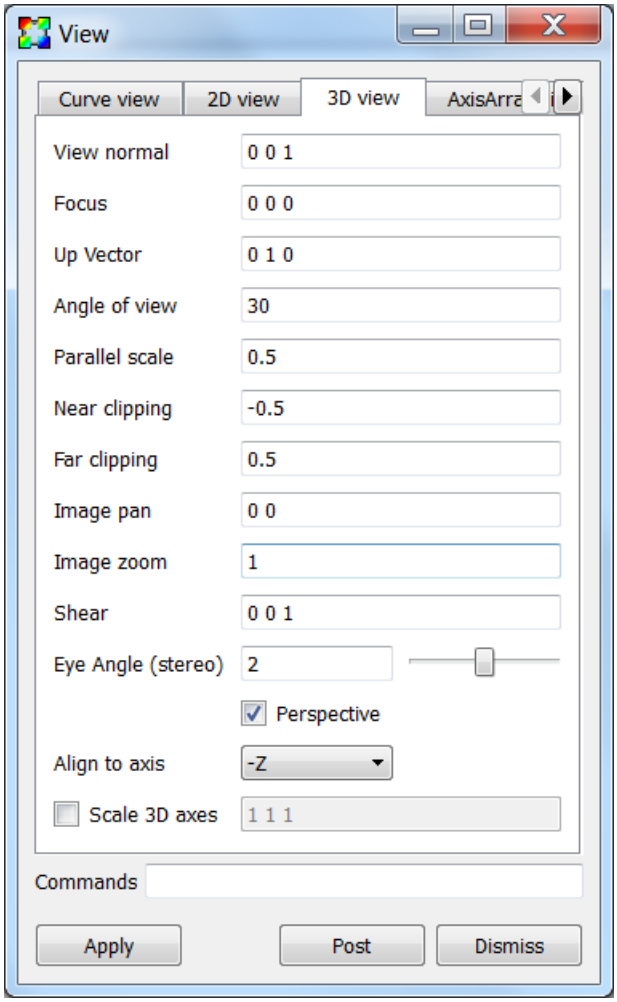


Fig. 4.384: The 3D view options

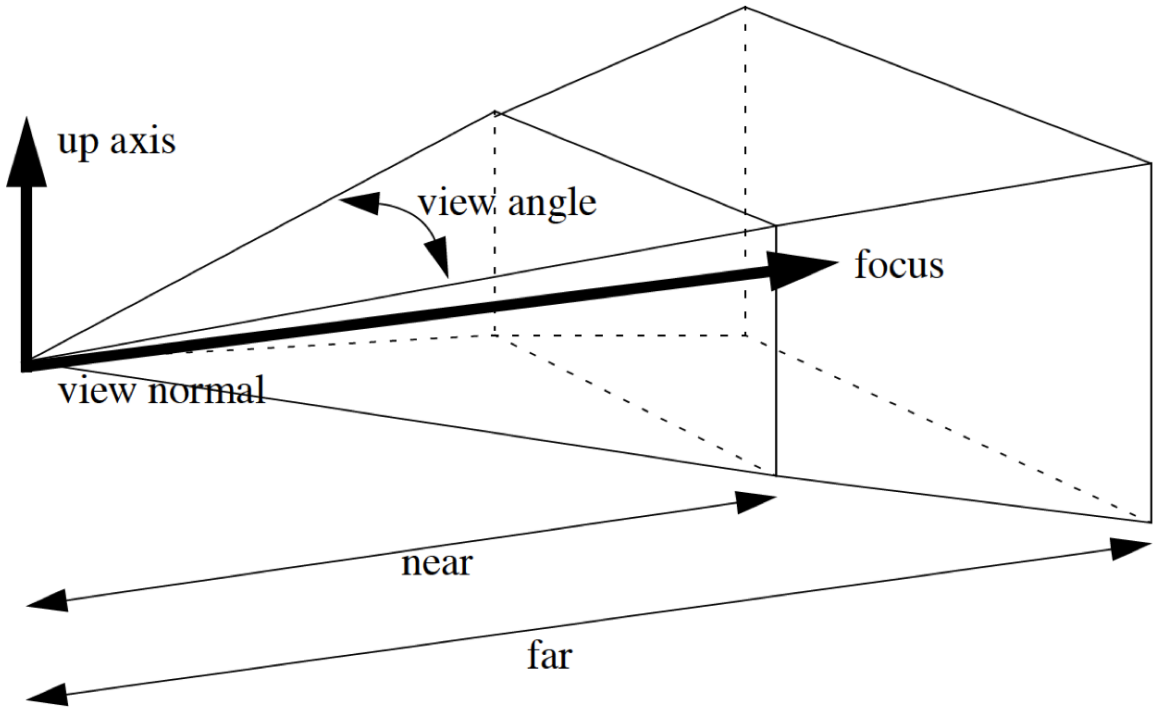


Fig. 4.385: The 3D perspective view volume

Parallel scale Acts like a zoom factor that zooms the camera towards the focus. For a parallel projection, it is half the height of an object in the window. For example, if you had a sphere of radius 10, setting the parallel scale to 10, would result in the top and bottom of the sphere touching the top and bottom of the image. Where the sphere touches on the left and right edges depends on the aspect ratio of the image. If it was 1:1, then the sphere would also touch the left and right edges of the image. When doing a perspective projection, it attempts to have the top and bottom of the sphere touch the top and bottom of the image.

Perspective Applies to 3D visualizations and it causes a more realistic view to be used where objects that are farther away are drawn smaller than closer objects of the same size. VisIt uses a perspective view for 3D visualizations by default.

VisIt supports stereo rendering, during which VisIt draws the image in the visualization window twice with the camera eye positioned in slightly different locations to mimic the differences in images seen by your left eye and your right eye. With the right stereo goggles, the image that you see appears to hover in 3D space within your monitor since the effect of the stereo image adds much more depth to the visualization. You can set the angle that VisIt uses to separate the cameras used to draw the images by typing a new angle into the **Eye angle** text field or by using the **Eye angle** slider.

The **Align to axis** menu provides a convenient way to get side, top, and bottom views of your 3D data. It provides six options corresponding to the six axis aligned directions and sets both the **View normal** and the **Up vector**.

Setting the axis array view

Visualization windows that contain Parallel Coordinate plots use a special type of view known as an axis array view. An axis array view consists of: viewport, domain, and range. The viewport is the area of the visualization window that will be occupied by the plots and is specified using X and Y values in the range [0,1]. The point (0,0) corresponds to the lower-left corner of the visualization window while the point (1,1) corresponds to the visualization window's

upper-right corner. To change the viewport, type new numbers into the **Viewport** text field on the **Curve view** tab of the **View Window** (Figure 4.386). The minimum and maximum X values should come first, followed by the minimum and maximum Y values.

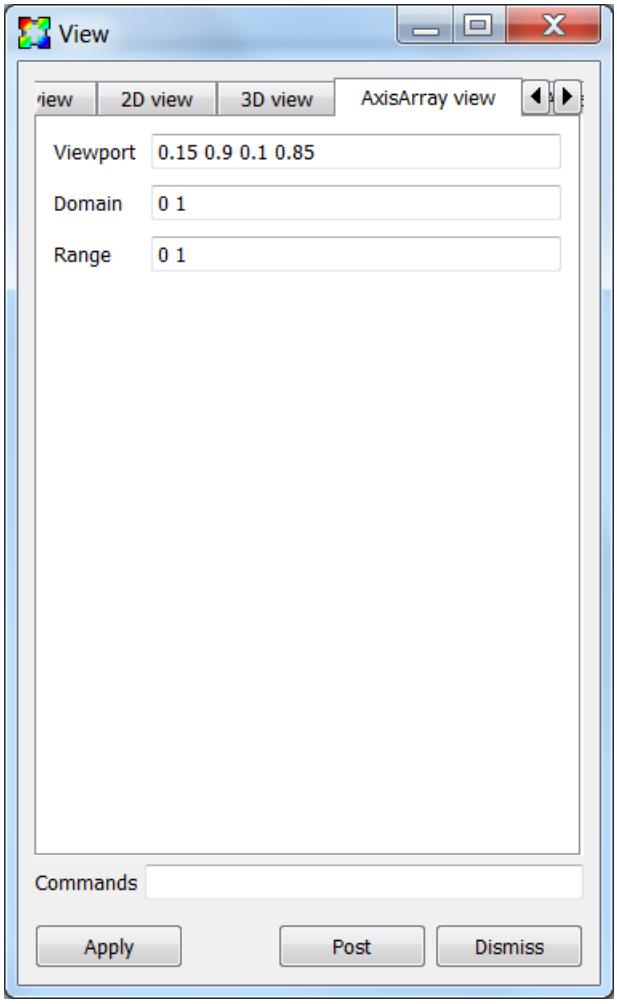


Fig. 4.386: The axis array view options

The **Domain** and **Range** settings are not very intuitive and we will give a short description followed by some examples. The domain controls the position and spacing of the parallel axes. The larger the value the more tightly they are spaced or the more axes that will fit in the view. For example, a domain of 0. to 2. would have room for exactly three coordinate axes, with the first one at the extreme left edge of the viewport and the third one at the extreme right edge of the viewport. Changing the domain to 1. to 3. would shift the second axis to the extreme left edge of the viewport and move the third axis to the center of the viewport. If there were only three axes, then the right half of the viewport would be empty. The range controls the height of the coordinate axes. The larger the value, the shorter the axes. For example, the default range of 0. to 1. results in the axes filling the height of the viewport. A range of 0. to 2. results in the axes filling the bottom half of the viewport. You can play with the controls to get a better understanding of the domain and range settings.

Advanced view features

The **View Window's Advanced** tab, shown in Figure 4.387, contains advanced features that are not needed by all users.

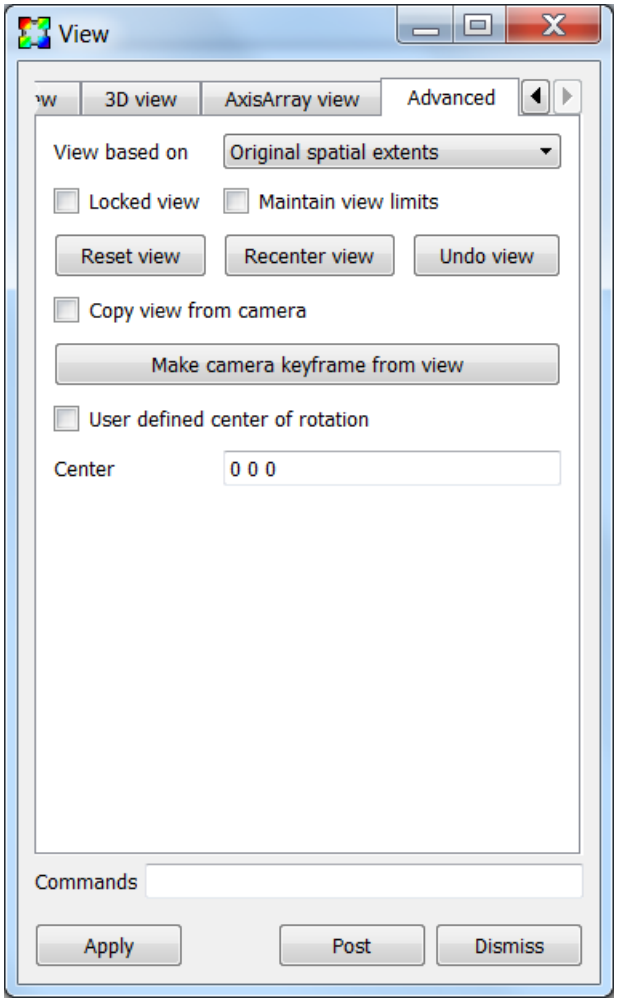


Fig. 4.387: The advanced view options

The **View based on** menu is used to specify if the view is set based on the original spatial extents of the plot or the actual current extents which are the plot's current extents after it has been subsetted in some way. By default, VisIt bases the view on the plot's original extents which leaves the remaining bits of a plot, after being subsetted, in the same space as the original plot. This makes it easy to see where the remaining pieces of the plot were situated relative to the whole plot but it does not always make best use of the visualization window. To fill up more of the visualization window, you might want to base the view on the actual current extents by selecting **Actual current extents** from the **View based on** menu.

When using more than one visualization window, such as when comparing plots using two different databases side by side, it is often useful for the plots being compared to have the same view. VisIt allows you to lock the views together for the multiple visualization windows so that when you change the view of any window whose view is locked, all other windows with locked views get the new view. To lock the view for a visualization window, click the **Locked view** check box or click on the Toolbar button to lock views.

Normally, VisIt will adjust the view to match the extents of the data. For example, if you are looking at data from a simulation whose extents expand over time, VisIt will automatically adjust the view so that the data fills roughly the same amount of space as the extents expand. Another example is when the extents move from left to right, VisIt will adjust the view so that the extents are always centered in the same portion of the screen. This behavior is not always desired in certain situations. To turn off this behavior and fix the view, no matter how the extents of the data change, click on the **Maintain view limits** check box.

The **Reset view**, **Recenter view**, and **Undo view** can be used to reset the view, recenter the view, and undo the last view change. Resetting the view resets all aspects of the view based on the data extents. Recentering the view resets all aspects of the view except the view orientation based on the data extents. Undoing the view returns the view to the last view setting. The last 10 views are stored so you can undo the view up to 10 times.

The **Locked view** check box, the **Maintain view limits** check box, the **Reset view** button, the **Recenter view** button, and **Undo view** buttons behave differently than the rest of the controls in the view window in that they effects take effect immediately, without having to press the **Apply** button.

The **Copy view from camera** check box and the **Make camera keyframe from view** button are deprecated and will be removed in the next release.

The center of rotation is the point about which plots are rotated when you set the view. You can type a new center of rotation into the **Center** text field and click the **User defined center of rotation** check box if you want to specify your own center of rotation. The center of rotation is, by default, the center of your plots' bounding box. When you zoom in to look at smaller plot features and then rotate the plot, the far away center of rotation causes the changes to the view to be large. Large view changes when you are zoomed in often make the parts of the plot that you were inspecting go out of the view frustum. If you are zoomed in, you should pick a center of rotation that is close to the surface of the plot that you are inspecting. You can also pick a center of rotation using the **Choose center** from the visualization window's **Popup** menu.

Using view commands

The **Commands** text field at the bottom of the **View Window** allows you to enter one or more semi-colon delimited legacy MeshTV commands to change the view. The following list has a description of the supported view commands:

pan x y Pans the 3D view to the left/right or up/down. The x, y arguments, which are floating point fractions of the screen in the range [0,1], determine how much the view is panned in the X and Y dimensions.

pan3 x y Same as pan.

panx x Pans the 3D view left or right. The x argument is a floating point fraction of the screen in the range [0,1].

pany y Pans the 3D view up or down. The y-argument is a floating point fraction of the screen in the range [0,1].

ytrans y Same as pany.

rotx x Rotates the 3D view about the X-axis x degrees.

rx x Same as rotx.

roty y Rotates the 3D view about the Y-axis y degrees.

rotz z Rotates the 3D view about the Z-axis z degrees.

rz z Same as rotz.

zoom val Scales the 3D zoom factor. If you provide a value of 2.0 for the val argument, the object being viewed will appear twice as large. A value of 0.5 for the val argument will make the object appear only half as large.

zf Same as zoom.

zoom3 Same as zoom.

vp x0 x1 y0 y1 Sets the window, which is how much space relative to the plot will be visible inside of the viewport, for the 2D view. All arguments are floating point numbers that are in the same range as the plot extents. The x0 and x1 arguments are the minimum and maximum values for the edges of the window in the X dimension. The y0 and y1 arguments are the minimum and maximum values for the edges of the window in the Y dimension.

wp x0 x1 y0 y1 Sets the window, which is how much space relative to the plot will be visible inside of the viewport, for the 2D view. All arguments are floating point numbers that are in the same range as the plot extents. The x0 and x1 arguments are the minimum and maximum values for the edges of the window in the X dimension. The y0 and y1 arguments are the minimum and maximum values for the edges of the window in the Y dimension.

reset Resets the 2D and 3D views.

recenter Recenters the 3D view.

undo Changes back to the previous view.

4.10 Animation

This chapter discusses how to use VisIt to create animations. There are three ways of creating animations using VisIt: flipbooks, keyframing, and scripting. For complex animations with perhaps hundreds or thousands of database time steps, it is often best to use scripting via VisIt's *Python command-line interface*. VisIt provides Python and Java language interfaces that allow you to program animation and save image files that get converted into a movie. The flipbook approach is strictly for static animations in which only the database time step changes. This method allows database behavior over time to be quickly inspected without the added complexity of scripting or keyframing. Keyframed animation can exhibit complex behavior of the view, plot attributes, and database time states over time. This chapter emphasizes the flipbook and keyframe approaches and explains how to create animations both ways.

Scripting is the recommended method of producing animations. Scripting is more difficult than other methods because users have to script each event by writing a Python or Java program to control VisIt's viewer. One clear strength of scripting is that it is very reproducible and can be used to generate animation frames in a batch computing environment. For in-depth information about writing Python scripts for Visit, consult the *Python command-line interface*. Scripting for purposes of animations is not described further here.

4.10.1 Animation basics

Animation is used mainly for looking at how scientific databases evolve over time. Databases usually consist of many discrete time steps that contain the state of a simulation at a specific instant in time. Creating visualizations using just one time step from the database does not reveal time-varying behavior. To be most effective, visualizations must be created for all time steps in the database.

The .visit file

Since scientific databases usually consist of dozens to thousands of time states. Those time states can reside in any number of actual files. Some database file formats support multiple time states in a single file while other formats require each time state to be located in its own file. When all time states are in their own file, it is important for VisIt to know which files comprise the database. VisIt attempts to use automatic file grouping to determine which files are in a database but sometimes it is better if you provide the actual list of files in a database when you want to generate an animation using VisIt. You can create a *.visit* file that contains a list of the files in the database. By having a list of files that make up the database, VisIt does not have to guess database membership based on file naming conventions. While this may appear to be inconvenient, it removes the possibility that VisIt will include a file that is not in the database. It also frees VisIt from having to know about dozens of ad hoc file naming conventions. Having a *.visit* file also allows VisIt to make certain optimizations when generating a visualization.

VisIt provides a **File grouping** combo box in the **File open** window (see Figure 4.388) to assist in grouping related time-varying files into a virtual database. A virtual database accomplishes the same function as a `.visit` file except that no extra file needs to be created. Selecting *On* or *Smart* will group files into a virtual database. The *On* setting applies file matching rules to group files with similar prefixes into a virtual database. VisIt will attempt to generate a pattern from a filename so sequences of numbers can be abstracted out. Multiple files that match the same pattern are added to the same virtual database. The *Smart* setting applies the same logic as well as some extra rules that permit additional file grouping. For instance, certain file extensions that include numbers such as `.hdf5` are excluded from the pattern generation so the number in the file extension does not prevent useful file groupings.

Flipbook animation

All that is needed to create a flipbook animation is a time-varying database. To view a flipbook animation, open a time-varying database, create plots as usual, and click the **Play** button in the **GUI** shown in Figure 4.389 highlighted in red or in the visualization window's **Animation Toolbar** . A flipbook animation repeatedly cycles through all of the time states in the database displaying the plots for the current time state in the visualization window. The result is an animation that allows you to see the database evolve over time. The **VCR** buttons, shown in Figure 4.389 , allow you to control how a flipbook animation plays. The animation controls are are also used for controlling keyframe animations. Clicking the **Play** button causes VisIt to advance the database timestep until the **Stop** button is clicked. As the plots are generated for each database time state, the animation proceeds only as fast as the compute engine can generate plots. As described in the *Animation Window* section, you have the option of caching the geometry for each time state so animations will play smoothly according to the animation playback speed once the plots for each database time state have been generated.

Setting the time state

There are several ways that you can set the time state for an animation. You can use the **VCR** controls to play animations or step through them one state at a time. You can also use the **Time slider** to access a specific animation time state. To set the animation time state using the **Time slider** , click on the time slider and drag horizontally to a new time state. The time state to which you drag it will be displayed in the **Cycle/Time** text field as you drag the time slider so you will know when to let go of the **Time slider** . Once you release the mouse button at a new time state, VisIt will calculate the visualized plots using the data at the specified time state.

If you prefer more precise control over the time state, you can type a cycle or time into the **Cycle/Time** text field to make VisIt jump to the closest cycle or time for the active database. You can also highlight a new time state for the active database in the **Selected files** list and then click the **Replace** button to make VisIt change the time state for the visualization.

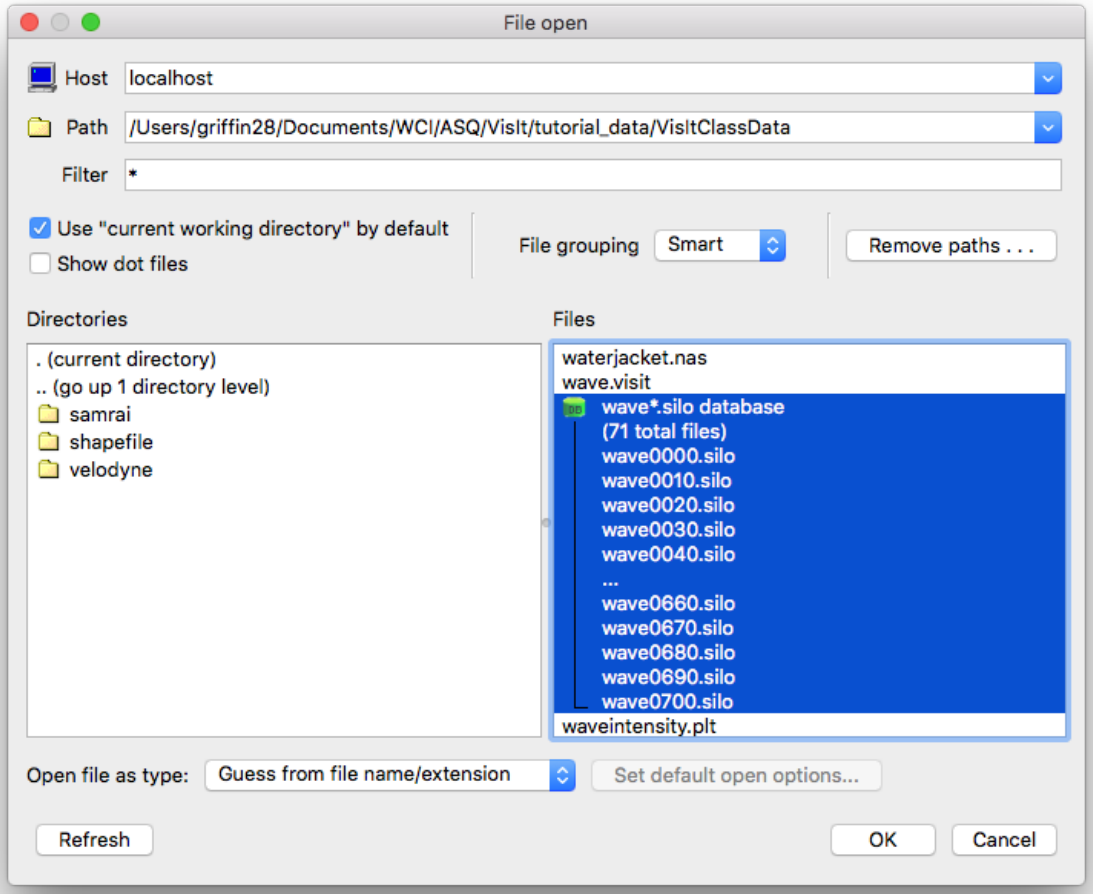


Fig. 4.388: File open window

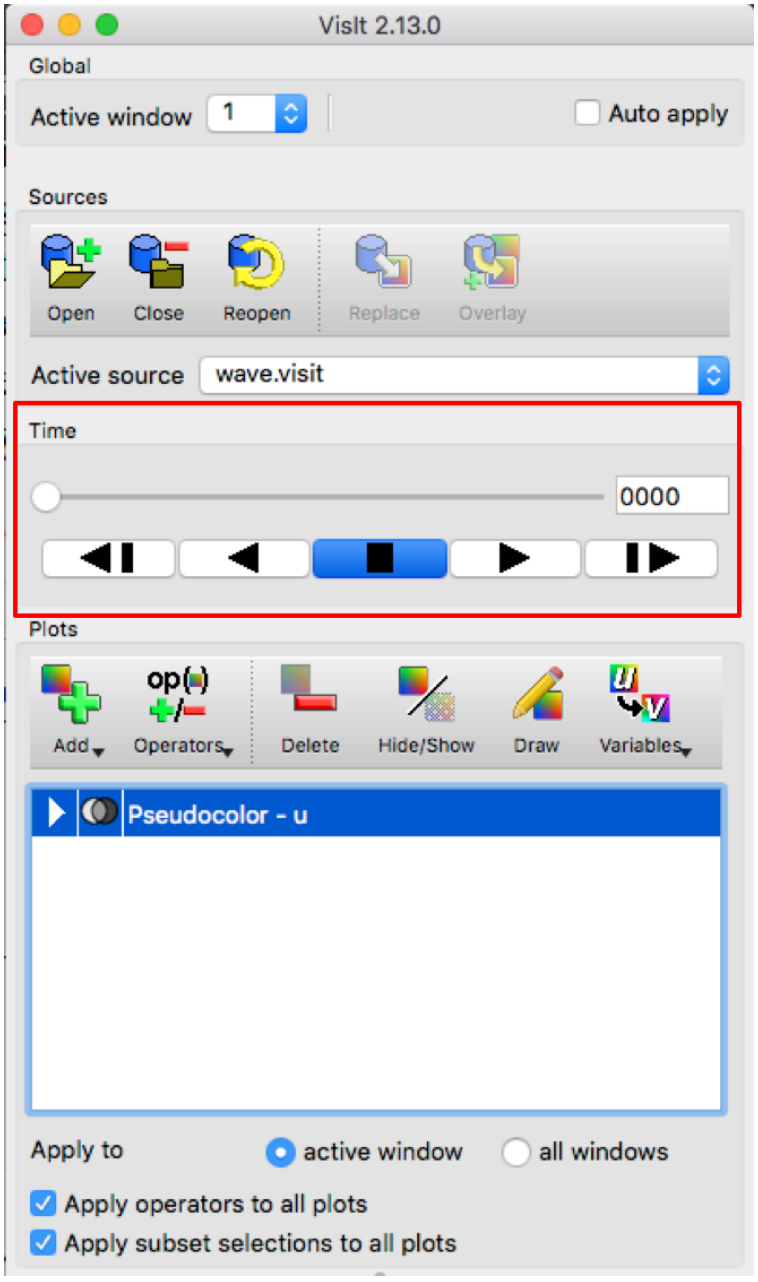


Fig. 4.389: Animation controls

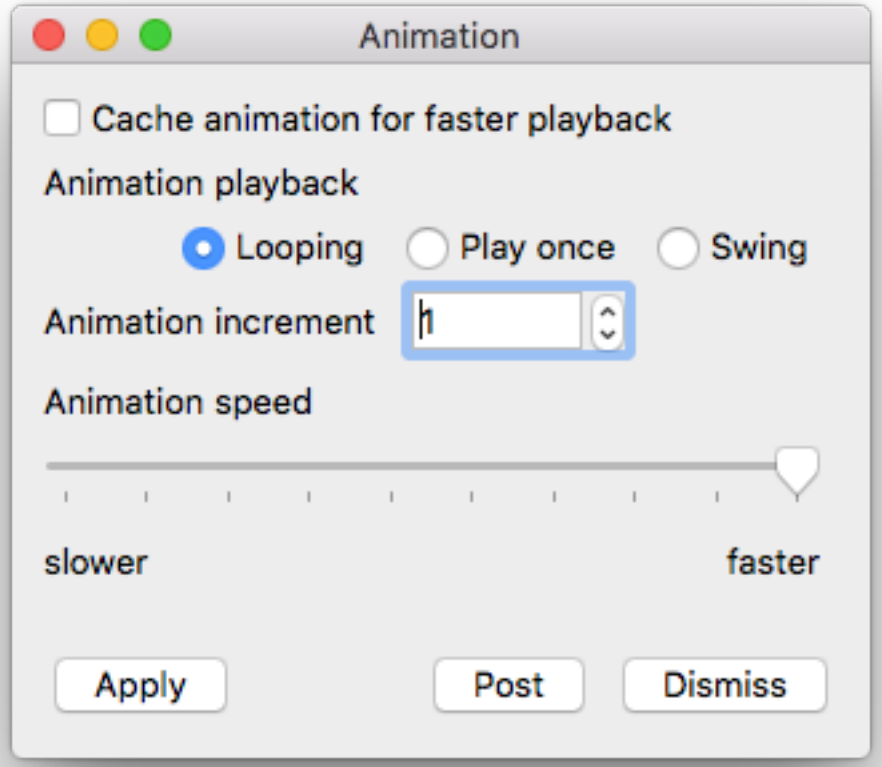


Fig. 4.390: Animation window

Animation Window

You can open the **Animation Window**, shown in Figure 4.390 , by clicking on the **Animation ...** option from the **Controls** menu. The **Animation Window** contains controls that allow you to turn off pipeline caching and adjust the animation playback mode and speed.

Animation playback speed

The animation playback speed is used when playing flipbook or keyframe animations. The playback speed determines how fast VisIt cycles through the database states that make up the animation. Rather than using states per second as a measurement for the playback speed, VisIt uses a simple scale of slower to faster. To set the animation playback speed, use the **Animation speed** slider. Moving the slider to the left and slower setting slows down animations so they change time states once every few seconds. Moving the slider to the right and faster setting will make VisIt play the animation as fast as the host graphics hardware allows.

Pipeline caching

When pipeline caching is enabled, VisIt tries to retain all of the geometric primitives that are used to draw a plot. This greatly speeds up animations once the geometry for all time states is cached. The downside to pipeline caching is that it can consume large amounts of memory. Pipeline caching is enabled by default, but sometimes it makes sense to turn it off. The deciding factors are the size of the database, the number of animation frames, and the number of plots in each animation frame. Try leaving pipeline caching enabled until you notice performance degradation. To turn off pipeline caching, uncheck the **Pipeline caching** check box in the **Animation Window** .

Animation playback mode

The animation playback mode determines how VisIt gets to the next time state after playing until the end of the animation. There are three animation playback modes: looping, play once, and swing. VisIt loops animations by default so once the end of the animation is reached, it starts playing from the beginning. When the animation mode is set to play once, VisIt plays the animation through until the end and then stops playing the animation. When VisIt reaches the end of the animation in swing mode, the animation starts playing in reverse until it gets to the start, at which point, it starts playing forward again. To set the animation mode, click on one of the **Looping**, **Play once** , and **Swing** radio buttons in the **Animation Window** .

4.10.2 Keyframing

Keyframing is an advanced form of animation that allows you create animations where certain animation attributes such as view or plot attributes can change as the animation progresses. You can design an entire complex animation upfront by specifying a number of animation frames to be created and then you can tell VisIt which plots exist over the animation frames and how their time states map to the frames. You can also specify the plot attributes so they remain fixed over time or you can make individual plot and operator attributes evolve over time. With keyframing, you can make a plot fade out as the animation progresses, you can make a slice plane move, you can make the view slowly change, etc. Keyframe animations allow for quite complex animation behavior.

There is a video tutorial that demonstrates the process of creating a keyframing animation and saving it as a movie.

Keyframing Window

Keyframe animations are designed using VisIt's **Keyframing Window** (see Figure 4.391), which you can open by selecting the **Keyframing** option from the **Controls** menu. The window is dominated by the **Keyframe area** , which

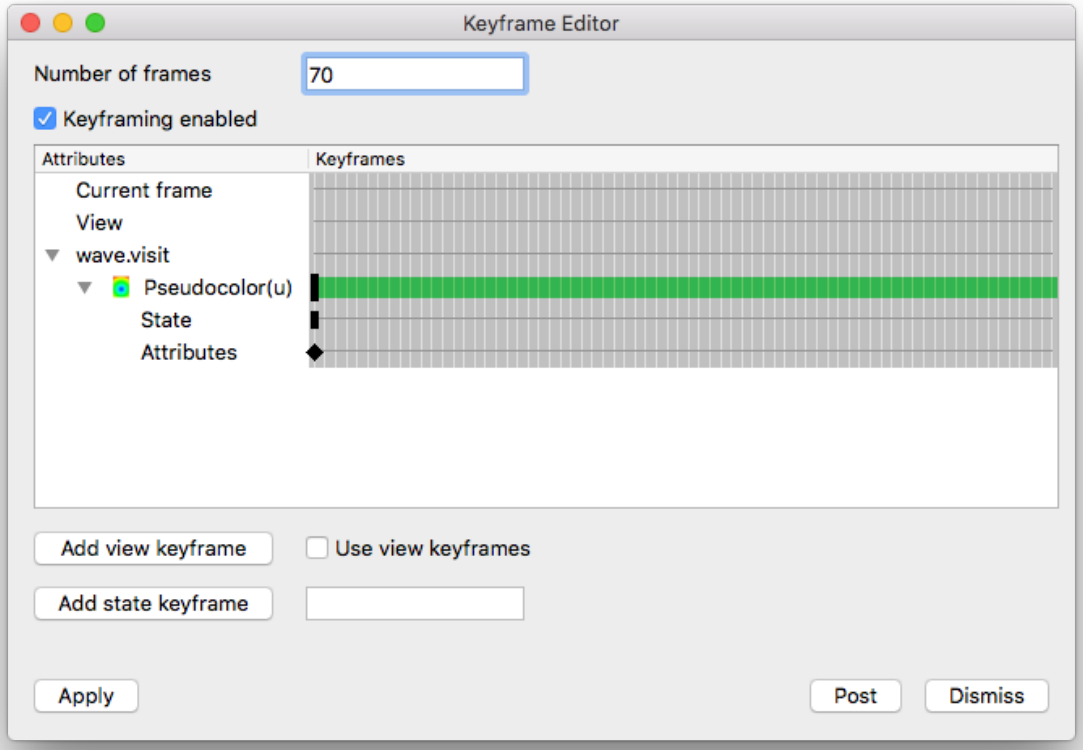


Fig. 4.391: Keyframing Window

consists of many vertical lines that correspond to each frame in the animation and horizontal lines, or **Keyframe lines** , that correspond to the state attributes that are being keyframed. The horizontal lines are the most important because they allow you to move and delete keyframes and set the plot range, which is the set of animation frames over which the plot is defined.

Keyframing mode

To create a keyframe animation, you must first open the **Keyframing Window** and check the **Keyframing enabled** check box. When VisIt is in keyframing mode, a keyframe is created for the active animation state each time you set plot or operator attributes and time is set using the **Animation** time slider. The Animation time slider is a special time slider that is made active when you enter keyframing mode and the animation frame can only be set using it. Changing time using any other time slider results in a new database state keyframe instead of changing the animation frame.

If you have created plots before entering keyframing mode, VisIt converts them into plots that can be keyframed when you enter keyframing mode. When you leave keyframing mode, extra keyframing attributes associated with plots are deleted, the animation containing the plots reverts to a flipbook animation, and the Animation time slider is no longer accessible.

Setting the number of frames

When you go into keyframing mode for the first time, having never set a number of keyframes, VisIt will use the number of states in the active database for the number of frames in the new keyframe animation. The number of frames in the keyframe animation will vary with the length of the database with the most time states unless you manually specify a number of animation frames, which you can do by entering a new number of frames into the **Keyframing Window's Number of frames** text field. Once you enter a number of frames, the number of frames will not change unless you change it.

Adding a keyframe

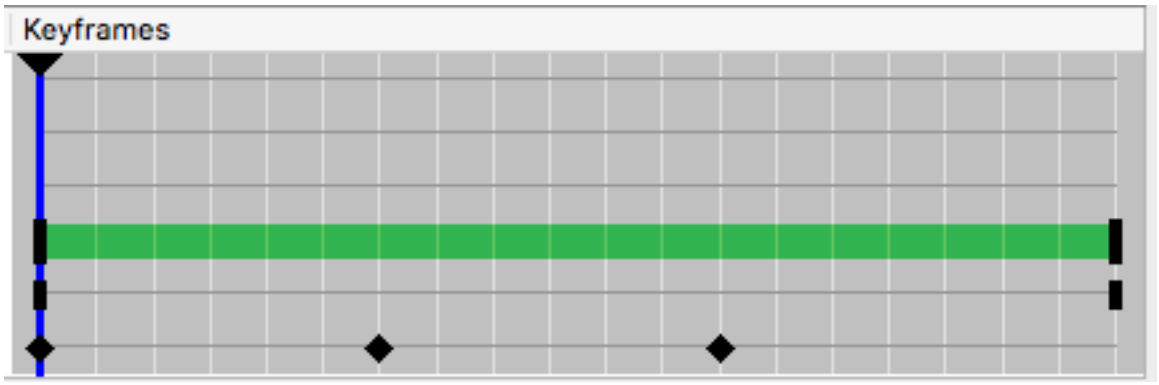


Fig. 4.392: Keyframe area

To add a keyframe, you must first have created some plots and put VisIt into keyframing mode by clicking the **Keyframing enabled** check box in the **Keyframing Window** . After you have plots and VisIt is in keyframing mode, you can add a keyframe by opening a plot's attribute window, changing settings, and clicking its **Apply** button. To set a keyframe for a later frame in the animation, move the **Keyframe time** slider, which is located under the **Keyframe area** (see Figure 4.392), to a later time and change the plot attributes again. Each time you add a keyframe to the animation, a small black diamond, called a **Keyframe indicator** , will appear along the **Keyframe line** for the plot. When you play through the animation using any of VisIt's animation controls, the plot attributes are calculated for each animation frame and they are used to influence how the plots look when they appear in the **Viewer** window.

Adding a database state keyframe

Each plot that exists at a particular animation frame must use a specific database state so the correct data will be plotted. When VisIt is in keyframing mode, the database state can also be keyframed so you can control the database state used for a plot at any given animation frame. The ability to set an arbitrary database state keyframe for a plot allows you to control the flow of time in novel ways. You can, for example, slow down time, stop time, or even make time flow backwards for a little while.

There are two ways to set database state keyframes in VisIt. The first way is to move the **Keyframe time** slider to the desired animation frame, enter a new number into the text field next to the **Keyframe Window's Add state keyframe** button, and the click the **Add state keyframe** button. As an alternative, you can use the ** Main Window's Time slider** to create a database state keyframe, provided the active time slider is not the Animation time slider. To set a database state keyframe using the **Time slider** , select a new database time slider from the Active time slider combo box and then change time states using the **Time slider** . Instead of changing the active state for the plots that use the specified database, VisIt uses the information to create a new database state keyframe for the active animation frame.

Adding a view keyframe

In addition to being able to add keyframes for plot attributes, operator attributes, and database states, you can also set view keyframes so you can create sophisticated flybys of your data. To create a view keyframe, you must interactively change the view in the **Viewer** window using the mouse or specify an exact view in the **View Window** . Once the view is where you want it for the active animation frame, open the **View Window** and click the **Make camera keyframe from view** button on the **Advanced** tab in order to make a view keyframe. Once the view keyframe has been added, a keyframe indicator will be drawn in the **Keyframing Window** .

VisIt will not use view keyframes by default when you are in keyframing mode because it can be disruptive for VisIt to set the view while you are still adding view keyframes. Once you are satisfied with your view keyframes, click the **Copy view from camera** button on the **Advanced** tab in the **View Window** in order to allow VisIt to set the view using the view keyframes when you change animation frames.

Deleting a keyframe

To delete a keyframe, move the mouse over a **Keyframe indicator** and right click on it with the mouse once the indicator becomes highlighted.

Moving a keyframe

To move a keyframe, move the mouse over a **Keyframe indicator** , click the left mouse button and drag the **Keyframe indicator** left or right to a different animation frame. If at any point you drag the **Keyframe indicator** outside of the green area, which is the plot time range, and release the mouse button, moving the keyframe is cancelled and the **Keyframe indicator** returns to its former animation frame.

Changing the plot time range

The plot time range determines when a plot appears or disappears in a keyframed animation. Since VisIt allows plots to exist over a subset of the animation frames, you can set a plot's plot range in the **Keyframe area** to make a plot appear later in an animation or be removed before the animation reaches the last frame. You may find it useful to set the plot range if you have increased the number of animation frames but found that the plot range did not expand to fill the new frames. To change the plot time range, you left-click on the beginning or ending edges of the **Plot time range** (the green band on the **Keyframe line**) in the **Keyframe area** and drag it to a new animation frame.

4.10.3 Movie tools

VisIt provides a command line utility based on VisIt's Command Line Interface that is called `visit -movie` . The `visit -movie` movie generation utility is installed with all versions of VisIt and can be used to generate movies using session files or Python scripts as input. If you want to design movies based on visualizations that you have created while using VisIt's GUI then you might also want to read about the **Save movie wizard** . If the visit command is in your path then typing `visit -movie` at the command prompt, regardless of the platform that you are using, will launch the `visit -movie` utility. The following list provides `visit -movie` command line arguments:

`-format fmt` The format option allows you to set the output format for your movie. The supported values for fmt are:

- mpeg : MPEG 2 movie.
- qt : QuickTime movie.
- sm : Streaming movie format.
- png : Save raw movie frames as individual PNG files.
- ppm : Save raw movie frames as individual PPM files.
- tiff : Save raw movie frames as individual TIFF files.
- jpeg : Save raw movie frames as individual JPEG files.
- bmp : Save raw movie frames as individual BMP (Windows Bitmap) files.
- rgb : Save raw movie frames as individual RGB (SGI format) files.

`-geometry size` The geometry option allows you to set the movie resolution. The size argument is of the form WxH where W is the width of the image and H is the height of the image. For example, if you want an image that is 1024 pixels wide and 768 pixels tall, you would provide: -geometry 1024x768.

`-sessionfile name` The sessionfile option lets you pick the name of the VisIt session to use as input for your movie. The VisIt session is a file that describes the movie that you want to make and it is created when you save your session from within VisIt's GUI after you set up your plots how you want them.

`-scriptfile name` The scriptfile option lets you pick the name of a VisIt Python script to use as input for your movie.

`-framestep name` The number of frames to advance when going to the next frame.

`-start frame` The frame at which to start.

`-end frame` The frame at which to end.

`-fps number` Sets the frames per second at which the movie should be played.

`-output` The output option lets you set the name of your movie.

The `visit -movie` utility always supports creation of series of image files but it does not always support creation of movie formats such as QuickTime, or Streaming movie. Support for movie formats varies based on the platform. QuickTime and Streaming movie formats are currently limited to computers running IRIX and the appropriate movie conversion tools (*makemovie*, *img2sm*) must be in your path or VisIt will create a series of image files instead of a single movie file. You can always use *visit -movie* to generate the individual movie frames and then use your favorite movie generation software to convert the frames into a single movie file.

If you browse the Windows file system and come across a VisIt session file, which ends with a `.session` extension, you can right click on the file and choose from several movie generation options. The movie generation options make one-click movie generation possible so you don't have to master the arguments for `visit -movie` like you do on other platforms. After selecting a movie generation option for a VisIt session file, Windows runs `visit -movie` implicitly with the right arguments and saves out the movie frames to the same directory that contains the session file,

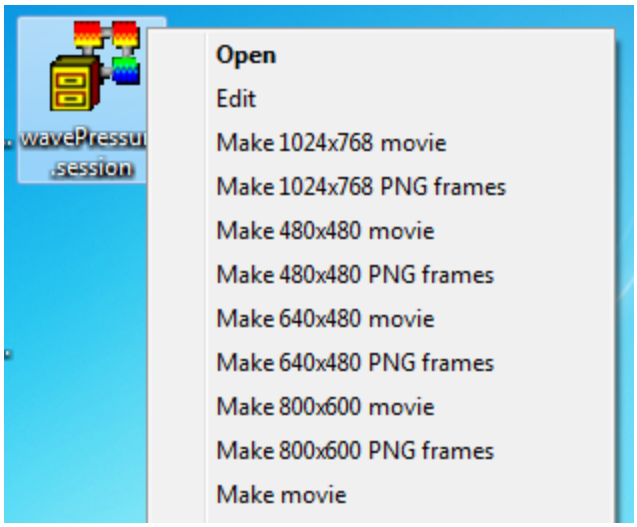


Fig. 4.393: Movie generation options for session files on Windows platform

and will have the same name as the session file. The movie generation options in a session file's context menu are shown in Figure 4.393.

4.11 Interactive Tools

An interactive tool is an object that can be added to a visualization window to set attributes for certain plots and operators such as the Parallel Coordinates plot or Slice operator. You can turn interactive tools on and off by clicking on the tool icons in a visualization window's **Toolbar** or **Popup menu** (see Figure 4.394). Note that some tools prefer to operate in visualization windows that contain plots of a certain dimension so some tools are not always available.

Once you enable a tool, its appears in the visualization window. Tools have one or more small red rectangles called *hot points* that cause the tool to perform an action when you click or drag the hot point with the mouse. When you use the mouse to manipulate a tool's hot point, all mouse events are delivered to the tool so it can respond to the mouse interaction. When the mouse is outside of a hot point, the mouse responds as it would if there were no tools activated so you can still rotate and zoom-in on plots while still having tools enabled.

4.11.1 Box Tool

The box tool, which is shown in Figure 4.395, allows you to move an axis-aligned box around in 3D space. You can use the box tool with the Box and Clip operators to interactively restrict plots to a certain volume. The box tool is drawn as a box with five hotpoints that allow you to move the box in 3D space or resize it in any or all dimensions.

You can move the box tool around the **Viewer** window by clicking on the origin hotpoint, which has the word "Origin" next to it, and dragging it around the **Viewer** window. When you move the box tool, it moves in a plane that is parallel to the screen. You can move the box tool backward and forward along an axis by holding down the keyboard's *Shift* key before you click and drag the origin hotpoint. When the box tool moves, red, green, and blue boxes appear to give a point of reference for the box with respect to the X, Y, and Z dimensions (see Figure 4.396).

You can extend one of the box's faces at a time by clicking on the appropriate hotpoint and moving the mouse up to extend the box or by moving the mouse down to shrink the box in the given dimension. Hotpoints for the box's back faces are drawn smaller than their front-facing counterparts. When the box is resized in a single dimension, reference planes are drawn in the dimension that is changing so you can see where the edges of the box are in relation to the bounding box for the visible plots. You can also resize all of the dimensions at the same time by clicking on the

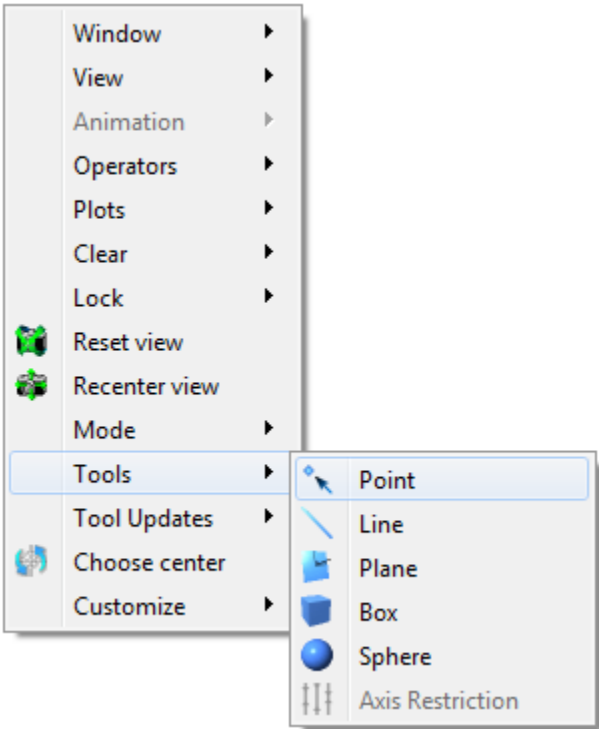


Fig. 4.394: Tools menu

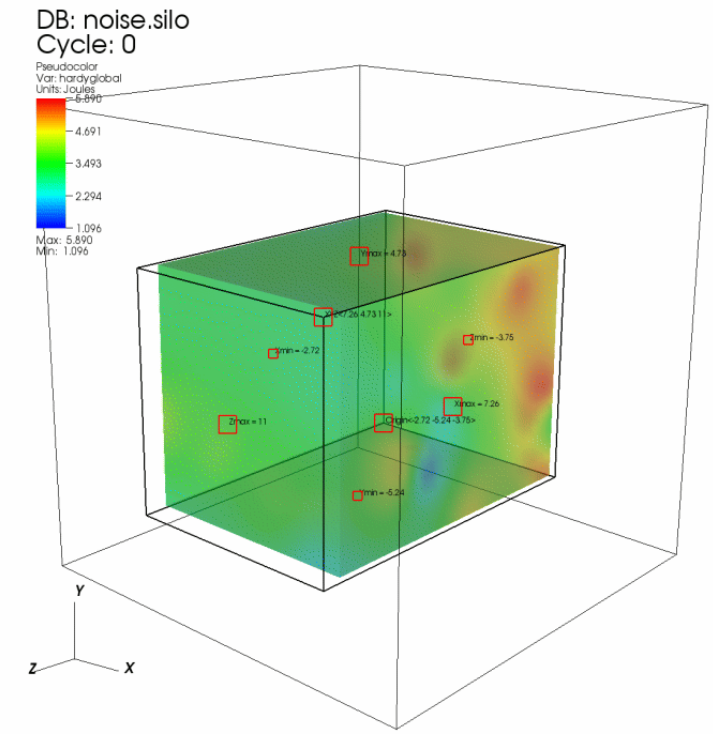


Fig. 4.395: Box tool with a plot restricted to the box

"Resize XYZ" hotpoint and dragging the mouse in an upward motion to scale the box to a larger size in X,Y, and Z or by dragging the mouse down to shrink the box. When all box dimensions are resized at the same time, the shape of the box remains the same but the scale of the box changes.

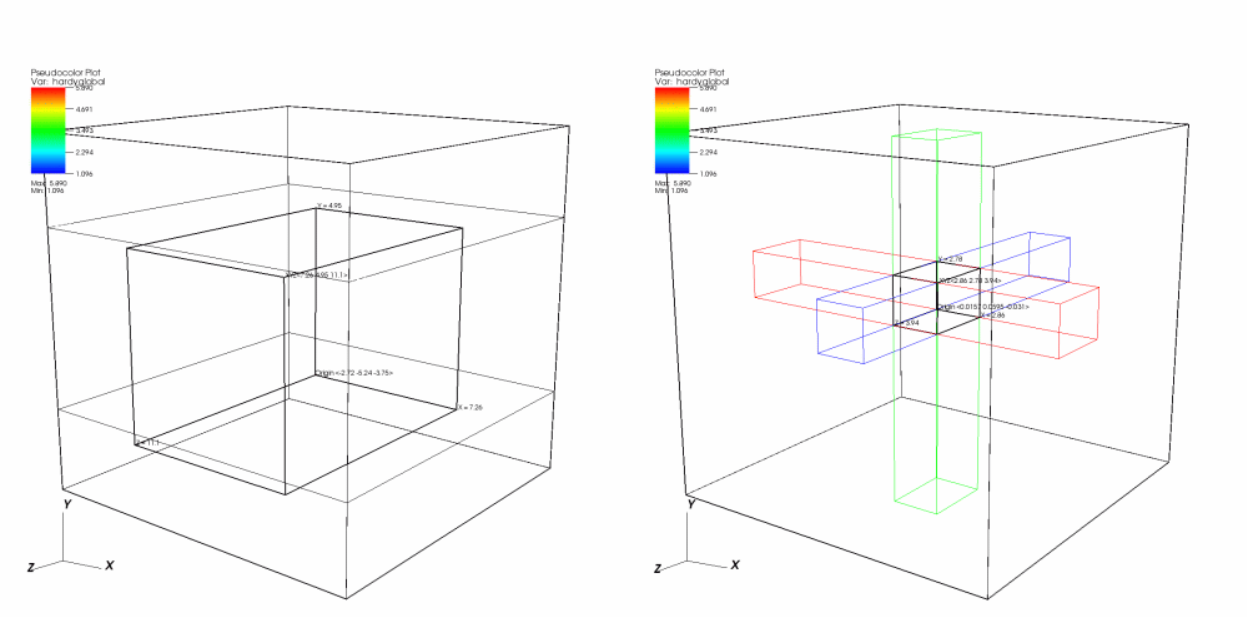


Fig. 4.396: Box tool while it is resized or moved

4.11.2 Line Tool

It is common to create Curve plots when analyzing a simulation database. Curve plots are created using VisIt's lineout mechanism where reference lines are drawn in a visualization window and Curve plots are created in another visualization window using the path described by the reference lines. VisIt's line tool allows reference lines to be moved after they are initially drawn. The line tool allows the user to see a representation of a line in a visualization window and position the line relative to plots that exist in the window.

The line tool is drawn as a thick line with three hot points positioned along the length of the line. Both of the line tool's endpoints, as well as its center, have a hotpoint. Since the line tool can be used for both 2D and 3D databases, the line tool's behavior is slightly different for 2D than it is for 3D. Clicking and dragging on either endpoint will move the selected endpoint causing the line to change shape. Another way of moving an endpoint is to hold down the *Ctrl* key and then click on the point and move the mouse up and down to extend or shorten the line. Clicking and dragging the middle hot point moves the entire line tool.

In 2D, the line endpoints can only be moved in the X-Y plane (Figure 4.397). In 3D, the line endpoints can be moved in any dimension. Since it is more difficult to see how the line is oriented relative to plots in 3D, when the line tool is moved, 3D crosshairs appear. The crosshairs intersect the bounding box and show the position of the line endpoint relative to the plots. Clicking and dragging endpoints will move them in a plane that is perpendicular to the screen. Moving the endpoints, while first pressing and holding down the *Shift* key, causes the selected endpoint to move back and forth in the dimension that most faces the screen. This allows endpoints to be moved in one dimension at a time. An example of the line tool in 3D is shown in Figure 4.398.

The line tool can be used to set the attributes for certain VisIt operators such as VisIt's *Lineout operator*. If a plot has a Lineout operator applied to it, and the Lineout operator's *interactive* option is turned on (see Lineout's *Interactive mode* for more details), the line tool is initialized with that operator's endpoints when it is first enabled. (*Note: Due to a current bug, the tool must be activated, deactivated, then activated a second time in order to be properly initialized*

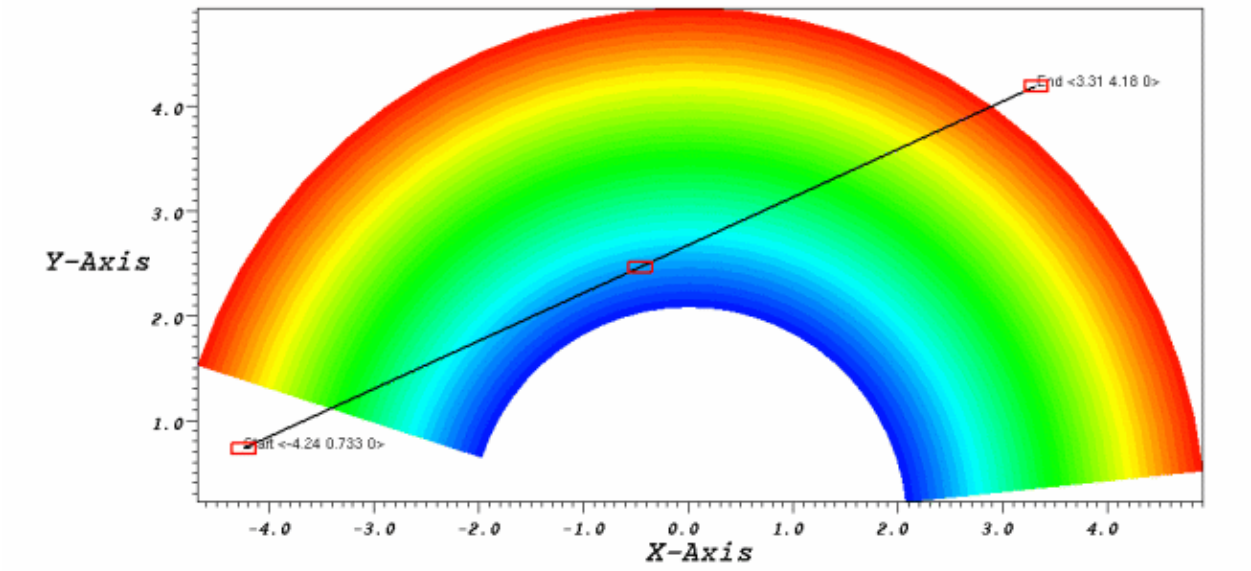


Fig. 4.397: Line tool with a 2D plot

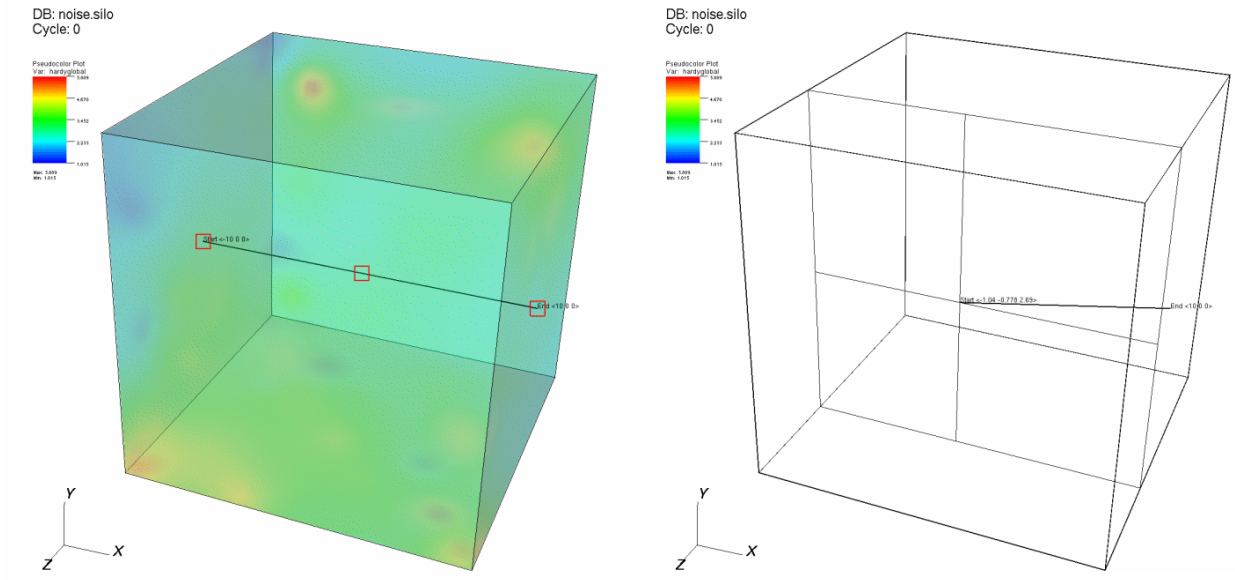


Fig. 4.398: Line tool in 3D

with the Lineout's endpoint values.) As the line tool is repositioned and reoriented, the line tool's line endpoints are given to the Lineout operator and the Curve plots that are fed by the Lineout operator are recalculated.

4.11.3 Plane Tool

The plane tool allows the user to see a representation of a slice plane in a visualization window and position the plane relative to plots that may exist in the window. The plane tool, shown in Figure 4.399, is represented as a set of 3D axes, a bounding rectangle, and text which gives the plane equation in origin-normal form. The plane tool provides several hot points positioned along the 3D axes that are used to position and orient the tool. The hot point nearest the origin allows the user to move the plane tool in a plane parallel to the computer screen. The hot point that lies in the middle of the plane's Z-axis translates the plane tool along its normal vector when the hotpoint is dragged up and down. The hot point on the end of the Z-axis causes the plane tool to rotate freely when the hot point is moved. When the plane tool is facing into the screen, the Z-axis vector turns red to indicate which direction the plane tool is pointing. The other hot points also rotate the plane tool but they restrict the rotation to a single axis of rotation.

You can use the plane tool to set the attributes for certain VisIt plots and operators. The *Slice operator*, for example, can update its plane equation from the plane tool's plane equation. If a plot has a Slice operator applied to it, the plane tool is initialized with that operator's slice plane when it is first enabled. As the plane tool is repositioned and reoriented, the plane tool's plane equation is given to the operator and the sliced plot is recalculated.

4.11.4 Point Tool

The point tool allows you to position a single point relative to plots that exist in the visualization window. The point tool provides one hot point at the tool's origin. Clicking on the hot point and moving the mouse moves the point tool's origin in a plane perpendicular to the screen. Holding down the *Shift* key before clicking on the hot point moves the point tool's origin along the plot axis that most faces the user. Holding down the *Ctrl* key moves the point tool along the plot axis that points up. Figure 4.400 shows the point tool being used to set the origin for the *ThreeSlice operator*.

4.11.5 Sphere Tool

The sphere tool allows you to position a sphere relative to plots that exist in the visualization window. The sphere tool, shown in Figure 4.401, provides several hot points that are used to position and scale the sphere. The hot point nearest the center of the sphere is the origin hot point and it is used to translate the sphere in a plane parallel to the screen. The other hot points are all used to scale the sphere. To scale the sphere, click on one of the scaling hot points and move the mouse towards the origin hot point to shrink the sphere or move the hot point away from the origin to enlarge the sphere.

You can use the sphere tool to set the attributes for certain VisIt plots and operators. The sphere tool is commonly used to set the attributes for the *SphereSlice operator*. After applying a SphereSlice operator to a plot, enable the Sphere tool to interactively position the sphere that slices the plot.

4.11.6 Axis Restriction Tool

The AxisRestriction tool is used in conjunction with the Parallel Coordinates plot allowing you to modify the axis restrictions used by the plot. The Axis Restriction tool, shown in Figure 4.402, provides triangular hot points that are originally positioned at the tops and bottoms of each axis in the plot. As the hot points are moved up or down the axis, the plot is changed to reflect the new min or max.

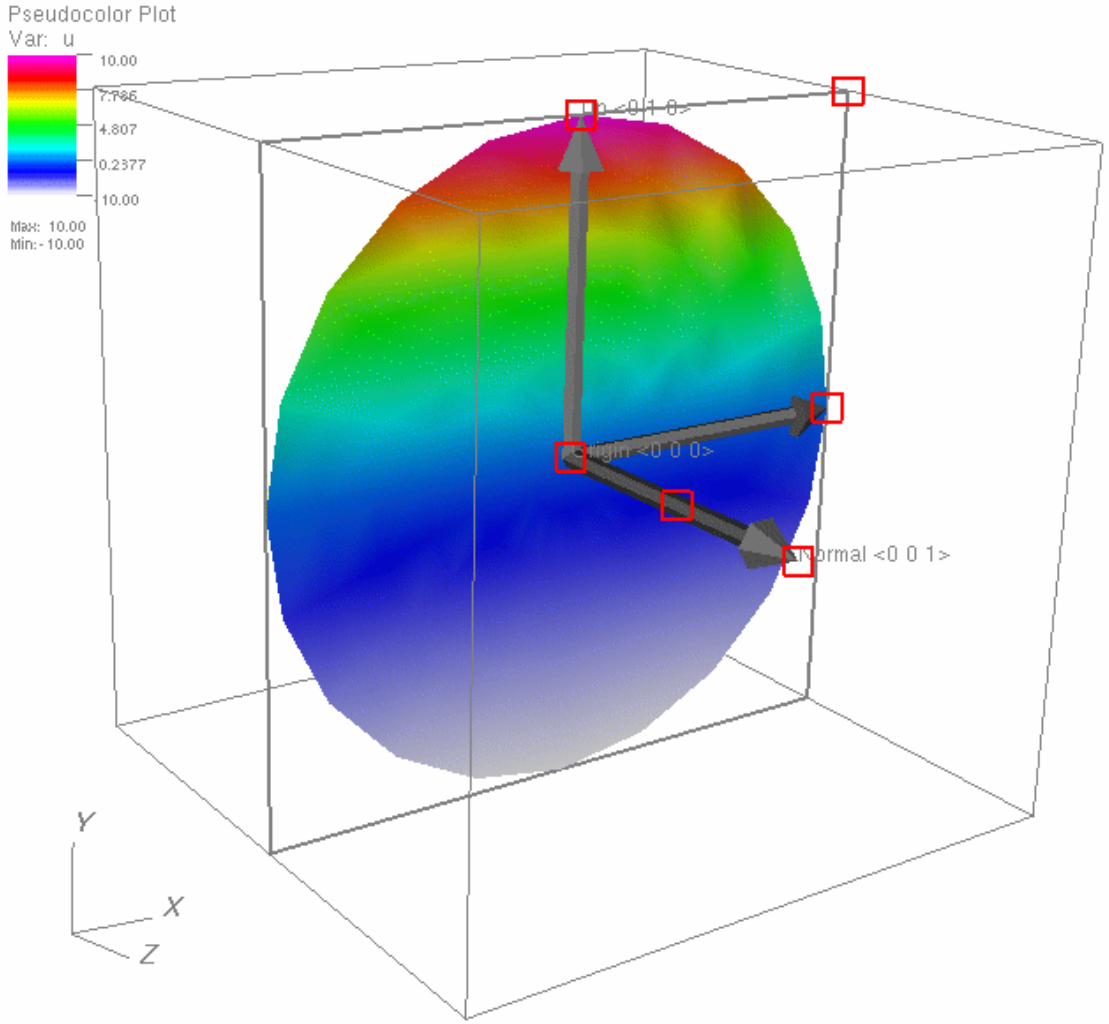


Fig. 4.399: Plane tool with sliced plot

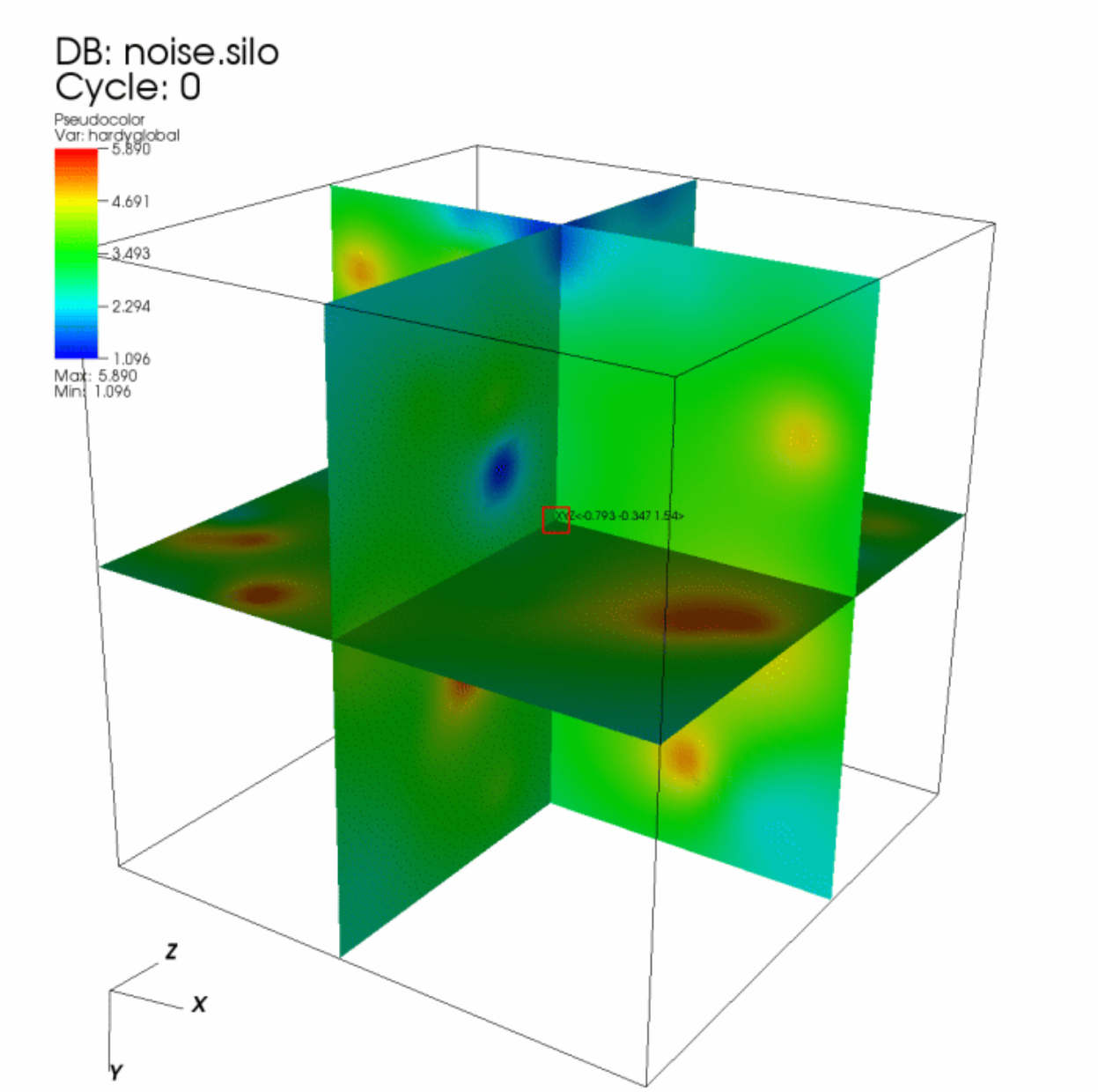


Fig. 4.400: Point tool

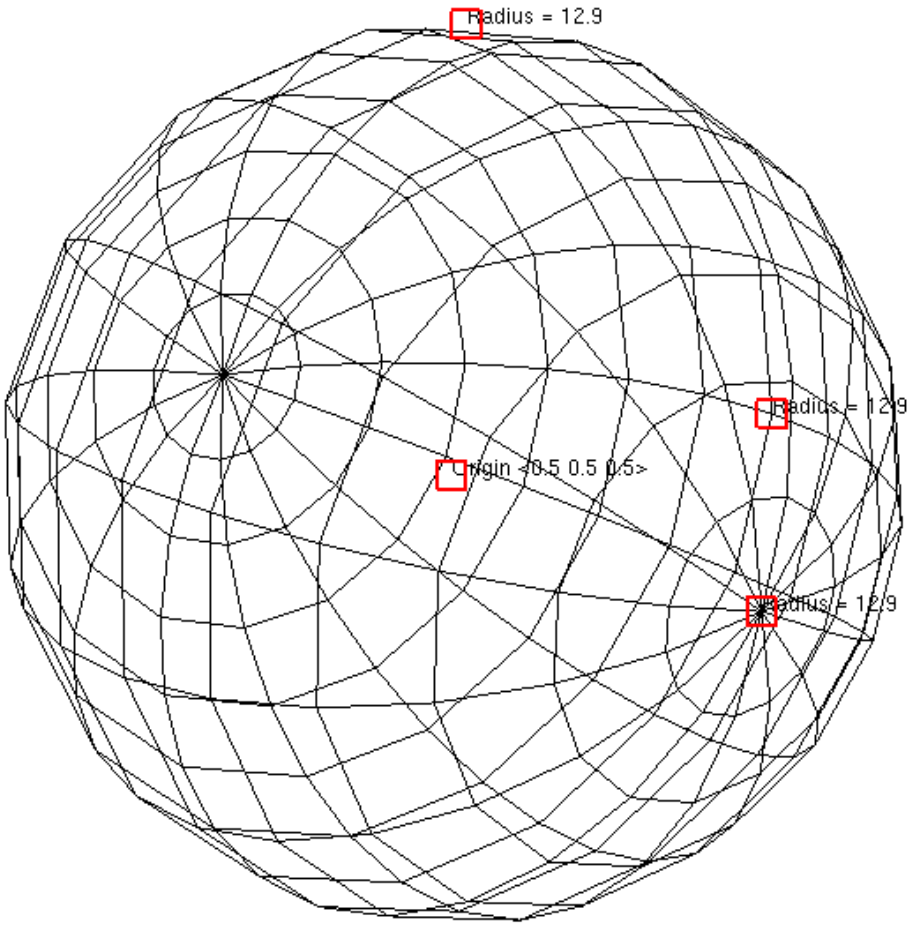


Fig. 4.401: Sphere tool

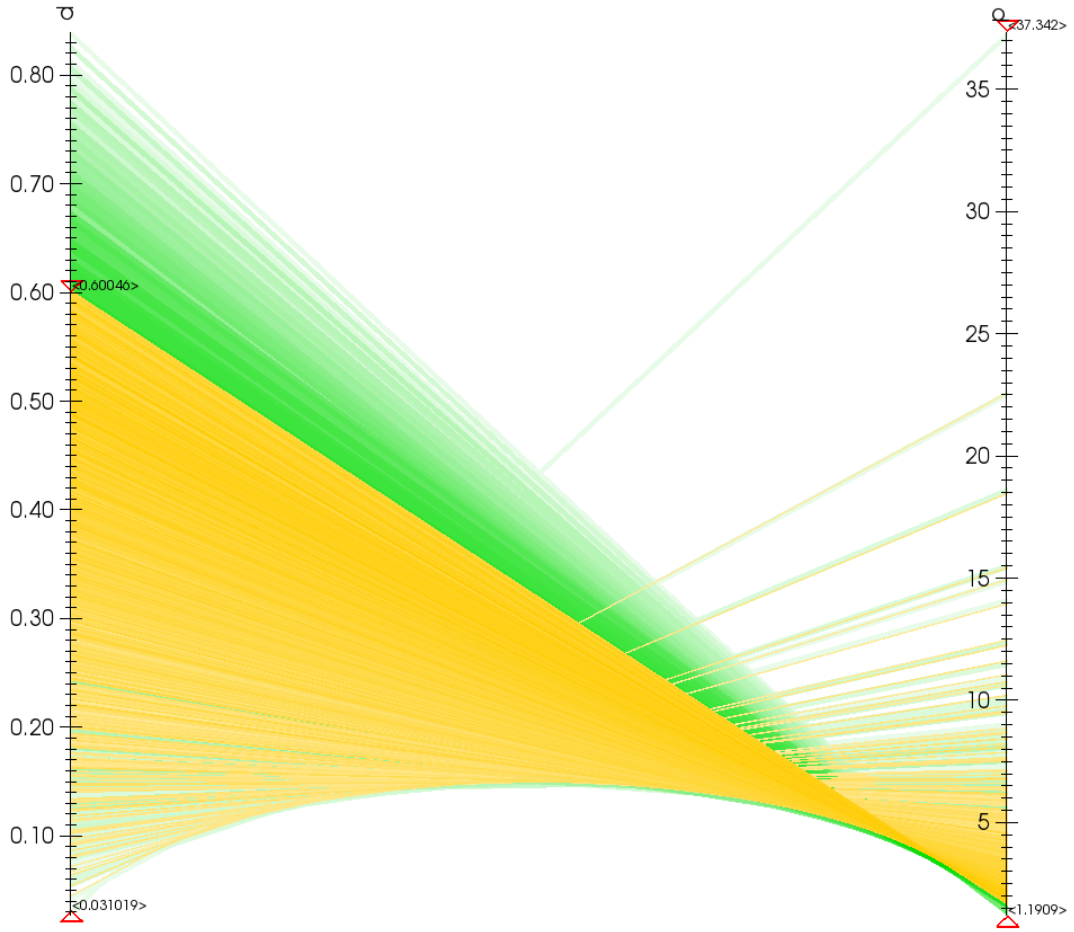


Fig. 4.402: Axis Restriction tool

4.12 Multiple Databases and Windows

In this chapter, we discuss how to use VisIt to visualize multiple databases using either a single window or multiple visualization windows that have been locked together. After a general discussion of databases, we move to database correlation, which is used to relate multiple time-varying databases together in some fashion. The use of database correlations will be explained in detail followed by a description of database comparisons, then common useful operations involving multiple visualization windows.

4.12.1 Databases

One main use of a visualization tool such as VisIt is to compare multiple related simulation databases. Simulations are often run over and over with a variety of different settings or physics models and this results in several versions of a simulation database that all describe essentially the same object or phenomenon. Simulations are also often run using different simulation codes and it is important for a visualization tool to compare the results from both simulations for validation purposes. You can use VisIt to open any number of databases at the same time so you can create plots from different simulation databases in the same window or in separate visualization windows that have been locked together.

Active database

VisIt can have any number of databases open simultaneously but there is still an active database that is used to create new plots. VisIt calls this the **Active source**. Each time you open a database, the newly opened database becomes the active source for any new plots that you decide to create. If you want to create a plot using a database that is open but is not your active source, you must make that database the active source. When a database becomes the active source, its variables are added to the menus for the various plot types. To changing the active source, select a database from the **Active source** combo box in the **Main Window** as shown in Figure 4.403.

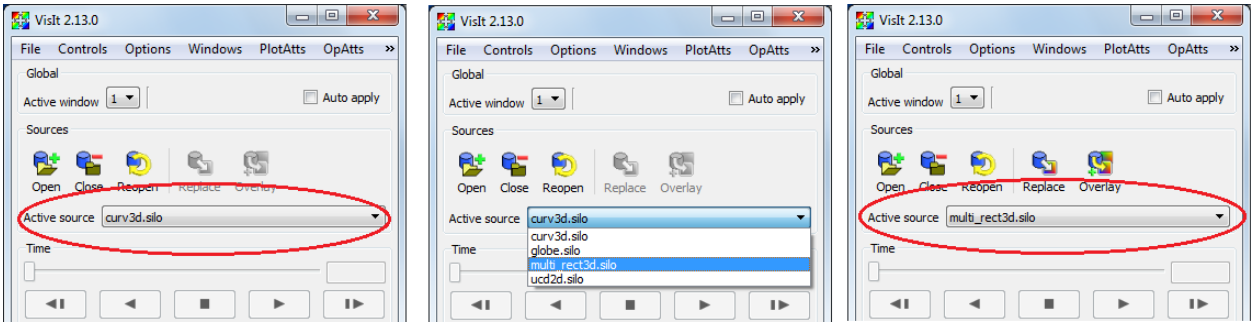


Fig. 4.403: Changing the active source.

Multiple time sliders

When your open databases all have only a single time state, the **Time slider** in the **Main Window** is disabled. When you have one database that has multiple time states, the **Time slider** is enabled and can be used exclusively to change time states for the database that has multiple time states; the database does not even have to be the active database. Things get a little more complicated when you have opened more than one time-varying database - especially if you have plots from more than one of them.

When you open a database in VisIt, it becomes the active database. If the database that you open has multiple time states, VisIt creates a new logical time slider for it so you can end up having a separate time slider for every open database with multiple time states. When VisIt has to create a time slider for a newly opened database, it also makes

the new database's (the *active source*) be the active time slider. There is only one **Time slider** control in the **Main Window** so when there are multiple logical time sliders, VisIt displays an **Active time slider** combo box (see Figure 4.404) that lets you choose which logical time slider to affect when you change time using the **Time slider**.

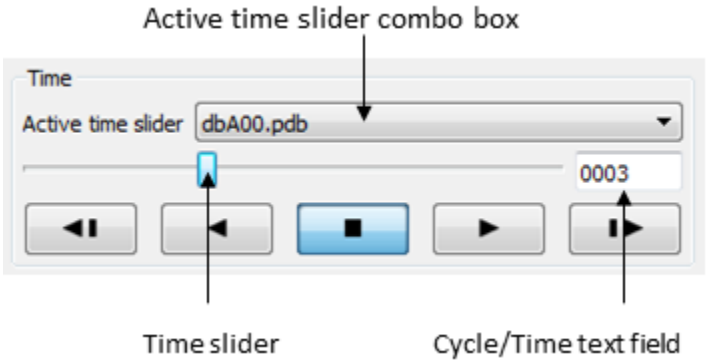
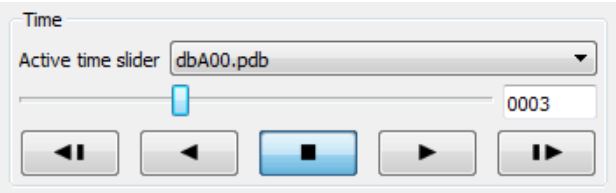
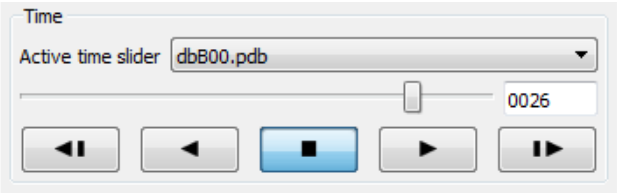


Fig. 4.404: Time slider and related controls

Since VisIt allows each time-varying database to have its own logical time slider, you can create plots from more than one time-varying database in a single visualization window and change time independently for each database. Another benefit of having multiple logical time sliders is that the databases plotted in the visualization windows are free to have different numbers of time states. Suppose you have opened time-varying databases A and B and created plots from both databases in the same visualization window. Assuming you opened database A and then database B, database B will be the active database. If you want to change time states for database A but not for database B, you can select database A from the **Active time slider** combo box and then change the time state using the **Time slider**. If you then wanted to change time states for database B, you could select it in the **Active time slider** combo box and then change the time state using the **Time slider** . If you wanted to change time states for both A and B at the same time, you have to use database correlations, which are covered next.





4.12.2 Database correlations

A database correlation is a map that relates one or more different time-varying databases so that when accessed with a common time state, the database correlation can tell VisIt which time state to use for any of the databases in the database correlation. VisIt supports multiple logical time sliders, so time states can be changed independently for different time-varying databases in the same window. No time slider for any database can have any effect on another database. Sometimes when comparing two different, but related, time-varying databases, it is useful to make plots of both databases and see how they behave over time. Since changing time for each database independently would be tedious, VisIt provides database correlations to simplify visualizing multiple time-varying databases.

Database correlations and time sliders

When you open a database for the first time, VisIt creates a trivial database correlation for that single database and creates a new logical time slider for it. Each database correlation has its own logical time slider. Figure 4.406 shows a database correlation as the active time slider.

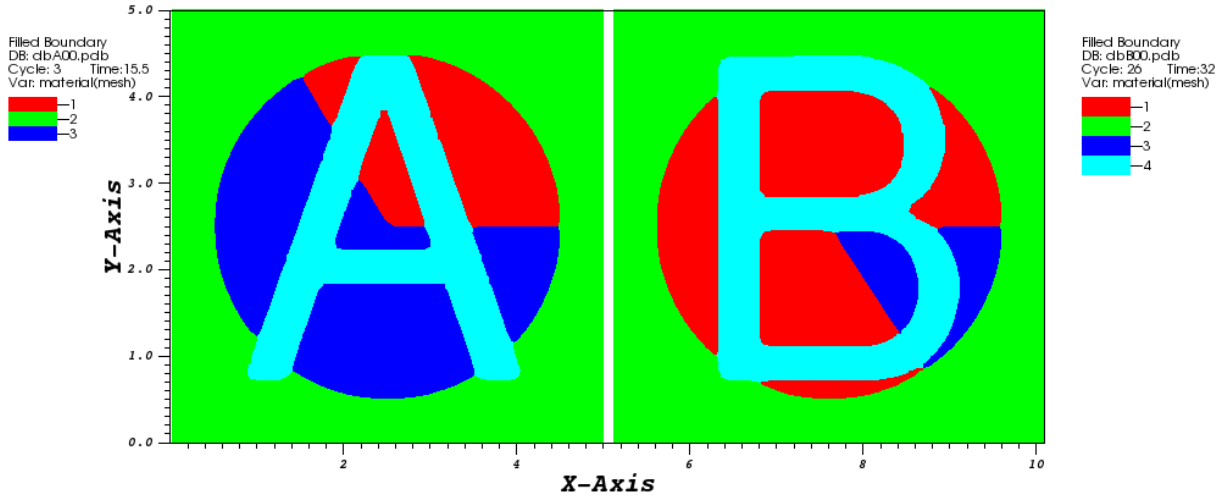


Fig. 4.405: Active time slider and time slider controls

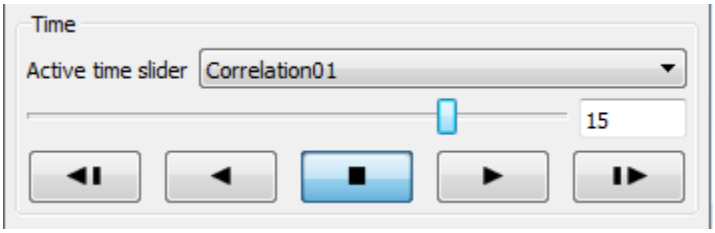


Fig. 4.406: Database correlation as the active time slider

Suppose you have plots from time-varying database A and database B in the same visualization window. You can use the logical time slider for database A to change database A's time state and you can use the logical time slider for database B to change database B's time state. If you want to change the time state for both databases at the same time using a single logical time slider, you can create a database correlation involving database A and database B and then change time states using the database correlation's logical time slider. When you change time states using a database correlation's time slider, the time state used in each plot is calculated by using the database correlation's time slider's time state to look up the plot's expected time state in the database correlation. Thus changing time states using a database correlation also updates the logical time slider for each database involved in the database correlation.

Types of database correlations

A database correlation is a map that relates one or more databases. When there is more than one database involved in a database correlation, the time states from each database are related using a correlation method. Database correlations currently have 4 supported correlation methods: padded index, stretched index, time, and cycle. This section describes each of the correlation methods and when you might want to use each method.

For illustration purposes, the examples describing each correlation method use two databases, though database correlations can have any number of databases. The examples refer to the databases as: database A and database B. Both databases consist of a rectilinear grid with a material variable. The material variable is used to identify the database using a large letter A or B and also to visually indicate progress through the databases' numbers of time states by sweeping out a red material like a clock in reverse. At the first time state, there is no red material but as time progresses, the read material increases and finally totally replaces the material that was blue. Database A has 10 time states and database B has 20 time states. The tables below list the cycles and times for each time state in each database so the time and cycle behavior of database A and database B will make more sense later when time database correlations and cycle database correlations are covered.

Table 4.1: Database A

Time state	0	1	2	3	4	5	6	7	8	9
Times	14	14.5	15	15.5	16	16.5	17	17.5	18	18.5
Cycles	0	1	2	3	4	5	6	7	8	9

Table 4.2: Database B (part 1)

Time state	0	1	2	3	4	5	6	7	8	9
Times	16	17	18	19	20	21	22	23	24	25
Cycles	10	11	12	13	14	15	16	17	18	19

Table 4.3: Database B (part 2)

Time state	10	11	12	13	14	15	16	17	18	19
Times	26	27	28	29	30	31	32	33	34	35
Cycles	20	21	22	23	24	25	26	27	28	29

Padded index database correlation

A padded index database correlation, like any other database correlation, involves multiple input databases where each database potentially has a different number of time states. A padded index database correlation has as many time states as the input database with the largest number of time states. All other input databases that have fewer time states than the longest database have their last time state repeated until they have the same number of time states as the input database with the largest number of time states. Using the example databases A and B, since B has 20 time states and A only has 10 time states, database A will have its last time state repeated 10 times to make up the difference in time states between A and B. Note how database A's last time state is repeated in Figure 4.407.

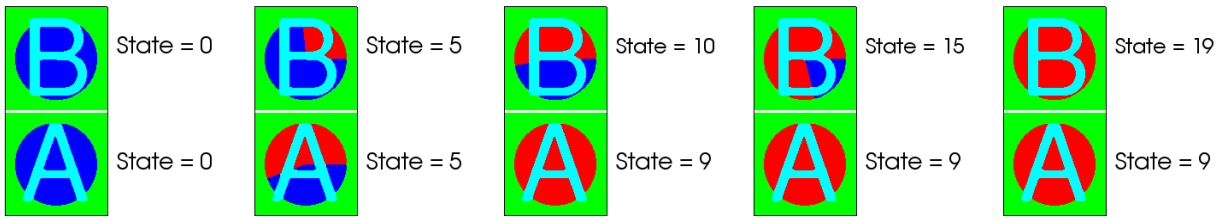


Fig. 4.407: Padded index database correlation of A and B (every 5th time state)

Stretched index database correlation

A stretched index database correlation, like any other database correlation, involves multiple input databases where each database potentially has a different number of time states. Like a padded index database correlation, a stretched index database correlation also has as many time states as the input database with the largest number of time states. The difference between the two correlation methods is in how the input databases are mapped to a larger number of time states. The padded index database correlation method simply repeated the last frame of the input databases that needed more time states to be made even with the length of the database correlation. Stretched index database correlations on the other hand do not repeat only the last frame; they repeat frames throughout the middle time states until shorter input databases have the same number of time states as the database correlation. The effect of repeating time states throughout the middle is to evenly spread out the time states over a larger number of time states.

Stretched index database correlations are useful for comparing related related simulation databases where one simulation wrote out data at 2x, 3x, 4x, ... the frequency of another simulation. Stretched index database correlations repeat the data for smaller databases, which makes it easier to compare the databases. Figure 4.408 shows example databases A and B related using a stretched index database correlation. Note how the plots for both databases, even though the databases contain a different number of time states, remain roughly in sync.

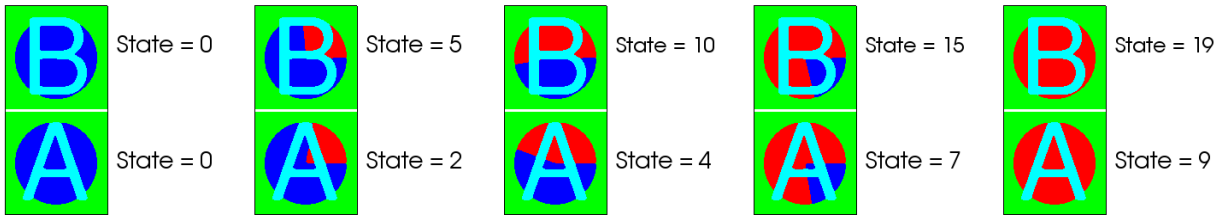


Fig. 4.408: Stretched index database correlation of A and B (every 5th time state)

Time database correlation

A time index database correlation, like any other database correlation, involves multiple input databases where each database potentially has a different number of time states. The number of time states in a time database correlation is not directly related to the number of time states in each input database. The number of time states in the database correlation are instead determined by counting the number of unique time values for every time state in every input database. The times from each input database are arranged on a number line and each unique time value is counted as one time state. Time values from different input databases that happen to have the same time value are counted as a single time state. Once the time values have been arranged on the number line and counted, VisIt calculates a list of time state indices for each database that identify the right time state to use for each database with respect to the time database correlation's time state. The first time state for each database is always the first time state index stored for a database. The first time state is used until the time exceeds the first time on the number line, and so on.

Time database correlations are useful in many of the same situations as stretched index database correlations since

they are both used to align different databases in time. Unlike a stretched index database correlation, the time database correlation does a better job of aligning unrelated databases in actual simulation time rather than just spreading out the time states until each input database has an equal number. Use a time database correlation when you are correlating two or more databases that were generated with different dump frequencies or databases that were generated by totally different simulation codes. Figure 4.409 shows the behavior of databases A and B when using a time database correlation.

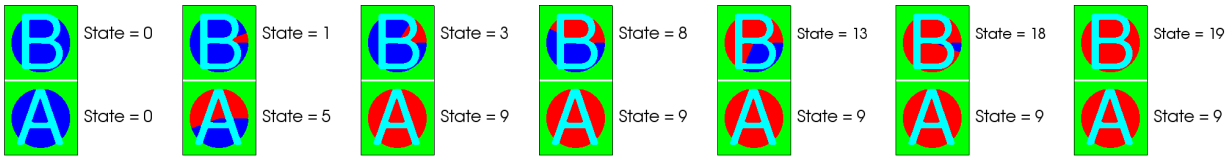


Fig. 4.409: Time database correlation of A and B (every 5th time state)

Cycle database correlation

Cycle database correlations operate in exactly the same way as time database correlations except that they correlate using the cycles from each input database instead of using times. Figure 4.409 shows the behavior of databases A and B when using a cycle database correlation.

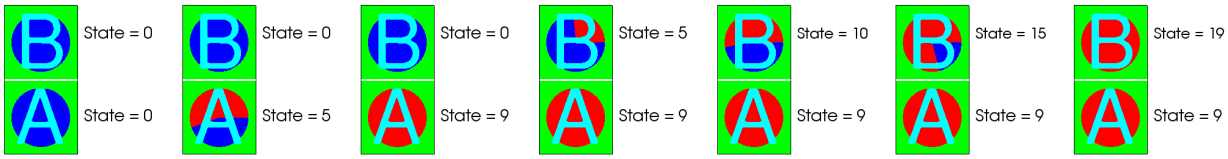


Fig. 4.410: Cycle database correlation of A and B (every 5th time state)

Managing database correlations

If you want to create a new database correlation or edit properties related to database correlations, you can use the **Database Correlation Window**. You can open the **Database Correlation Window**, shown in Figure 4.411, by clicking on the **Database correlations** option in the **Main Window's Controls** menu. The **Database Correlation Window** contains the list of database correlations, along with controls that allow you to create new database correlations, edit existing database correlations, delete database correlations, or set global settings that tell VisIt when to automatically create database correlations.

Creating a new database correlation

If you want to create a new database correlation to relate time-varying databases that you have opened, you can do so by opening the **Database Correlation Window**. The **Database Correlation Window** contains a list of trivial database correlations for the time-varying databases that you have opened. You can create a new, database correlation by clicking on the New button to the left of the list of database correlations. Clicking the **New** button opens a **Database Correlation Properties Window** (Figure 4.412) that you can use to edit properties for the database correlation.

New database correlations are automatically named when you first create them but you can change the name of the database correlation to something more memorable by entering a new name into the **Name** text field. Once you have entered a name, you should set the correlation method that the database correlation will use to relate the time states from all of the input databases. The available choices, shown in Figure 4.413, are: padded index, stretched index, time, and cycle.

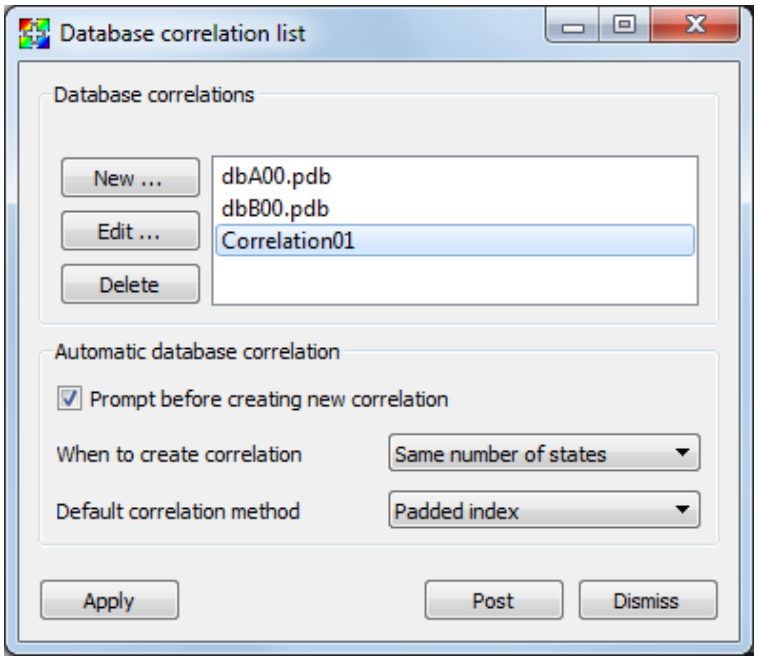


Fig. 4.411: Database Correlation Window

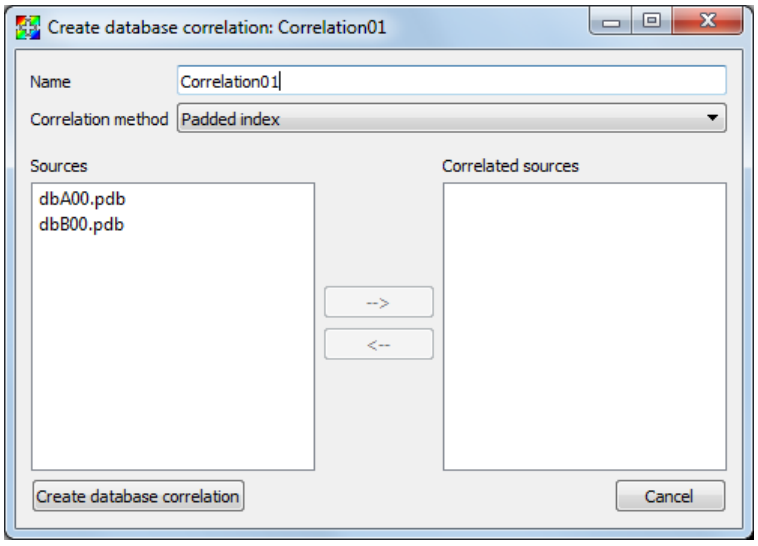


Fig. 4.412: Database Correlation Properties Window

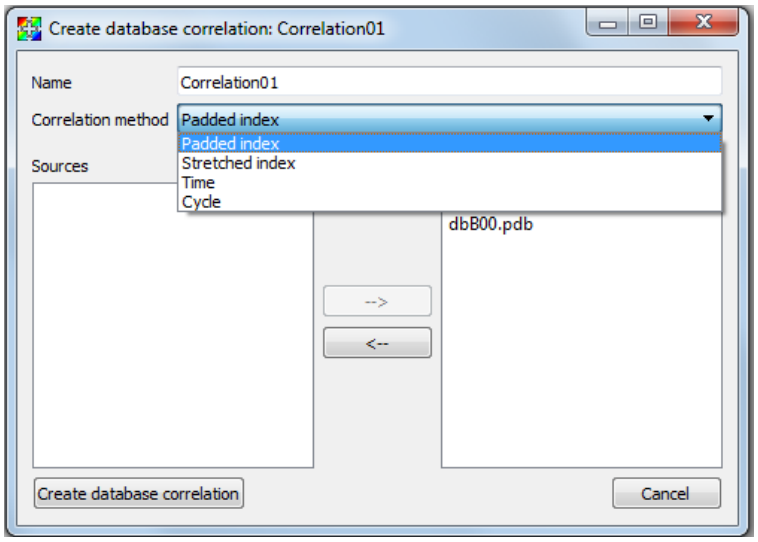


Fig. 4.413: Correlation methods

Once you have chosen a correlation method, it is time to choose the input databases for the correlation. The input databases, or sources as they are sometimes called in VisIt, are listed in the **Sources** list (see Figure 4.414). The **Sources** list only contains the databases that you have opened so far. If you do not see a database that you would like to have in the database correlation, you can either click the **Cancel** button to cancel creating the new database correlation or you can continue creating the database correlation and then add the other database to the correlation later after you have opened it. To add databases to the new database correlation, click on the them in the **Sources** list to highlight then and then click on the **Right arrow** button to move the highlighted databases into the database correlation's **Correlated sources** list. If you want to remove a database from the **Correlated sources** list, highlight the database in the **Correlated sources** list and then click the **Left arrow** button to move it back to the **Sources** list. Once you are satisfied with the new database correlation, click the **Create database correlation** button to create a new database correlation.

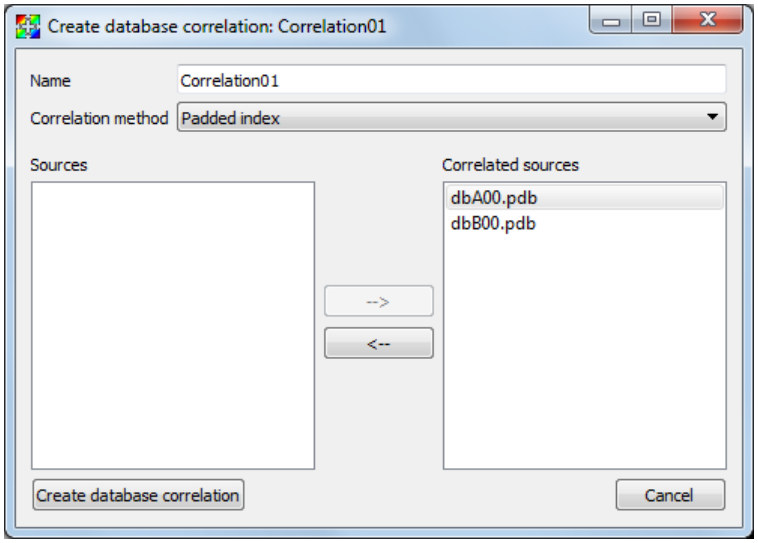


Fig. 4.414: Sources list and Correlated sources list

When you create a new database correlation, VisIt also creates a new time slider for the new database correlation. The database correlation's active time state is initially set to the first time state, which might not match the time state of

individual plots in the vis window. Once you change time states using the **Time slider**, the plots in the vis window will be updated using the correct time state with respect to the correlation's active time state. As always, if you want to update the time state for only one database, you can select a different time slider using the **Active time slider** combo box and then change time states using the **Time slider**. Any time state changes made to an individual database that is also an input database for a database correlation has no effect on the database correlations that involve the changed database. Time state changes for a database correlation can only happen if you have selected the database correlation as your active time slider.

Altering an existing database correlation

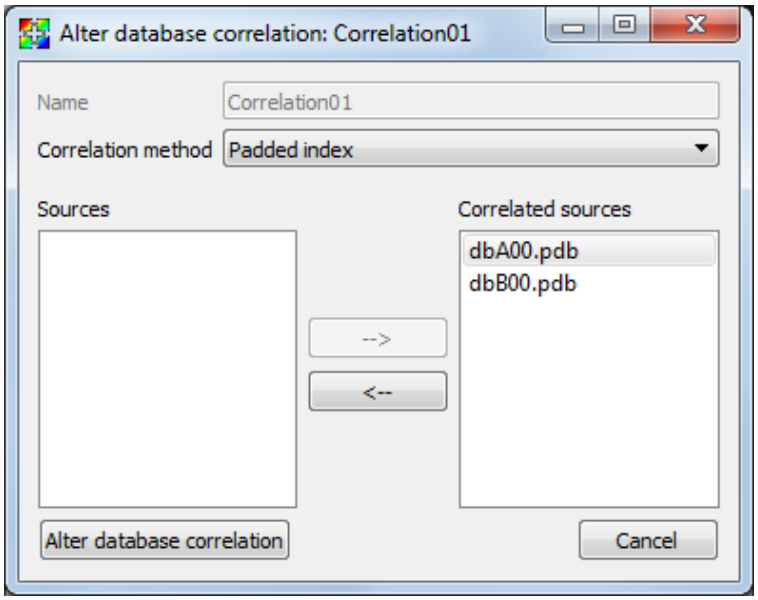


Fig. 4.415: Altering a database correlation

Once a database correlation has been created, you can alter it at any time by highlighting it in the **Correlation** list in the **Database Correlation Window** and clicking the **Edit** button to the left of the **Correlation** list. Clicking the **Edit** button opens the **Database Correlation Properties Window** and allows you to change the correlation method and the input databases. Once the desired changes are made, clicking the **Alter database correlation** button will make the specified database correlation use the new options and all plots in all vis windows that are subject to the changed database correlation will update to the new time states prescribed by the altered database correlation.

Using the **Database Correlation Properties Window** explicitly alters a database correlation. Reopening a file or refreshing the file list can implicitly alter a database correlation if after reopening the affected databases, there are different numbers of time states in the databases. When reopened databases that are input databases to database correlations have a new number of time states, VisIt recalculates the indices used to access the input databases via the time slider and updates any plots that were affected. In addition to the time state indices changing, the number of time states in the database correlation and its time slider can also change.

Deleting a database correlation

Database correlations are automatically deleted when you close a database that you are not using anymore provided that the closed database is not an input database to any database correlation except for that database's trivial database correlation. You can delete non-trivial database correlations that you have created by highlighting a database correlation in the **Correlation** list in the **Database Correlation Window** and clicking the **Delete** button to the left of the **Correlation** list. When you delete a database correlation, the new active time slider will be set to the active database's

time slider if the active database has more than one time state. Otherwise, the new active time slider, if any, will be set to the time slider for the first source that has more than one time state.

Automatic database correlation

VisIt can automatically create database correlations when they are needed if you enable certain global settings to control the creation of database correlations. By default, VisIt will prompt you when it wants to create a database correlation. VisIt can automatically create a database correlation when you add a plot of a multiple time-varying database to a vis window that already contains a plot from a different time-varying database. VisIt first looks for the most suitable existing database correlation and if the one it picks must be modified to accommodate a new input database or if an entirely new database correlation must be created, VisIt will prompt you using a **Correlation question** dialog (Figure 4.416). If you prevent VisIt from creating a database correlation or altering the most suitable correlation, you will no longer be prompted to create a database correlation for the list of databases listed in the **Correlation question** dialog.

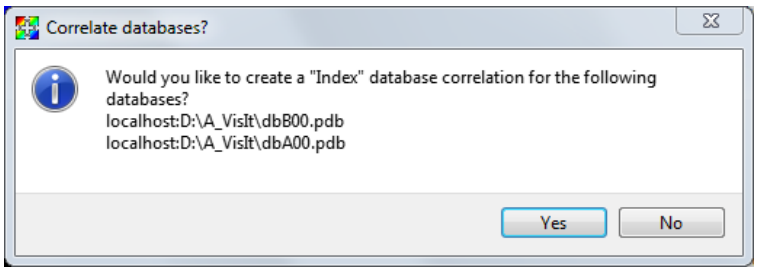


Fig. 4.416: Correlation question dialog

By default, VisIt will only attempt to create a database correlation for you if the new plot's database has the same number of time states as the existing plot. You can change when VisIt creates a database correlation for you by selecting a different option from the **When to create correlation** combo box in the **Database Correlation Window**. The available options are: **Always**, **Never**, and **Same number of states**. You can change the default correlation method by selecting a new option from the **Default correlation method** combo box. Finally, you can prevent VisIt from prompting you when it needs to create a database correlation if you turn off the **Prompt before creating new correlation** check box.

4.12.3 Database comparison

Comparing the results of multiple related simulation databases is one of VisIt's main uses. Users can plot multiple databases in the same window or adjacent windows, allowing comparison of plots visually. In addition to these *visual* modes of comparison, VisIt also supports more direct *numerical* comparison through the expression system. Database comparison allows users to plot direct differences between two databases or between different time states in the same database including even in the definition of time derivatives.

Numerical database comparisons use special expressions called *Cross-Mesh Field Evaluation (CMFE)* expressions, *pos_cmfe()* and *conn_cmfe()*, which are capable of mapping a field from one mesh, the *donor*, onto another mesh, the *target*. The name *conn_cmfe()* stands for *connectivity-based cross mesh field evaluation* (CMFE). It is a specialization of *position-based cmfe*, *pos_cmfe()*, for cases in which donor and target meshes be topologically congruent (e.g. size, connectivity, decomposition, etc. are identical). More information on CMFE expressions are found in the *Cross-Mesh Field Evaluation (CMFE)* section of the *Exprssions* chapter. There is also a helpful *wizard*, the *Data Level Comparison Wizard*, that simplifies the process of defining comparison expressions. Here, we walk through a few basic examples of using *CMFE* expressions and demonstrate how to use them in comparisons.

Plotting the difference between two databases

The typical case is where two slightly different databases time series have been generated from the same simulation code and the user wishes to work with the *difference* between the two databases *and* to have this difference update as the time slider is changed.

<mesh/ireg> - conn_cmfe(</usr/local/visit/data/dbB00.pdb[0]id:mesh/ireg>, <mesh>)

In the above expression, the first argument to *conn_cmfe()* serves as the *donor* field and the second argument is the *target* mesh. This expression is a simple difference operation of database A minus database B. Note the special *[0]id time specification syntax* before the colon but after the file system path in the first argument *conn_cmfe()*. The *i* means to interpret the number in brackets, *[0]* as a time state *index*. The *d* means to further interpret that number as an index *difference* from the *current* time slider index. This *syntax* is described in greater detail in the section describing *pos_cmfe()*.

The assumption made by this expression is that database A is the *active* database and the user wishes to map database B onto it to subtract it from database A's *mesh/ireg* variable. In this example, database B's *mesh/ireg* field is being mapped onto database A's mesh and their difference is then being taken. Figure 4.417 illustrates the database differencing operation.

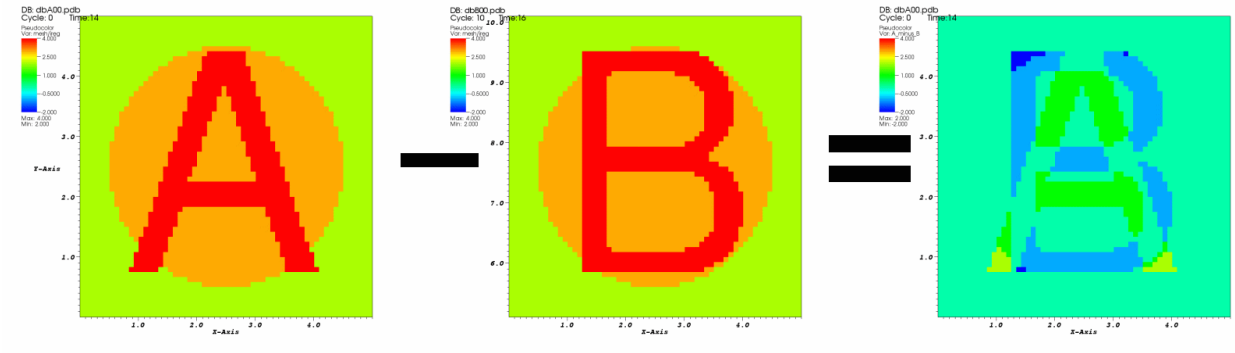


Fig. 4.417: Database B subtracted from database A

Plotting values relative to the first time state

Plotting a variable relative to its initial values can be important for understanding how the variable has changed over time. The conn_cmfe expression is also used to plot values from one time state relative to the values at the current time state. Consider the following expression:

<mesh/ireg> - conn_cmfe(</usr/local/visit/data/dbA00.pdb[0]i:mesh/ireg>, mesh)

The above expression subtracts the value of *mesh/ireg* at time state zero (in the *[0]i* without the *d* means to always map *absolute* time index zero from the *donor*) from the value of *mesh/ireg* at the current time As the time slider is changed, the values for the *active* database will change but the part of the expression using conn_cmfe, which in this case uses the first database time state, will not change. This allows users to create expressions that compare the current time state to a fixed time state.

Plotting time derivatives

Plotting time derivatives is much like plotting the difference between the current time state and a fixed time state except that instead of being fixed, the second time state being compared is free to move relative to the current time state. To

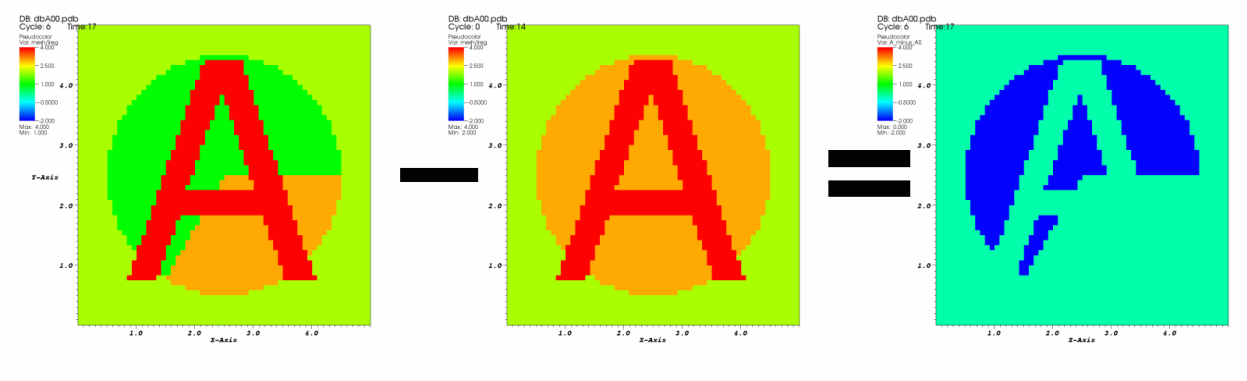


Fig. 4.418: Time state 6 minus time state 0

plot a simple time derivative such as the current time state minus the last time state, create an expression similar to the following expression:

<mesh/ireg> - conn_cmfe(</usr/local/visit/data/dbA00.pdb[-1]id:mesh/ireg>, mesh)

The important piece of the above expression is its use of *"[-1]id"* to specify a time state delta of -1, which means add -1 to the current time state to get the time state whose data will be used in the conn_cmfe calculation. You could provide different values for the time state in the *[]* operator. Substituting a value of *3*, for example, would make the conn_cmfe expression consider the data for 3 time states beyond the current time state. If you use a time state delta, which always uses the *"d"* suffix, the time state being considered is always relative to the current time state. This means that as you change time states for the active database using the time slider, the plots that use the conn_cmfe expression will update properly. Figure 4.419 shows an example plot of a time derivative.

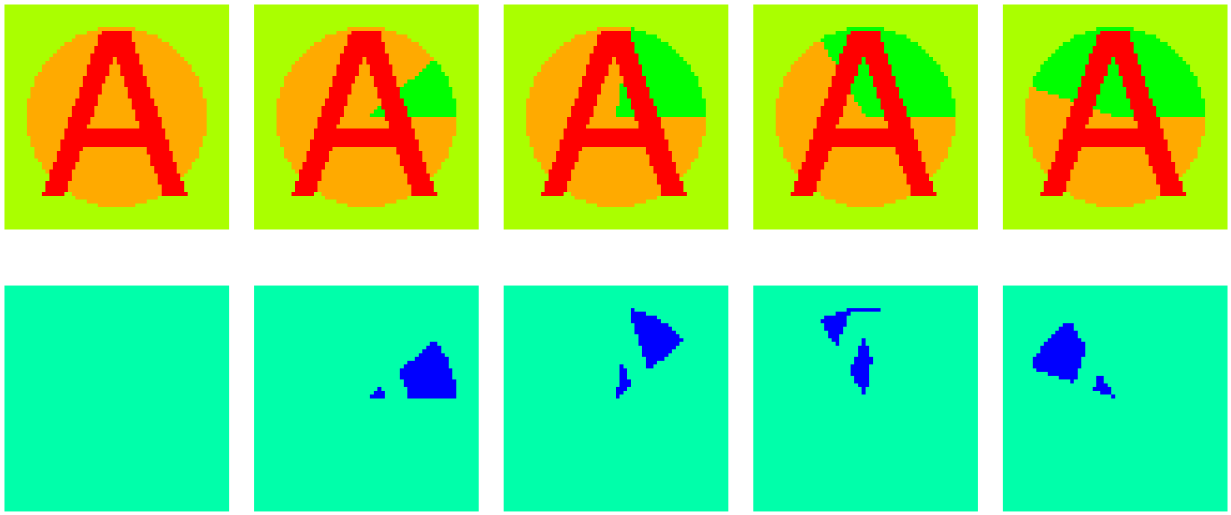


Fig. 4.419: Plot of a variable and its time derivative plot

4.12.4 Multiple window operations

This section focuses on some of the common techniques for exploring multiple databases when you have multiple visualization windows.

Reflection and Translation

When you visualize multiple related databases, they often occupy the same space in the visualization window since they may have been generated using the same computational mesh but with different physics. When this is the case, you can modify the location of the plots from one of the databases in two immediately obvious ways. First of all, if you simulated the same object and it does not make use of any symmetry then you could use the *Transform operator* to translate the coordinate system of one of the plots out of the way of the other plot so you can look at the two plots from the different databases side by side in the same visualization window. If your databases make use of symmetry (maybe you only simulated half of the problem) then you can apply the *Reflect operator* to one of the plots to show them side by side but reflected to show the entire problem. Each method has its merits.

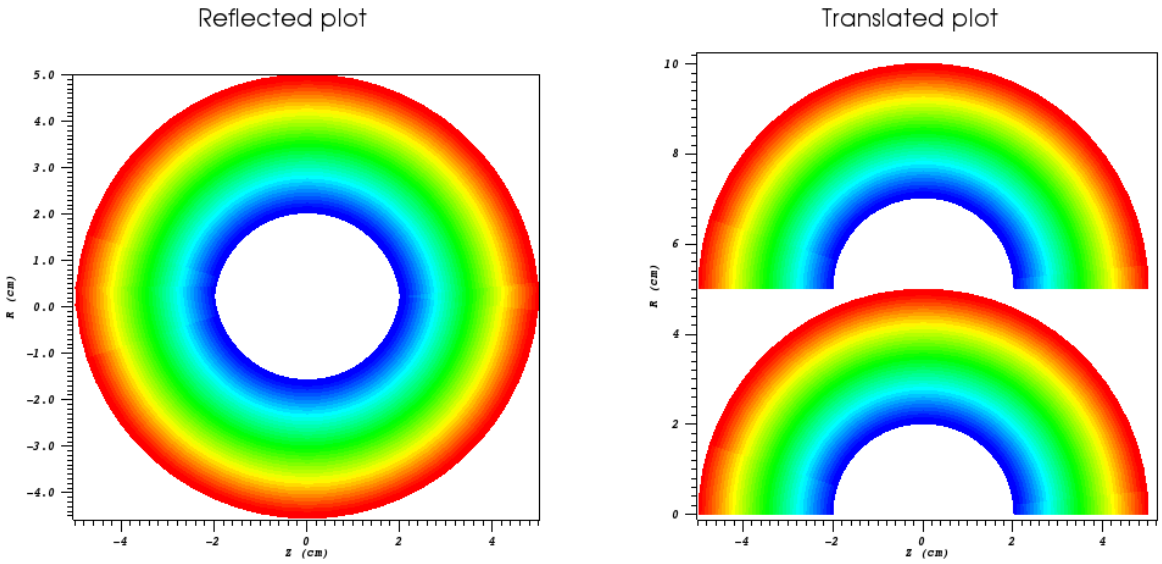


Fig. 4.420: Plots side by side using the Reflect or Transform operator

Copying Windows

If you visualize multiple databases and you want to create identical plots for each database but have them placed in different visualization windows then you can either have VisIt copy windows on first reference or you can clone an existing window and then replace the database used in the new window's plots with a different database.

If you have already created multiple visualization windows, perhaps as the result of a change to VisIt's layout, then you can make VisIt copy the attributes of the active window to another visualization window when you switch active windows by enabling **Clone window on first reference** in the **Preferences Window**. To open the **Preferences Window**, choose the **Preferences** option from the **Main Window's Options** menu. This form of window cloning copies the plots, lights, colors, etc from the active window to a pre-existing visualization window when you access it for the first time. If you have already accessed a visualization window but you would still like to copy plots, lights, colors, etc from another visualization window, you can make the destination visualization window be the active window and then copy everything from the source visualization window using the **Copy everything** menu option in the **Main Window's Windows** menu.

If you have no empty visualization window to contain plots for the another database, you can click the **Clone** option in the **Main Window's Windows** menu to create a new visualization window with the same plots and settings as the active window. Once the new window has been created, you could visualize a new database by choosing a new database in the **Active source** combo box and clicking the **Replace** button.

Locking Windows

When you visualize databases using multiple visualization windows, it is often convenient to keep the time state and view in sync between windows so you can concentrate on comparing plots instead of dealing with the intricacies of setting the view or time state for each visualization window. VisIt's visualization windows can be locked with respect to time, view, or interactive tools. To lock visualization windows, use the **Popup menu**, **Toolbar**, or the **Lock** options from the **Main Window's Windows** menu as shown in Figure 4.421.

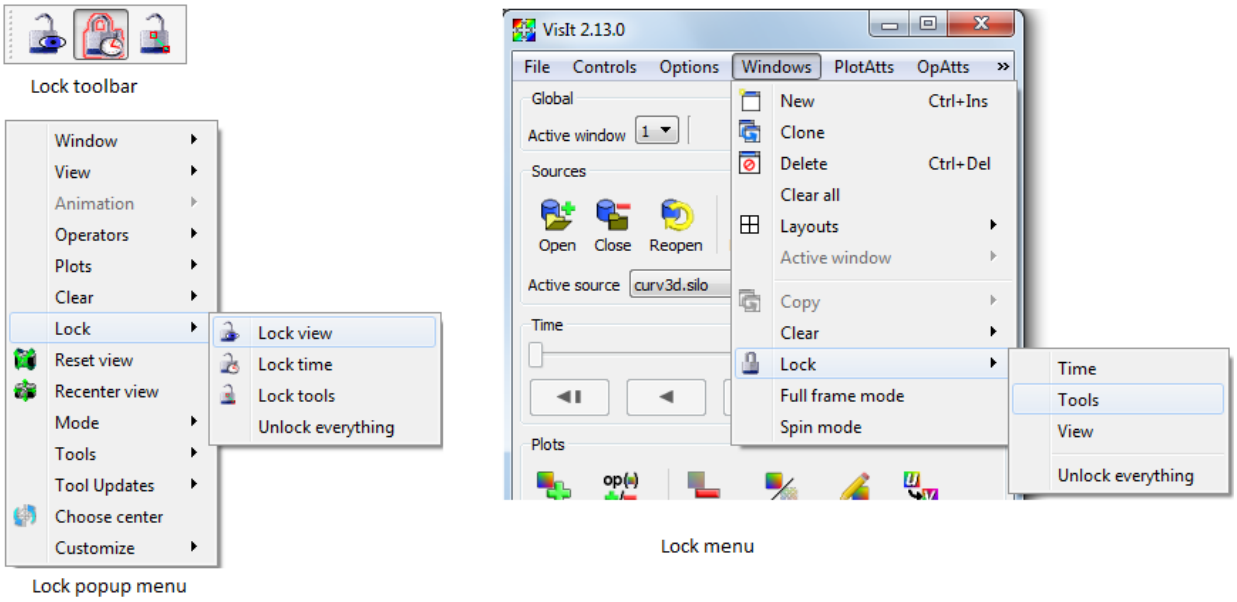


Fig. 4.421: Mechanisms for locking windows

Locking views

If you have created plots from related databases in multiple visualization windows, you can lock the views for the visualization windows together so that as you change the view in one of the visualization windows with a locked view, the other visualization windows with locked views also update to have the same view. There are four types of views in VisIt: curve, 2D, 3D, and AxisArray. If you have 2D plots in a visualization window, the visualization window is considered to be 2D. Locking that 2D visualization window's view will only update other visualization windows that are also 2D and vice-versa. The same is true for curve, 3D and AxisArray views.

Locking time

If you have created plots from related databases in multiple visualization windows, you can lock the visualization windows together in time so that as you change time in one visualization window, it updates in all other visualization windows that are locked in time.

Locking visualization windows together in time may cause VisIt to prompt you to create a new database correlation that involves all of the databases in the visualization windows that are locked in time. VisIt creates a database correlation because the visualization windows must use a common time slider to really be locked in time. If the visualization windows did not use a common time slider then changing time in one visualization window would not cause other visualization windows to update. Once VisIt creates a suitable database correlation for all windows, the active time slider is set to that database correlation in all visualization windows that are locked in time. If you alter a database correlation at this point, it will cause the time state in each locked visualization window to change. Since the same

database correlation is used in all locked visualization windows, changing the time state for the database correlation changes the time state in all of the locked windows. This frees you to examine time-varying database behavior without having to set the time state independently in each visualization window. See *Database correlations* for more information.

Locking tools

In addition to locking visualization windows together with respect to the view and time, you can also lock their tools. This capability can be useful when exploring data that often requires the use of an operator whose attributes can be set interactively using a tool since the same tool can be used to set the operator attributes for operators in more than one visualization window. See *Interactive Tools* for information on the different tools and how they are used.

Consider the following scenario: you have two related 3D databases and you want to examine the same slice plane for each database and you want each database to be plotted in a separate visualization window. You can set up separate visualization windows and slice the plots from each database independently but locking tools is easier and requires much less setup.

Start off by opening the first 3D database and create the desired plots from it. If you want to maintain a 3D view of the plots, you can clone the visualization window to get a new window with the same plots or you can apply a *Slice operator* to the plots. Apply a Slice operator but make sure the slice is *not* projected to 2D and also be sure that its **Interactive** check box is turned on. Turn on VisIt's plane tool and make sure that tools are locked. Clone the visualization window twice and for each of the new visualization windows, make sure that their Slice operator projects to 2D. There should now be four visualization windows if you opted to keep a 3D view of the data. In the last visualization window, replace the database with another related database that you want to compare to the first database.

Now that all of the setup steps are complete, you can save a session file so you can get back to this state when you run VisIt next time. Now, in the window that still has a slice in 3D, use the plane tool to reposition the slice. Both of the 2D visualization windows should also update so they use the new slice plane attributes calculated by the plane tool. The four visualization windows, arranged in a 2x2 window layout are shown in Figure 4.422.

4.13 Client Server

Scientific simulations are almost always run on a powerful supercomputer and accessed using desktop workstations. This means that the databases usually reside on remote computers. In the past, the practice was to copy the databases to a visualization server, a powerful computer with very fast computer graphics hardware. With ever increasing database sizes, it no longer makes sense to copy databases from the computer on which they were generated. Instead, it makes more sense to examine the data on the powerful supercomputer and use local graphics hardware to draw the visualization. VisIt can run in a client-server mode that allows this exact use case. The GUI and viewer run locally (client) while the database server and parallel compute engine run on the remote supercomputer (server). Running VisIt in client-server mode is almost as easy as running all components locally. This chapter explains the differences between running locally and remotely and describes how to run VisIt in client-server mode.

4.13.1 Client-Server Mode

When you run VisIt locally, you usually select files and create plots using the open database. Fortunately, the procedure for running VisIt in client-server mode is no different than it is for running in single-computer mode. You begin by launching the *File Open Window* and typing the name of the computer where the files are stored into the **Host** text field.

Once you have told VisIt which host to use when accessing files, VisIt launches the VisIt Component Launcher (VCL) on the remote computer. The VCL is a VisIt component that runs on remote computers and is responsible for launching other VisIt components such as the metadata server (mdserver) and compute engine. (Figure 4.423). Once you are

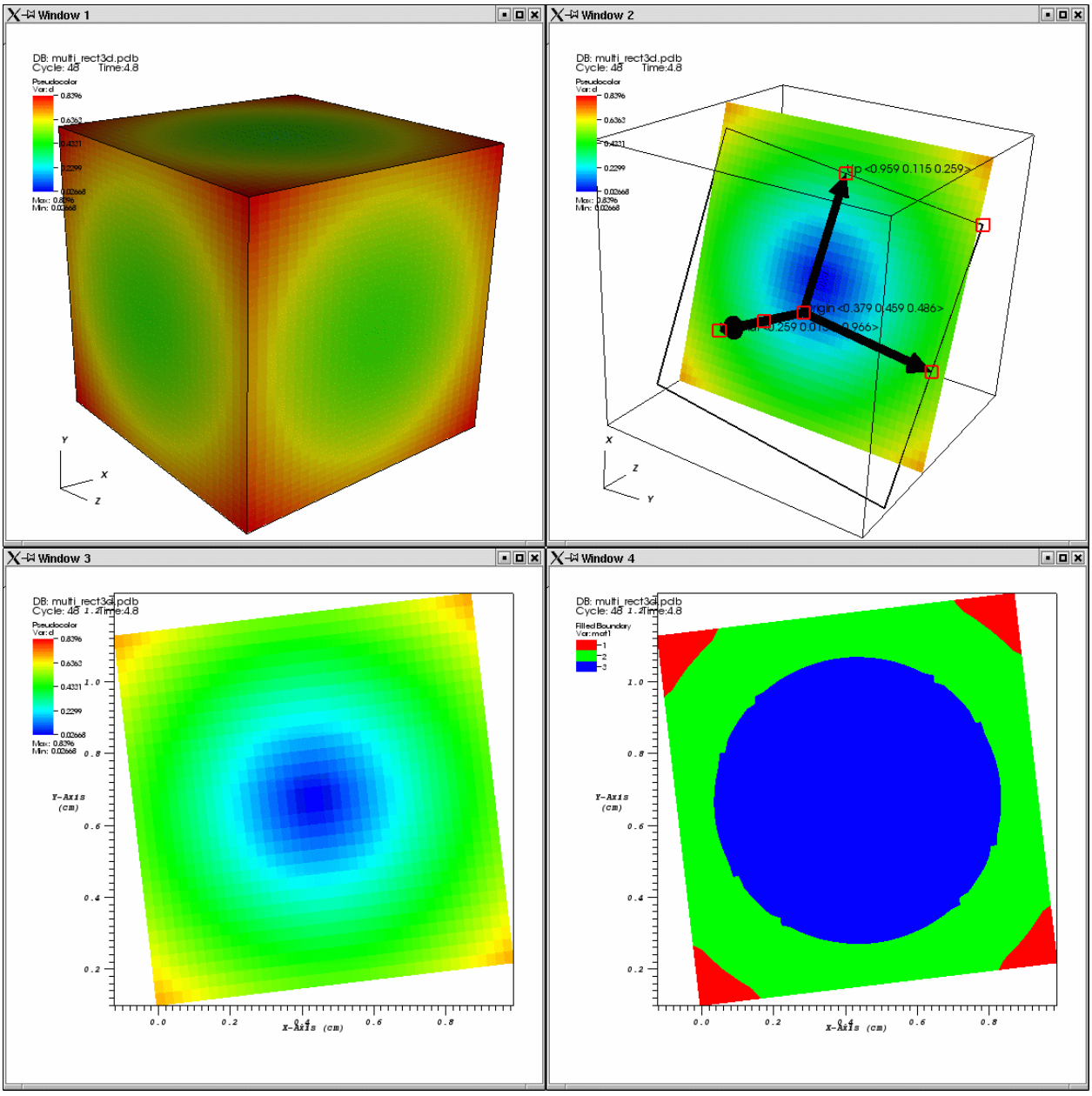


Fig. 4.422: Multiple visualization windows with locked tools

connected to the remote computer and VCL is running, you won't have to enter a password again for the remote computer because VCL stays active for the life of your VisIt session and it takes care of launching VisIt components on the remote computer.

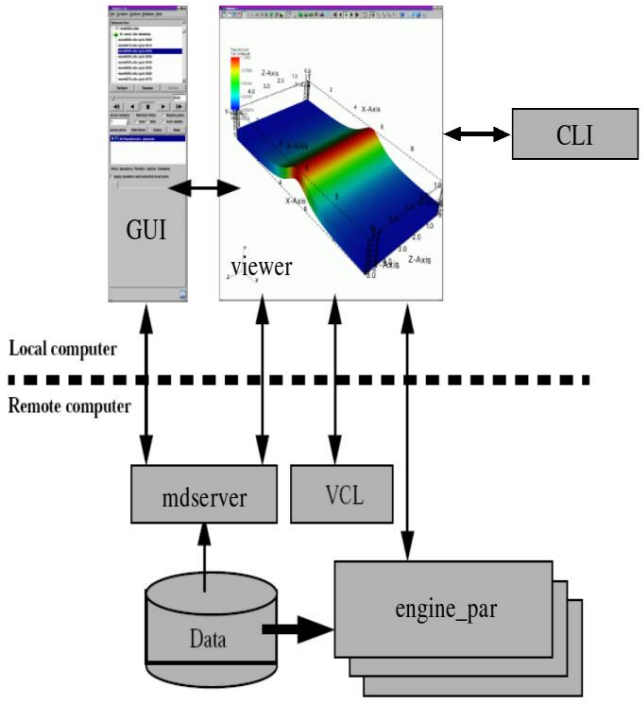


Fig. 4.423: VisIt's Architecture

If VCL was able to launch on the remote computer and if it was able to successfully launch the metadata server, the files for the remote computer will be listed in the **Files** pane of the **File Open Window**, just as if you were running locally. You then select the file or virtual database and click **OK**. Now that you have files from the remote computer at your disposal, you can create plots as usual.

Passwords

Sometimes when you try to access files on a remote computer, VisIt prompts you for a password by opening a **Password Window** (Figure 4.424). If you are prompted for a password, type your password into the window and click the **Ok** button. If the password window appears and you decide to abort the launch of the remote component, you can click the **Password Window's Cancel** button to stop the remote component from being launched.

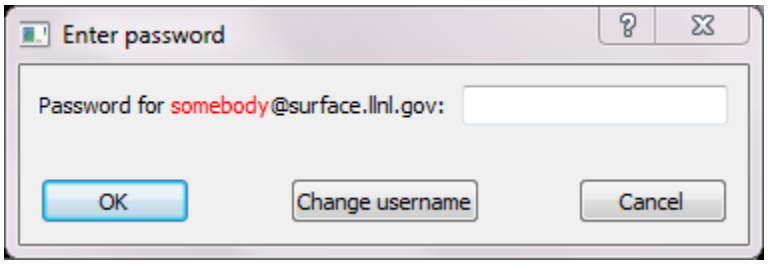


Fig. 4.424: Password Window

If your username for the remote machine is not listed correctly, you can click on the **Change username** button and a new window will pop up allowing you to enter the proper username for the remote system. (Figure 4.425). Enter the correct username in the text field provided and click **Confirm username**. Proceed with entering the password in the **Password Window**.

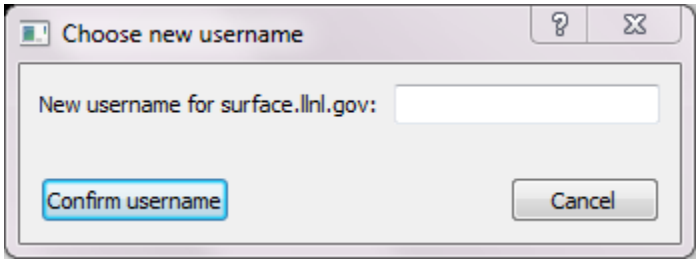


Fig. 4.425: Change Username Window

VisIt uses *ssh* for authentication and you can set up ssh so that passwords are not required. This is called *passwordless* ssh and once it is set up for a computer, VisIt will no longer need to prompt for a password.

Setting Up Password-less SSH

The following instructions describe how to set up **ssh** to allow password-less authentication among a collection of machines.

On the Local Machine

If you do not already have a **~/.ssh/id_rsa.pub** file, generate the key:

```
cd

ssh-keygen -t rsa
```

Accept default values by pressing <*Enter*>. This will generate two files, **~/.ssh/id_rsa** and **~/.ssh/id_rsa.pub**. The **~/.ssh/id_rsa.pub** file contains your public key in one very long line of text. This information needs to be concatenated to the **authorized_keys** file on the remote machine, so copy it to a temp file on the remote machine:

```
scp ~/.ssh/id_rsa.pub <your-user-name>@<the.remote.machine>:tmp
```

On the Remote Machine

If you do not already have a **~/.ssh** directory, create one with **r-w-x** permission for the owner only:

```
cd

mkdir .ssh

chmod 700 .ssh
```

If you do not already have a **~/.ssh/authorized_keys** file, create an empty one with permission for the owner only:

```
cd ~/.ssh

touch authorized_keys

chmod 600 authorized_keys
```

Concatenate the temporary file you copied into authorized_keys:

```
cd ~/.ssh

cat authorized_keys ~/tmp > authorized_keys

rm ~/tmp
```

Completing the Process

If you have more remote machines you want to access from the same local machine using *passwordless* ssh, repeat the process starting with copying the **~/.ssh/id_rsa.pub** file from the local machine to the remote, and continuing from there.

You can also repeat the above sections, reversing the local and remote machines, in order to allow *passwordless* ssh to the local machine from the remote machine.

Environment

It is important to have VisIt in your default search path instead of specifying the absolute path to VisIt when starting it. This is not as important when you run VisIt locally, but VisIt may not run properly in client-server mode if it is not in your default search path on remote machines. If you regularly run VisIt using the network configurations provided for LLNL computers then VisIt will have host profiles, which are sets of information that tell VisIt how to launch its components on a remote computer. The provided host profiles have special options that tell the remote computer where it can expect to find the installed version of VisIt so it is not required to be in your path. If you did not opt to install the provided network configurations or if you are at a site that requires other network configurations then you will probably not have host profiles by default and it will be necessary for you to add VisIt to your path on the remote computer. You can add VisIt to your default search path on Linux systems by editing the initialization file for your command line shell.

Launch Progress Window

When VisIt launches a compute engine or metadata server, it opens the **Launch Progress Window** when the component cannot be launched in under four seconds. An exception to this rule is that VisIt will always show the **Launch Progress Window** when launching a parallel compute engine or any compute engine on macOS. VisIt's components frequently launch fast enough that it is not necessary to show the **Launch Progress Window** but you will often see it if you launch compute engines using a batch system.

The **Launch Progress Window** indicates VisIt is waiting to hear back from the component being launched on the remote computer and gives you some indication that VisIt is still alive by animating a set of moving dots representing the connection from the local computer to the remote computer. The icon used for the remote computer will vary depending on whether a serial or parallel VisIt component is being launched. The **Launch Progress Window** for a parallel compute engine is shown in Figure 4.426. The window is visible until the remote compute engine connects back to the viewer or the connection is cancelled. If you get tired of waiting for a remote component to launch, you can cancel it by clicking the **Cancel** button. Once you cancel the launch of a remote component, you can return to your VisIt session. Note that if the remote compute is a parallel compute engine launched via a batch system, the engine

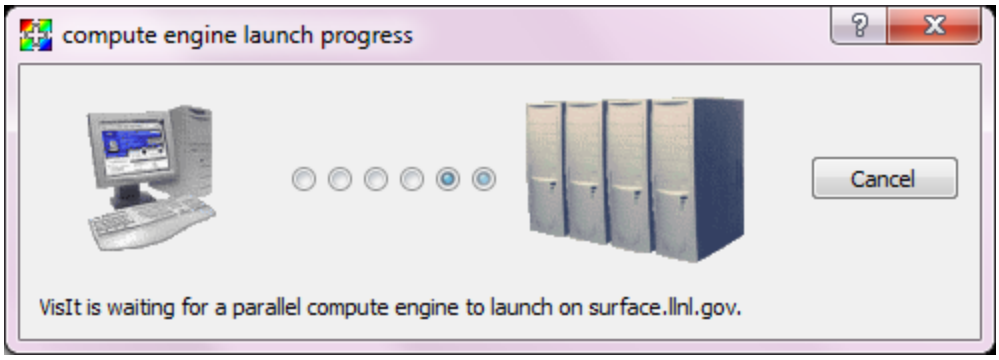


Fig. 4.426: Launch Progress Window

will still run when it is finally scheduled but it will immediately die since VisIt has stopped listening for it. On heavily saturated batch systems, it might be prudent for you to manually remove your compute engine job from the queue.

4.13.2 Host Profiles

When VisIt launches a component on a remote computer, it looks for something called a *host profile*. A host profile contains information that VisIt uses to launch components on a remote computer. Host profiles allow you to specify information like the remote username, the number of processors, the parallel launch method, etc. You can have multiple launch profiles for any given host, most often a serial profile and one or more parallel profiles.

Host profiles window

VisIt provides a **Host profiles** window, shown in Figure 4.427, that you can use to manage your host profiles. You can open the **Host profiles** window by choosing **Host profiles** from the **Options** dropdown menu. The **Host profiles** window is divided into two main areas. The left area contains a list of host profiles currently installed, as well as controls to create, delete, copy and export profiles. The right area contains two vertical tabs: **Remote Profiles**, used for installing profiles retrieved from a remote location; and **Machines**, which displays all attributes for the selected host profile. The **Remote Profiles** tab is useful for obtaining profiles that were not installed with VisIt. **Machines** has two sections contained in tabs displayed horizontally across the top: **Host Settings** and **Launch Profiles**. The **Host Settings** tab displays information for the selected machine, including the nickname, the full host name, aliases, the username, and connection information. The **Launch Profiles** tab displays a list of available profiles in the top section, and information for the selected launch profile in tabs on the bottom.

If the **Hosts** section in the left pane of the **Host profiles** window has no hosts listed, you have two options for installing already generated profiles. The first is to install one or more of the pre-defined host profiles shipped with VisIt while the second is to install one or more of the pre-defined host profiles from the VisIt repository. See *Installing pre-defined host profiles shipped with VisIt* and *Installing pre-defined host profiles from the VisIt repository*.

Click **Apply** when you are finished making changes in this window, and remember to save your settings (*How to Save Settings*) before exiting VisIt in order for your changes to be available in future sessions of VisIt.

Creating a new host profile

You click the **New Host** button to create a new host profile. The host profile will have a default name corresponding to the machine on which you are running VisIt. When you change the **Host nickname** the new name will be reflected in the Hosts list. See *Setting general options*, *Managing launch profiles* and *Setting parallel options* for more information on the available settings.

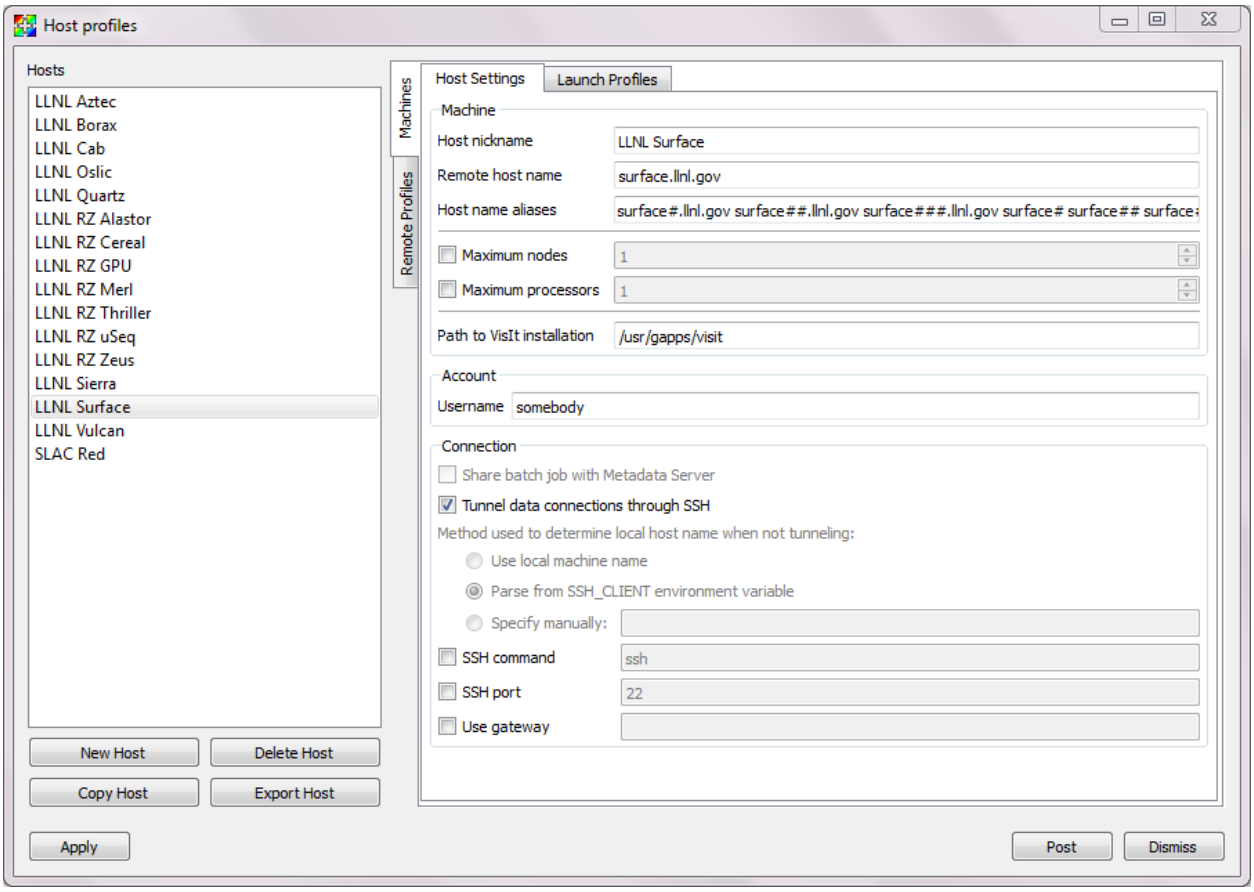


Fig. 4.427: Host profiles window

Deleting a host profile

If a host profile is no longer useful, you can click on it in the hosts list to select it and then click the **Delete Host** button to delete it.

Copying a host profile

To copy a host profile, select the desired source host from the **Hosts** list, then click the **Copy Host** button at the bottom of the Hosts list. A new host profile called `Copy of XXX` (where `XXX` is the name of the host you chose to copy) will be added to the **Hosts** list. Select this new host from the list and modify it's **Host Settings** and **Launch Profiles** appropriately. Once you change the **Host nickname** the new name will be reflected in the **Hosts** list.

Exporting a host profile

The **Export Host** button is useful for saving a host profile installed on your machine to share with someone else. Select the host profile you wish to export, and click the **Export Host** button. The exported host will be saved to your user VisIt directory (~/.visit/hosts on Linux). The name of the host profile file will start with `hosts_`, followed by the **Host nickname**, where letters are all converted to lower case and blanks are converted to underscores, followed by ".xml".

To share the host profile with someone else have them copy the host profile to their VisIt directory. It is recommended that you don't change the name of the file, but if you do, be aware that VisIt will only recognize it as a host profile if it starts with `hosts_` of `HOSTS_` and ends with `.xml` or `.XML`.

Setting general options

The **Host Settings** tab allows you to set general attributes for all launch profiles on the host.

Host nickname

Change the **Host nickname** to the name as you would like it to appear in the **Hosts** list in the left pane.

Remote host name

The **Remote host name** should be the fully qualified host name *(hostname.domain.net)*.

Host name aliases

Some clustered systems have one overall host name but also have names for the individual compute nodes that comprise the system. The compute nodes are often named by appending the node number to the host name. For example, if the clustered system is called `cluster`, you might be logged into node `cluster023`. When you launch a remote component, VisIt will not find any host profiles if the host name in the host profiles is: `cluster`.

To ensure that VisIt correctly matches a computer's node name to one of VisIt's host profiles, you should include host name aliases in the host profile for a clustered system. Host name aliases typically consist of the host name with different wildcard characters appended to it. Three wildcards are supported. The `?` wildcard character lets any one character replace it while the `*` wildcard character lets any character or group of characters replace it and the `#` wildcard character lets any numeric digit replace it. Appropriate host aliases for the previous example would be:

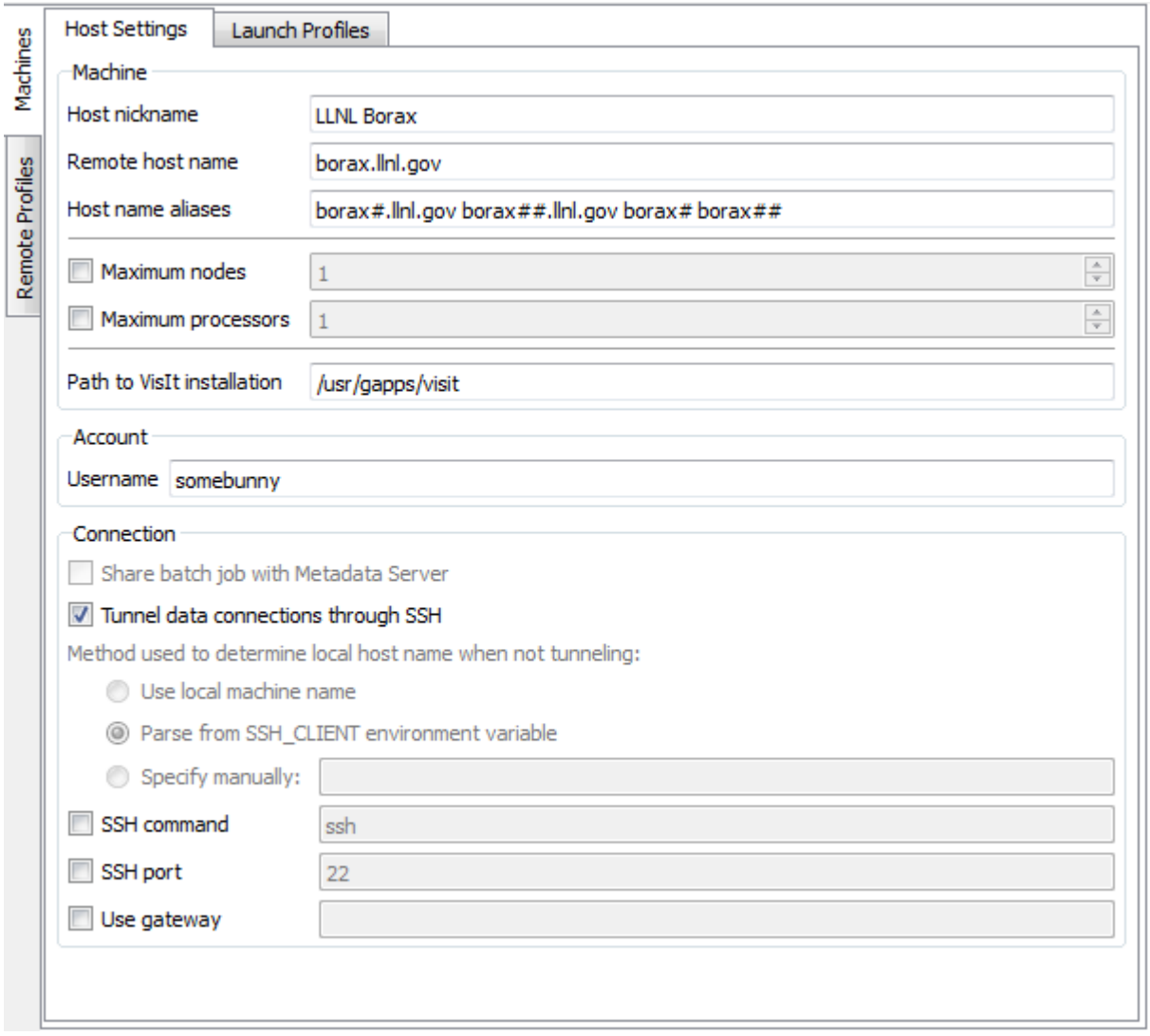


Fig. 4.428: Host Settings tab

`cluster#`, `cluster##` , `cluster###`, etc. If you need to enter host name aliases for the host profile, type them into the **Host name aliases** text field.

Maximum nodes/processors

If the host has a maximum number of nodes and/or processors that can be allocated, these can be specified by checking the **Maximum nodes** or **Maximum processors** checkboxes and entering a number in the corresponding text fields.

Path to VisIt installation

Most of the host profiles that are installed with VisIt specify the expected installation directory for VisIt so VisIt does not have to be in your path on remote computers. Enter the path to VisIt on the host in the **Path to VisIt installation** text field. It should be the full path up-to but not including the `bin` directory.

Account

The remote user name is the name of the account that you want to use when you access the remote computer. The remote user name does not have to match your local user name and it is often the case that your desktop user name will not match your remote user name. To change the remote user name, type a new user name into the **Username** text field.

Sharing a compute job

Some computers place restrictions on the number of interactive sessions that a single user can have on the computer. To allow VisIt to run on computer systems that enforce these kinds of restrictions, VisIt can optionally force the metadata server and parallel compute engine to share the same job in the batch system. If you want to make the database server and parallel compute engine share the same batch job, you can click the **Share batch job with Metadata Server** check box.

Determining the host name

There are many different network naming schemes and each major operating system type seems to have its own variant. While being largely compatible, the network naming schemes sometimes present problems when you attempt to use a computer that has one idea of what its name is with another computer that may use a somewhat different network naming scheme. Since VisIt users are encouraged to use client-server mode because it provides fast local graphics hardware without sacrificing computing power, VisIt must provide a way to reconcile the network naming schemes when 2 different computer types are used.

Workstations often have a host name that was arbitrarily set when the computer was installed and that host name has nothing to do with the computer's network name, which ultimately resolves to an IP address. This condition is common on computers running MS Windows though other operating systems can also exhibit this behavior. When VisIt launches a component on a remote computer, it passes information that includes the host name of the local computer so the remote component will know how to connect back to the local computer. If the local computer did not supply a valid network name then the remote component will not be able to connect back to the local computer and VisIt will wait for the connection until you click the **Cancel** button in the **Launch progress window**.

By default, VisIt tunnels data connections through SSH. If you don't want to tunnel, or SSH tunneling is not working you can turn it off by unchecking **Tunnel data connections through SSH** in the **Connection** section. If you want VisIt to rely on the the name obtained from the local computer, click on **Use local machine name**. If you choose the **Parse from SSH_CLIENT environment variable** option then VisIt will not pass a host name for the local computer but

will instead tell the remote computer to inspect the `SSH_CLIENT` environment variable to determine the IP address of the local computer that initiated the connection. This option usually works if you have a local computer that does not accurately report its host name. If you don't trust the output of any implicit scheme for getting the local computer's name, you can provide the name of the local computer by typing its name or IP address into the text field next to the **Specify manually** radio button.

SSH command

VisIt uses ssh for its connections to remote computers. On Windows, VisIt packages its own putty-based ssh program: `qtssh.exe`. Regardless of the system, you can override VisIt's SSH by clicking the **SSH command** checkbox and entering the full path to the ssh command you want to use in the text box.

SSH port

VisIt uses secure shell (ssh) to launch its components on remote computers. Secure shell often uses port 22 but if you are attempting to communicate with a computer that does not use port 22 for ssh then you can specify a port for ssh by clicking the **SSH port** check box and then typing a new port number into the adjacent text field.

In addition to relying on remote computers' ssh port, VisIt listens on its own ports (5600-5605) while launching components. If your desktop computer is running a firewall that blocks ports 5600-5605 then any remote components that you launch will be unable to connect back to the viewer running on your local computer. If you are not able to successfully launch VisIt components on remote computers, be sure that you make sure your firewall does not block VisIt's ports. Windows' default software firewall configurations block VisIt's ports so if you run those software firewall programs, you will have to unblock VisIt's ports if you want to run VisIt in client-server mode.

Gateway

If access to the compute nodes on your remote cluster is controlled by a gateway computer, then check the **Use gateway** checkbox, and enter the fully qualified name of the gateway computer in the text field. In order for VisIt to tunnel SSH connections through the gateway computer, passwordless-ssh needs to be set up from the gateway computer to the hose where you ultimately want to run VisIt. See *Setting Up Password-less SSH* for instructions on how to do this.

Managing launch profiles

The **Launch Profiles** tab (Figure 4.429) displays the launch profiles available for the selected host, generally a serial profile and one or more parallel profiles. There are controls for creating, deleting and copying launch profiles as well as tabs for setting the launch profile attributes.

Creating a new launch profile

Click the **New Profile** button. Give the profile an appropriate name by filling in the **Profile name** text box. The new name will be reflected in the profiles list as soon as it is entered. After filling out all the necessary attributes, click **Apply** in the lower left corner of the window in order to use the new profile immediately. The new profile to be available in future sessions of VisIt.

Deleting a launch profile

Select the profile to be deleted by clicking on its name in the list, then click the **Delete Profile** button. If you have made a mistake in deleting the profile, you must exit VisIt and restart. Saving your settings will make the change

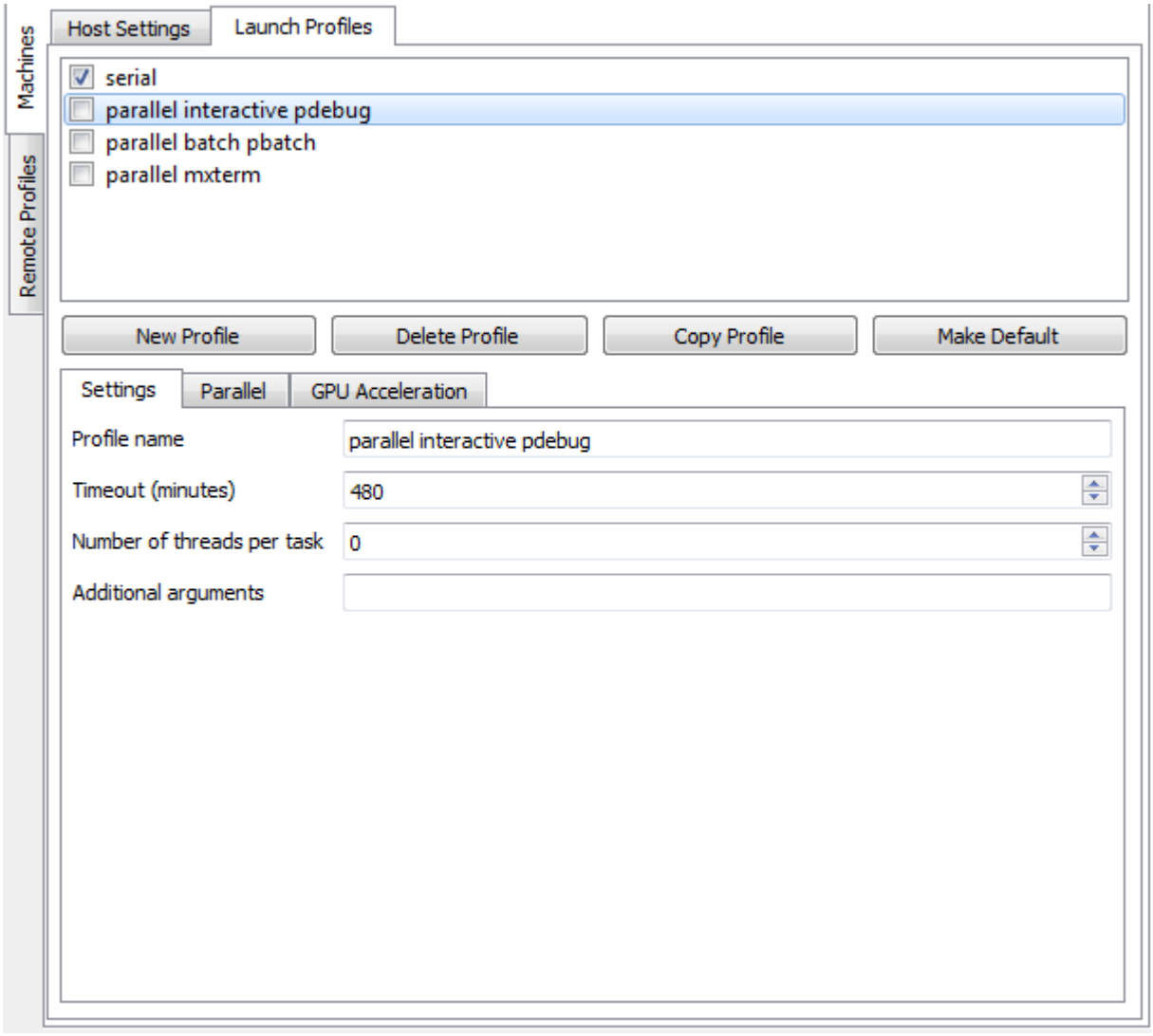


Fig. 4.429: Launch Profiles tab

permanent for future sessions.

Activating a launch profile

Only one launch profile can be active for any given host. When VisIt launches a remote component, it looks for the active launch profile for the host where the component is to be launched. The currently active launch profile is the one with the box to the left of the name checked in the list of launch profile names. To activate a different launch profile, select it from the list and click the **Apply** button. The VCL and the metadata server use the active launch profile but VisIt will prompt you for a launch profile to use before launching a compute engine if you have more than one launch profile or your only launch profile has parallel options set for the compute engine.

Setting the timeout

The compute engine and metadata server have a timeout mechanism that causes them to exit if no requests have been made of them for a certain period of time so they do not run indefinitely if their connection to VisIt's viewer is severed. You can set this period of time, or timeout, by typing in a new number of minutes into the **Timeout** text field. You can also increase or decrease the timeout by clicking on the up and down arrows next to the **Timeout** text field.

Setting the number of threads

If VisIt is running in threading mode, the number of threads per task can be set by typing in the desired number of threads in the **Number of threads per task** text field, or by utilizing the up and down arrows next to the text field.

Providing additional command line options

The **Launch Profiles** tab allows you to provide additional command line options to the compute engine and metadata server through the **Additional arguments** text field. When you provide additional command line options, you should type them, separated by spaces, into the **Additional arguments** text field. Command line options influence how the compute engine and metadata server are executed. For more information on VisIt's command line options, see *Startup Options*.

Setting parallel options

The chief purpose of host profiles is to make launching compute engines easier. This is even more the case when host profiles are used to launch parallel compute engines on large computers that often have complex scheduling programs that determine when parallel jobs can be executed. It is easy to forget how to use the scheduling programs on a large computer because each scheduling program requires different arguments. In order to make launching compute engines easy, VisIt hides the details of the scheduling program used to launch parallel compute engines. VisIt instead allows you to set some common parallel options and then figures out how to launch the parallel compute engine on the specified computer using the parallel options specified in the host profile. Furthermore, once you create a host profile that works for a computer, you rarely need to modify it.

To enable parallel options open the **Parallel** tab of the **Launch Profiles** tab, and click the **Launch parallel engine** checkbox.

Setting the parallel launch method

The parallel launch method option allows you to specify which launch program should be used to execute the parallel compute engine. This setting depends on the computer where you plan to run the compute engine and how the

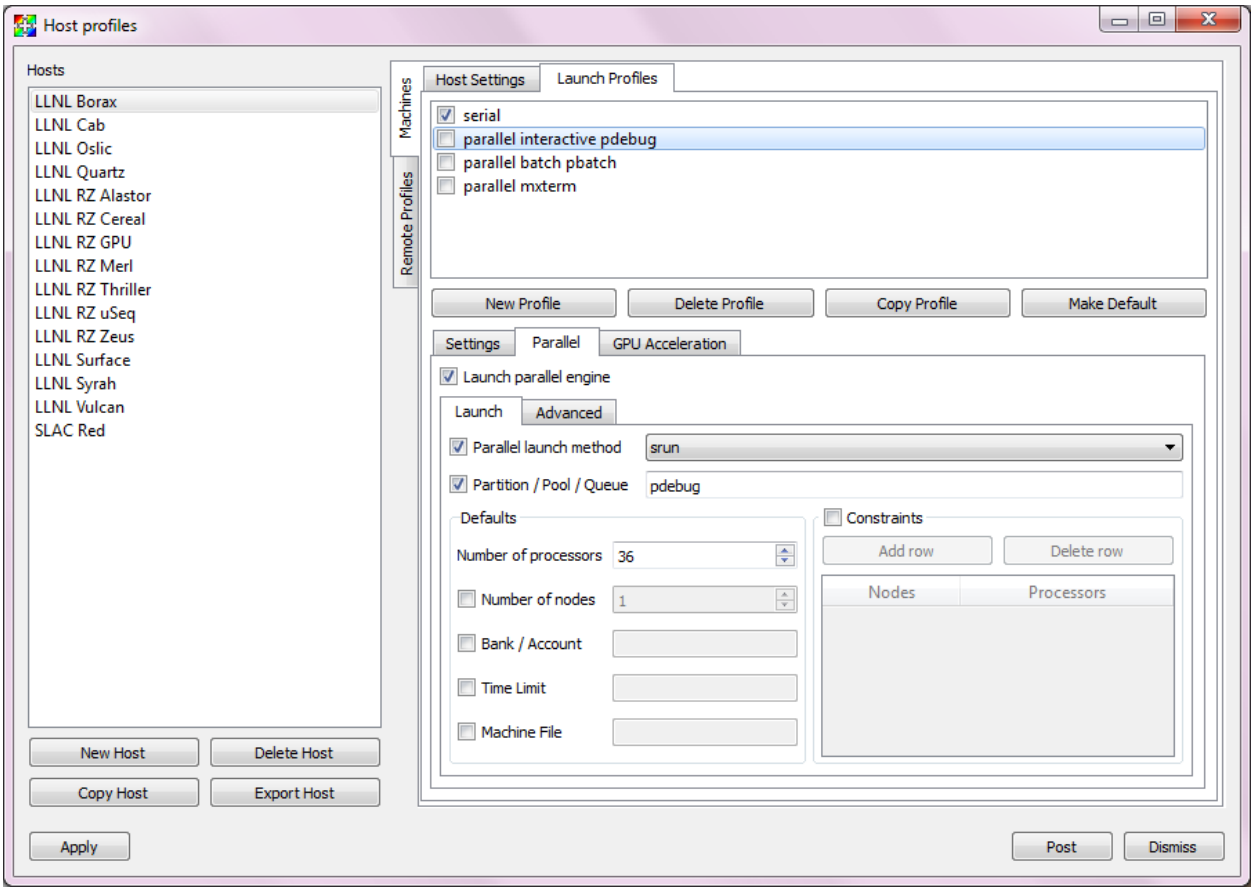


Fig. 4.430: Parallel options

computer is configured. Some computers have multiple launch programs depending on which part of the parallel machine you want to use. Figure 4.431 shows some common parallel-launch options that VisIt currently supports.

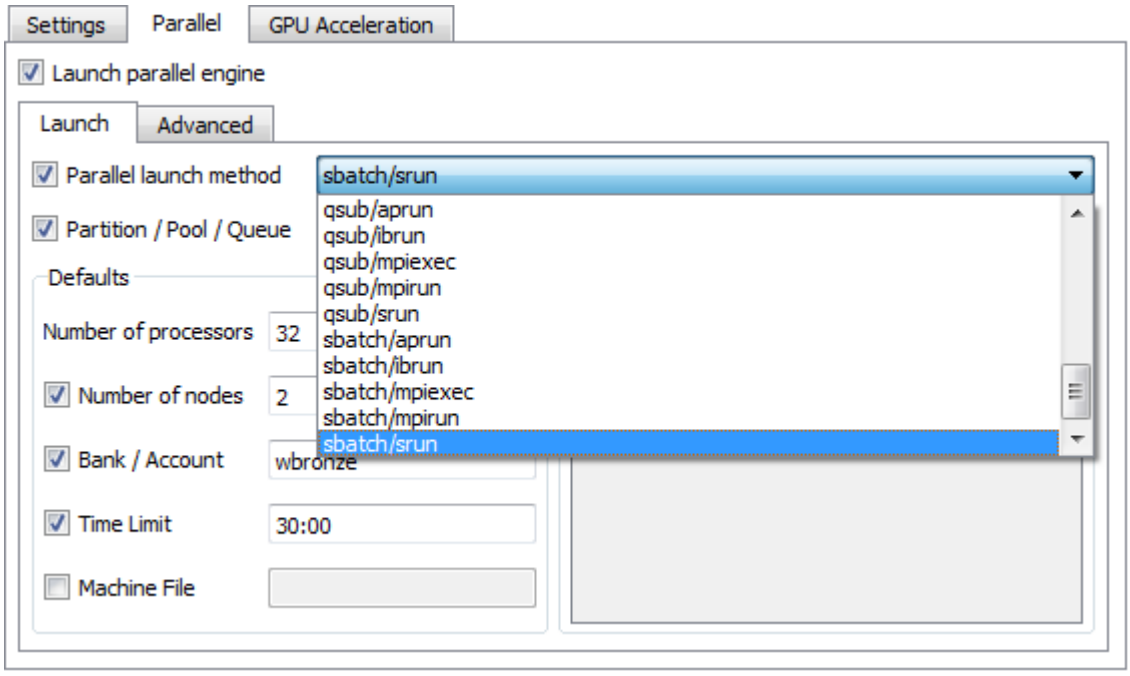


Fig. 4.431: Parallel launch method options

Setting the partition/pool/queue

Some parallel computers are divided into partitions so that batch processes might be executed on one part of the computer while interactive processes are executed on another part of the computer. You can use launch profiles to tell VisIt which partition to use when launching the compute engine on systems that have multiple partitions. To set the partition, check the **Partition/Pool/Queue** check box and type a partition name into the text field.

Setting the number of processors

You can set the number of processors by typing a new number of processors into the **Number of processors** text field in the **Defaults** section. When the number of processors is greater than 1, VisIt will attempt to run the parallel version of the compute engine. You can also click on the up and down arrows next to the text field to increase or decrease the number of processors. If VisIt finds a parallel launch profile, you will have the option of changing the number of processors before the compute engine is actually launched.

Setting the number of nodes

The number of nodes refers to the number of compute nodes that you want to reserve for your parallel job. Each compute node typically contains more than one processor (often 2, 4, 16) and the number of nodes required is usually the ceiling of the number of processors divided by the number of processors per node. It is only necessary to set the number of nodes if you want to use fewer processors than the number of processors that exist on a compute node. This option is not available on some computers as it is meant primarily for compute clusters. To set the number of nodes, check the **Number of nodes** check box and type a new number into the text field.

Setting the default bank

Some computers, if they are large enough, have scheduling systems that break up the number of processors into banks, which are usually reserved for particular projects. Users who contribute to a project take processors from their default bank of processors. By default, VisIt uses environment variables to get your default bank when submitting a parallel job to the batch system. If you want to override those settings, you can click the **Bank/Account** check box to turn it on and then type your desired bank into the text field next to the check box.

Setting the parallel time limit

The parallel time limit is the amount of time given to the scheduling program to tell it the maximum amount of time, usually in minutes, that your program will be allowed to run. The parallel time limit is one of the factors that determines when your compute engine will be run and smaller time limits often have a greater likelihood of running before jobs with large time limits. To specify a parallel time limit, click the **Time Limit** check box and enter a number of minutes or hours into the text field. If you want to specify minutes, be sure to append `m` to the number or append an `h` for hours. If you want to specify a timeout of 30 minutes, you would type: `30m`.

Specifying a machine file

When using VisIt with some versions of MPI on some clustered computers, it may be necessary to specify a *machine file*, which is a file containing a list of the compute nodes where the VisIt compute engine should be executed. If you want to specify a machine file when you execute VisIt in parallel on a cluster that requires a machine file, click on the **Machine File** check box and type the name of the machine file that you want to associate with your host profile into the text field.

Specifying constraints

Some machines constrain the processor-to-node ratio. In order to prevent accidentally requesting nodes/processors outside those constraints, they can be entered in table form by clicking the **Constraints** checkbox to enable the controls. Click **Add row** to add a new row to the table, and **Delete row** to remove a row from the table. For each row, enter number of nodes and appropriate associated number of processors in appropriate columns. When the launch engine dialog pops up, users won't be able to specify node-processor combinations outside of the constraints.

Advanced host profile options

The **Advanced** tab (see Figure 4.433) in the **Launch Profiles tab** lets you specify advanced networking options to ensure that the VisIt components running on the remote computer use resources correctly and can connect back to the viewer running on your local workstation.

Load balancing

Load balancing refers to how well tasks are distributed among computer processors. The goal is to make each computer processor have roughly the same amount of work so they all finish at the same time. VisIt's compute engine supports two forms of load balancing. The first form is static load balancing where the entire problem is distributed among processors and that distribution of work never changes. The second form of load balancing is dynamic load balancing. In dynamic load balancing, the work is redistributed as needed each time work is done. Idle processors independently ask for work until the entire task is complete. VisIt allows you to specify the form of load balancing that you want to use. You can choose to use static or dynamic load balancing by clicking the **Static** or **Dynamic** radio buttons. There is also a default setting that uses the most appropriate form of load balancing.

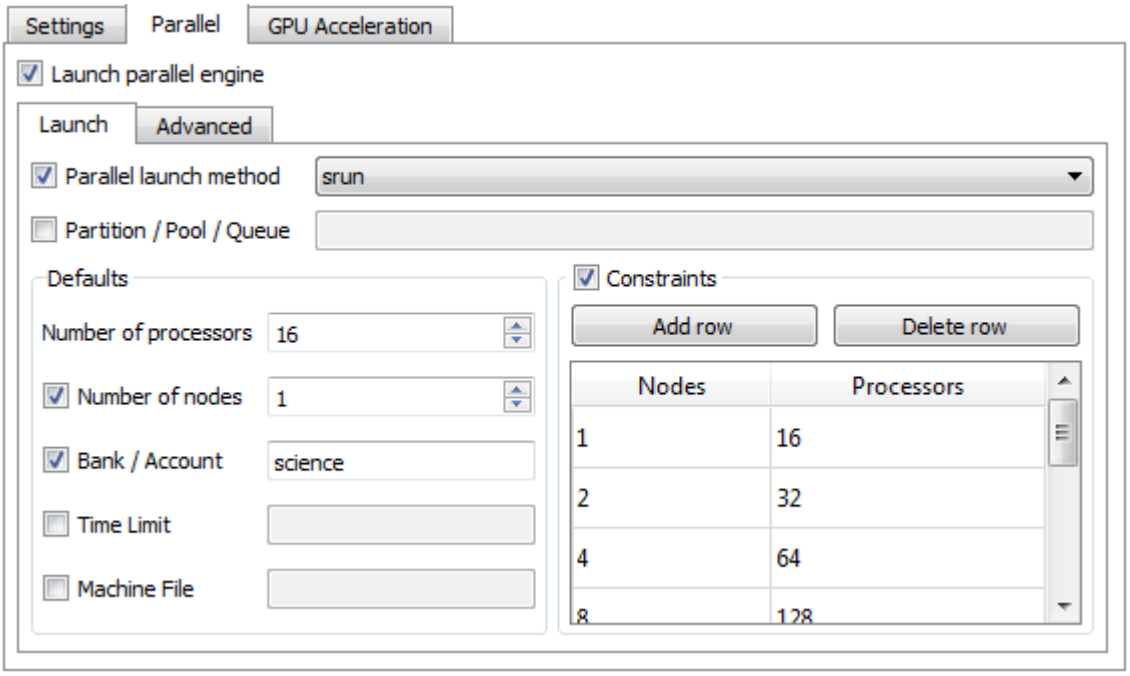


Fig. 4.432: Parallel launch constraints

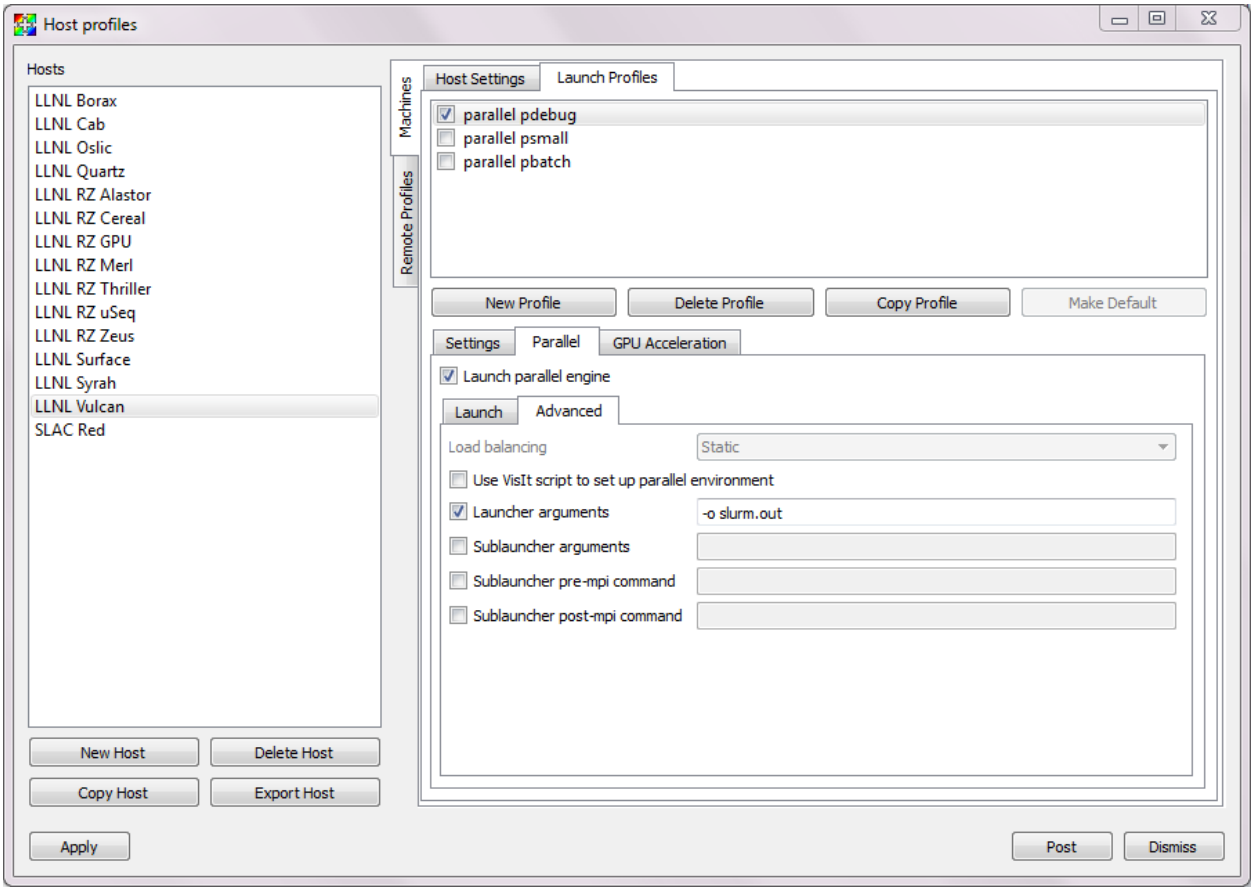


Fig. 4.433: Advanced options tab

Setting up the parallel environment

VisIt is usually executed by a script called: `visit`, which sets up the environment variables required for VisIt to execute. When the visit script is told to launch a parallel compute engine, it sets up the environment variables as it usually does and then invokes an appropriate parallel launch program that takes care of either spawning the VisIt parallel compute engine processes or scheduling them to run in a batch system. When VisIt is used with some versions of MPI on some clusters, the parallel launch program does not replicate the environment variables that the visit script set up, preventing the VisIt parallel compute engine from running. On clusters where the parallel launch program does not replicate the VisIt environment variables, VisIt provides an option to start each process of the VisIt compute engine under the visit script. This ensures that the environment variables that VisIt requires in order to run are indeed set up before the parallel compute engine processes are started. To enable this feature, click on the **Use VisIt script to set up parallel environment** check box.

Setting launcher arguments

In addition to choosing a launch program, you can also elect to give it additional command line options to influence how it launches your compute engine. To give additional command line options to the launch program, click the **Launcher arguments** check box and type command line options into the text field to the right of that check box.

Setting sublauncher options

To give additional command line options to the sublauncher program, click the **Sublauncher arguments**, **Sublauncher pre-mpi command** or **Sublauncher post-mpi command** check box and type options into the text field to the right of that check box.

Installing pre-defined host profiles shipped with VisIt

The **Setup Host Profiles And Configuration** window is used to install pre-defined host profiles that are shipped with VisIt. It can be accessed from the **Options** dropdown. It will list all the pre-defined host profiles shipped with the installation, listed according to location. From the list, you can choose one or more locations and all the host profiles for the selected locations will be installed. However, you will need to exit and restart VisIt for them to become available for use. With this window, you can also specify a default configuration for VisIt to use. Don't forget to click **Install** before dismissing the window. (Figure 4.434)

Installing pre-defined host profiles from the VisIt repository

The **Remote Profiles** tab can be used to install pre-defined host profiles from the VisIt repository. The advantage to using the VisIt repository is that it may have additional host profiles defined after a particular release of VisIt was released. To do so, click on the **Remote Profiles** vertical tab in the middle of the **Host Profiles** window. The top section of the tab allows you to choose the remote location (currently, only VisIt's repository is available).

(Figure 4.435)

If you click the **Update** button, the list of host profiles available from the remote location will be displayed. (Figure 4.436)

Scroll through the list, clicking on the arrow next to a location to view the profiles available for that location, then highlight a profile and click the **Import** button. (Figure 4.437) The selected host profile will now show up in the hosts list in the left pane.

It is important to save your settings before exiting VisIt in order to save the newly imported host profiles for future sessions.

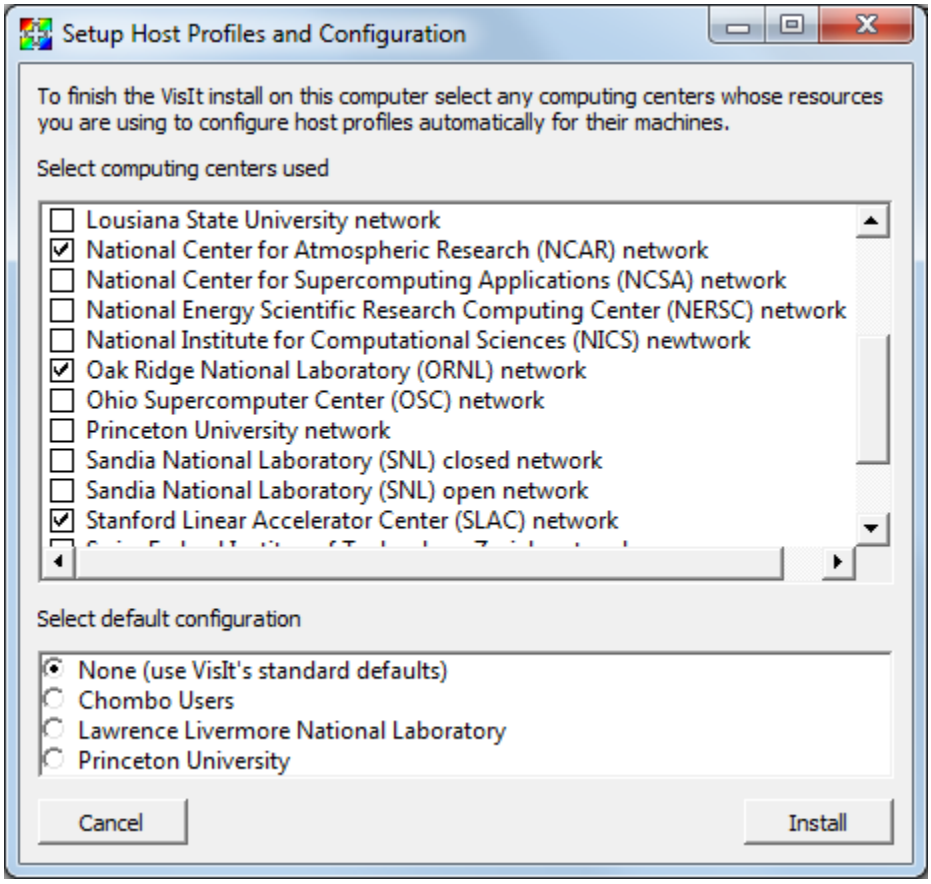


Fig. 4.434: The Host Profile Configuration Window

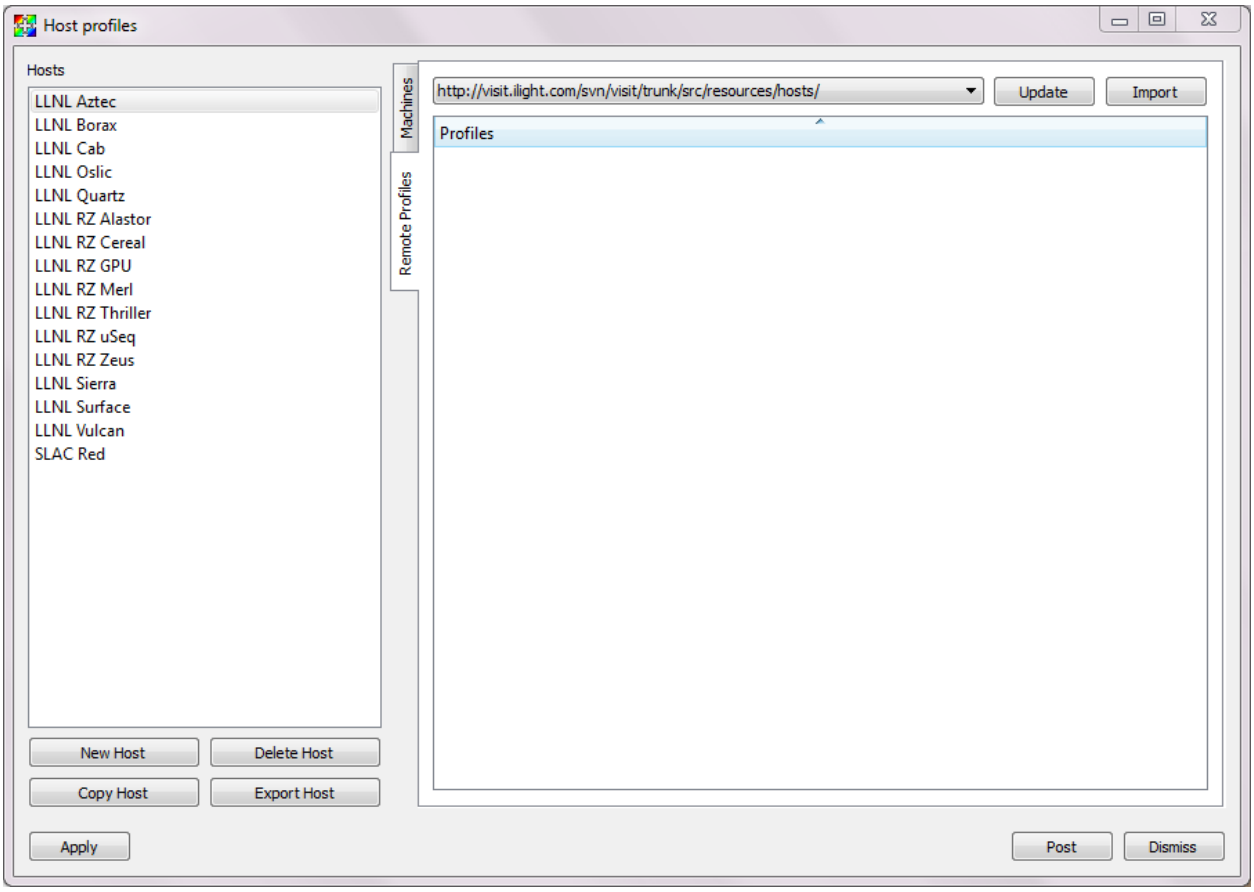


Fig. 4.435: Remote Profiles tab

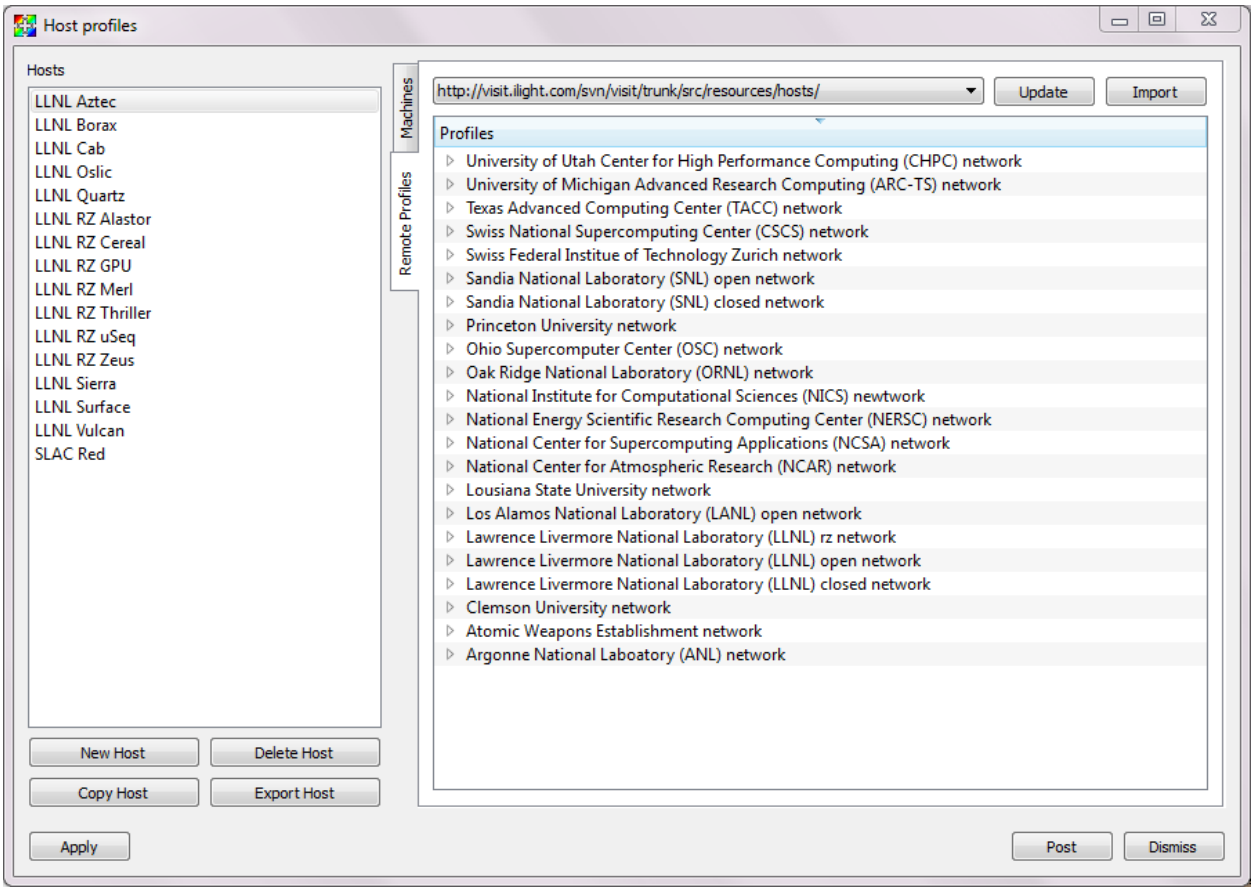


Fig. 4.436: Remote Profiles tab with updated content

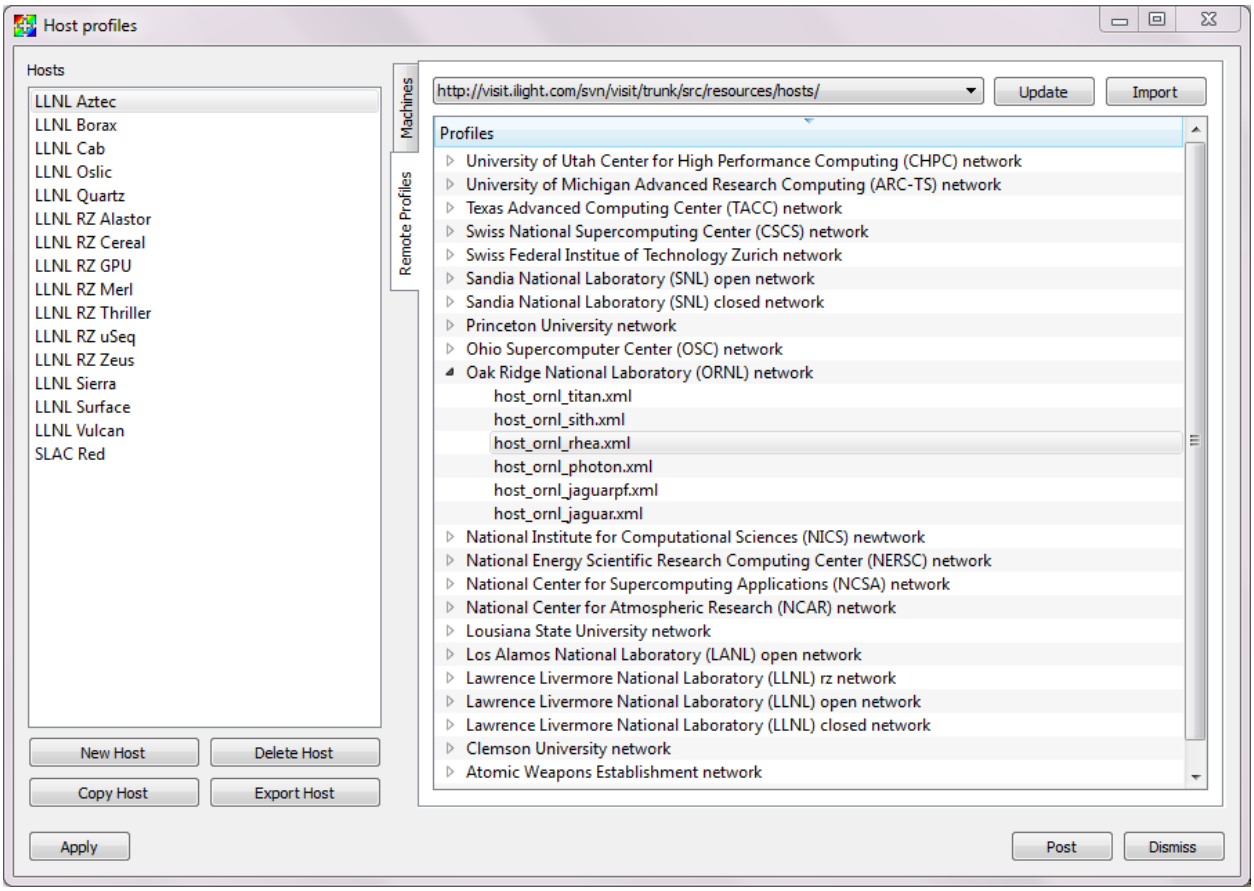


Fig. 4.437: Remote Profiles tab with host selected for import

Engine launch options window

The engine launch options window, shown in (Figure 4.438), is used to pick a launch profile to use when there are multiple launch profiles for a host or if there are any parallel launch profiles. When there is a single serial host profile or no host profiles, the window is not activated and VisIt launches a serial compute engine. The window's primary purpose is to select a launch profile and set some parallel options such as the number of processors. This window is provided as a convenience so host profiles do not have to be modified each time you want to launch a parallel engine to run with a different number of processors.

The engine launch options window has a list of launch profiles from which to choose. The active profile for the host is selected by default though another profile can be used instead. Once a launch profile is selected, the parallel options such as the number of processors/nodes, processor count, can be changed to fine-tune how the compute engine is launched. After making any changes, click the window's **OK** button to launch the compute engine. Clicking the **Cancel** button prevents the compute engine from being launched.

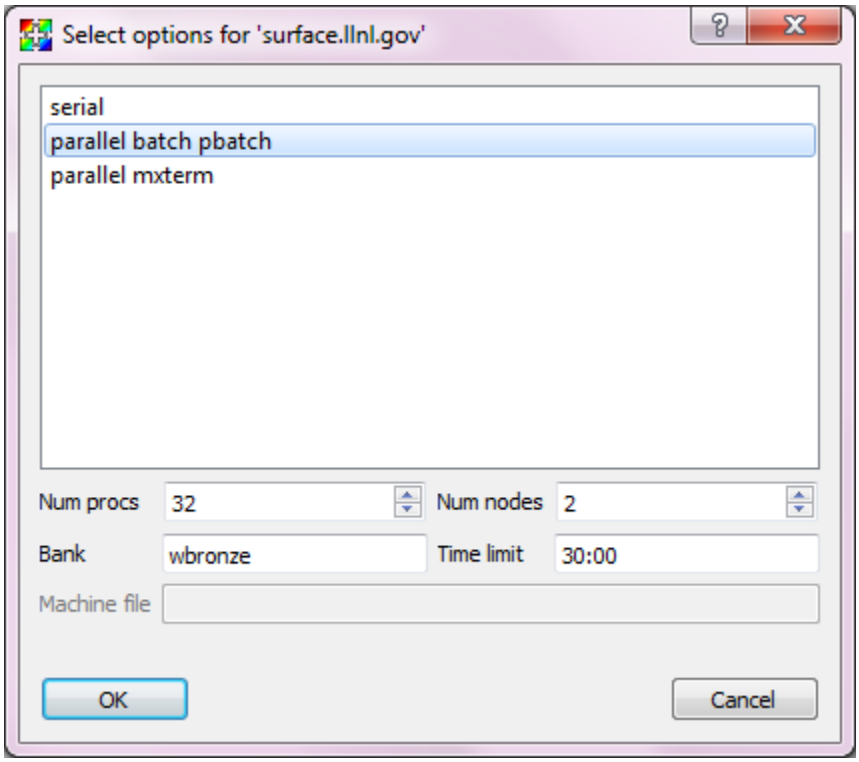


Fig. 4.438: Engine launch options window

Setting the number of processors

The number of processors determines how many processors are used by VisIt's compute engine. Generally, a higher number of processors yields higher performance but it depends on the host platform and the database being visualized. The **Num procs** text field initially contains the number of processors used in the active host profile but you can change it by typing a new number of processors. The number of processors can also be incremented or decremented by clicking the up/down buttons next to the text field.

Setting batch queue options

Many compute environments schedule parallel jobs in batch queues. The engine launch options window provides a few controls that are useful for batch queue systems. The first option is the number of nodes which determines the number of smaller portions of the computer that are allocated to a particular task. Typically the number of processors is evenly divisible by the number of nodes but the window allows you to specify the number of nodes such that not all processors within a node need be active. You can set the number of nodes, by typing a new number into the **Num nodes** text field or you can increment or decrement the number by clicking on the arrow buttons to the right of the text field. The second option is the bank which is a large collection of nodes from which nodes can be allocated. To change the bank, you can type a new bank name into the **Bank** text field. The final option that the window allows to be changed is the time limit. The time limit is an important piece of information to set because it can help to determine when the compute engine is scheduled to run. A smaller time limit can increase the likelihood that a task will be scheduled to run sooner than a longer running task. You can change the time limit by typing a new number of minutes into the **Time limit** text field.

Setting the machine file

Some compute environments use machine files, text files that contain the names of the nodes to use for executing a parallel job, when running a parallel job. If you are running VisIt in such an environment, the engine launch options window provides a text field called **Machine file**. The **Machine file** text field allows you to enter the name of a new machine file if you want to override which machine file is used for the selected host profile.

4.14 Compute Engines

VisIt can have many compute engines running at the same time. Much of the time the compute engines are those that are installed with VisIt but on occasion, simulation codes may be instrumented to act as VisIt compute engines capable of performing visualization operations on simulation data as it is created. When a simulation is used as a VisIt compute engine, VisIt can access data directly from the simulation without the need to translate data into another format or write it out to disk. When simulations are instrumented to become VisIt compute engines, they have all of the capabilities of a standard VisIt compute engine and more. Specifically, simulations can accept additional simulation-defined control commands that direct them to perform actions such as writing a restart file. Since simulations offer extra capabilities over a normal VisIt compute engine, VisIt provides different windows in order to manage them. To manage compute engines and check on progress, VisIt provides a **Compute Engine Window**. VisIt provides the **Simulation Window** to manage simulations, display their progress, and provide extra controls for the simulations.

4.14.1 Compute Engines Window

You can open the **Compute Engines Window**, shown in Figure 4.439, by selecting the **Compute engines** option from the **Main Window's File** menu. The main purpose of the **Compute Engines Window** is to display the progress of a compute engine as it completes a task. The window has two status bars. The top status bar indicates the progress of the overall task. The bottom status bar indicates that compute engine's progress through the current processing stage. The window also provides buttons for interrupting and closing compute engines, as well as an **Engine Information Area** that indicates how many processors the engine uses and its style of load balancing.

Picking a compute engine

The **Compute Engines Window** has the concept of an active compute engine. Only the active compute engine's progress is displayed in the status bars. The active compute engine is also the engine that is interrupted or closed. To pick a new active compute engine, choose a compute engine name from the **Engine** menu. The **Engine** menu contains the names of all compute engines that VisIt is running.

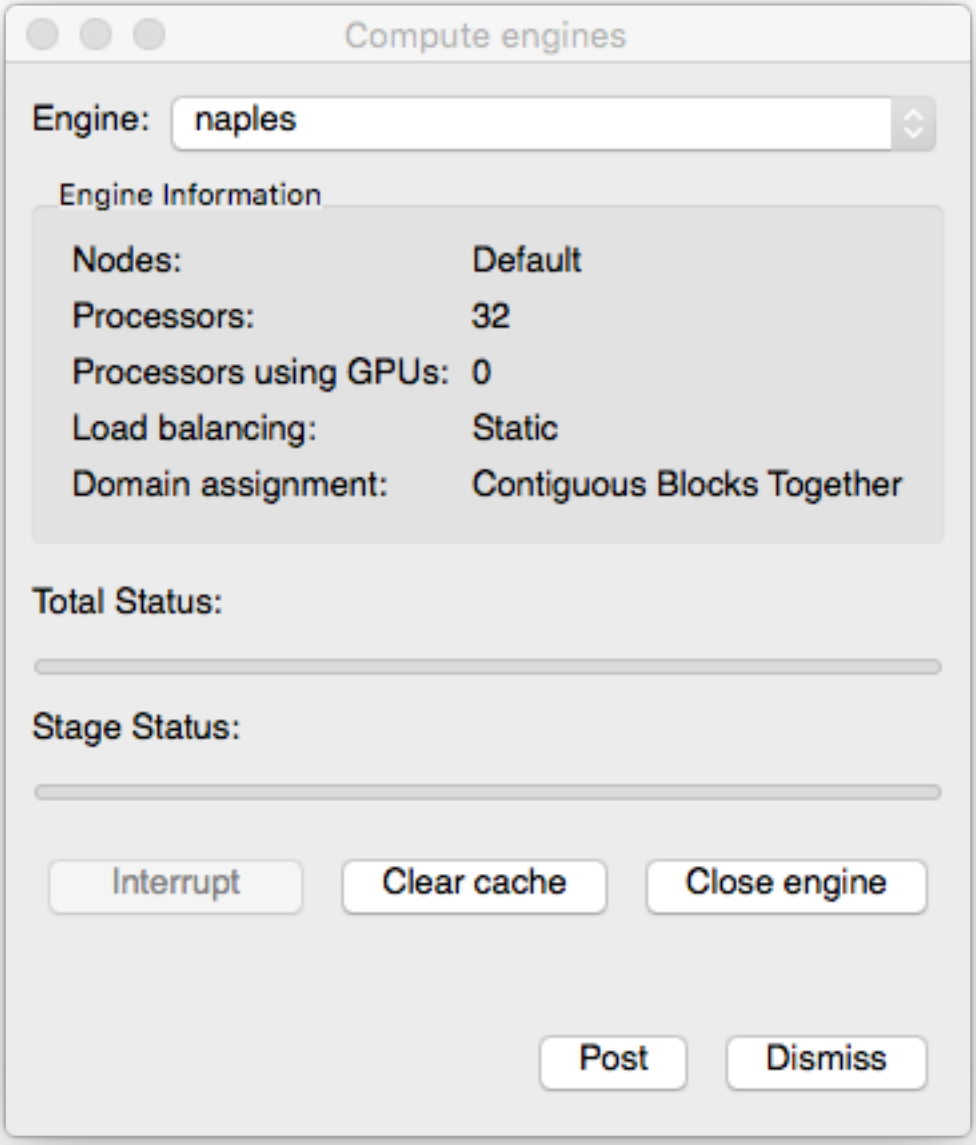


Fig. 4.439: Compute Engines Window

Interrupting a compute engine

Some operations in VisIt may take a long time to complete so most computations are broken down into stages. In the event that you do not want to wait for an operation to complete, or if you realize that you made a mistake, you can interrupt a compute engine. When you click the **Interrupt engine** button a signal is sent to the compute engine that tells it to stop its work. The compute engine handles the interrupt requests after it completes the current stage so there can be a small delay before the compute engine is interrupted. Any plots being generated when a compute engine is interrupted are sent into the error state and are listed in red in the **Plot list** until they are regenerated.

Closing a compute engine

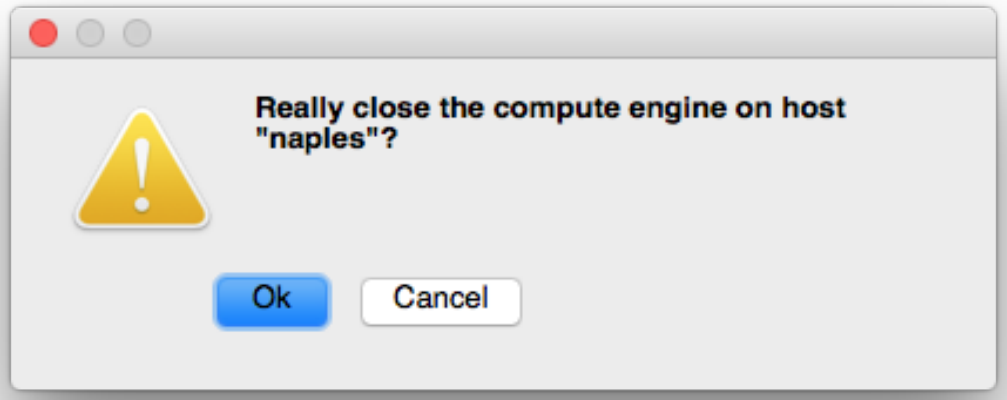


Fig. 4.440: Close compute engine confirmation dialog

You can close a compute engine when you no longer need it by clicking the **Close engine** button. The compute engine is closed only after you click **Yes** in a confirmation dialog window.

Clearing a compute engine's cache

As the compute engine processes data, it caches calculation results in case they are needed again. This includes meshes and variables that have been read from databases as well as the results from more complicated calculations involving expressions and operators. VisIt's compute engine periodically clears the cache of items that it no longer needs but if you want to explicitly clear the cache to free up more memory, you can click the **Clear cache** button in the **Compute Engine Window**.

4.14.2 Simulation Window

You can open the **Simulation Window**, shown in Figure 4.441, by selecting the **Simulations** option from the **Main Window's File** menu. The main purpose of the **Simulation Window** is to display the progress of a simulation that is acting as a VisIt compute engine as it completes its visualization tasks. The **Simulation Window** also provides

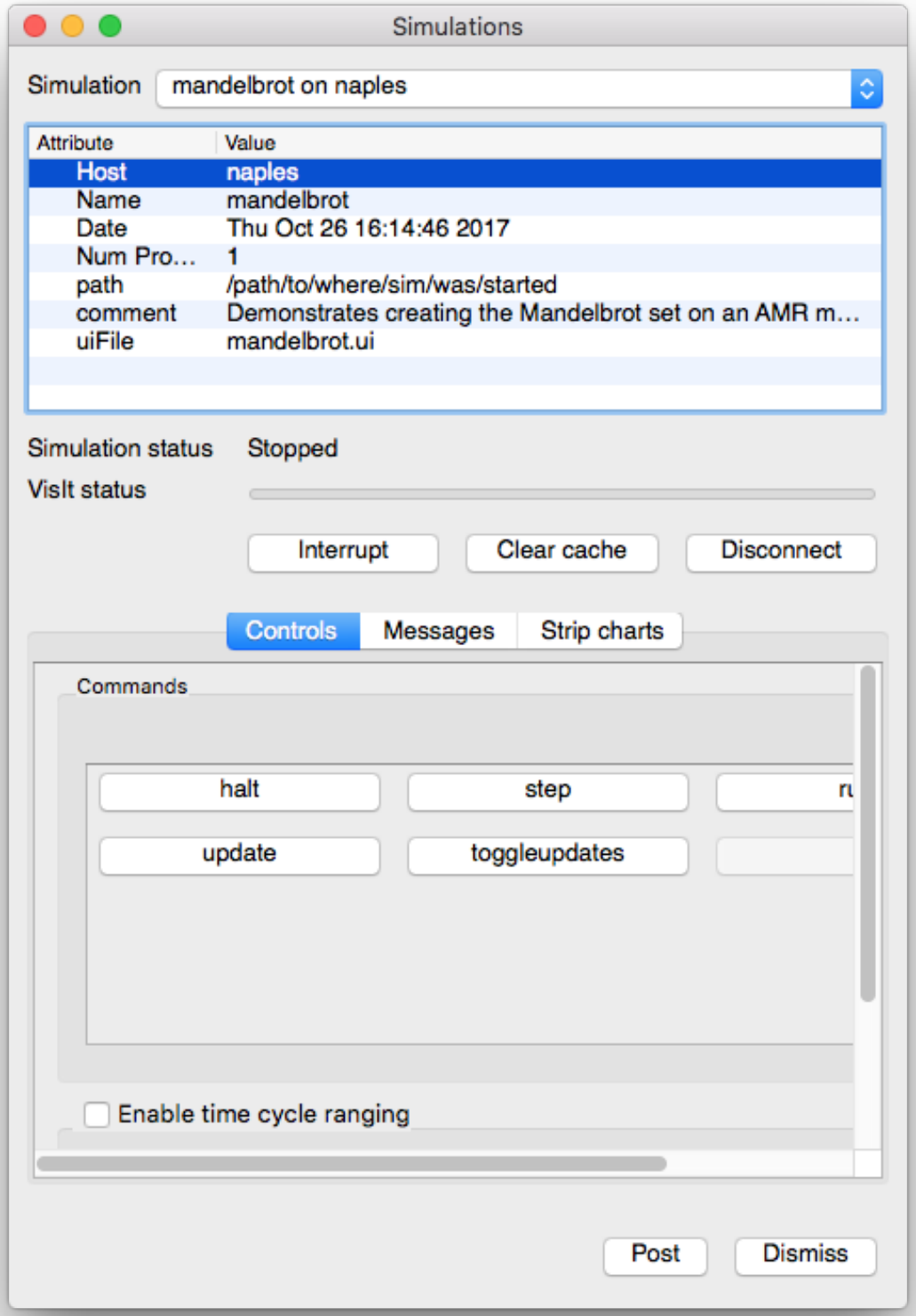


Fig. 4.441: Simulation Window

buttons that direct the simulation to perform simulation-defined commands such as saving out a restart dump. The list of commands depends on the functionality that the simulation exposes to VisIt when instrumented.

The **Simulation Window** is divided up into two main areas. The top of the window, called the **Simulation attributes** area, displays various attributes of the simulation such as its name, when it was started, the name of the computer where it is running, the number of processors, etc. Below the **Simulation attributes** area, you will find controls that are also present in the **Compute Engines Window** such as the **Interrupt** button and **Clear cache** button. The **Disconnect** button is specific to the **Simulation Window** and when you click it, VisIt will detach from the running simulation, allowing it to continue its calculation. You can reconnect to the simulation later to check on the its progress or create more visualizations.

Below the **Simulation attributes** area, you can access **Commands**, **Messages**, and **Strip Charts**. The **Commands** tab displays buttons for simulation-defined commands. When a simulation is instrumented to act as a VisIt compute engine, it publishes a list of commands that it will accept when connected to VisIt. This allows the simulation to provide hooks that allow the user to tell the simulation to execute certain commands like writing a restart file. Depending on the complexity of the commands exposed, VisIt could ultimately be used to steer the simulation as well as visualize its results. The **Messages** tab displays messages from the simulation. The **Strip Charts** tab shows traces of specific quantities published from the simulation to VisIt.

4.15 Command Window

In this section, we describe the **Command Window** which provides a convenient interface from the GUI to VisIt's *Python command-line interface*.

4.15.1 VisIt's Python Command Line via the Command Window

It is possible for VisIt's GUI and Python Interface to share the same viewer component at runtime. When you invoke visit at the command line, VisIt's GUI is launched. When you invoke `visit -cli` at the command line, VisIt's CLI (Python interface) is launched. If you want to use both components simultaneously then you can use VisIt's **Command Window** . The **Command Window** can be opened by clicking on the **Command** menu option from the **Controls** menu. The **Command Window** consists of a set of eight tabs in which you can type Python scripts. When you type a Python script into one of the tabs, you can then click the tab's **Execute** button to make VisIt try and interpret your Python code. If VisIt detects that it has no Python interpreting service available, it will launch the CLI (connected to the same viewer component) and then tell the CLI to execute your Python code. Note that the **Command Window** is just for editing Python scripts. Any output that results from the Python code's execution will be displayed in the CLI program window (see Figure 4.442).

Saving the Command Window's Python scripts

The **Command Window** is meant to be a sandbox for experimenting with small Python scripts that help you visualize your data. You will often hit upon small scripts that can be used over and over. The scripts in each of the eight tabs in the **Command Window** can be saved for future VisIt sessions if you save your settings. Once you save your settings, any Python scripts that are present in the **Command Window** are preserved for future use.

Clearing a Python script from a tab

If a Python script in one of the **Command Window's** tabs is no longer useful then you can click that tab's **Clear** button to clear out the contents of the tab so you can begin creating a new script in that tab. If you want VisIt to permanently delete the script from the tab then you must save your settings after clicking the **Clear** 3button.

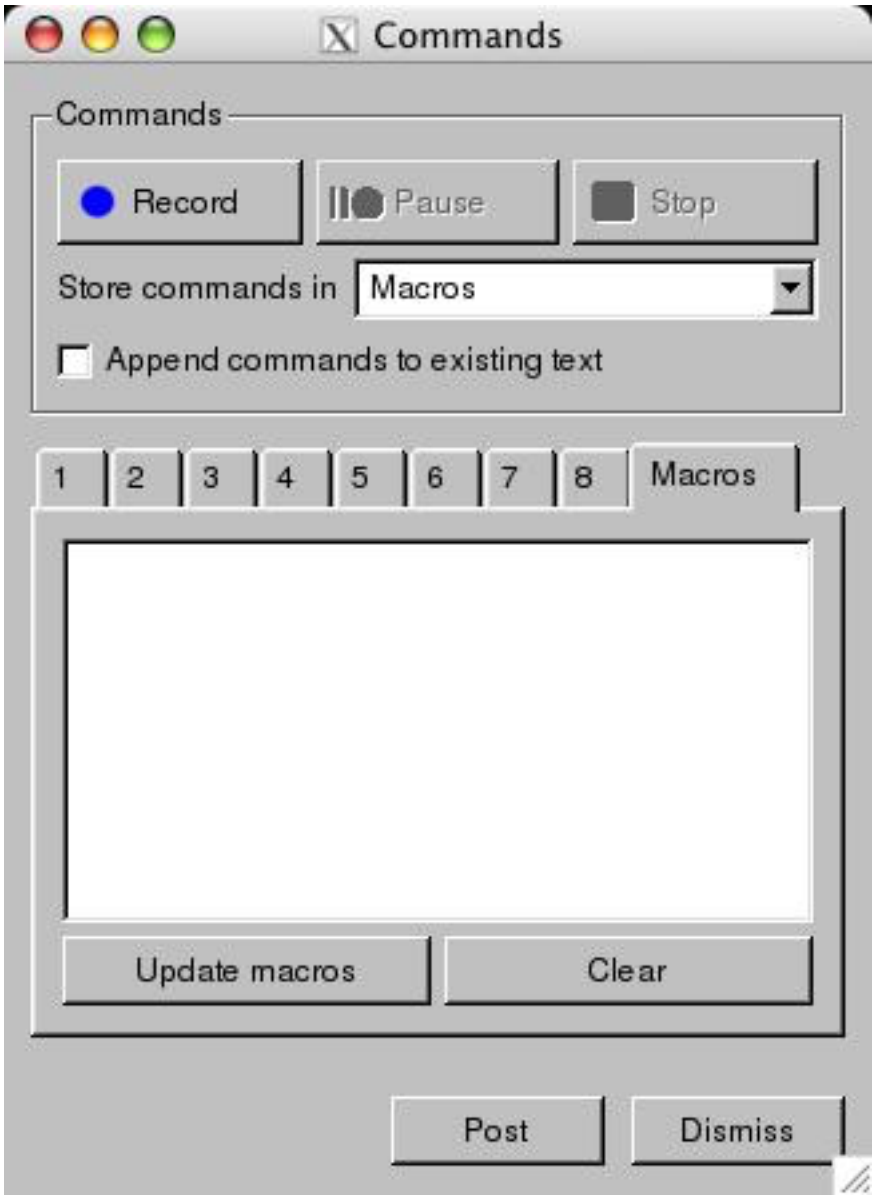


Fig. 4.442: Command Window

Using the GUI and CLI to design a script

Writing a Python script that performs visualization from scratch can be difficult. The process of setting up a complex visualization can be simplified by using both the GUI and the CLI at the same time. For example, you can use VisIt's GUI to set up the plots that you initially want to visualize and then you can save out a session file that captures that setup. Next, you can open a text editor and create a new Python script. The first line of your Python script can use VisIt's *RestoreSession* command to restore the session file that you set up with the GUI from within the Python scripting environment. For more information on functions and objects available in VisIt's Python interface, see the *VisIt_ Python Interface* manual. After using the *RestoreSession* function to set VisIt situated with all of the right plots, you can proceed with more advanced Python scripting to alter the view or move slice planes, etc. Once you have completed your Python script in a text editor, you can pasted it into the **Command Window** to test it or you can pass it along to VisIt's command line movie tools to make a movie.

4.15.2 Macros

VisIt's **Command** window contains controls that allow you to record most GUI actions and view Python scripting code needed to accomplish those actions. The **Command** window provides 8 conventional tabs that serve as destinations for recorded Python coding. In addition to those 8 tabs, there is a special tab called **Macros** that shows the contents of the `visitrc` file. If you record Python code to the **Macros** tab then that Python code is turned into a function that can be called in response to a button click from a button in the **Macros** window.

Recording a macro

Here are the steps involved in *recording* a macro.

1. Open the **Command** window and choose to **Store commands in Macros**.
2. Click the **Record** button
3. Perform any GUI actions that you want to record to a single button click.
4. Click the **Stop** button in the **Command** window.
5. Enter the name of a Python function in which to store your set of recorded commands.
6. Enter the text for the macro button as it will appear in the **Macro** window.
7. Now, the **Macros** tab will contain a function for your recorded commands and it will call the RegisterMacro function from the VisIt Python Interface to associate your Python function with the named button. **Note:** Remembe that you can edit the recorded Python code to suit your needs. You can generalize the code so it can, for example, operate on the active database instead of a specific database. The state information that you need to generalize can often be returned by the GetGlobalAttributes(), GetWindowInformation(), or GetMetaData() functions.
8. Click the **Update macros** button to make VisIt update the buttons in the **Macros** window so it will contain your new button.
9. No further steps need to be taken to save your macro since the macro definitions in the **Macros** tab of the **Command** window will be automatically saved to your `visitrc` file.
10. Click the new button in the **Macros** window whenever you want to replay the recorded set of commands.

Warning: Users should be aware that VisIt's macro recording feature records only those operations that use methods in VisIt's *python* interface. Any operations that would not ordinarily *require* VisIt's python methods to complete are not recorded. In particular, operations involving interaction with the file system such as changing the current working directory and creating a directory (or folder) are not recorded.

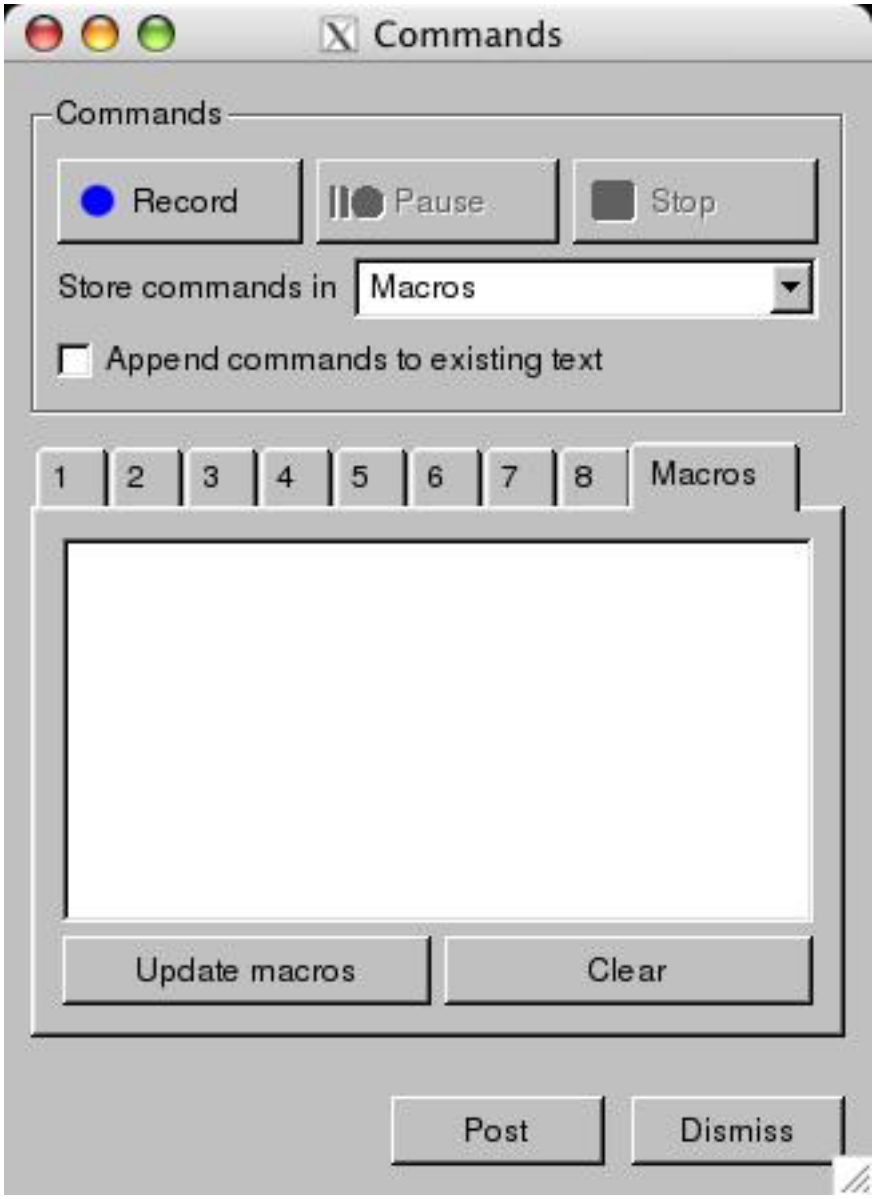


Fig. 4.443: Command Window Macros Tab

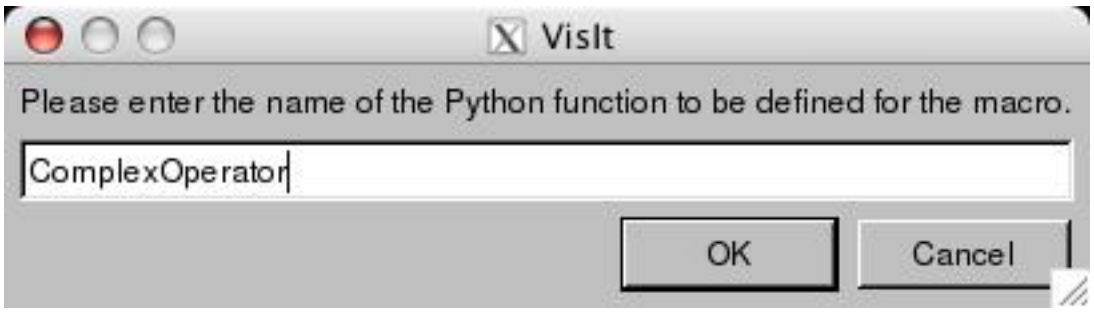


Fig. 4.444: Setting the Python funtion name

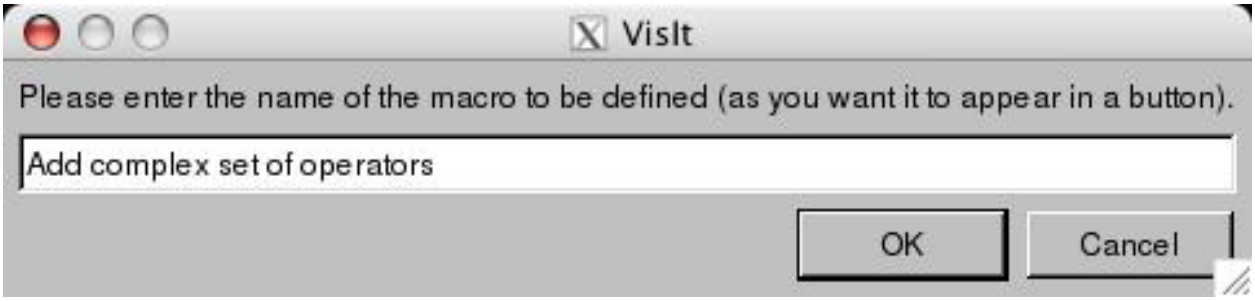


Fig. 4.445: Setting the Macro Button text

Fig. 4.446: The final Macro Button that is produced

4.15.3 VisIt Run Commands (RC) File

VisIt supports a run commands or an *rc* file called the `visitrc` file which is typically *located* in `~/.visit`. The `visitrc` file is a Python source code file that contains Python scripting commands that VisIt executes whenever the CLI is started either from the shell or from within the GUI through the *Command Window*.

The `visitrc` file is most often used to define Python functions associated with VisIt *macros*. However, users can use the file to run whatever Python code they wish during VisIt CLI startup. This could include opening a frequently used database, defining a set of frequently used expressions, etc. See the *Python command-line interface* manual for more information about the commands available in VisIt's Python interface.

4.16 Preferences

In this chapter, we will discuss how to set and save user preferences. User preferences affect the default values for plots and operators as well as window properties like the background color. This chapter reveals where those settings are saved and how to modify them.

4.16.1 How VisIt Uses Preferences

VisIt's preferences are saved into two levels of XML files that are stored in the user's home directory and in the global VisIt installation directory. The global preferences are read first and they allow the system administrator to set global preferences for all users. After VisIt reads the global preferences, it reads the preferences file for the current user.

These settings include things like the color of the GUI and the initial directory from which to read files. Most of the attributes that are settable in VisIt can be saved to the preferences files for future VisIt sessions.

4.16.2 Setting Default Values

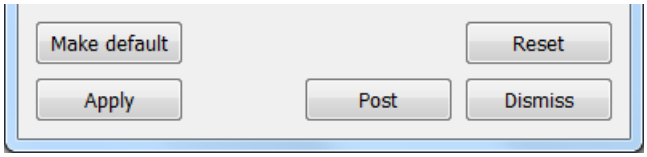


Fig. 4.447: The make default button

Some windows have a button called **Make default** that sets the default attributes for the window. This is typically the case for plot and operator attribute windows. Other windows that have a **Make default** button include the **Annotation**, **Lighting**, **Material Reconstruction Options**, **Mesh Management Options**, **Pick**, **QueryOverTime** and **Interactors** windows. Setting the attributes with the **Apply** button sets the attributes for the active plots or operators. Setting the default attributes sets the attributes for future plots and operators. When saving the settings using **Save Settings** from the **Options** menu, the default attributes are saved. An example of a **Make default** button is shown in Figure 4.447.

4.16.3 How to Save Settings

To save preferences in VisIt, select **Save settings** from the **Main** window's **Options** menu. When VisIt saves the current settings to the users preferences file they are used to set the initial state the next time the user runs VisIt. VisIt does not automatically save settings when changes are made to the default attributes for plots, operators, or various control windows. For windows that only have current attributes (windows without a **Make default** button), the current attributes are saved. For windows that have current and default attributes (windows with a **Make default** button), the default attributes are saved.

To save the entire state of VisIt, which includes things such as the plots in the window and the operators applied to the plots for each visualization window, select either **Save session** or **Save session as** from the **Main** window's **File** menu. When using **Save session**, if a session has already been restored or saved, VisIt will overwrite the existing session file. If a session has not already been restored or saved, VisIt will bring up a dialog window that will allow the user to specify the location and name of the session file. When using **Save session as** VisIt will always bring up a dialog window that will allow the user to specify the location and name of the session file and prompt the user to confirm before overwriting an existing session file.

VisIt saves two preference files, the first of which stores preferences for VisIt's GUI while the second file stores preferences for VisIt's state. When running VisIt on UNIX and MacOS X systems, the preference files are called: `guiconfig` and `config` and they are saved in the `.visit` directory in the users home directory. The Windows version of the `.visit` directory is `%USERPROFILE%\Documents\VisIt`, which may be something like: `C:\Users\<your-user-name>\Documents\VisIt`.

To run VisIt without reading the saved settings, add `-noconfig` to the command line when running VisIt. The `-noconfig` argument is often useful when running an updated version of VisIt that is incompatible with the saved settings. VisIt settings are usually compatible between different versions but this is not always the case and some users have had trouble on occasion when transitioning to a newer version. If VisIt has stability problems when it starts up after upgrading to a newer version, add the `-noconfig` option to the command line and save the settings to write over any older preference files.

4.16.4 Appearance Window

The **Appearance** window is responsible for setting preferences for the appearance of the GUI windows. The **Appearance** window shown in Figure 4.448 is brought up by selecting **Appearance** from the main window's **Options** menu. It can be used to set the GUI colors as well as other attributes such as the style and orientation. In order to change any of the appearance attributes, the user must first uncheck the **Use default system appearance** check box.

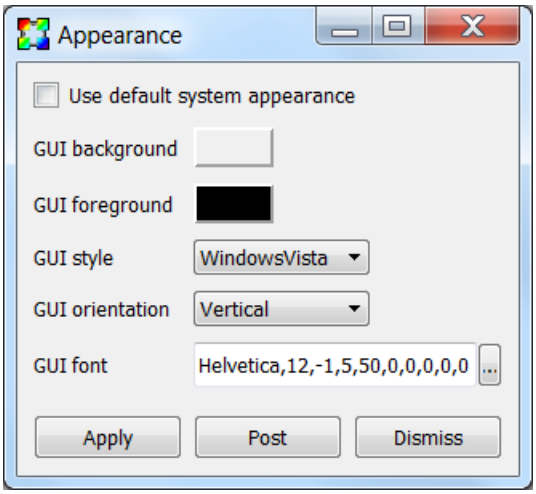


Fig. 4.448: The appearance window

Changing GUI colors

To change the GUI colors using the **Appearance** window, click on the color button next to the color to be changed. To change the background color (the color of the GUI's windows), click on the **GUI background** color button and select a new color from the **Popup color** menu. To change the foreground color (the color used to draw text), click the **GUI foreground** color button and select a new color from the **Popup color** menu.

VisIt will issue an error message if the colors chosen for both the background and foreground colors are close enough that they cannot be distinguished so that the user does not accidentally get into a situation where the controls in VisIt's GUI become too difficult to read. Some application styles, such as Aqua, do not use the background color so setting the background has no effect unless an application style like Windows is chosen, which does use the background color.

Changing GUI Style

VisIt's GUI adapts its look and feel, or application style, to the platform on which it is running. It is also possible to make the GUI use other application styles, although for the most part they look fairly similar.

To change the style select a new style from the **GUI style** menu. It is frequently necessary to change the GUI font by either changing the font description in the **GUI font** text box or selecting a new font from the font selection window, which is brought up by clicking on the **...** button to the right of the **GUI font** text field.

Changing GUI Orientation

By default, VisIt's **Main** window appears as a vertical window to the left of the visualization windows. The default configuration often makes the best use of the display with wide aspect ratio screens. It has become very rare to encounter screens where the horizontal orientation makes better use of the display, so it is not recommended and will most likely be deprecated in future versions of VisIt.

4.16.5 Plugin Manager Window

The **Plugin Manager** window , shown in Figure 4.449, allows the user to see which plugins are available for plots, operators, and databases. Not all plugins have to be loaded, in fact, many operator plugins are not loaded by default. The **Plugin Manager** window allows the user to specify which plugins are loaded when VisIt is started. The **Plugin Manager** window is brought up by selecting **Plugin Manager** from the **Main** window's **Options** menu.

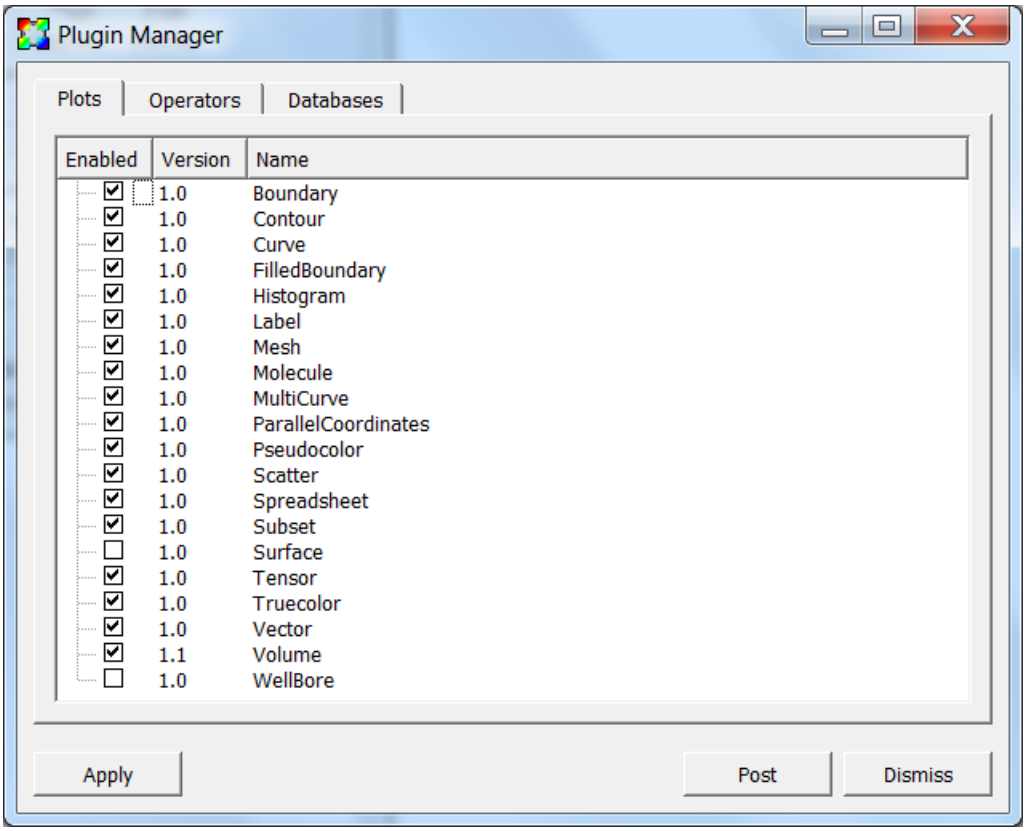


Fig. 4.449: The plugin manager window

Enabling and Disabling Plugins

All of VisIt's plots, operators, and database readers are implemented as plugins that are loaded when VisIt first starts up. Some plugins are not likely to be used by most people so they should not be loaded. The **Plugin Manager** window provides a mechanism to turn plugins on and off. The window has three tabs: **Plots** , **Operators** , and **Databases**. Each tab displays a list of plugins that can be loaded by VisIt. If a plugin is enabled, it has a check by its name.

Plugins can be turned on and off by checking or unchecking the check box next to a plugin's name. Plugins are loaded at startup, so enabling or disabling plugins will not take effect unless the preferences are saved and VisIt is restarted.

If plots or operators are disabled, they will not appear in the **Add**, **Operator**, **PlotAtts** and **OpAtts** menus. Similarly, disabled databases will not show up in the list of **Open file type as** menu in the **File open** window.

4.16.6 Rendering Options Window

The **Rendering options** window (shown in Figure 4.450) contains controls that set global options that affect how the plots in the active visualization window are drawn, as well as, look at information related to the performance of the

graphics hardware VisIt is running on. The **Rendering options** window can be brought up by selecting **Rendering** from the **Main** window's **Preferences** menu. The **Rendering options** window contains three tabs. The **Basic** tab contains basic rendering options, the **Advanced** tab contains advanced rendering options, and the **Information** tab contains information about the rendering performance of the graphics hardware VisIt is running on.

Basic Rendering Options

The **Antialiasing**, and **Specular lighting** options are covered in the *Making It Pretty* chapter.

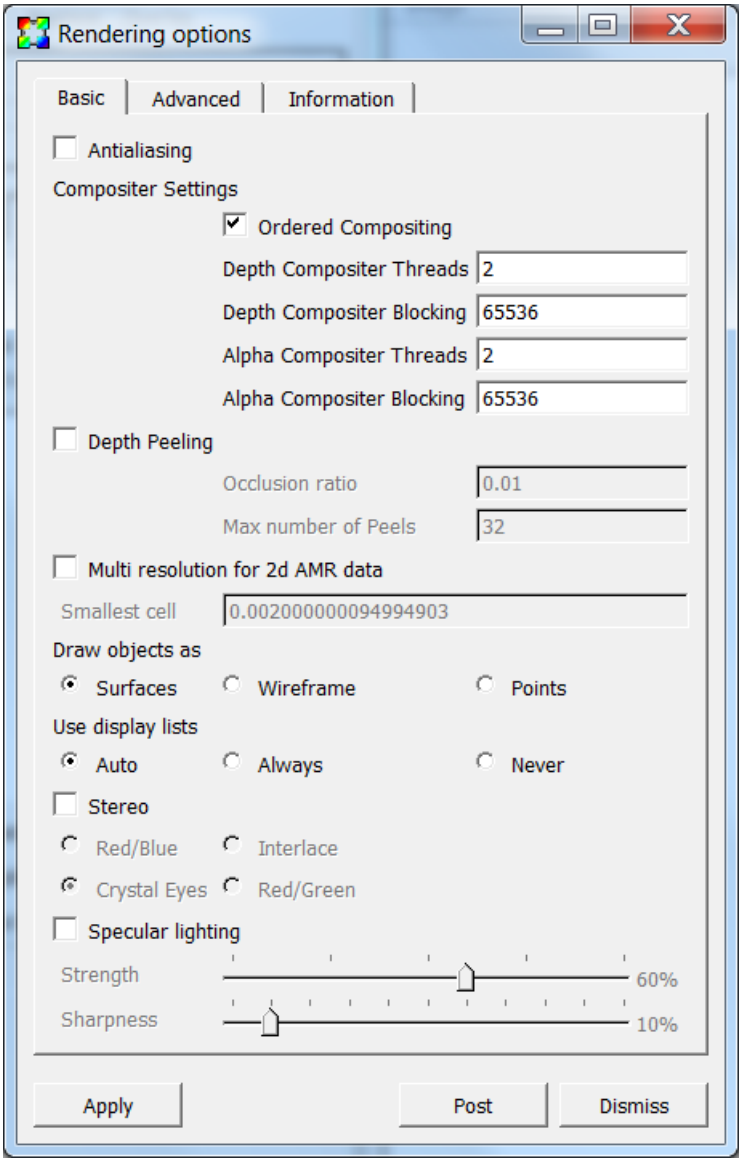


Fig. 4.450: The basic rendering options

Changing surface representations

Sometimes when visualizing large or complex databases, drawing plots with all of their shaded surfaces can take too long to be interactive, even for fast graphics hardware. To combat this problem, VisIt provides an option to view all of

the plots in the visualization window as wireframe outlines or point clouds instead of as shaded surfaces (see Figure 4.451). While being less visually informative, plots drawn as wireframe outlines or as clouds of points can still be useful for visualizations since it is possible to do the setup work like setting the view before switching back to a surface representation that is more costly to draw. To change the surface representation used to draw plots click on either the **Surfaces**, **Wireframe** or **Points** radio buttons below the **Draw objects as** label.

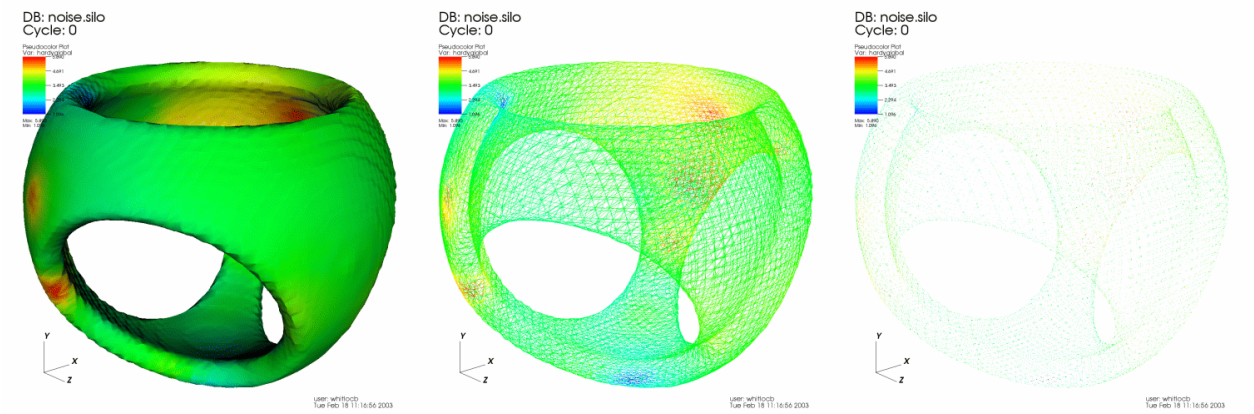


Fig. 4.451: The different surface representations

Using display lists

VisIt benefits from the use of hardware accelerated graphics and one of the concepts central to hardware accelerated graphics is the display list. A display list is a sequence of simple graphics commands that are stored in a computer's graphics hardware so the hardware can draw the object described by the display list several times more quickly than it could if the graphics commands were issued directly. VisIt tries to make maximum use of display lists when necessary so it can draw plots as fast as possible.

By default, VisIt decides when to and when not to use display lists. Typically, when running VisIt on a local workstation with plots that result in fewer than a couple million graphics primitives, VisIt does not use display lists because the cost of creating them is more expensive than just drawing the graphics primitives without display lists. When running on a Unix version of VisIt on a remote computer and displaying the results back to a workstation using an X-server, it is almost always advantageous to create display lists for plot geometry. Without display lists, VisIt must transmit the plot geometry over the network to the X-server every time it renders an image. VisIt can be set to either use or not use display lists all the time. To change the way VisIt uses display lists click on either the **Auto**, **Always** or **Never** radio buttons below the **Use display lists** label.

Stereo images

Stereo images, which are composites of left and right eye images, can convey additional depth information that cannot be expressed by images that are generated using a single eye point. VisIt provides four forms of stereo images: red/blue, red/green, interlace, and crystal eyes. A red/blue stereo image (see Figure 4.452) is similar to frames from early 3D movies in that it appears stereo only when using red/blue stereo glasses. Unfortunately, red/blue stereo images are not very useful for visualization because colors are lost since most of the color ends up in the magenta range when the red and blue color channels are merged. Red/green stereo suffers a similar color loss. Interlaced images alternate lines in the image with left and right eye views so that squinting makes the image look somewhat 3D. VisIt's crystal eyes option requires the use of special virtual reality goggles for images to appear to be 3D but this option is by far the best since it allows interactive frame rates with images that really appear to stand out from the computer monitor. VisIt does not use stereo imaging by default since it makes images draw slower because an image must be drawn for both the left eye and the right eye. To enable stereo images, check the **Stereo** check box. To change the type of stereo

images generated, click on either the **Red/Blue**, **Interlace**, **Crystal Eyes** or **Red/Green** radio boxes under the **Stereo** check box.

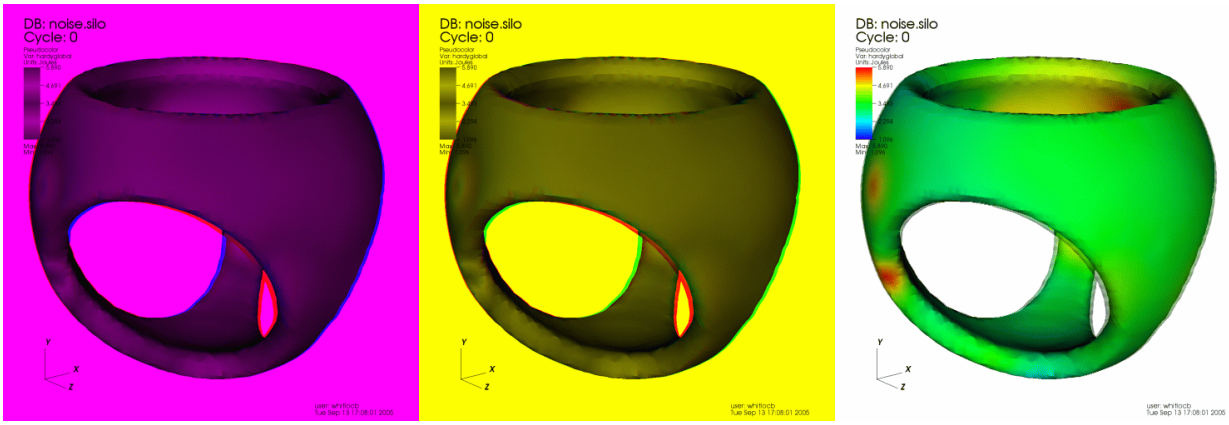


Fig. 4.452: Some various stereo image types

Advanced Rendering Options

The **Shadows**, and **Depth Cueing** options are covered in the *Making It Pretty* chapter.

Scalable rendering

VisIt typically uses graphics hardware on the local computer to very quickly draw plots once they have been generated by the compute engine. This becomes impractical for very large databases because the amount of memory needed to store the graphics commands that draw the plots quickly exceeds the amount of memory in the graphics hardware. Large sets of graphics commands can also degrade performance when they must be shipped over slow networks from the compute engine to the VisIt's viewer. VisIt provides a scalable rendering option that can improve both of these situations by creating the actual plot images, in parallel, on the compute engine, compressing them, and then transmitting only an image to the viewer where the image can be displayed.

Scalable rendering can be orders of magnitude faster for large databases than VisIt's conventional image drawing strategy because large databases are typically processed using a parallel compute engine. When using scalable rendering with a parallel compute engine, VisIt is able to draw small pieces of the plot on each processor in parallel and then glue the image together before sending it to the viewer to be displayed. Not only has the image likely been created faster, but the size of the image is usually on the order of a megabyte instead of the tens or hundreds of megabytes needed to transmit graphics commands, which results in faster transmission of the image to the viewer. The drawback of scalable rendering is that it is usually not as interactive as graphics hardware because each time the view is changed or some other change is made to the plots, the image must be resent to the viewer over the network.

VisIt can automatically determine when to stop sending geometry to the viewer in favor of sending scalably rendered images. The scalable rendering threshold determines when VisIt switches between sending geometry and doing scalable rendering. The threshold is based on the number of polygons to be rendered. The scalable rendering threshold can be changed by entering a new number of polygons into the **When polygon count exceeds** spin box. The number is specified in thousands of polygons.

It is also possible to have VisIt always or never use scalable rendering. To change the scalable rendering mode, click on either the **Auto**, **Always** or **Never** radio boxes under the **Use scalable rendering** label.

Rendering Information

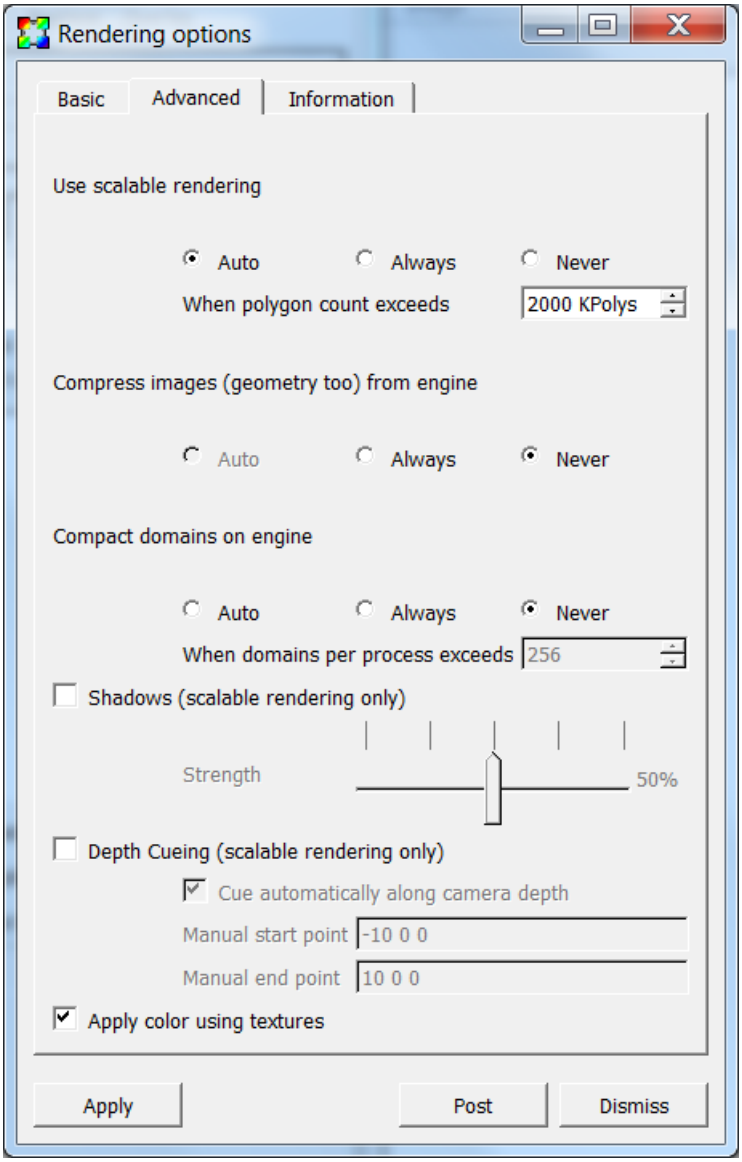


Fig. 4.453: The advanced rendering options

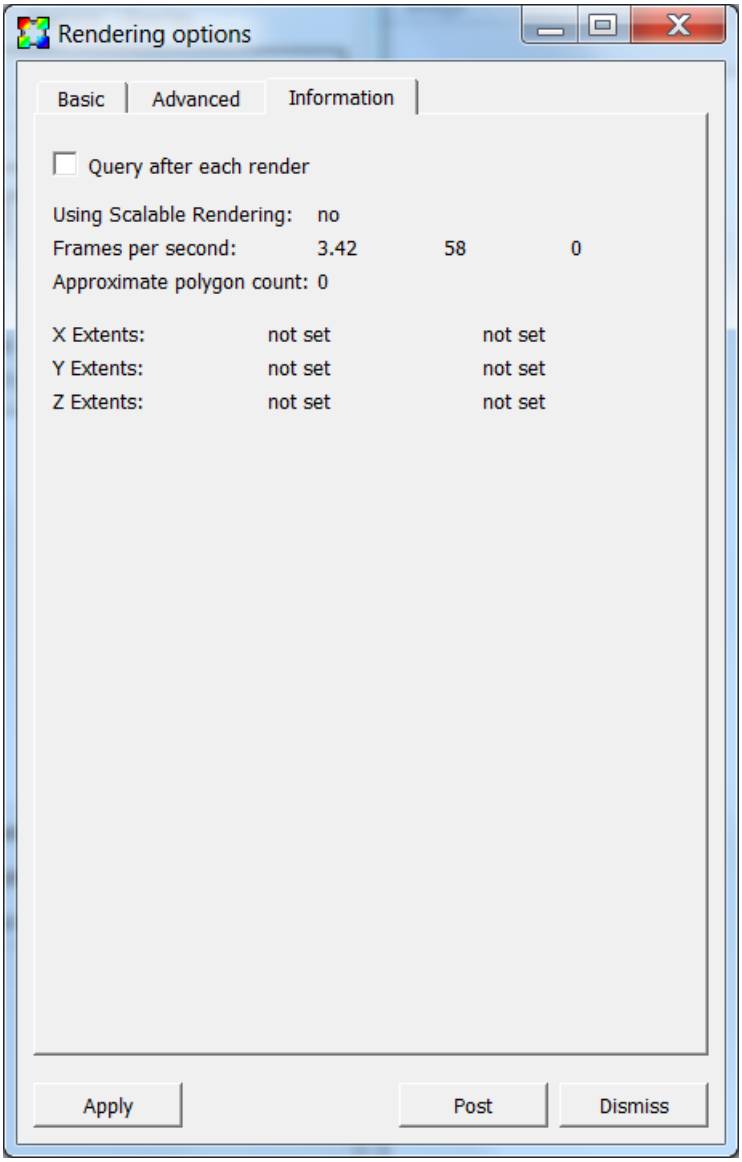


Fig. 4.454: The rendering information

Scalable rendering

The scalable rendering indicates if the compute engine used scalable rendering to render the image displayed in the viewer. The use of scalable rendering is indicated next to the **Use Scalable Rendering:** label.

Frames per second

The frames per second refers to the number of times that VisIt can draw the plots in the visualization window in the course of a second. VisIt displays the minimum, average, and maximum frame rates achieved during the last draw operation, like rotating the image with the mouse. They are displayed next to the **Frames per second:** label. Some actions that force a redraw do not cause the information to update. An example of this is resizing the visualization window. To make VisIt update the frame rate information after each time it draws the plots in the visualization window, check the **Query after each render** check box.

Polygon count

The polygon count refers to the number of polygons used to represent the plots in the visualization window. VisIt displays the polygon count next to the **Approximate polygon count:** label.

Plot extents

The plot extents are the minimum and maximum locations of the plot in each spatial dimension. The plot extents are the smallest bounding box that can contain the plots in the visualization window. VisIt displays the plot extents for each dimension next to the **X Extents:**, **Y Extents:** and **Z Extents:** labels. .

4.16.7 Preferences Window

The **Preferences** window, shown in Figure 4.455, contains controls that allow setting global options that influence VisIt's behavior. The **General** tab contains a collection of miscellaneous options. This is followed by options that are grouped by functionality. The groups are contained in the **Database**, **Session file** and **File panel** tabs.

Copying Plots On First Reference

The **Clone windows on first reference** option clones all attributes of the active window to a new window when a window is made active for the first time. To control this behavior check or uncheck the **Clone window on first reference** check box.

Posting Windows By Default

When a postable window, such as a plot attributes window is brought up, the window manager is free to show the window wherever it likes. When using VisIt on a large display where the windows might pop up very far away from VisIt's **Main** window, it is sometimes convenient to make sure that windows that can be posted to the **Notepad** area are initially posted to the **Notepad** area instead of popping up wherever the window manager puts them. To make postable windows post to the **Notepad** area by default when they are shown, check the **Post windows when shown** check box.

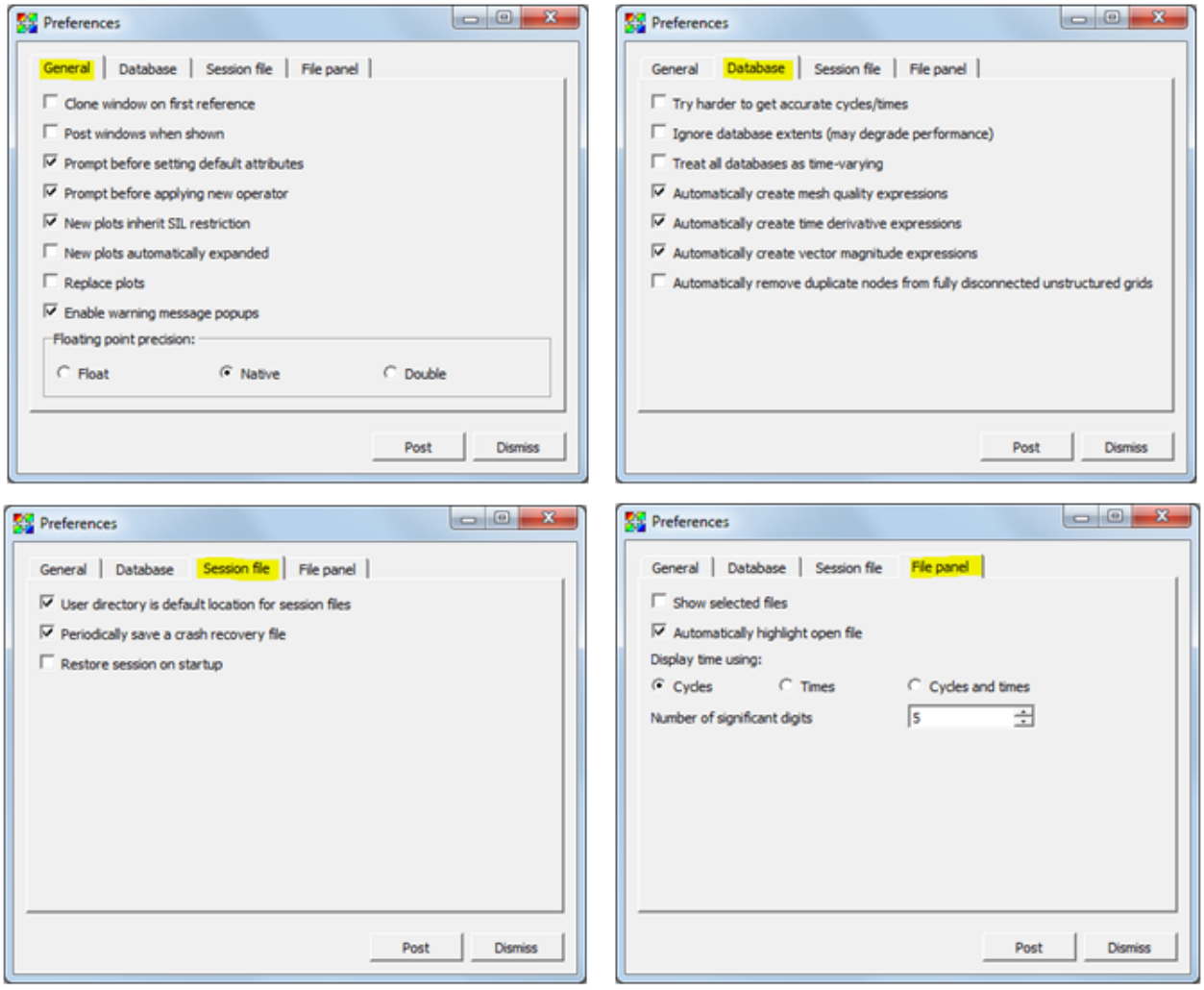


Fig. 4.455: The preferences window

Reading Accurate Cycles and Times From Databases

Many of the file formats that VisIt reads contain a single time state, making accurate cycles and times unavailable in VisIt's metadata for all but the open time state. To get accurate times and cycles for these types of files, VisIt would have to open each file in the database, which can be a costly operation. VisIt does not go to this extra effort unless **Try harder to get accurate cycles/times** option is enabled. This option allows VisIt to create meaningful cycle or time-based database correlations for groups of single time state databases. Note that databases that are already open will need to be reopened in order for VisIt to retrieve updated cycles and times.

File Panel Properties

The **File panel** is a deprecated feature that will be removed in a future release of VisIt. The **File panel** is enabled by checking the **Show selected files** check box. It is not recommended for use.

4.16.8 File Locations

VisIt manages various files associated with its operation. In most cases where VisIt saves or loads data from files, the user is presented with a file browser dialog and can explicitly choose arbitrary locations on the file system to look for or store files. However, this is not universally true. The locations and names of some files are *prescribed*. In this section we provide some additional details about various file locations and names involved with the operation of VisIt.

Factors Effecting Prescribed File Location and Names

To complicate matters, the *prescribed* location of these files depends on a few different factors including

- Which platform is running VisIt.
- How VisIt was launched.
- Whether VisIt is running in *client/server mode*.

The Platform and the User's *Home* Directory

Typically, on UNIX and macOS systems, prescribed configuration files are stored in `~/.visit` whereas on Windows systems, they are, by default, in `%USERPROFILE%\Documents\VisIt`, which may be something like `C:\Users\<user-name>\Documents\VisIt`. Furthermore, on Windows, Visit honors the `CSIDL_PERSONAL` and `CSIDL_MYDOCUMENTS` CSIDL environment variables. Depending on the how the system is configured, these might actually resolve to a networked drive, but most commonly, to the values described previously. Finally, Windows users can also set the `VISITUSERHOME` environment variable to point to whatever location they desire, and VisIt will use that location instead. For the rest of this section, we use the symbol `VUSER_HOME` as a way to refer to whatever this location happens to be on whatever platform VisIt is running.

The Launch Method and the *Current Working Directory*

The launch method effects what VisIt uses as the current working directory or `CWD`. On Windows and macOS it is most common to start VisIt by clicking an icon. In these cases, VisIt uses the user's `$HOME` or login directory as the current working directory.

However, when VisIt is started by typing a command-line at a shell terminal prompt, then VisIt uses whatever that shell's `CWD` is at the time of launch.

Client/Server Operation

When running VisIt in *client/server mode*, the user will need to be aware of what VisIt uses as `VUSER_HOME` and `CWD` on both the client and the server. These cases are pointed out in the descriptions below.

Files in `VUSER_HOME`

Most of the files associated with VisIt configuration are managed in in `VUSER_HOME`. When running in client/server, it is the configuration files on the *local client* that effect behavior. This means it is always the files on the *local* machine and not the *remote* system that effect behavior. Any configuration files that might also be on the remote server do not play a role in effecting behavior in client/server mode.

Settings/Preferences File

- Location and file name: `VUSER_HOME/config`
- Purpose: Holds user settings from **Preferences Window** plus numerous other settings such as default attributes for operators and plots, default database read options, default color tables, as well as the enabled/disabled state of various plot, operator and database plugins.
- Written: When user *saves settings*.
- Read: On VisIt startup but this can be overridden by the `-noconfig` command-line *startup option*.
- Format: ASCII XML

GUI Configuration File

- Location and file name: `VUSER_HOME/guiconfig`
- Purpose: Holds positions and sizes of various GUI windows. Also holds the list of recently used paths to open databases.
- Otherwise operates identically to `VUSER_HOME/config`.

Host Profile Files

- Location and file name(s): `VUSER_HOME/hosts/host_<site-name>_<resource-name>.xml` where `<site-name>` is something like `ornl`, `llnl`, `anl` etc. and `<resource-name>` is a machine name such as `summit`, `sierra`, `theta`.
- Purpose: Stores information on how to connect to and launch jobs on a specific compute resource. In many cases, there are separate sets of host profile files for all the compute resources at a commonly used site such as LLNL CZ or RZ, ANL, ORNL, etc. Often sites will *post* VisIt host profile files in places for users to easily find and *install* them. Installing them is just a matter of copying them to `VUSER_HOME`. In addition, updated profiles can be downloaded and installed automatically by VisIt from the **Host Profiles** window.
- Written: When user *saves settings* or when user hits the **Export Host** button from the **Host Profiles** window.
- Read: On VisIt startup. All host profiles in `VUSER_HOME/hosts/host*.xml` are read on VisIt startup but this can be overridden by `-noconfig`. Users should be aware of this behavior. If the user passes `-noconfig` for the purposes of avoiding the loading of preferences, s/he will also be without any host profiles.
- Format: ASCII XML

VisIt Run Commands (rc) File

- Location and file name: `VUSER_HOME/visitrc`
- Purpose: Holds Python code to be executed each time VisIt is launched.
- Written: Whenever user hits the **Update Macros** button in the *Command Window*.
- Read: On VisIt startup of the CLI.
- Format: Python source code. However, there is no `.py` file extension in the file name.

Command Window Tabs Script Files

- Location and file name(s): `VUSER_HOME/script<K>.py` where `K` is an integer in the range [1...8].
- Purpose: Hold the python code associated with each tab in the **Command Window**.
- Written: When user *saves settings*.
- Read: On VisIt startup but this can be overridden by `-noconfig`.
- Format: Python source code.

Color Table Files

- Location and file name(s): `VUSER_HOME/<color-table-name>.ct`
- Purpose: Store a single color table for easy sharing with other users.
- Written when the user hits the **Export** button in the *color table window* from *Controls -> Color table...*.
- Read: On VisIt startup. All color table files in `VUSER_HOME/*.ct` are read and loaded into VisIt. However, this behavior is overridden by `-noconfig`.
- Format: ASCII XML specifying the *colors and color control points* for the color table.

Custom Plugin Files

- Location and file name(s): There are separate directories in `VUSER_HOME` for *private*, user-specific operator, database and plot plugins. On UNIX/macOS, these are
  - `VUSER_HOME/<visit-version>/<visit-arch>/plugins/operators/`
  - `VUSER_HOME/<visit-version>/<visit-arch>/plugins/databases/`
  - `VUSER_HOME/<visit-version>/<visit-arch>/plugins/plots/`

  where `<visit-version>` and `<visit-arch>` are the VisIt version number and VisIt architecture moniker. On Windows, these diretories are
  - `VUSER_HOME/operators/`
  - `VUSER_HOME/databases/`
  - `VUSER_HOME/plots/`

  If the `-public` command-line option to `xml2cmake` is used when building a plugin and the user performing this operation has appropriate permissions, the plugin will instead be installed to the VisIt *public* installation directory for *all* users of that installation. If a previous version of this plugin exists there, it will be overwritten by this operation.

A single plugin involves a set of related files for the mdserver, engine and those common all VisIt components. For example, on UNIX the files for the Silo database plugin are `libESiloDatabase_par.so`, `libESiloDatabase_ser.so`, `libISiloDatabase.so`, and `libMSiloDatabase.so`.

- Purpose: Directories to hold custom plugin shared library files.
- Written: When the user makes and installs or copies the shared libraries for a custom plugin.
- Read: On VisIt startup, all *enabled* plugin *info* files are read. The remaining plugin files are read only when the plugin is actually used. In client/server mode, it is important to ensure that the same plugin files have been installed on *both* the client and the server.
- Format: Binary shared library files in the machine format of the host architecture.

Usage Tracking Files

- Location and file name(s): `VUSER_HOME/stateA.B.C.txt` where `A`, `B` and `C` form a VisIt version number.
- Purpose: Holds a single ASCII integer indicating the number of times the associated VisIt version has been run. This is to facilitate suppression of the release notes and help after the *first* run of a new version of VisIt.
- Written: Each time VisIt is started, the integer value in the associated state tracking file is updated.
- Read: Each time VisIt is started, the value in the associated state tracking file is read.
- Format: ASCII text

Crash Recovery Files

- Location and file name(s): `VUSER_HOME/crash_recovery.$pid.session` and `VUSER_HOME/crash_recovery.$pid.session.gui` where `$pid` is the process id of the VisIt viewer component.
- Purpose: Hold the most recently saved last good state of VisIt and VisIt's GUI windows prior to a crash.
- Written: Periodically from VisIt automatically. Disabled if the preference `Periodically save a crash recovery file` is unchecked in the **Preferences Window**. In client/server mode, crash recovery files are always written on the client.
- Read: When user starts VisIt and answers `yes` when queried whether to start up from the most recent crash recovery file or when user explicitly specifies the crash recovery file as an argument to the `-sessionfile` command-line *startup option*.
- Format: ASCII XML, same as any other VisIt *session files*.

Files In Other Locations

There are several other kinds of files VisIt reads and writes to locations other than `VUSER_HOME`. These are breifly described in this section.

Database Files

- Location and file name(s): User uses *File* → *Open...* to bring up a file browser to select the name and location of database files.
- Purpose: Database files store the data that VisIt is used to analyze and visualize for scientific insights.

- Written: By data producers, simulation codes or instruments, upstream of VisIt in the scientific analysis workflow.
- Read: On demand when user selects *File* → *Open...*. The `-o` command-line *startup option* can be used to select a database file to open at startup. VisIt uses the *file's extension* to decide what type of database the file is and then select the appropriate plugin to read it.
- Format: Varies by database type.

VisIt Debug Log (`.vlog`) Files

- Location and file name(s): The location of these files depends on whether VisIt is being run in *client/server mode*. When running client/server, some logs are written on the client and some on the server. On Windows, the logs on the client are always located in `VUSER_HOME` but on UNIX/macOS the logs on the client are written to whatever the `CWD` was when VisIt was started. If started by clicking on an icon, this is most likely the the user's login directory. If started from a command-line, it is whatever the shell's `CWD` for that command-line was. On the server, the logs are written to the user's login (home) directory. In a typical client/server scenario, the user gets gui and viewer logs locally in the `CWD` and mdserver and engine logs on the remote system in their login (home) directory. In a purely local scenario, all logs are written to the `CWD`.

  On UNIX/macOS, the names of the log files are of the form `<letter>.<component-name>.<mpi-rank-or-$pid>.<debug-level>.vlog` where `<letter>` is one of `A` through `E`, `<component-name>` is one of `gui`, `mdserver`, `viewer`, `engine_ser`, `engine_par`, `<mpi-rank-or-$pid>` is the MPI rank for a parallel engine (`engine_par`) or, optionally if `-pid` is given as a command-line *startup option*) the component's process id, and `<debug-level>` is the integer argument for the `-debug` command-line *startup option*. For example the file names are `A.mdserver.5.vlog` or `C.engine_par.123.2.vlog`.

  On Windows, the names of the log files are slightly different and are of the form `<component-name>.exe.<$pid>.<debug-level>.vlog` or `<component-name>.exe.<mpi-rank>.<$pid>.<debug-level>.vlog` for a parellel engine. On Windows, the `-pid` command-line *startup option*) is ignored and `<$pid>` is always included in the file names.
- Purpose: Capture streaming debugging messages from various VisIt components.
- Written: Continuously by VisIt if `-debug L` where `L` is the debug *level* and is an integer in the range `[1...5]` is given on the command-line that starts VisIt or buffered if a `b` is given immediately afte the debug level integer. In addition, on UNIX/macOS VisIt maintains the 5 most recently written logs from the 5 most recent component executions each beginning with the letters `A` through `E`, `A` being the most recent.
- Format: Various, ad-hoc ASCII, mostly human readable.

Plot and Operator Attribute Files

- Location and file name(s): User is prompted with a file browser to select the name and location of these files.
- Purpose: Hold the settings for a single, specific plot or operator for easy sharing with other users.
- Written: Whenever user hits the **Save** button in a plot or operator attributes window.
- Read: Whenever user hits the **Load** button in a plot or operator attributes window.
- Format: ASCII XML.

Session Files

- Location and file name(s): User is prompted with a file browser to select the name and location of these files.

- Purpose: *Session files* are used to save and restore the entire state of a VisIt session.
- Written: On demand when user selects *File* → *Save session...*
- Read: On demand when user selects *File* → *Restor session...* or when the `-sessionfile` command-line *startup option* is used to specify a session file to open at startup.
- Format: ASCII XML.

Save Window Files

- Location and file name(s): User uses the *File* → *Set save options...* to specify the name and location of subsequent saved window files as well as many other properties of a saved window.
- Purpose: Save the *visually relevant* aspects of the data displayed in the currently active window usually but not always to an image file.
- Written: On demand when user selects *File* → *Save Window* or hits the **Save** button in the **Set save options** window. In client/server mode, keep in mind that the files are written only on the *client*.
- Read: Yes, saved images can be read into VisIt like any other database. On demand when user selects *File* → *Open...*
- Format: Various, see *Set save options* window.

Export Database Files

- Location and file name(s): User uses *File* → *Export database...* to bring up a file browser to select the name and location of exported database files.
- Purpose: Exported database files are often used to share computed results among users, to convert among database formats, or to create a new more convenient database to load back into VisIt for further analysis.
- Written: On demand when user selects *File* → *Export database...*. While VisIt *reads* over 130 different types of databases, only about 20 of those types does it *write*. And some of those output types support only limited kinds of data. In client/server mode, keep in mind that the files are saved only on the server.
- Read: On demand when user selects *File* → *Open...*
- Format: Varies by database type.

Save Window vs. Export Database Files

As far as file location are concerned, the key issue for users to keep in mind regarding **Save Window** and **Export Database** operations has to do with client/server operation. In client/server mode, **Save Window** produces files always on the client whereas **Export Database** produces files always on the server.

Apart from file locations, another key issue is understanding when to use **Save Window** vs. **Export Database**. In some circumstances, these operations can be highly similar and confusing to decide which to use.

In general, the **Save Window** operation is used to save *visually relevant* aspects of the data most often to an *image* file whereas the **Export Database** operation is to output a wholly new VisIt *database* file. The cases where these two operations can get confused is when non-image formats are used by **Save Window** such as STL, VTK, OBJ, PLY (3D formats) and Curve or Ultra (2D, xy curve formats) formats. These non-image formats support object and visually relevant object attributes in 2 and 3 dimensions for input to other high end graphics tools such as for 3D printing or rendering engines. In particular, these formats typically support aspects of the *rendering* process such as object colors, textures, lighting and view. This is the key to what makes a **Save Window** in these formats different from **Export Database**.

Adjusting Configuration

Probably the easiest way to change VisIt configuration is to start a new VisIt session, make the desired changes through the GUI and then *save settings*. Sometimes starting the GUI to just adjust configuration is inconvenient.

Sometimes, users need to temporarily change their configuration either to work around or diagnose an issue. Since the majority of content in these files is ASCII, it is possible to manually edit files without having to start VisIt.

The user can also move (or rename) files so that VisIt will either find or not find them. For example, a common trick is to change the name of `VUSER_HOME/config` to `VUSER_HOME/config.orig` so that the majority of *settings/preferences* are not seen during VisIt startup but other things such as host profiles still work. The most dramatic variation of this approach is to move the whole `VUSER_HOME` directory which on UNIX platforms might be a command like `mv ~/.visit ~/.visit.old`.

4.17 Help

In this chapter, we will discuss how to use VisIt's online help. VisIt's online help consists of release notes, copyright information, Frequently Asked Questions (FAQ), and the contents of this manual. The release notes help page lists the complete set of bug fixes and enhancements for the current version of VisIt with links to the release notes for older versions. The copyright information help page lists VisIt's copyright agreement. The FAQ help page lists commonly asked questions and the answers to those questions. Beginning VisIt users should read through the FAQ help page to find the answers to commonly asked questions. Finally, the contents of this manual are available as online help.

4.17.1 About VisIt

VisIt provides a **Splash screen** (Figure 4.456) that appears when the tool is launched. The **Splash screen** has three purposes: entertainment, displaying startup progress, and telling the user about VisIt. As VisIt launches, the **Splash screen** cycles through a handful of images that show some of VisIt's capabilities and it also tells the user what happens while VisIt is launching. Once VisIt is launched, you can look at some information about VisIt by selecting the **About** option from the **Main Window's Help** menu. Choosing that menu option displays the **Splash screen** which can be hidden by clicking its **Dismiss** button.

4.17.2 Help Window

VisIt's **Help Window**, shown in Figure 4.457, displays all of VisIt's online help content. You can open the **Help Window** by choosing the **Help** option from the **Main Window's Help** menu. The **Help Window** has a toolbar along the top of the window while the rest of the window is divided vertically into two main areas. The left side of the window is used to select online help pages and it is further divided with tabs for help contents, help index, and bookmarks. The right side of the window displays the content for the online help pages.

Help Window Toolbar

The **Help Window's** toolbar exposes buttons for navigation, changing font size, and adding bookmarks. You can hide the toolbar by double-clicking on the handle located at the far left of the toolbar. The toolbar can also be moved to other parts of the **Help Window** by clicking on its handle and dragging it to the top, sides, or the bottom of the **Help Window**.

Fig. 4.456: Splash screen

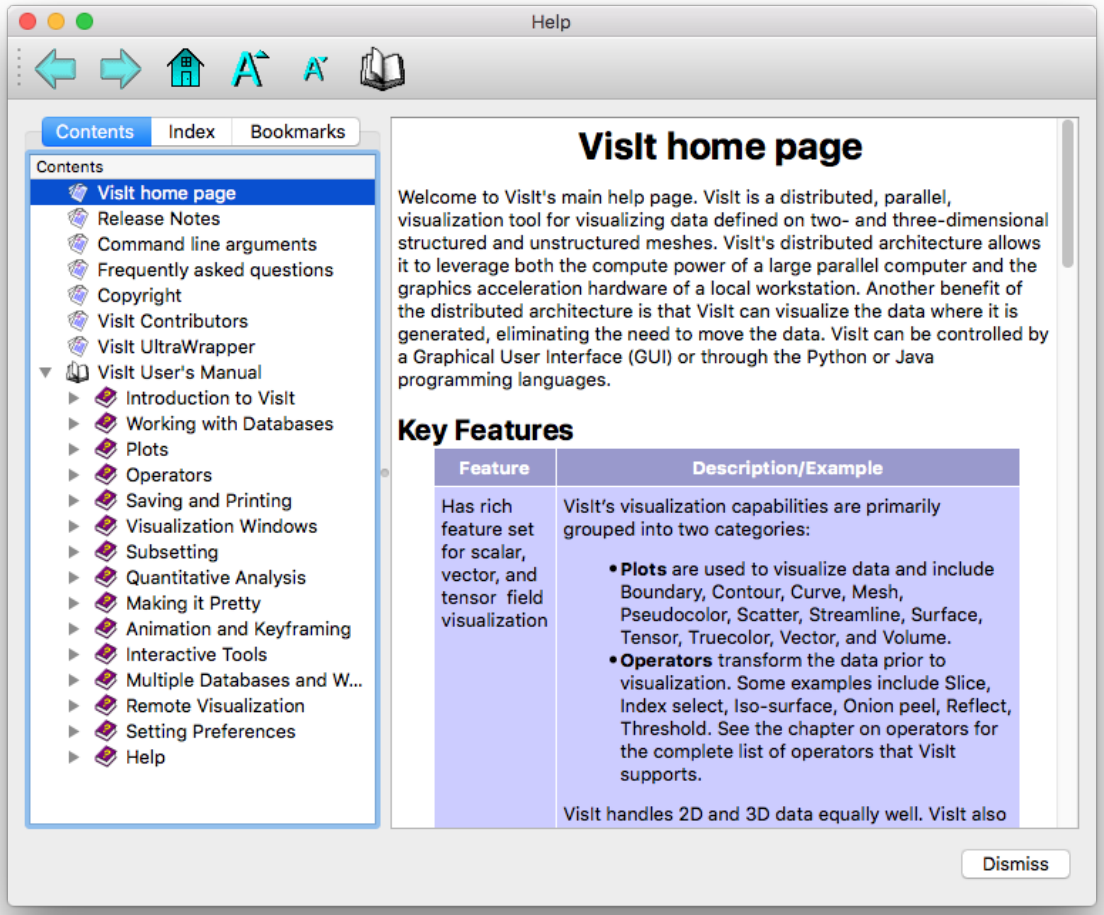


Fig. 4.457: Help window

Navigation

The toolbar contains buttons that you can use to cycle forward and backward in the list of visited help pages. The **Back** button has an arrow icon that points to the left and the button changes the active help page to the last visited help page. The **Forward** button has an arrow icon that points to the right and it switches the help page to the page that was active before the **Back** button was clicked. If have not visited any help pages, both of these buttons are disabled. The toolbar also contains a **Home** button which has a house icon. The **Home** button displays the VisIt home page, which describes VisIt's features.

Changing font size

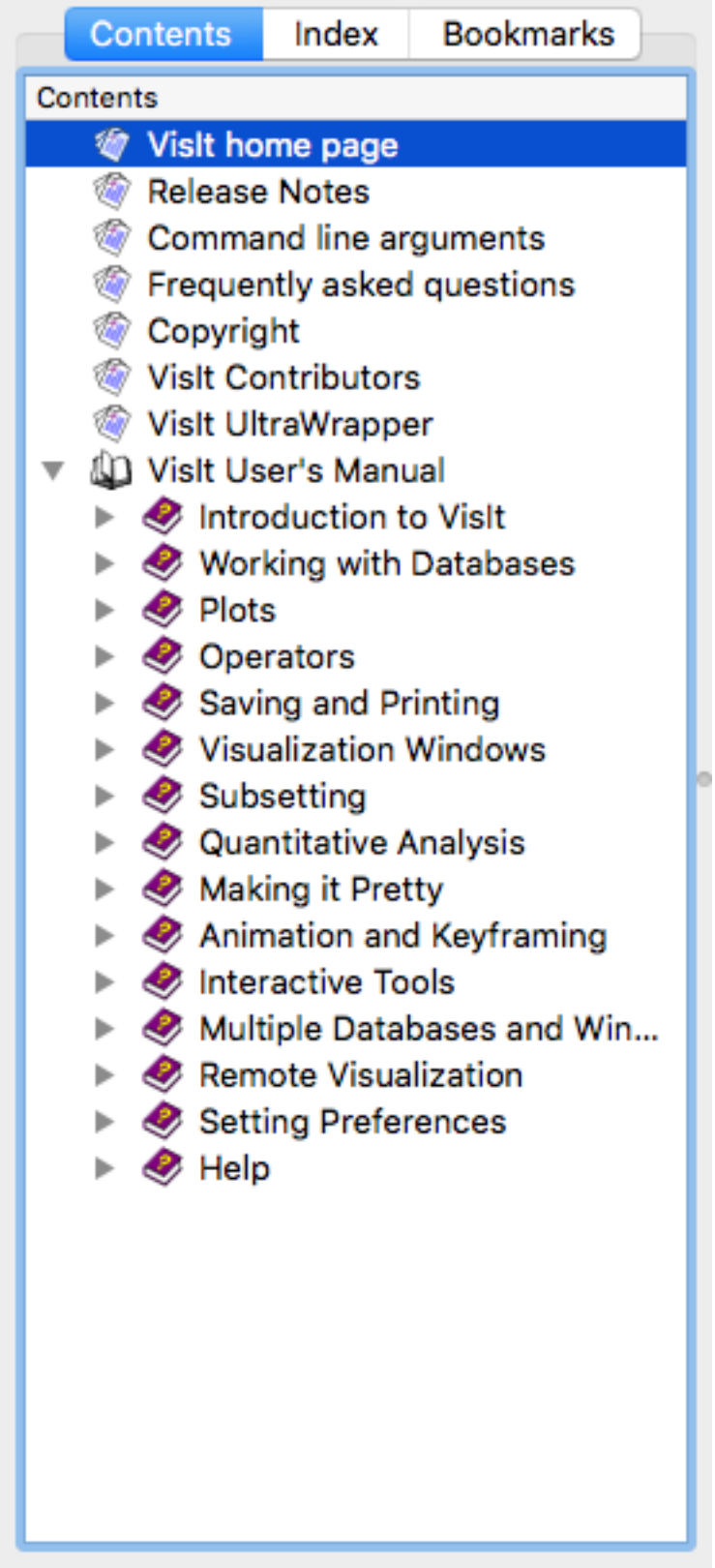
The toolbar contains two buttons that allow you to change the font size used to display online help. The **Larger font** button is distinguished by a large capital *'A'* and a small triangle which points up. When the **Larger font** button is clicked, the font size is increased and the active help page is redrawn with the larger font. The **Smaller font** button looks similar to the **Larger font** button except that its icon's triangle points down and its *'A'* is smaller. The **Smaller font** button decreases the font size and immediately redraws the active help page using the new smaller font.

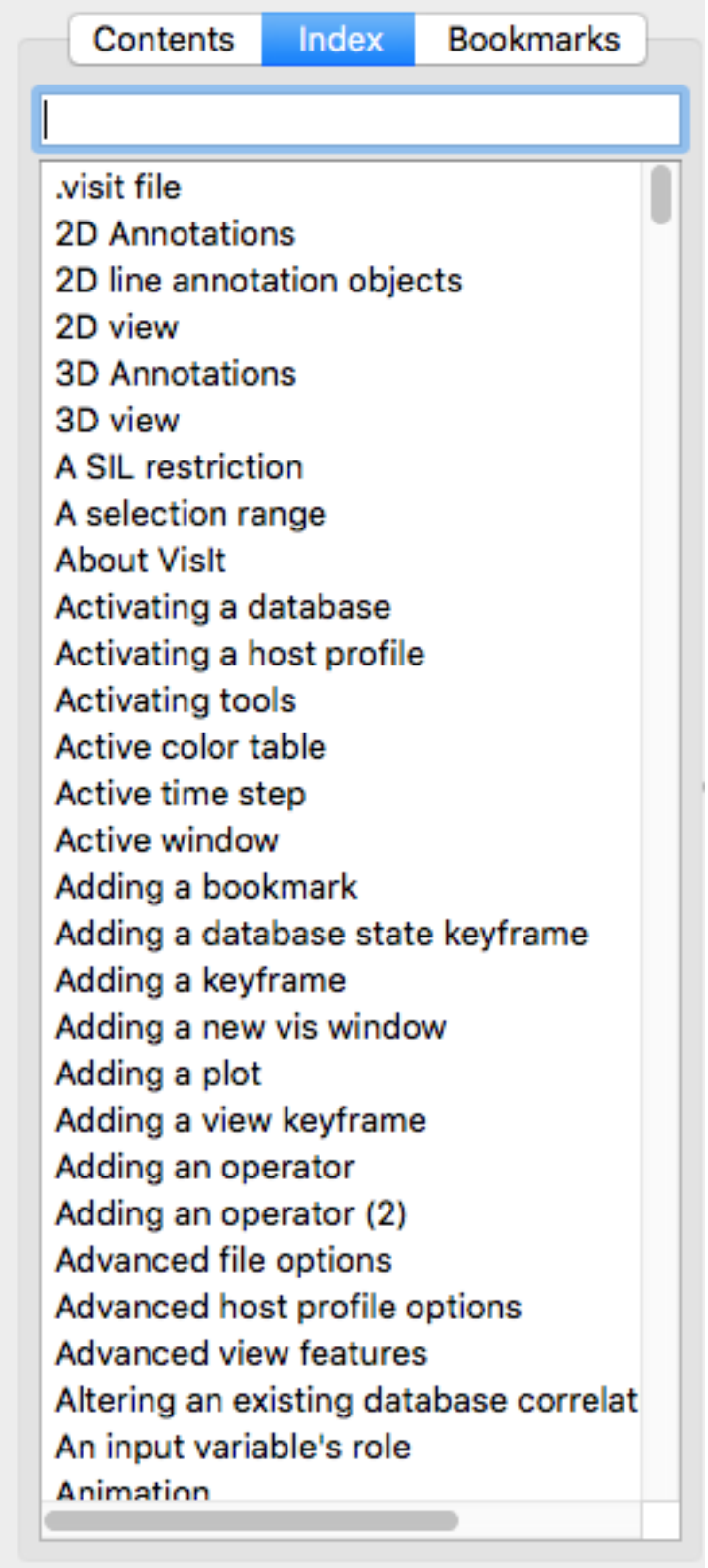
Adding a bookmark

VisIt's **Help Window** provides the ability to create and save personal bookmarks. This allows you to easily come back to frequently-used sections of the online help. The toolbar contains an Add bookmark button that adds the current help page to the list of bookmarks. The **Bookmarks** tab in the left part of the **Help Window** also has this feature.

Selecting a help page

The **Help Window** has three tabs, shown in Figure 4.458, that allow a help page to be located in different ways. The first tab is the **Contents** tab and it lists all of the online help pages and allows them to be grouped into related topics. The **Index** tab lists all of the online help pages in an alphabetized list that can be searched for a particular help topic. The **Bookmarks** tab shows all bookmarked help pages which can be recalled by clicking on a bookmark.


Contents
Index
Bookmarks
Contents
VisIt home page
Release Notes
Command line arguments
Frequently asked questions
Copyright
VisIt Contributors
VisIt UltraWrapper
VisIt User's Manual
Introduction to VisIt
Working with Databases
Plots
Operators
Saving and Printing
Visualization Windows
Subsetting
Quantitative Analysis
Making it Pretty
Animation and Keyframing
Interactive Tools
Multiple Databases and Win...
Remote Visualization
Setting Preferences
Help


Contents
Index
Bookmarks
.visit file
2D Annotations
2D line annotation objects
2D view
3D Annotations
3D view
A SIL restriction
A selection range
About VisIt
Activating a database
Activating a host profile
Activating tools
Active color table
Active time step
Active window
Adding a bookmark
Adding a database state keyframe
Adding a keyframe
Adding a new vis window
Adding a plot
Adding a view keyframe
Adding an operator
Adding an operator (2)
Advanced file options
Advanced host profile options
Advanced view features
Altering an existing database correlat
An input variable's role
Animation

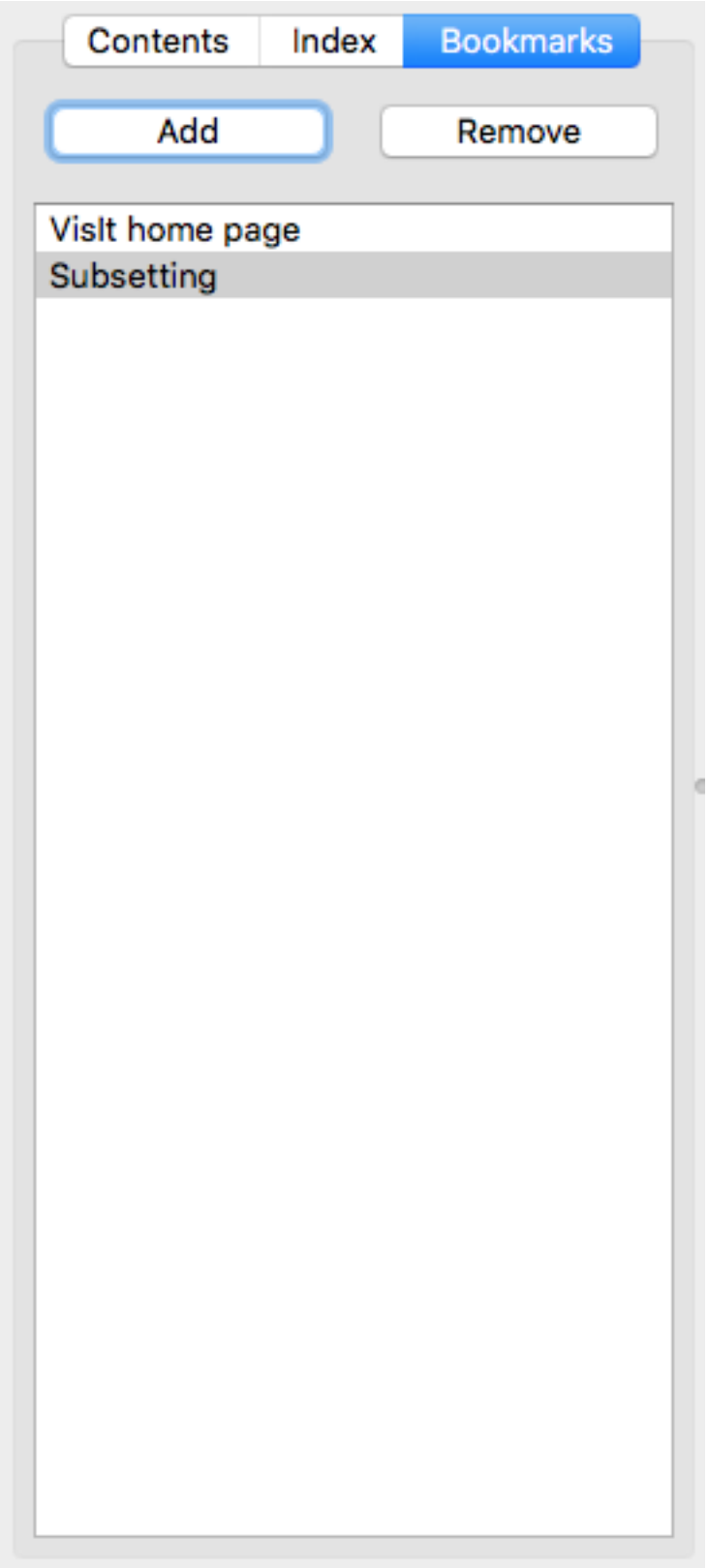


Fig. 4.458: Help tabs

Contents tab

The **Contents** tab lists all of the online help pages and groups them into related topics which are sometimes organized in tree format. When items are organized into a tree, an entry in the list of help pages often has a book icon next to it indicating that the topic contains other help topics. When an item has a book icon, it can be opened by double-clicking on its title or by clicking the check box to the left of the title. Items that have an icon that looks like a stack of papers contain the actual help content and clicking on them displays the help page in the right half of the **Help Window**.

Index tab

The **Index** tab lists all of the help topics alphabetically in a single searchable list. Help topics can be selected by clicking on an item in the list or by typing a help topic into the text field above the list. As words are typed into the text field, the closest match is found in the list of help topics and the topic is displayed in the right half of the **Help Window**.

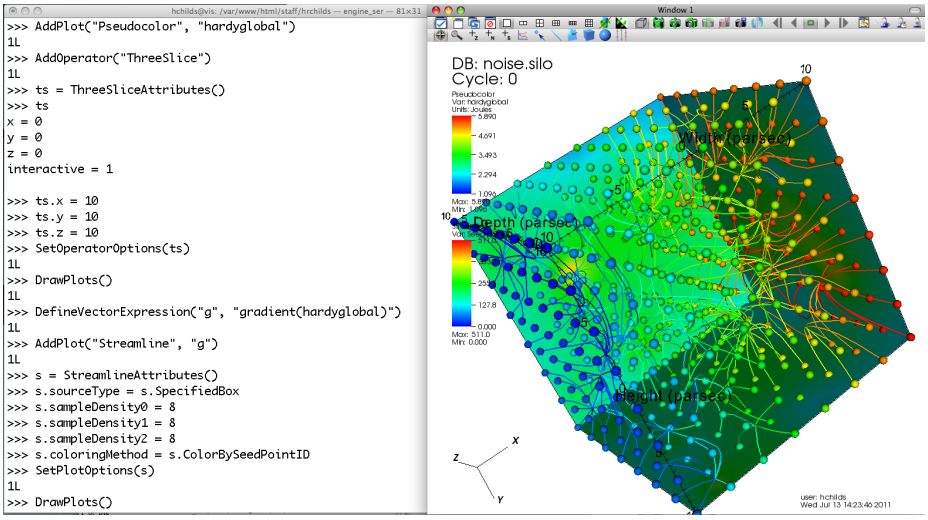
Bookmarks tab

The **Bookmarks** tab lists all of the help topics that have been bookmarked for further use. To view a page that has been previously bookmarked, simply click on its title in the bookmark list. To add a bookmark for the current help page, click the **Add** button in the **Bookmarks** tab or in the **Help Window's** toolbar. To remove a bookmark, click on its title in the bookmark list and then click the **Remove** button.

CHAPTER

FIVE

PYTHON SCRIPTING



5.1 Introduction to Python Scripting

5.1.1 Starting VisIt's Python Interface

You can invoke VisIt's Python scripting interface from the command line by typing:

```
visit -cli
```

VisIt provides a separate Python module if you instead wish to include VisIt functions in an existing Python script. In that case, you must first import the VisIt module into Python and then call the `Launch()` function to make VisIt launch and dynamically load the rest of the VisIt functions into the Python namespace. VisIt adopts this somewhat unusual approach to module loading since the lightweight front-end module, named `visit`, can be installed as one of your Python's site packages yet still dynamically load the real control functions from different versions of VisIt selected by the user.

If you do not install the `visit` module as a Python site package, you can tell the Python interpreter where it is located either by appending a new path to the `sys.path` variable as in

```
import sys
sys.path.append("/path/to/visit/<version>/<architecture>/lib/site-packages")
```

or by setting the `PYTHONPATH` environment variable as in

```
env PYTHONPATH=/path/to/visit/<version>/<architecture>/lib/site-packages ./myscript.py
```

Here is how to import all functions into the global Python namespace:

```
from visit import *
Launch()
```

Here is how to import all functions into a "visit" module namespace:

```
import visit
visit.Launch()
import visit
```

If you are running VisIt at an HPC center where multiple versions of VisIt are installed, the default version of VisIt may not always match the version you expect. To avoid confusion, you should use:

```
import visit
visit.AddArgument("-v")
visit.AddArgument("<version>") # for example: "3.2.0"
visit.Launch()
import visit
```

5.1.2 Python 3 vs Python 2

Python 2 has reached end of life and Python 3 is now preferred. VisIt was ported to use Python 3 as part of VisIt's 3.2 release. Some Python 2 syntax and common patterns no longer work in Python 3.

For example, this is no longer valid in Python 3:

```
print "Hello from VisIt"
```

In Python 3 you must call `print` like a function:

```
print("Hello from VisIt")
```

Since many VisIt scripts in the wild are written for Python 2 we provide limited on-the-fly support to convert Python 2 style scripts to valid Python 3 and execute them. The command line option `-py2to3` enables this automatic conversion logic.

When `-py2to3` is used, VisIt will attempt to convert the input script passed with `-s` and any scripts run using `visit.Source()` on-the-fly. For example, if you create script called `hello_visit.py` that includes the Python 2 style print above and run it as follows:

```
visit -nowin -cli -py2to3 -s hello_visit.py
```

On-the-fly conversion and execution will succeed and you will see:

```
Running: cli -dv -nowin -py2to3 -s hello_visit.py
VisIt CLI: Automatic Python 2to3 Conversion Enabled
Running: viewer -dv -nowin -noint -host 127.0.0.1 -port 5600
Hello from VisIt
```

You can also toggle this support in VisIt's CLI using:

```
visit_utils.builtin.SetAutoPy2to3(True) # or False
```

You can check the current value using:

```
visit_utils.builtin.GetAutoPy2to3()
```

We want to emphasize the limited aspect of the automatic support. The best long term path is to port your Python 2 style scripts to Python 3.

Python 3 installs provide a utility called `2to3` that you can use to help automate porting, see https://docs.python.org/3/library/2to3.htm for more details.

If you need help porting your trusty (or favorite) VisIt script, please reach out to the VisIt team.

5.1.3 Mixing and Matching Python Extension Modules

Danger: Mixing and matching independently compiled Python extension modules can result in subtle and hard to diagnose failures.

Care must be taken when combining a variety of Python modules especially if any are extension modules and not *pure python*. A pure python module is one that is written *entirely* in Python and is highly portable. Most python modules involve a combination of compiled C/C++/Fortran code wrapped with a small amount of Python. These are less portable. When these kinds of modules are used, a number of additional factors impact their ability to be combined in a single Python script. These include

- The Python library (header files) used to compile the module.
- The compiler used to compile the module.
- The compiler used to compile the Python interpreter where the modules are being combined.

It is a best practice to ensure that all modules being combined are compiled with the same compiler and the same Python library. However, each team supporting installations of a given Python module on a given platform makes their own decisions regarding these choices. Consequently, when using combinations of Python modules installed by others, its very easy to encounter situations where the installations are incompatible and fail in subtle and hard to diagnose ways. Worse, things may work for the most part and only intermittently produce invalid results with no warning.

5.1.4 Getting started

Typically, one of the first things you do with VisIt is open a database and draw a plot. Here is a simple example of opening a database, adding a "Pseudocolor" plot and drawing it.

```
OpenDatabase("/usr/local/visit/data/multi_curv3d.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
```

To see a list of the available plots and operators when you use the Python scripting interface, use the Operator Plugins and Plot Plugins functions. Each of those functions returns a tuple of strings that contain the names of the currently loaded plot or operator plugins. Each plot and operator plugin provides a function for creating an attributes object to set the plot or operator attributes. The name of the function is the name of the plugin in the tuple returned by the OperatorPlugins or PlotPlugins functions plus the word "Attributes". For example, the "Pseudocolor" plot provides a function called PseudocolorAttributes. To set the plot attributes or the operator attributes, first use the attributes creation function to create an attributes object. Assign the newly created object to a variable name and set the fields in the object. Each object has its own set of fields. To see the available fields in an object, print the name of the variable at the Python prompt and press the Enter key. This will print the contents of the object so you can see the

fields contained by the object. After setting the appropriate fields, pass the object to either the SetPlotOptions function or the SetOperatorAttributes function.

Example:

```
OpenDatabase("/usr/local/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddOperator("Slice")
p = PseudocolorAttributes()
p.colorTableName = "rainbow"
p.opacity = 0.5
SetPlotOptions(p)
a = SliceAttributes()
a.originType = a.Point
a.normal, a.upAxis = (1,1,1), (-1,1,-1)
SetOperatorOptions(a)
DrawPlots()
```

That's all there is to creating a plot using VisIt's Python Interface. For more information on creating plots and performing specific actions in VisIt, refer to the documentation for each function later in this manual.

5.2 Python

5.2.1 Overview

Python is a general purpose, interpreted, extensible, object-oriented scripting language that was chosen for VisIt's scripting language due to its ease of use and flexibility. VisIt's Python interface was implemented as Python module and it allows you to enhance your Python scripts with coding to control VisIt. This chapter explains some of Python's syntax so it will be more familiar when you examine the examples found in this document. For more information on programming in Python, there are a number of good references, including on the Internet at http://www.python.org.

5.2.2 Indentation

One of the most obvious features of Python is its use of indentation for new scopes. You must take special care to indent all program logic consistently or else the Python interpreter may halt with an error, or worse, not do what you intended. You must increase indentation levels when you define a function, use an if/elif/else statement, or use any loop construct.

Note the different levels of indentation:

```
def example_function(n):
  v = 0
  if n > 2:
    print "n greater than 2."
  else:
    v = n * n
  return v
```

5.2.3 Comments

Like all good programming languages, Python supports the addition of comments in the code. Comments begin with a pound character (#) and continue to the end of the line.

```
# This is a comment
a = 5 * 5
```

5.2.4 Identifiers

The Python interpreter accepts any identifier that contains letters 'A'-'Z', 'a'-'z' and numbers '0'-'9' as long as the identifier does not begin with a number. The Python interpreter is case-sensitive so the identifier "case" would not be the same identifier as "CASE". Be sure to case consistently throughout your Python code since the Python interpreter will instantiate any identifier that it has not seen before and mixing case would cause the interpreter to use multiple identifiers and cause problems that you might not expect. Identifiers can be used to refer to any type of object since Python is flexible in its treatment of types.

5.2.5 Data types

Python supports a wide variety of data types and allows you to define your own data types readily. Most types are created from a handful of building-block types such as integers, floats, strings, tuples, lists, and dictionaries.

Strings

Python has built-in support for strings and you can create them using single quotes or double quotes. You can even use both types of quotes so you can create strings that include quotes in case quotes are desired in the output. Strings are sequence objects and support operations that can break them down into characters.

```
s = 'using single quotes'
s2 = "using double quotes"
s3 = 'nesting the "spiffy" double quotes'
```

Tuples

Python supports tuples, which can be thought of as a read-only set of objects. The members of a tuple can be of different types. Tuples are commonly used to group multiple related items into a single object that can be passed around more easily. Tuples support a number of operations. You can subscript a tuple like an array to access its individual members. You can easily determine whether an object is a member of a tuple. You can iterate over a tuple. There are many more uses for tuples. You can create tuples by enclosing a comma-separated list of objects in parenthesis.

```
# Create a tuple
a = (1,2,3,4,5)
print "The first value in a is:", a[0]
# See if 3 is in a using the "in" operator.
print "3 is in a:", 3 in a
# Create another tuple and add it to the first one to create c.
b = (6,7,8,9)
c = a + b
# Iterate over the items in the tuple
for value in c:
  print "value is: ", value
```

Lists

Lists are just like tuples except they are not read-only and they use square brackets [] to enclose the items in the list instead of using parenthesis.

```
# Start with an empty list.
L = []
for i in range(10):
  # Add i to the list L
  L = L + [i]
print L
# Assign a value into element 6
L[5] = 1000
print L
```

Dictionaries

Dictionaries are Python containers that allow you to store a value that is associated with a key. Dictionaries are convenient for mapping 1 set to another set since they allow you to perform easy lookups of values. Dictionaries are declared using curly braces and each item in the dictionary consists of a key: value pair with the key and values being separated by a colon. To perform a lookup using a dictionary, provide the key whose value you want to look up to the subscript [] operator.

```
colors = {"red" : "rouge", "orange" : "orange", \
"yellow" : "jaune", "green" : "vert", "blue" : "bleu"}
# Perform lookups using the keys.
for c in colors.keys():
   print "%s in French is: %s" % (c, colors[c])
```

NumPy Arrays

VisIt is often used together with NumPy. It is natural to want to use NumPy arrays in assignments to members of VisIt attribute and dictionary objects or as arguments to functions. For the most part, NumPy arrays do work in assignment to members of attribute objects or via setter functions on attribute objects. In cases where entries in a NumPy array are to be interpreted as arguments to a function, it is possible to *dereference* a NumPy array by preceding it with $\star$. For example, this does not work

```
NumPyArray = numpy.array([1.1,2.2,3.3,4.4])
va = VolumeAttributes()
va.SetMaterialProperties(NumPyArray)
```

whereas this does

```
NumPyArray = numpy.array([1.1,2.2,3.3,4.4])
va = VolumeAttributes()
va.SetMaterialProperties(*NumPyArray)
```

as well as this

```
NumPyArray = numpy.array([1.1,2.2,3.3,4.4])
va = VolumeAttributes()
va.materialProperties = NumPyArray
```

Whenever passing NumPy arrays to VisIt is causing problems, it is possible to *convert* them to *list* objects using the `.tolist()` method as in

```
NumPyArray = numpy.array([1.1,2.2,3.3,4.4])
va = VolumeAttributes()
va.SetMaterialProperties(NumPyArray.tolist())
```

5.2.6 Control flow

Python, like other general-purpose programming languages provides keywords that implement control flow. Control flow is an important feature to have in a programming language because it allows complex behavior to be created using a minimum amount of scripting.

if/elif/else

Python provides if/elif/else for conditional branching. The if statement takes any expression that evaluates to an integer and it takes the if branch if the integer value is 1 other wise it takes the else branch if it is present.

```
# Example 1
if condition:
     do_something()

# Example 2
if condition:
     do_something()
else:
     do_something_else()

# Example 3
if condition:
     do_domething()
elif conditionn:
     do_something_n()
else:
     do_something_else()
```

For loop

Python provides a for loop that allows you to iterate over all items stored in a sequence object (tuples, lists, strings). The body of the for loop executes once for each item in the sequence object and allows you to specify the name of an identifier to use in order to reference the current item.

```
# Iterating through the characters of a string
for c in "characters":
   print c

# Iterating through a tuple
for value in ("VisIt", "is", "coolness", "times", 100):
   print value

# Iterating through a list
for value in ["VisIt", "is", "coolness", "times", 100]:
   print value

# Iterating through a range of numbers [0,N) created with range(N).
N = 100
```

(continues on next page)

(continued from previous page)

```
for i in range(N):
   print i, i*i
```

While loop

Python provides a while loop that allows you to execute a loop body indefinitely based on some condition. The while loop can be used for iteration but can also be used to execute more complex types of loops.

```
token = get_next_token()
while token != "":
  do_something(token)
  token = get_next_token()
```

5.2.7 Functions

Python comes with many built-in functions and modules that implement additional functions. Functions can be used to execute bodies of code that are meant to be re-used. Functions can optionally take arguments and can optionally return values. Python provides the def keyword, which allows you to define a function. The def keyword is followed by the name of the function and its arguments, which should appear as a tuple next to the name of the function.

```
# Define a function with no arguments and no return value.
def my_function():
    print "my function prints this..."

# Define a function with arguments and a return value.
def n_to_the_d_power(n, d):
    value = 1
    if d > 0:
        for i in range(d):
            value = value * n
    elif d < 0:
        value = 1. / float(n_to_the_d_power(n, -d))

return value
```

5.2.8 Finding Stuff from Python Prompt

If you are having trouble finding the right functions or objects, you can use `apropos(regex)`, where regex is a regular expression string, and you will get back a list of all objects and functions whose names, doc strings or stringified instances (for objects only) match the regular expression. For example,

```
>>> apropos("icosahedron")
['SubsetAttributes', 'ScatterAttributes', 'PseudocolorAttributes',
 ↪'FilledBoundaryAttributes', 'MeshAttributes']
>>> apropos(".*mir.*")
['SetDefaultMaterialAttributes', 'MaterialAttributes', 'GetMaterialAttributes',
 ↪'SetMaterialAttributes']
>>> apropos(".*reconstruct.*")
['GetMeshManagementAttributes', 'SetDefaultMeshManagementAttributes',
 ↪'SetMeshManagementAttributes', 'SetDefaultMaterialAttributes',
 ↪'GetMaterialAttributes', 'SetMaterialAttributes']
```

`apropos()` also always does case-insensitive searches.

In Python Regular Expressions the `.*` is needed for an arbitrary number of unspecified characters. See this HOWTO for more information about Python Regular Expressions. The function name `apropos` was inspired by a function of similar name and purpose in the Unix operating system.

Just use `help()`

One drawback with using Python's *standard* `help(thing)` facility is that it requires prior knowledge of the name(s) (including correct capitalization) of the thing. Consequently, VisIt's Python environment adjusts the default behavior of `help(thing)` in a couple of ways. First, it accepts string arguments and then returns the result of `apropos(thing)`. Second, for non-string arguments it will attempt to present output from Python's *standard* help but then to also follow that up with the output of `apropos(thing)`. For example, `help("materials")` (e.g. passing a string to `help()`) produces...

```
>>> help("materials")
NOTE: The following VisIt functions and objects also mention 'materials'...
['GetActiveTimeSlider', 'GetMaterialAttributes', 'GetMaterials', 'ListDomains',
↪'ListMaterials', 'MaterialAttributes', 'SetViewExtentsType', 'TurnDomainsOff',
↪'TurnDomainsOn', 'TurnMaterialsOff', 'TurnMaterialsOn']
```

whereas passing the name of a function such as `TurnMaterialsOn` produces the help for that function followed by a list of other items that mention that funciton...

```
>>> help(TurnMaterialsOn)
Help on built-in function TurnMaterialsOn:


TurnMaterialsOn(...)
    TurnMaterialsOn

.
.
.

NOTE: The following VisIt functions and objects also mention 'TurnMaterialsOn'...
['TurnDomainsOff', 'TurnDomainsOn', 'TurnMaterialsOff', 'TurnMaterialsOn']
```

For another example,

```
>>> help("copy")

Help on module copy:

NAME
    copy - Generic (shallow and deep) copying operations.

MODULE REFERENCE
    https://docs.python.org/3.7/library/copy

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.
```

(continues on next page)

(continued from previous page)

```
DESCRIPTION
    Interface summary:

             import copy

             x = copy.copy(y)         # make a shallow copy of y
             x = copy.deepcopy(y)     # make a deep copy of y

    For module specific errors, copy.Error is raised.

.
.
.

NOTE: The following VisIt functions and objects also mention 'copy'...
['CopyAnnotationsToWindow', 'CopyLightingToWindow', 'CopyPlotsToWindow',
↪'CopyViewToWindow', 'GetPlotList', 'InitializeNamedSelectionVariables']
```

If there is a need to bypass VisIt's override of `help()`, `python_help()` is an alias for Python's *default* `help()`. Likewise, `visit_help()` is an alias for VisIt's overridden `help()`.

Use lsearch to limit search results

VisIt's *CLI* provides a large set of functions. The scope of a search can be limited using the `lsearch()` helper function in the visit_utils module:

```
from visit_utils.common import lsearch
lsearch(dir(),"Material")
lsearch(apropos("subset"),"domain"))
```

lsearch() returns a python list of strings with the names that match the given pattern. Here is another example that prints each of the result strings on a separate line.

```
from visit_utils.common import lsearch
for value in lsearch(dir(),"Material"):
    print value
```

5.3 Quick Recipes

5.3.1 Overview

This manual contains documentation for over two hundred functions and several dozen extension object types. Learning to combine the right functions in order to accomplish a visualization task without guidance would involve hours of trial and error. To maximize productivity and start creating visualizations using Visit's Python Interface as fast as possible, this chapter provides some common patterns, or "quick recipes" that you can combine to quickly create complex scripts.

5.3.2 How to start

The most important question when developing a script is: "Where do I start?". You can either use session files that you used to save the state of your visualization to initialize the plots before you start scripting or you can script every

aspect of plot initialization.

Using session files

VisIt's session files contain all of the information required to recreate plots that have been set up in previous interactive VisIt sessions. Since session files contain all of the information about plots, etc., they are natural candidates to make scripting easier since they can be used to do the hard part of setting up the complex visualization, leaving the bulk of the script to animate through time or alter the plots in some way. To use session files within a script, use the RestoreSession function.

```
# Import a session file from the current working directory.
RestoreSession('/home/juan/visit/visit.session', 0)
# Now that VisIt has restored the session, animate through time.
for state in range(TimeSliderGetNStates()):
  TimeSliderSetState(state)
  SaveWindow()
```

Getting something on the screen

If you don't want to use a session file to begin the setup for your visualization then you will have to dive into opening databases, creating plots, and animating through time. This is where all of hand-crafted scripts begin. The first step in creating a visualization is opening a database. VisIt provides the OpenDatabase function to open a database. Once a database has been opened, you can create plots from its variables using the AddPlot function. The AddPlot function takes a plot plugin name and the name of a variable from the open database. Once you've added a plot, it is in the new state, which means that it has not yet been submitted to the compute engine for processing. To make sure that the plot gets drawn, call the DrawPlots function.

```
# Step 1: Open a database
OpenDatabase('~juanita/silo/stuff/wave.visit')

# Step 2: Add plots with default properties
AddPlot('Pseudocolor', 'pressure')
AddPlot('Mesh', 'quadmesh')

# Step 3: Draw the plots with default view
DrawPlots()

# Step 4: Iterate through time and save images
for state in range(0,TimeSliderGetNStates(),10):
  TimeSliderSetState(state)
  SaveWindow()
```

5.3.3 Using VisIt with the system Python

There are situations where you may want to import the VisIt module into the system Python. Some common use cases are using VisIt as part of a larger Python workflow or when you need to use a Python module that VisIt's Python does not include. You should always try to use VisIt's Python interpreter directly, since importing VisIt's Python module may not always work.

When importing the VisIt module into the system Python, at a minimum the major version numbers must match and ideally the major and minor version numbers would match. In general, there are three things you must do to import the VisIt module into the system Python.

1. Tell the Python interpreter where the standard C++ library used to compile VisIt is located. This needs to be done before any modules other than *ctypes* are imported.
2. Tell the Python interpreter where the VisIt module is located.
3. Specify the version of VisIt you are using if you have multiple versions of VisIt installed in the same directory.

Not all of the steps are necessary. For example, if VisIt was compiled with the default system compiler then you do not need to perform the first step. It is important that the steps are done in the order specified above.

In this example VisIt is imported into the system Python and used to save an image from one of our sample datasets. The paths specified for the location of the standard C++ library and the VisIt module will need to be changed as appropriate for your system.

```
python3
Python 3.7.2 (default, Feb 26 2019, 08:59:10)
[GCC 4.9.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ctypes
>>> ctypes.cdll.LoadLibrary('/usr/tce/packages/gcc/gcc-7.3.0/lib64/libstdc++.so.6')
<CDLL '/usr/tce/packages/gcc/gcc-7.3.0/lib64/libstdc++.so.6', handle 6d3e30 at␣
↪0x2aaaac14b7f0>
>>> import sys
>>> sys.path.append("/usr/gapps/visit/3.3.3/linux-x86_64/lib/site-packages/")
>>> import visit
>>> visit.AddArgument("-v")
>>> visit.AddArgument("3.3.3")
>>> visit.LaunchNowin()
Running: viewer3.3.3 -nowin -forceversion 3.3.3 -noint -host 127.0.0.1 -port 5601
True
>>> import visit
>>> visit.OpenDatabase("/usr/gapps/visit/data/noise.silo")
Running: mdserver3.3.3 -forceversion 3.3.3 -host 127.0.0.1 -port 5601
Running: engine_ser3.3.3 -forceversion 3.3.3 -dir /usr/gapps/visit -idle-timeout 480 -
↪host 127.0.0.1 -port 5601
1
>>> visit.AddPlot("Pseudocolor", "hardyglobal")
1
>>> visit.DrawPlots()
1
>>> visit.SaveWindow()
VisIt: Message - Rendering window 1...
VisIt: Message - Saving window 1...
VisIt: Message - Saved visit0000.png
'visit0000.png'
>>> quit()
```

Sometimes telling Python where the standard C++ library used to compile VisIt is located does not work and instead you can tell it where VisIt's Python library is located.

```
python3
Python 3.9.12 (main, Apr 15 2022, 09:20:22)
[GCC 10.3.1 20210422 (Red Hat 10.3.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ctypes
>>> ctypes.CDLL('/usr/gapps/visit/3.3.3/linux-x86_64-toss4/lib/libpython3.7m.so')
<CDLL '/usr/gapps/visit/3.3.3/linux-x86_64-toss4/lib/libpython3.7m.so', handle 508b30␣
↪at 0x155546edf4f0>
>>> import sys
```

(continues on next page)

(continued from previous page)

```
>>> sys.path.append("/usr/gapps/visit/3.3.3/linux-x86_64-toss4/lib/site-packages")
>>> import visit
>>> visit.AddArgument("-v")
>>> visit.AddArgument("3.3.3")
>>> visit.LaunchNowin()
Running: viewer3.3.3 -nowin -forceversion 3.3.3 -noint -host 127.0.0.1 -port 5600
True
>>> import visit
>>> visit.OpenDatabase("/usr/gapps/visit/data/noise.silo")
Running: mdserver3.3.3 -forceversion 3.3.3 -host 127.0.0.1 -port 5600
Running: engine_ser3.3.3 -forceversion 3.3.3 -dir /usr/gapps/visit -idle-timeout 480 -
↪host 127.0.0.1 -port 5600
1
>>> visit.AddPlot("Pseudocolor", "hardyglobal")
1
>>> visit.DrawPlots()
1
>>> visit.SaveWindow()
VisIt: Message - Rendering window 1...
VisIt: Message - Saving window 1...
VisIt: Message - Saved visit0000.png
'visit0000.png'
>>> quit()
```

5.3.4 Handling Command line arguments

In some cases, a VisIt python script also needs to handle its own command line arguments. This is handled using the `Argv()` method. For example, to run the python script, `myscript.py` with two arguments like so

```
visit -nowin -cli -s myscript.py myarg1 myarg2
```

these arguments can be accessed using the `Argv()` method which returns the Python tuple, `('myarg1', 'myarg2')`. Similarly, `sys.argv` will return the Python list, `['myscript.py', 'myarg1', 'myarg2']` which includes the script name itself as the zeroth argument.

5.3.5 Saving images

Much of the time, the entire purpose of using VisIt's Python Interface is to create a script that can save out images of a time-varying database for the purpose of making movies. Saving images using VisIt's Python Interface is a straight-forward process, involving just a few functions.

Setting the output image characteristics

VisIt provides a number of options for saving files, including: format, fileName, and image width and height, to name a few. These attributes are grouped into the SaveWindowAttributes object. To set the options that VisIt uses to save files, you must create a SaveWindowAttributes object, change the necessary attributes, and call the SetSaveWindowAttributes function. Note that if you want to create images using a specific image resolution, the best way is to use the *-geometry* command line argument with VisIt's Command Line Interface and tell VisIt to use screen capture. If you instead require your script to be capable of saving several different image sizes then you can turn off screen capture and set the image resolution in the SaveWindowAttributes object.

```
# Prepare to save a BMP file at 1024x768 resolution
s = SaveWindowAttributes()
s.format = s.BMP
s.fileName = 'mybmpfile'
s.width, s.height = 1024,768
s.screenCapture = 0
SetSaveWindowAttributes(s)
# Subsequent calls to SaveWindow() will use these settings
```

Saving an image

Once you have set the SaveWindowAttributes to your liking, you can call the SaveWindow function to save an image. The SaveWindow function returns the name of the image that is saved so you can use that for other purposes in your script.

```
# Save images of all timesteps and add each image filename to a list.
names = []
for state in range(TimeSliderGetNStates()):
  SetTimeSliderState(state)
  # Save the image
  n = SaveWindow()
  names = names + [n]
print(names)
```

5.3.6 Working with databases

VisIt allows you to open a wide array of databases both in terms of supported file formats and in terms how databases treat time. Databases can have a single time state or can have multiple time states. Databases can natively support multiple time states or sets of single time states files can be grouped into time-varying databases using .visit files or using virtual databases. Working with databases gets even trickier if you are using VisIt to visualize a database that is still being generated by a simulation. This section describes how to interact with databases.

Opening a database

Opening a database is a relatively simple operation - most complexities arise in how the database treats time. If you only want to visualize a single time state or if your database format natively supports multiple timestates per file then opening a database requires just a single call to the OpenDatabase function.

```
# Open a database (no time specified defaults to time state 0)
OpenDatabase("/Users/amina/data/pdb_test_data/allinone00.pdb")
```

Opening a database at specific time

Opening a database at a later timestate is done just the same as opening a database at time state zero except that you must specify the time state at which you want to open the database. There are a number of reasons for opening a database at a later time state. The most common reason for doing so, as opposed to just changing time states later, is that VisIt uses the metadata from the first opened time state to describe the contents of the database for all timestates (except for certain file formats that don't do this, i.e. SAMRAI). This means that the list of variables found for the first time state that you open is used for all timestates. If your database contains a variable at a later timestate that does not exist at earlier time states, you must open the database at a later time state to gain access to the transient variable.

```
# Open a database at a specific time state to pick up transient variables
OpenDatabase("/Users/amina/data/pdb_test_data/allinone00.pdb", 17)
```

Opening a virtual database

VisIt provides two ways for accessing a set of single time-state files as a single time- varying database. The first method is a .visit file, which is a simple text file that contains the names of each file to be used as a time state in the time-varying database. The second method uses "virtual databases", which allow VisIt to exploit the file naming conventions that are often employed by simulation codes when they create their dumps. In many cases, VisIt can scan a specified directory and determine which filenames look related. Filenames with close matches are grouped as individual time states into a virtual database whose name is based on the more abstract pattern used to create the filenames.

```
# Opening just one file, the first, in series wave0000.silo, wave0010.silo, ...
OpenDatabase("~juanita/silo/stuff/wave0000.silo")

# Opening a virtual database representing all available states.
OpenDatabase("~juanita/silo/stuff/wave*.silo database")
```

Opening a remote database

VisIt supports running the client on a local computer while also allowing you to process data in parallel on a remote computer. If you want to access databases on a remote computer using VisIt's Python Interface, the only difference to accessing a database on a local computer is that you must specify a host name as part of the database name.

```
# Opening a file on a remote computer by giving a host name
# Also, open the database to a later time slice (17)
OpenDatabase("thunder:~juanita/silo/stuff/wave.visit", 17)
```

5.3.7 Opening a compute engine

Sometimes it is advantageous to open a compute engine before opening a database. When you tell VisIt to open a database using the OpenDatabase function, VisIt also launches a compute engine and tells the compute engine to open the specified database. When the VisIt Python Interface is run with a visible window, the **Engine Chooser Window** will present itself so you can select a host profile. If you want to design a script that must specify parallel options, etc in batch mode where there is no **Engine ChooserWindow** then you have few options other than to open a compute engine before opening a database. To open a compute engine, use the OpenComputeEngine function. You can pass the name of the host on which to run the compute engine and any arguments that must be used to launch the engine such as the number of processors.

```
# Open a local, parallel compute engine before opening a database
# Use 4 processors on 2 nodes
OpenComputeEngine("localhost", ("-np", "4", "-nn", "2"))
OpenDatabase("~juanita/silo/stuff/multi_ucd3d.silo")
```

The options for starting the compute engine are the same as the ones used on the command line. Here are the most common options for launching a compute engine.

```
-l    <method>       Launch in parallel using the given method.
-np   <# procs>      The number of processors to use.
-nn   <# nodes>      The number of nodes to allocate.
```

(continues on next page)

(continued from previous page)

```
-p    <part>          Partition to run in.
-b    <bank>          Bank from which to draw resources.
-t    <time>          Maximum job run time.
-machinefile <file>   Machine file.
```

The full list of parallel launch options can be obtained by typing `visit --fullhelp`. Here is a more complex example of launching a compute engine.

```
# Use the "srun" job launcher, the "batch" partition, the "mybank" bank,
# 72 processors on 2 nodes and a time limit of 1 hour
OpenComputeEngine("localhost",
    ("-l", "srun", "-p", "batch", "-b", "mybank",
     "-np", "72", "-nn", "2", "-t", "1:00:00"))
```

You can also launch a compute engine using one of the existing host profiles defined for your system. In this particular case we know that the second profile is for the "parallel" profile. If you didn't know this you could print "p" to get all the properties.

```
# Set the user name to "user1" and use the 2nd profile,
# (indexed by 1 from zero) overriding a few of its properties
p = GetMachineProfile("quartz.llnl.gov")
p.userName="user1"
p.activeProfile = 1
p.GetLaunchProfiles(1).numProcessors = 6
p.GetLaunchProfiles(1).numNodes = 2
p.GetLaunchProfiles(1).timeLimit = "00:30:00"
OpenComputeEngine(p)
```

5.3.8 Working with plots

Plots are viewable objects, created from a database, that can be displayed in a visualization window. VisIt provides several types of plots and each plot allows you to view data using different visualization techniques. For example, the Pseudocolor plot allows you to see the general shape of a simulated object while painting colors on it according to the values stored in a variable's scalar field. The most important functions for interacting with plots are covered in this section.

Creating a plot

The function for adding a plot in VisIt is: AddPlot. The AddPlot function takes the name of a plot type and the name of a variable that is to be plotted and creates a new plot and adds it to the plot list. The name of the plot to be created corresponds to the name of one of VisIt's plot plugins, which can be queried using the PlotPlugins function. The variable that you pass to the AddPlot function must be a valid variable for the opened database. New plots are not realized, meaning that they have not been submitted to the compute engine for processing. If you want to force VisIt to process the new plot you must call the DrawPlots function.

```
# Names of all available plot plugins as a python tuple
x = PlotPlugins()

# print(x) will produce something like...
#     ('Boundary', 'Contour', 'Curve', 'FilledBoundary', 'Histogram',
#     'Label', 'Mesh', 'Molecule', 'MultiCurve', 'ParallelCoordinates',
#     'Pseudocolor', 'Scatter', 'Spreadsheet', 'Subset', 'Tensor',
#     'Truecolor', 'Vector', 'Volume')
```

(continues on next page)

(continued from previous page)

```
print(x)

# Create plots with AddPlot(<plot-plugin-name>,<database-variable-name>)
AddPlot("Pseudocolor", "pressure")
AddPlot("Mesh", "quadmesh")

# Draw the plots
DrawPlots()
```

Plotting materials

Plotting materials is a common operation in VisIt. The Boundary and FilledBoundary plots enable you to plot material boundaries and materials, respectively.

```
# Plot material boundaries
AddPlot("Boundary", "mat1")
# Plot materials
AddPlot("FilledBoundary", "mat1")
```

Setting plot attributes

Each plot type has an attributes object that controls how the plot generates its data or how it looks in the visualization window. The attributes object for each plot contains different fields. You can view the individual object fields by printing the object to the console. Each plot type provides a function that creates a new instance of one of its attribute objects. The function name is always of the form: plotname + "Attributes". For example, the attributes object creation function for the Pseudocolor plot would be: PseudocolorAttributes. To change the attributes for a plot, you create an attributes object using the appropriate function, set the properties in the returned object, and tell VisIt to use the new plot attributes by passing the object to the SetPlotOptions function. Note that you should set a plot's attributes before calling the DrawPlots method to realize the plot since setting a plot's attributes can cause the compute engine to recalculate the plot.

```
# Creating a Pseudocolor plot and setting min/max values.
AddPlot("Pseudocolor", "pressure")
p = PseudocolorAttributes()

# print p to find the names of members you want to change
#
# print(p) will produce output somewhat like...
#     scaling = Linear  # Linear, Log, Skew
#     skewFactor = 1
#     limitsMode = OriginalData  # OriginalData, ActualData
#     minFlag = 0
#     min = 0
#     useBelowMinColor = 0
#     belowMinColor = (0, 0, 0, 255)
#     maxFlag = 0
#     max = 1
#     .
#     .
#     .
print(p)

# Set the min/max values
```

(continues on next page)

(continued from previous page)

```
p.min, p.minFlag = 0.0, 1
p.max, p.maxFlag = 10.0, 1
SetPlotOptions(p)
```

Working with multiple plots

When you work with more than one plot, it is sometimes necessary to set the active plots because some of VisIt's functions apply to all of the active plots. The active plot is usually the last plot that was created unless you've changed the list of active plots. Changing which plots are active is useful when you want to delete or hide certain plots or set their plot attributes independently. When you want to set which plots are active, use the SetActivePlots function. If you want to list the plots that you've created, call the ListPlots function.

```
# Create more than 1 plot of the same type
AddPlot("Pseudocolor", "pressure")
AddPlot("Pseudocolor", "temperature")

# List the plots. The second plot should be active.
ListPlots()

# The output from ListPlots() will look something like...
#     Plot[0]|id=5;type="Pseudocolor";database="localhost:/Users/miller86/visit/visit/
↪data/silo_hdf5_test_data/tire.silo";
#         var=pressure;active=0;hidden=0;framerange=(0, 0);keyframes={0};database␣
↪keyframes={0};operators={};
#         activeOperator=-1
#     Plot[1]|id=6;type="Pseudocolor";database="localhost:/Users/miller86/visit/visit/
↪data/silo_hdf5_test_data/tire.silo";
#         var=temperature;active=1;hidden=0;framerange=(0, 0);keyframes={0};database␣
↪keyframes={0};operators={};
#         activeOperator=-1
# Note that active=1 for Plot[1] meaning plot #1 is the active plot

# Draw the plots
DrawPlots()

# Hide the first plot
SetActivePlots(0) # makes plot 0 the active plot
HideActivePlots()

# Set both plots' color table to "hot"
p = PseudocolorAttributes()
p.colorTableName = "hot"
SetActivePlots((0,1)) # makes both plots active
SetPlotOptions(p)

# Show the first plot again.
SetActivePlots(0)
HideActivePlots()

# Delete the second plot
SetActivePlots(1)
DeleteActivePlots()
ListPlots()
```

Plots in the error state

When VisIt's compute engine cannot process a plot, the plot is put into the error state. Once a plot is in the error state, it no longer is displayed in the visualization window. If you are generating a movie, plots entering the error state can be a serious problem because you most often want all of the plots that you have created to animate through time and not disappear in the middle of the animation. You can add extra code to your script to prevent plots from disappearing (most of the time) due to error conditions by adding a call to the DrawPlots function.

```
# Open the database at state 20 and add plots for "pressure" and "transient".
# "transient" variable exists only in states 18...51.
OpenDatabase(silo_data_path("wave.visit"),20)
AddPlot("Pseudocolor","pressure")
AddPlot("Pseudocolor","transient")
DrawPlots()

# Start saving images from every 10th state starting at state 20
# but take care to clean up when we get an error.
for state in range(20,TimeSliderGetNStates(),10):

    TimeSliderSetState(state)

    if DrawPlots() == 0:

        # Find plot(s) in error state and remove them
        pl = GetPlotList()
        for i in range(pl.GetNumPlots()):
            if pl.GetPlots(i).stateType == pl.GetPlots(i).Error:
                SetActivePlots((i,))
                DeleteActivePlots()

        # Clear the last error message
        GetLastError(1)

    SaveWindow()
```

5.3.9 Operators

Operators are filters that are applied to database variables before the compute engine uses them to create plots. Operators can be linked one after the other to form chains of operators that can drastically transform the data before plotting it.

Adding operators

The list of available operators is returned by the `OperatorPlugins()` function. Any of the names returned from `OperatorPlugins()` can be used to add an operator using the `AddOperator()` function. Adding an operator is similar to adding a plot in that you call a function with the name of the operator to be added. Operators are added *only* to the *active* plots by default but you can also force VisIt to add them to all plots in the plot list. You can also use the `SetActivePlots()` function to adjust the list of active plots before adding an operator to exercise finer grained control over which plots an operator is added to.

```
# Names of all available operator plugins as a python tuple
x = OperatorPlugins()

print(x)
```

(continues on next page)

(continued from previous page)

```
# will produce output something like...
# ('AMRStitchCell', 'AxisAlignedSlice4D', 'BoundaryOp', 'Box', 'CartographicProjection
↪',
#  'Clip', 'Cone', 'ConnectedComponents', 'CoordSwap', 'CreateBonds', 'Cylinder',
#  'DataBinning', 'DeferExpression', 'Displace', 'DualMesh', 'Edge', 'Elevate',
#  'EllipsoidSlice', 'Explode', 'ExternalSurface', ...
#  ..., 'TriangulateRegularPoints', 'Tube')

# We need at least one plot that we can add operators to
AddPlot("Pseudocolor", "dx")
AddPlot("Mesh","mesh1")

# Add Isovolume and Slice operators using whatever their default attributes are.
# The non-zero 2nd arg means to add the operator to all plots. If the 2nd argument
# is not present or zero, it means to add the operator only to the *active* plots
# (by default, the *active* plots are just the last plot added).
AddOperator("Isovolume", 1)
AddOperator("Slice", 1)
DrawPlots()
```

Setting operator attributes

Each plot gets its own instance of an operator which means that you can set each plot's operator attributes independently. Like plots, operators use objects to set their attributes. These objects are returned by functions whose names are of the form: operatorname + "Attributes". Once you have created an operator attributes object, you can pass it to the SetOperatorOptions to set the options for an operator. Note that setting the attributes for an operator nearly always causes the compute engine to recalculate the operator. You can use the power of VisIt's Python Interface to create complex operator behavior such as in the following code example, which moves slice planes through a Pseudocolor plot.

```
OpenDatabase("~juanita/silo/stuff/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
AddOperator("Slice")
s = SliceAttributes()
s.originType = s.Percent
s.project2d = 0
SetOperatorOptions(s)
DrawPlots()

nSteps = 20
for axis in (0,1,2):
  s.axisType = axis
  for step in range(nSteps):
    t = float(step) / float(nSteps - 1)
    s.originPercent = t * 100.
    SetOperatorOptions(s)
    SaveWindow()
```

5.3.10 Quantitative operations

This section focuses on some of the operations that allow you to examine your data more quantitatively.

Defining expressions

VisIt allows you to create derived variables using its powerful expressions language. You can plot or query variables created using expressions just as you would if they were read from a database. VisIt's Python Interface allows you to create new scalar, vector, tensor variables using the DefineScalarExpression, DefineVectorExpression, and DefineTensorExpression functions.

```
# Creating a new expression
OpenDatabase("~juanita/silo/stuff/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
DrawPlots()
DefineScalarExpression("newvar", "sin(hardyglobal) + cos(shepardglobal)")
ChangeActivePlotsVar("newvar")
```

Pick

VisIt allows you to pick on cells, nodes, and points within a database and return information for the item of interest. To that end, VisIt provides several pick functions. Once a pick function has been called, you can call the GetPickOutput function to get a string that contains the pick information. The information in the string could be used for a multitude of uses such as building a test suite for a simulation code.

```
OpenDatabase("~juanita/silo/stuff/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()
s = []
# Pick by a node id
PickByNode(300)
s = s + [GetPickOutput()]
# Pick by a cell id
PickByZone(250)
s = s + [GetPickOutput()]
# Pick on a cell using a 3d point
Pick((-2., 2., 0.))
s = s + [GetPickOutput()]
# Pick on the node closest to (-2,2,0)
NodePick((-2,2,0))
s = s + [GetPickOutput()]
# Print all pick results
print(s)
# Will produce output somewhat like...
# ['\nA:  noise.silo\nMesh2D \nPoint: <-10, -7.55102>\nNode:...
#  '\nD:  noise.silo\nMesh2D \nPoint: <-1.83673, 1.83673>\nNode:...
#  ...\nhgslice:  <nodal> = 4.04322\n\n']
```

Lineout

VisIt allows you to extract data along a line, called a *lineout*, and plot the data using a *Curve* plot.

```
p0 = (-5,-3, 0)
p1 = ( 5, 8, 0)
OpenDatabase("~juanita/silo/stuff/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
DrawPlots()
Lineout(p0, p1)
```

(continues on next page)

(continued from previous page)

```
# Specify 65 sample points
Lineout(p0, p1, 65)
# Do three variables ("default" is "hardyglobal")
Lineout(p0, p1, ("default", "airVf", "radial"))
```

The above steps produce the lineout(s), visually, as curve plot(s) in a new viewer window. Read more about how VisIt *determines which window* it uses for these curve plot(s). What if you want to access the actual lineout data and/or save it to a file?

```
# Set active window to one containing Lineout curve plots (typically #2)
SetActiveWindow(2)
# Get array of x,y pairs for first curve plot in window
SetActivePlots(0)
hg_vals = GetPlotInformation()["Curve"]
# Get array of x,y pairs for second curve plot in window
SetActivePlots(1)
avf_vals = GetPlotInformation()["Curve"]
# Get array of x,y pairs for third curve plot in window
SetActivePlots(2)
rad_vals = GetPlotInformation()["Curve"]

# Write it as CSV data to a file
for i in range(int(len(hg_vals) / 2)):
    idx = i*2+1 # take only y-values in each array
    print("%g,%g,%g" % (hg_vals[idx], avf_vals[idx], rad_vals[idx]))
```

Query

VisIt can perform a number of different queries based on values calculated about plots or their originating database.

```
OpenDatabase("~juanita/silo/stuff/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
DrawPlots()
Query("NumNodes")
print("The float value is: %g" % GetQueryOutputValue())
Query("NumNodes")
```

Finding the min and the max

A common operation in debugging a simulation code is examining the min and max values. Here is a pattern that allows you to print out the min and the max values and their locations in the database and also see them visually.

```
# Define a helper function to get node/zone id's from query string.
def GetMinMaxIds(qstr):
    import string
    s = qstr.split(' ')
    retval = []
    nextGood = 0
    idType = 0
    for token in s:
        if token == "(zone" or token == "(cell":
            idType = 1
            nextGood = 1
```

(continues on next page)

(continued from previous page)

```
            continue
        elif token == "(node":
            idType = 0
            nextGood = 1
            continue
        if nextGood == 1:
            nextGood = 0
            retval = retval + [(idType, int(token))]
    return retval

# Set up a plot
OpenDatabase("~juanita/silo/stuff/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()
Query("MinMax")

# Do picks on the ids that were returned by MinMax.
for ids in GetMinMaxIds(GetQueryOutputString()):
    idType = ids[0]
    id = ids[1]
    if idType == 0:
        PickByNode(id)
    else:
        PickByZone(id)
```

Note that the above example parses information from the query output *string* returned from `GetQueryOutputString()`. In some cases, it will be more convenient to use `GetQueryOutputValue()` or `GetQueryOutputObject()`.

Creating a CSV file of a query over time

VisIt has the ability to perform queries over time to create one or more curves. It generates one curve per scalar value returned by the query. Frequently, the user wants to process the result of a query over time. A CSV file is a convenient way to output the data for further processing.

Here is a pattern where we loop over the time steps writing the results of a `Time` query and a `PickByNode` to a text file in the form of a CSV file.

```
OpenDatabase("~juanita/silo/stuff/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()

n_time_steps = TimeSliderGetNStates()
f = open('points.txt', 'w', encoding='utf-8')
f.write('time, x, y, z, u, v, w\n')
for time_step in range(0, n_time_steps):
    TimeSliderSetState(time_step)
    pick = PickByNode(domain=0, element=3726, vars=["u", "v", "w"])
    Query("Time")
    time = GetQueryOutputValue()
    f.write('%g, %g, %g, %g, %g, %g, %g\n' % (time, pick['point'][0], pick['point
↪'][1], pick['point'][2], pick['u'], pick['v'], pick['w']))
f.close()
```

You can substitute the `Time` and `PickByNode` queries with your favorite query, such as the `MinMax` query used in the previous quick recipe.

5.3.11 Subsetting

VisIt allows the user to turn off subsets of the visualization using a number of different methods. Databases can be divided up any number of ways: domains, materials, etc. This section provides some details on how to remove materials and domains from your visualization.

Turning off domains

VisIt's Python Interface provides the TurnDomainsOn and TurnDomainsOff functions to make it easy to turn domains on and off.

```
OpenDatabase("~juanita/silo/stuff/multi_rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()

# Turning off every other domain
d = GetDomains()
i = 0
for dom in d:
    if i%2:
        TurnDomainsOff(dom)
    i += 1

# Turn all domains off
TurnDomainsOff()

# Turn on domains 3,5,7
TurnDomainsOn((d[3], d[5], d[7]))
```

Turning off materials

VisIt's Python Interface provides the TurnMaterialsOn and TurnMaterialsOff functions to make it easy to turn materials on and off.

```
OpenDatabase("~juanita/silo/stuff/multi_rect2d.silo")
AddPlot("FilledBoundary", "mat1")
DrawPlots()
# Get the material names
GetMaterials()
# GetMaterials() will return a tuple of material names such as
#     ('1', '2', '3')
# Turn off material with name "2"
TurnMaterialsOff("2")
```

5.3.12 View

Setting up the view in your Python script is one of the most important things you can do to ensure the quality of your visualization because the view concentrates attention on an object of interest. VisIt provides different methods for setting the view, depending on the dimensionality of the plots in the visualization window but despite differences in how the view is set, the general procedure is basically the same.

Setting the 2D view

The 2D view consists of a rectangular window in 2D space and a 2D viewport in the visualization window. The window in 2D space determines what parts of the visualization you will look at while the viewport determines where the images will appear in the visualization window. It is not necessary to change the viewport most of the time.

```
OpenDatabase("~juanita/silo/stuff/noise.silo")
AddPlot("Pseudocolor", "hgslice")
AddPlot("Mesh", "Mesh2D")
AddPlot("Label", "hgslice")
DrawPlots()
print("The current view is:", GetView2D())
# Get an initialized 2D view object.
# Note that DrawPlots() must be executed prior to getting
# the view to ensure current view parameters are obtained
v = GetView2D()
v.windowCoords = (-7.67964, -3.21856, 2.66766, 7.87724)
SetView2D(v)
```

Setting the 3D view

The 3D view is much more complex than the 2D view. For information on the actual meaning of the fields in the View3DAttributes object, refer to page 214 or the VisIt User Manual. VisIt automatically computes a suitable view for 3D objects and it is best to initialize new View3DAttributes objects using the GetView3D function so most of the fields will already be initialized. The best way to get new views to use in a script is to interactively create the plot and repeatedly call GetView3D() after you finish rotating the plots with the mouse. You can paste the printed view information into your script and modify it slightly to create sophisticated view transitions.

```
OpenDatabase("~juanita/silo/stuff/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
AddPlot("Mesh", "Mesh")
DrawPlots()
# Note that DrawPlots() must be executed prior to getting
# the view to ensure current view parameters are obtained
v = GetView3D()
print("The view is: ", v)
v.viewNormal = (-0.571619, 0.405393, 0.713378)
v.viewUp = (0.308049, 0.911853, -0.271346)
SetView3D(v)
```

Flying around plots

Flying around plots is a commonly requested feature when making movies. Fortunately, this is easy to script. The basic method used for flying around plots is interpolating the view. VisIt provides a number of functions that can interpolate View2DAttributes and View3DAttributes objects. The most useful of these functions is the EvalCubicSpline function. The EvalCubicSpline function uses piece-wise cubic polynomials to smoothly interpolate between a tuple of N like items. Scripting smooth view changes using EvalCubicSpline is rather like keyframing in that you have a set of views that are mapped to some distance along the parameterized space [0., 1.]. When the parameterized space is sampled with some number of samples, VisIt calculates the view for the specified parameter value and returns a smoothly interpolated view. One benefit over keyframing, in this case, is that you can use cubic interpolation whereas VisIt's keyframing mode currently uses linear interpolation.

```
OpenDatabase("~juanita/silo/stuff/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()

# Create the control points for the views.
c0 = View3DAttributes()
c0.viewNormal = (0, 0, 1)
c0.focus = (0, 0, 0)
c0.viewUp = (0, 1, 0)
c0.viewAngle = 30
c0.parallelScale = 17.3205
c0.nearPlane = 17.3205
c0.farPlane = 81.9615
c0.perspective = 1

c1 = View3DAttributes()
c1.viewNormal = (-0.499159, 0.475135, 0.724629)
c1.focus = (0, 0, 0)
c1.viewUp = (0.196284, 0.876524, -0.439521)
c1.viewAngle = 30
c1.parallelScale = 14.0932
c1.nearPlane = 15.276
c1.farPlane = 69.917
c1.perspective = 1

c2 = View3DAttributes()
c2.viewNormal = (-0.522881, 0.831168, -0.189092)
c2.focus = (0, 0, 0)
c2.viewUp = (0.783763, 0.556011, 0.27671)
c2.viewAngle = 30
c2.parallelScale = 11.3107
c2.nearPlane = 14.8914
c2.farPlane = 59.5324
c2.perspective = 1

c3 = View3DAttributes()
c3.viewNormal = (-0.438771, 0.523661, -0.730246)
c3.focus = (0, 0, 0)
c3.viewUp = (-0.0199911, 0.80676, 0.590541)
c3.viewAngle = 30
c3.parallelScale = 8.28257
c3.nearPlane = 3.5905
c3.farPlane = 48.2315
c3.perspective = 1

c4 = View3DAttributes()
c4.viewNormal = (0.286142, -0.342802, -0.894768)
c4.focus = (0, 0, 0)
c4.viewUp = (-0.0382056, 0.928989, -0.36813)
c4.viewAngle = 30
c4.parallelScale = 10.4152
c4.nearPlane = 1.5495
c4.farPlane = 56.1905
c4.perspective = 1

c5 = View3DAttributes()
c5.viewNormal = (0.974296, -0.223599, -0.0274086)
```

(continues on next page)

(continued from previous page)

```
c5.focus = (0, 0, 0)
c5.viewUp = (0.222245, 0.97394, -0.0452541)
c5.viewAngle = 30
c5.parallelScale = 1.1052
c5.nearPlane = 24.1248
c5.farPlane = 58.7658
c5.perspective = 1

# Make the last point loop around to the first
c6 = c0

# Create a tuple of camera values and x values. The x values
# determine where in [0,1] the control points occur.
cpts = (c0, c1, c2, c3, c4, c5, c6)
x=[]
for i in range(7):
    x = x + [float(i) / float(6.)]

# Animate the view using EvalCubicSpline.
nsteps = 100
for i in range(nsteps):
    t = float(i) / float(nsteps - 1)
    c = EvalCubicSpline(t, x, cpts)
    c.nearPlane = -34.461
    c.farPlane = 34.461
    SetView3D(c)
```

5.3.13 Working with annotations

Adding annotations to your visualization improve the quality of the final visualization in that you can refine the colors that you use, add logos, or highlight features of interest in your plots. This section provides some recipes for creating annotations using scripting.

Using gradient background colors

VisIt's default white background is not necessarily the best looking background color for presentations. Adding a gradient background under your plots is an easy way to add a small professional touch to your visualizations. VisIt provides a few different styles of gradient background: radial, top to bottom, bottom to top, left to right, and right to left. The gradient style is set using the *gradientBackgroundStyle* member of the AnnotationAttributes object. The before and after results are shown in Figure 5.1.

```
# Set a blue/black, radial, gradient background.
a = AnnotationAttributes()
a.backgroundMode = a.Gradient
a.gradientBackgroundStyle = a.Radial
a.gradientColor1 = (0,0,255,255) # Blue
a.gradientColor2 = (0,0,0,255) # Black
SetAnnotationAttributes(a)
```

Adding a banner

Banners are useful for providing titles for a visualization or for marking its content (see Figure 5.2). To add an "Unclassified" banner to a visualization, use the following bit of Python code:

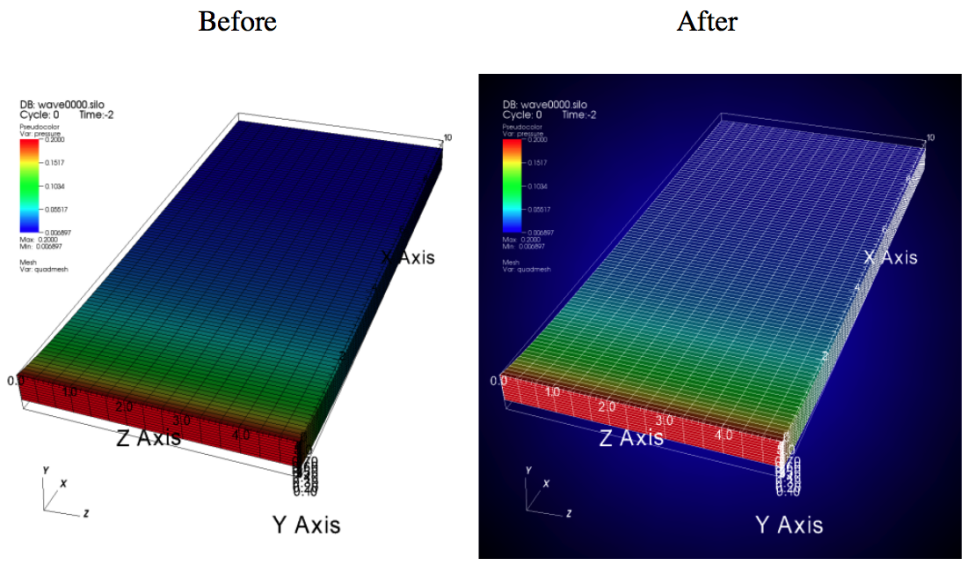


Fig. 5.1: Before and after image of adding a gradient background.

```
# Create a Text2D object to indicate the visualization is unclassified.

# Note the annoation object is added to the viewer window the moment it is created.
banner = CreateAnnotationObject("Text2D")

# Note text is updated in the viewer window the moment it is changed.
banner.text = "Unclassified"

banner.position = (0.37, 0.95)
banner.fontBold = 1

# print the attributes to see what you can set in the Text2D object.
print(banner)
# print(banner) will print something like...
#    visible = 1
#    position = (0.5, 0.5)
#    height = 0.03
#    textColor = (0, 0, 0, 255)
#    useForegroundForTextColor = 1
#    text = "2D text annotation"
#    fontFamily = Arial  # Arial, Courier, Times
#    fontBold = 0
#    fontItalic = 0
#    fontShadow = 0
```

Adding a time slider

Time sliders are important annotations for movies since they convey how much progress an animation has made as well as how many more frames have yet to be seen. The time slider is also important for showing the simulation time as the animation progresses so users can get a sense of when in the simulation important events occur. VisIt's time slider annotation object is shown in Figure 5.3.

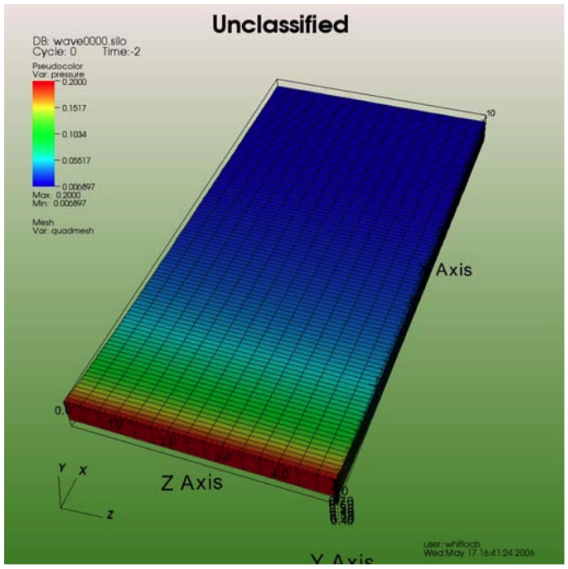


Fig. 5.2: Adding a banner

```
# Add a time slider in the lower left corner
slider = CreateAnnotationObject("TimeSlider")

# Adjust the height. Takes effect immediately as the value is assigned.
slider.height = 0.07

# Print members that are available in the time slider object
print(slider)
# will produce something like...
#    visible = 1
#    active = 1
#    position = (0.01, 0.01)
#    width = 0.4
#    height = 0.05
#    textColor = (0, 0, 0, 255)
#    useForegroundForTextColor = 1
#    startColor = (0, 255, 255, 255)
#    endColor = (255, 255, 255, 153)
#    text = "Time=$time"
#    timeFormatString = "%g"
#    timeDisplay = AllFrames  # AllFrames, FramesForPlot, StatesForPlot, UserSpecified
#    percentComplete = 0
#    rounded = 1
#    shaded = 1
```

Adding a logo

Adding a logo to a visualization is an important part of project identification for movies and other visualizations created with VisIt. If you have a logo image file stored in TIFF, JPEG, BMP, or PPM format then you can use it with VisIt as an image annotation (see Figure 5.4). Note that this approach can also be used to insert images of graphs, plots, portraits, diagrams, or any other form of image data into a visualization.

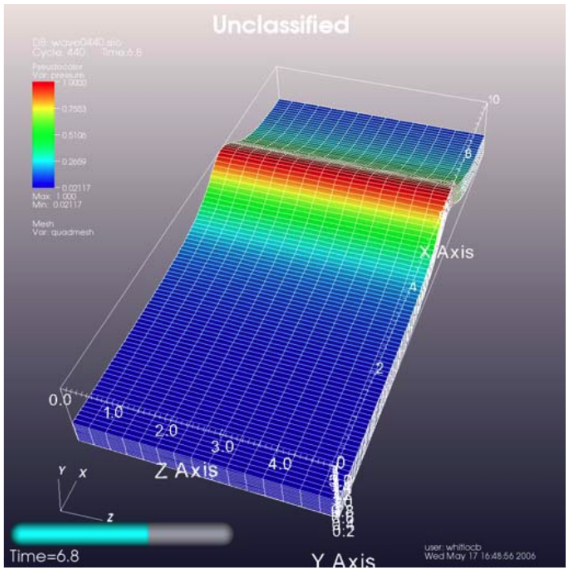


Fig. 5.3: Time slider annotation in the lower left corner

```
# Incorporate LLNL logo image (llnl.jpeg) as an annotation
image = CreateAnnotationObject("Image")
image.image = "llnl.jpeg"
image.position = (0.02, 0.02)

# Print the other image annotation options
print(image)
# Will print something like...
#    visible = 1
#    active = 1
#    position = (0, 0)
#    transparencyColor = (0, 0, 0, 255)
#    useTransparencyColor = 0
#    width = 100.000000
#    height = 100.000000
#    maintainAspectRatio = 1
#    image = ("")
```

Modifying a legend

VisIt's plot legends can be customized. To obtain the proper annotation object, you must use the name of the plot, which is a unique name that identifies the plot. Once you have the plot's name, you can obtain a reference to its legend annotation object and start setting properties to modify the legend.

```
# Open a file and make a plot
OpenDatabase("~juanita/silo/stuff/noise.silo")
AddPlot("Mesh", "Mesh")
AddPlot("Pseudocolor", "hardyglobal")
DrawPlots()
# Get the legend annotation object for the Pseudocolor plot, the second
# plot in the list (0-indexed).
plotName = GetPlotList().GetPlots(1).plotName
```

(continues on next page)

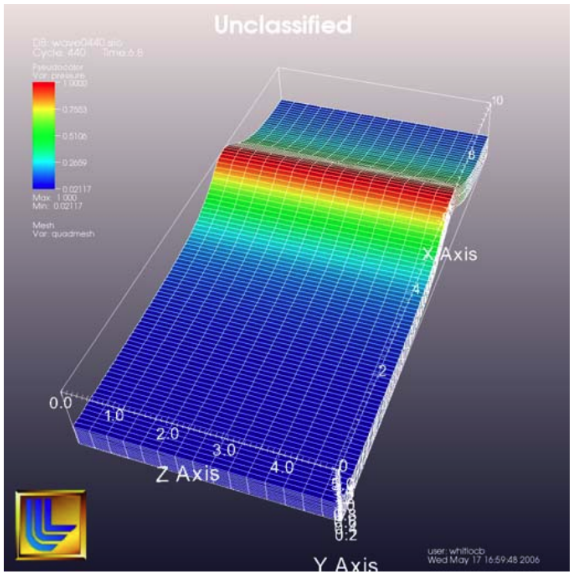


Fig. 5.4: Image annotation used to incorporate LLNL logo

(continued from previous page)

```
legend = GetAnnotationObject(plotName)
# See if we can scale the legend.
legend.xScale = 3.
legend.yScale = 3.
# the bounding box.
legend.drawBoundingBox = 1
legend.boundingBoxColor = (180,180,180,230)
# Make it horizontal
legend.orientation = legend.HorizontalBottom
# moving the legend
legend.managePosition = 0
legend.position = (0.7,0.15)
# text color
InvertBackgroundColor()
legend.useForegroundForTextColor = 0
legend.textColor = (255, 0, 0, 255)
# number format
legend.numberFormat = "%1.4e"
# the font.
legend.fontFamily = legend.Arial
legend.fontBold = 1
legend.fontItalic = 1
# turning off the labels.
legend.fontItalic = 0
legend.drawLabels = 0
legend.drawMinMax = 0
# turning off the title.
legend.drawTitle = 0
# Use user-supplied labels, rather than numeric values.
legend.controlTicks=0
legend.drawLabels = legend.Labels
# suppliedLabels must be strings, only valid when controlTicks is 0
legend.suppliedLabels=("A", "B", "C", "D", "E")
```

(continues on next page)

(continued from previous page)

```
# Give the legend a custom title
legend.useCustomTitle=1
legend.customTitle="my custom title"
```

5.3.14 Working with color tables

Sometimes it is helpful to create a new color table or manipulate an existing user defined color table. Color tables consist of `ControlPoints` which specify color and position in the color spectrum as well as a few other standard options.

Existing color tables can be retreived by name via `GetColorTable` as in:

```
hotCT = GetColorTable("hot")
print(hotCT)
# results of print
#    GetControlPoints(0).colors = (0, 0, 255, 255)
#    GetControlPoints(0).position = 0
#    GetControlPoints(1).colors = (0, 255, 255, 255)
#    GetControlPoints(1).position = 0.25
#    GetControlPoints(2).colors = (0, 255, 0, 255)
#    GetControlPoints(2).position = 0.5
#    GetControlPoints(3).colors = (255, 255, 0, 255)
#    GetControlPoints(3).position = 0.75
#    GetControlPoints(4).colors = (255, 0, 0, 255)
#    GetControlPoints(4).position = 1
#    smoothing = Linear  # NONE, Linear, CubicSpline
#    equalSpacingFlag = 0
#    discreteFlag = 0
```

The `colors` field of the `ControlPoint` represent the (Red,Green,Blue,Alpha) channels of the color and must be in the range (0, 255). The numbers indicate the contribution each channel makes to the overall color. Higher values mean higher saturation of the color. For example, *red* would be represented as *(255, 0, 0, 255)*, while *yellow* would be *(255, 255, 0, 255)*, an equal combination of red and green. Playing with the `Color selection dialog` from the `Popup color menu` in the gui can help in determining RGBA values for desired colors. See Figure 4.343.

Alpha indicates the level of transparency of the color, with a value of 255 meaning fully opaque, and 0, fully transparent. Not all plots in VisIt make use of the Alpha channel of a color table. For instance, for Pseudocolor plots, one must set the `opacityType` to `ColorTable` in order for a Color table with semi-transparent colors to have any effect.

The `position` field of the `ControlPoint` is in the range (0, 1) and should be in ascending order.

General information on VisIt's color tables can be found in the *Color Tables* section of *Using VisIt*.

In all the examples below, `silo_data_path()` refers to a function specific to VisIt testing that returns the path to silo example data. When copying the examples don't forget to modify that reference according to your needs.

Modifying existing color tables

User-defined color tables can be modified by Adding or Removing ControlPoints, and by changing ControlPoint colors and position:

```
OpenDatabase(silo_data_path("rect2d.silo"))
AddPlot("Pseudocolor", "d")

pc = PseudocolorAttributes()
```

(continues on next page)

(continued from previous page)

```
pc.centering=pc.Nodal
# set color table name
pc.colorTableName = "hot"
SetPlotOptions(pc)

DrawPlots()
# put the plot in full-frame mode
v = GetView2D()
v.fullFrameActivationMode= v.On
SetView2D(v)
```

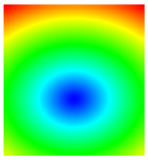
```
hotCT = GetColorTable("hot")

# Remove a couple of control points
hotCT.RemoveControlPoints(4)
hotCT.RemoveControlPoints(3)

# We must use a different name, as VisIt will not allow overwriting of built-in color␣
↪tables
SetColorTable("hot_edited", hotCT)

# set color table name so changes to it will be reflected in plot
pc.colorTableName = "hot_edited"
SetPlotOptions(pc)
```

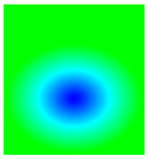
```
# Change colors
hotCT.GetControlPoints(0).colors = (255,0,0,255)
hotCT.GetControlPoints(1).colors = (255, 0, 255, 255)
SetColorTable("hot_edited", hotCT)
```

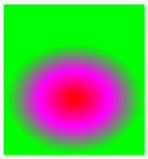
```
# Turn on equal spacing
hotCT.equalSpacingFlag = 1
# Create a new color table by providing a different name
SetColorTable("hot2", hotCT)

# tell the Pseudocolor plot to use the new color table
pc.colorTableName = "hot2"
SetPlotOptions(pc)
```

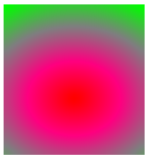
```
# Change positions so that the first and last are at the endpoints
hotCT.equalSpacingFlag=0
hotCT.GetControlPoints(0).position = 0
hotCT.GetControlPoints(1).position =0.5
hotCT.GetControlPoints(2).position = 1
SetColorTable("hot3", hotCT)

pc.colorTableName = "hot3"
SetPlotOptions(pc)
```

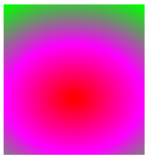
In the set of images below we can see how the plot changes as the color table it uses is modified: with original 'hot' color table; after removing control points; after changing colors; after using equal spacing and after changing positions.











Creating a continous color table from scratch

Creating a continuous color table involves creating a `ColorControlPoint` for each color you want, setting its `colors` and `position` fields and then adding them to a `ColorControlPointList`. The `ColorControlPointList` is then passed as an argument to `AddColorTable`.

```
# create control points (red, green, blue, position).
ct = ((1,0,0,0.), (1,0.8,0.,0.166), (1,1,0,0.333), (0,1,0,0.5),
      (0,1,1,0.666), (0,0,1,0.8333), (0.8,0.1,1,1))

ccpl = ColorControlPointList()

# add the control points to the list
for pt in ct:
    p = ColorControlPoint()
    # colors is RGBA and must be in range 0...255
    p.colors = (pt[0] * 255, pt[1] * 255, pt[2] * 255, 255)
    p.position = pt[3]
    ccpl.AddControlPoints(p)
AddColorTable("myrainbow", ccpl)
```

(continues on next page)

(continued from previous page)

```
OpenDatabase(silo_data_path("globe.silo"))
AddPlot("Pseudocolor", "speed")

# Make the plot use the new color table
pc = PseudocolorAttributes(1)
pc.colorTableName = "myrainbow"
SetPlotOptions(pc)

DrawPlots()

v = GetView3D()
v.viewNormal = (-0.693476, 0.212776, 0.688344)
v. viewUp = (0.161927, 0.976983, -0.138864)
SetView3D(v)
```

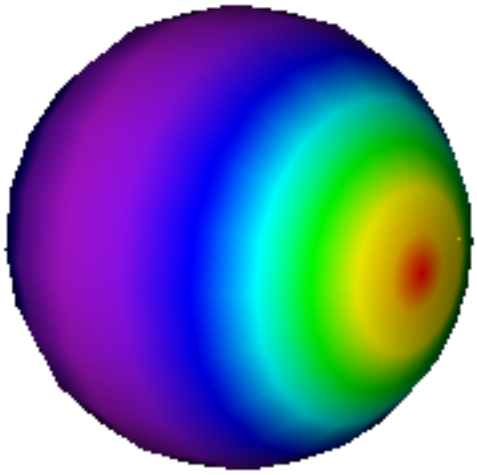


Fig. 5.5: Pseudocolor plot resulting from using the "myrainbow" color table.

Creating a discrete color table from scratch

Sometimes you may not want to figure out RGBA values for colors you want to use. In that case you can import a color module that will give you the RGBA values for named colors that are part of the module. Here's an example of creating a discrete color table using named colors from the vtk module:

```
try:
    import vtk # for vtk.vtkNamedColors
except:
```

(continues on next page)

(continued from previous page)

```
    return

# to see list of all color names available:
# print(vtk.vtkNamedColors.GetColorNames())

# choose some colors from vtk.vtkNamedColors
colorNames = ["tomato", "turquoise", "van_dyke_brown", "carrot",
              "royalblue", "naples_yellow_deep", "cerulean", "warm_grey",
              "venetian_red", "seagreen", "sky_blue", "pink"]
# Create a color control point list
ccpl = ColorControlPointList()
# Make it discrete
ccpl.discreteFlag=1
# Add color control points corresponding to color names
for name in colorNames:
    p = ColorControlPoint()
    p.colors=vtk.vtkNamedColors().GetColor4ub(name)
    ccpl.AddControlPoints(p)
# add a color table based on the color control points
AddColorTable("mylevels", ccpl)

OpenDatabase(silo_data_path("multi_rect2d.silo"))
AddPlot("Subset", "domains")
s = SubsetAttributes()
s.colorType = s.ColorByColorTable
s.colorTableName = "mylevels"
SetPlotOptions(s)
DrawPlots()
```

Volume Plot's special handling of its color table

Volume plot's Color Table is stored directly as a `ColorControlPointList` rather than indirectly from a color table name.

The colorControlPoints object is initialized with 5 values corresponding to the `hot` Color Table. The size of the colorControlPoints list can be adjusted in several ways: via `AddControlPoints`, `RemoveControlPoints` or `SetNumControlPoints` called on the colorControlPoints object, or via `SetColorControlPoints` called on the VolumeAttributes object.

Here is an example using `RemoveControlPoints`:

```
OpenDatabase(silo_data_path("noise.silo"))

AddPlot("Volume", "hardyglobal")

# Modify colors. The default color table has 5 control points. Delete
# all but 2 of them and then change their colors.
v = VolumeAttributes()
v.colorControlPoints.RemoveControlPoints(4)
v.colorControlPoints.RemoveControlPoints(3)
v.colorControlPoints.RemoveControlPoints(2)
v.colorControlPoints.GetControlPoints(0).colors = (255,0,0,255)
v.colorControlPoints.GetControlPoints(0).position = 0.
v.colorControlPoints.GetControlPoints(1).colors = (0,0,255,255)
v.colorControlPoints.GetControlPoints(1).position = 1.
```

(continues on next page)

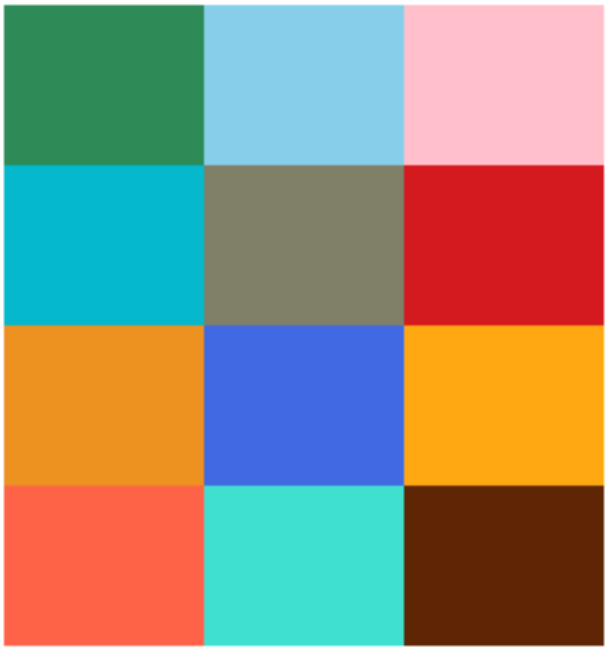


Fig. 5.6: Subset plot using the “mylevels” color table.

(continued from previous page)

```
SetPlotOptions(v)
DrawPlots()
ResetView()
```

Here is an example using `AddControlPoints`:

```
# there are a default of 5 control points, add 3 more and change
# positions of original  so everything is evenly spaced
v = VolumeAttributes()
v.colorControlPoints.GetControlPoints(0).position = 0
v.colorControlPoints.GetControlPoints(1).position = 0.142857
v.colorControlPoints.GetControlPoints(2).position = 0.285714
v.colorControlPoints.GetControlPoints(3).position = 0.428571
v.colorControlPoints.GetControlPoints(4).position = 0.571429
tmp = ColorControlPoint()
tmp.colors = (255,255,0,255)
tmp.position = 0.714286
v.GetColorControlPoints().AddControlPoints(tmp)
tmp.colors = (0,255,0,255)
tmp.position = 0.857143
v.GetColorControlPoints().AddControlPoints(tmp)
tmp.colors = (0,255,255,255)
tmp.position = 1
v.GetColorControlPoints().AddControlPoints(tmp)
SetPlotOptions(v)
```

Here is an example using `SetNumControlPoints`:

```
v = VolumeAttributes()
# there are a default of 5, this resizes to 6
v.colorControlPoints.SetNumControlPoints(6)
v.colorControlPoints.GetControlPoints(4).position = 0.92
# GetControlPoints(5) will cause a segfault without the call to SetNumControlPoints
v.colorControlPoints.GetControlPoints(5).position = 1
v.colorControlPoints.GetControlPoints(5).colors = (128,0,128,255)
SetPlotOptions(v)
```

Here is an example using a named color table and `SetColorControlPoints`:

```
OpenDatabase(silo_data_path("noise.silo"))
AddPlot("Volume", "hardyglobal")
v = VolumeAttributes()
v.lightingFlag = 0
v.rendererType = v.RayCasting
v.sampling = v.KernelBased
ct = GetColorTable("hot_desaturated")
v.SetColorControlPoints(ct)
SetPlotOptions(v)
```

Available color table names can be found via `ColorTableNames()`.

5.3.15 Creating an expression that maps materials to values

A use case that has come up with some VisIt users is the ability to associate scalar values with material numbers. The following example defines a function that creates an expression that maps material numbers to scalar values. It takes a

list of *pairs* of material number and scalar value. The material number of the last pair is ignored. Its value is used for any unspecified materials. The expression generated by calling the function is then used in a Psuedocolor plot.:

```
# Create an expression that maps material numbers to scalar values.
#
# var is the name of the expression.
# mat is the name of the material variable.
# mesh is the name of the mesh variable.
# pairs is a list of tuples of material number and scalar value.
# The material number of the last tuple of the list is ignored and the value
# will be used for all the remaining materials.

def create_mat_value_expr(var, mat, mesh, pairs):
    expr=""
    parens=""
    nlist = len(pairs)
    ilist = 0
    for pair in pairs:
        ilist = ilist + 1
        parens = parens + ")"
        if (ilist == nlist):
            expr = expr + "zonal_constant(%s,%f" % (mesh, pair[1]) + parens
        else:
            expr=expr + "if(eq(dominant_mat(%s),zonal_constant(%s,%d)),zonal_constant(
↪%s,%f)," % (mat, mesh, pair[0], mesh, pair[1])

    DefineScalarExpression(var, expr)

# Call the function to create the expression.
mat_val_pairs = [(1, 0.75), (3, 1.2), (6, 0.2), (7, 1.6), (8, 1.8), (11, 2.2), (-1, 2.
↪5)]

create_mat_value_expr("myvar", "mat1", "quadmesh2d", mat_val_pairs)

# Create a pseudocolor plot of the expression.
OpenDatabase(silo_data_path("rect2d.silo"))
AddPlot("Pseudocolor", "myvar")
DrawPlots()
```

5.4 Functions

Many functions return an integer where 1 means success and 0 means failure. This behavior is represented by the type CLI_return_t in an attempt to distinguish it from functions that may utilize the full range of integers.

5.4.1 ActivateDatabase

Synopsis:

```
ActivateDatabase(argument) -> integer
```

argument [string] The name of the database to be activated.

return type [CLI_return_t] ActivateDatabase returns 1 on success and 0 on failure.

Description:

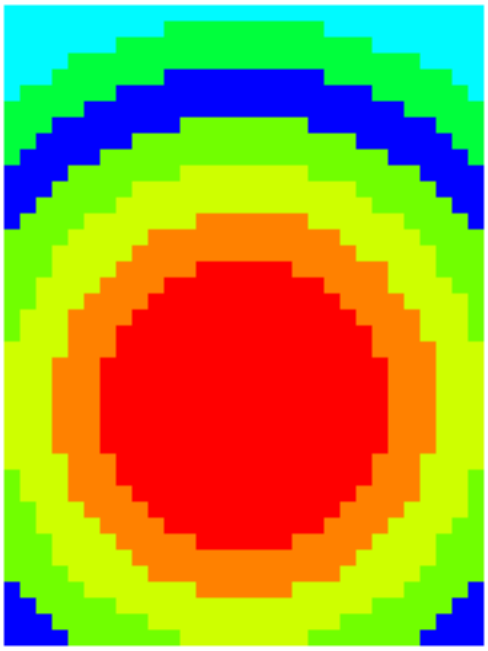


Fig. 5.7: Pseudocolor plot mapping materials to values.

The ActivateDatabase function is used to set the active database to a database that has been previously opened. The ActivateDatabase function only works when you are using it to activate a database that you have previously opened. You do not need to use this function unless you frequently toggle between more than one database when making plots or changing time states. While the OpenDatabase function can also be used to set the active database, the ActivateDatabase function does not have any side effects that would cause the time state for the new active database to be changed.

Example:

```
#% visit -cli
dbs = ("/usr/gapps/visit/data/wave.visit", "/usr/gapps/visit/data/curv3d.silo")
OpenDatabase(dbs[0], 17)
AddPlot("Pseudocolor", "u")
DrawPlots()
OpenDatabase(dbs[1])
AddPlot("Pseudocolor", "u")
DrawPlots()
# Let's add another plot from the first database.
ActivateDatabase(dbs[0])
AddPlot("Mesh", "quadmesh")
DrawPlots()
```

5.4.2 AddArgument

Synopsis:

```
AddArgument(argument)
```

argument [string] A string object that is added to the viewer's command line argument list.

Description:

The AddArgument function is used to add extra command line arguments to VisIt's viewer. This is only useful when VisIt's Python interface is imported into a stand-alone Python interpreter because the AddArgument function must be called before the viewer is launched. The AddArgument function has no effect when used in VisIt's cli program because the viewer is automatically launched before any commands are processed.

Example:

```
import visit
visit.AddArgument("-nowin") # Add the -nowin argument to the viewer.
```

5.4.3 AddMachineProfile

Synopsis:

```
AddMachineProfile(MachineProfile) -> integer
```

MachineProfile : MachineProfile object

Description:

Sets the input machine profile in the HostProfileList, replaces if one already exists Otherwise adds to the list

5.4.4 AddOperator

Synopsis:

```
AddOperator(operator) -> integer
AddOperator(operator, all) -> integer
```

operator [string] The name of the operator to be applied.

all [integer] This is an optional integer argument that applies the operator to all plots if the value of the argument is not zero.

return type [CLI_return_t] The AddOperator function returns an integer value of 1 for success and 0 for failure.

Description:

The AddOperator function adds a VisIt operator to the active plots. The operator argument is a string containing the name of the operator to be added to the active plots. The operatore name must be a valid operator plugin name that is a member of the tuple returned by the OperatorPlugins function. The all argument is an integer that determines whether or not the operator is applied to all plots. If the all argument is not provided, the operator is only added to active plots. Once the AddOperator function is called, the desired operator is added to all active plots unless the all argument is a non-zero value. When the all argument is a non-zero value, the operator is applied to all plots regardless of whether or not they are selected. Operator attributes are set through the SetOperatorOptions function.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddPlot("Mesh", "mesh1")
AddOperator("Slice", 1) # Slice both plots
DrawPlots()
```

5.4.5 AddPlot

Synopsis:

```
AddPlot(plotType, variableName) -> integer
AddPlot(plotType, variableName, inheritSIL) -> integer
AddPlot(plotType, variableName, inheritSIL, applyOperators) -> integer
```

plotType [string] The name of a valid plot plugin type.

variableName [string] A valid variable name for the open database.

inheritSIL [integer] An integer flag indicating whether the plot should inherit the active plot's SIL restriction.

applyOperators [integer] An integer flag indicating whether the operators from the active plot should be applied to the new plot.

return type [CLI_return_t] The AddPlot function returns an integer value of 1 for success and 0 for failure.

Description:

The AddPlot function creates a new plot of the specified type using a variable from the open database. The plotType argument is a string that contains the name of a valid plot plugin type which must be a member of the string tuple that is returned by the PlotPlugins function. The variableName argument is a string that contains the name of a variable in the open database. After the AddPlot function is called, a new plot is created and it is made the sole active plot.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Subset", "mat1") # Create a subset plot
DrawPlots()
```

5.4.6 AddWindow

Synopsis:

```
AddWindow()
```

Description:

The AddWindow function creates a new visualization window and makes it the active window. This function can be used to create up to 16 visualization windows. After that, the AddWindow function has no effect.

Example:

```
import visit
visit.Launch()
visit.AddWindow() # Create window #2
visit.AddWindow() # Create window #3
```

5.4.7 AlterDatabaseCorrelation

Synopsis:

```
AlterDatabaseCorrelation(name, databases, method) -> integer
```

name [string] The name of the database correlation to be altered.

databases [tuple or list of strings] The databases argument must be a tuple or list of strings containing the fully qualified database names to be used in the database correlation.

method [integer] The method argument must be an integer in the range [0,3].

Correlation method	Value
IndexForIndexCorrelation	0
StretchedIndexCorrelation	1
TimeCorrelation	2
CycleCorrelation	3

return type [CLI_return_t] The AlterDatabaseCorrelation function returns 1 on success and 0 on failure.

Description:

The AlterDatabaseCorrelation method alters an existing database correlation. A database correlation is a VisIt construct that relates the time states for two or more databases in some way. You would use the AlterDatabaseCorrelation function if you wanted to change the list of databases used in a database correlation or if you wanted to change how the databases are related - the correlation method. The name argument is a string that is the name of the database correlation to be altered. If the name that you pass is not a valid database correlation then the AlterDatabaseCorrelation function fails. The databases argument

is a list or tuple of string objects containing the fully-qualified (host:/path/filename) names of the databases to be involved in the database query. The method argument allows you to specify a database correlation method.

Example:

```
dbs = ("/usr/gapps/visit/data/wave.visit", "/usr/gapps/visit/data/wave*.silo database
↪")
OpenDatabase(dbs[0])
AddPlot("Pseudocolor", "pressure")
OpenDatabase(dbs[1])
AddPlot("Pseudocolor", "d")
# VisIt created an index for index database correlation but we
# want a cycle correlation.
AlterDatabaseCorrelation("Correlation01", dbs, 3)
```

5.4.8 ApplyNamedSelection

Synopsis:

```
ApplyNamedSelection(name) -> integer
```

name [string] The name of a named selection. (This should have been previously created with a CreateNamedSelection call.)

return type [CLI_return_t] The ApplyNamedSelection function returns 1 for success and 0 for failure.

Description:

Named Selections allow you to select a group of elements (or particles). One typically creates a named selection from a group of elements and then later applies the named selection to another plot (thus reducing the set of elements displayed to the ones from when the named selection was created).

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/wave*.silo database"
OpenDatabase(db)
AddPlot("Pseudocolor", "pressure")
AddOperator("Clip")
c = ClipAttributes()
c.plane1Origin = (0,0.6,0)
c.plane1Normal = (0,-1,0)
SetOperatorOption(c)
DrawPlots()
CreateNamedSelection("els_above_at_time_0")
SetTimeSliderState(40)
RemoveLastOperator()
ApplyNamedSelection("els_above_at_time_0")
```

5.4.9 ChangeActivePlotsVar

Synopsis:

```
ChangeActivePlotsVar(variableName) -> integer
```

variableName [string] The name of the new plot variable.

return type [CLI_return_t] The ChangeActivePlotsVar function returns an integer value of 1 for success and 0 for failure.

Description:

The ChangeActivePlotsVar function changes the plotted variable for the active plots. This is a useful way to change what is being visualized without having to delete and recreate the current plots. The variableName argument is a string that contains the name of a variable in the open database.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
SaveWindow()
ChangeActivePlotsVar("v")
```

5.4.10 CheckForNewStates

Synopsis:

```
CheckForNewStates(name) -> integer
```

name [string] The name of a database that has been opened previously.

return type [CLI_return_t] The CheckForNewStates function returns 1 for success and 0 for failure.

Description:

Calculations are often run at the same time as some of the preliminary visualization work is being performed. That said, you might be visualizing the leading time states of a database that is still being created. If you want to force VisIt to add any new time states that were added since you opened the database, you can use the CheckForNewStates function. The name argument must contain the name of a database that has been opened before.

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/wave*.silo database"
OpenDatabase(db)
AddPlot("Pseudocolor", "pressure")
DrawPlots()
SetTimeSliderState(TimeSliderGetNStates() - 1)
# More files appear on disk
CheckForNewStates(db)
SetTimeSliderState(TimeSliderGetNStates() - 1)
```

5.4.11 ChooseCenterOfRotation

Synopsis:

```
ChooseCenterOfRotation() -> integer
ChooseCenterOfRotation(screenX, screenY) -> integer
```

screenX [double] A double that is the X coordinate of the pick point in normalized [0,1] screen space.

screenY [double] A double that is the Y coordinate of the pick point in normalized [0,1] screen space.

return type [CLI_return_t] The ChooseCenterOfRotation function returns 1 if successful and 0 if it fails.

Description:

The ChooseCenterOfRotation function allows you to pick a new center of rotation, which is the point about which plots are rotated when you interactively rotate plots. The function can either take zero arguments, in which case you must interactively pick on plots, or it can take two arguments that correspond to the X and Y coordinates of the desired pick point in normalized screen space. When using the two argument version of the ChooseCenterOfRotation function, the X and Y values are floating point values in the range [0,1]. If the ChooseCenterOfRotation function is able to actually pick on plots, yes there must be plots in the vis window, then the center of rotation is updated and the new value is printed to the console.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlots("Pseudocolor", "u")
DrawPlots()
# Interactively choose the center of rotation
ChooseCenterOfRotation()
# Choose a center of rotation using normalized screen
# coordinates and print the value.
ResetView()
ChooseCenterOfRotation(0.5, 0.3)
print("The new center of rotation is:{}".format(GetView3D().centerOfRotation))
```

5.4.12 ClearAllWindows

Synopsis:

```
ClearAllWindows() -> integer
```

return type [CLI_return_t] 1 on success, 0 on failure.

Description:

The ClearWindow function is used to clear out the plots from the active visualization window. The plots are removed from the visualization window but are left in the plot list so that subsequent calls to the DrawPlots function regenerate the plots in the plot list. The ClearAllWindows function preforms the same work as the ClearWindow function except that all windows are cleared of their plots.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
AddWindow()
SetActiveWindow(2) # Make window 2 active
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Subset", "mat1")
DrawPlots()
ClearWindow() # Clear the plots in window 2.
DrawPlots() # Redraw the plots in window 2.
ClearAllWindows() # Clear the plots from all windows.
```

5.4.13 ClearCache

Synopsis:

```
ClearCache(host) -> integer
ClearCache(host, simulation) -> integer
```

host [string] The name of the computer where the compute engine is running.

simulation [string] The name of the simulation being processed by the compute engine.

return type [CLI_return_t] 1 on success and 0 on failure.

Description:

Sometimes during extended VisIt runs, you might want to periodically clear the compute engine's network cache to reduce the amount of memory being used by the compute engine. Clearing the network cache is also useful when you want to change what the compute engine is working on. For example, you might process a large database and then decide to process another large database. Clearing the network cache beforehand will free up more resources for the compute engine so it can more efficiently process the new database. The host argument is a string object containing the name of the computer on which the compute engine is running. The simulation argument is optional and only applies to when you want to instruct a simulation that is acting as a VisIt compute engine to clear its network cache. If you want to tell more than one compute engine to clear its cache without having to call ClearCache multiple times, you can use the ClearCacheForAllEngines function.

Example:

```
#%visit -cli
OpenDatabase("localhost:very_large_database")
# Do a lot of work
ClearCache("localhost")
OpenDatabase("localhost:another_large_database")
# Do more work
OpenDatabase("remotehost:yet_another_database")
# Do more work
ClearCacheForAllEngines()
```

5.4.14 ClearCacheForAllEngines

Synopsis:

```
ClearCacheForAllEngines() -> integer
```

return type [CLI_return_t] 1 on success and 0 on failure.

Description:

Sometimes during extended VisIt runs, you might want to periodically clear the compute engine's network cache to reduce the amount of memory being used by the compute engine. Clearing the network cache is also useful when you want to change what the compute engine is working on. For example, you might process a large database and then decide to process another large database. Clearing the network cache beforehand will free up more resources for the compute engine so it can more efficiently process the new database. The host argument is a string object containing the name of the computer on which the compute engine is running. The simulation argument is optional and only applies to when you want to instruct a simulation that is acting as a VisIt compute engine to clear its network cache. If you want to tell more than one compute engine to clear its cache without having to call ClearCache multiple times, you can use the ClearCacheForAllEngines function.

Example:

```
#%visit -cli
OpenDatabase("localhost:very_large_database")
# Do a lot of work
ClearCache("localhost")
OpenDatabase("localhost:another_large_database")
# Do more work
OpenDatabase("remotehost:yet_another_database")
# Do more work
ClearCacheForAllEngines()
```

5.4.15 ClearMacros

Synopsis:

```
ClearMacros()
```

Description:

The ClearMacros function clears out the list of registered macros and sends a message to the gui to clear the buttons from the Macros window.

Example:

```
ClearMacros()
```

5.4.16 ClearPickPoints

Synopsis:

```
ClearPickPoints()
```

Description:

The ClearPickPoints function removes pick points from the active visualization window. Pick points are the letters that are added to the visualization window where the mouse is clicked when the visualization window is in pick mode.

Example:

```
#% visit -cli
# Put the visualization window into pick mode using the popup
# menu and add some pick points.
# Clear the pick points.
ClearPickPoints()
```

5.4.17 ClearReferenceLines

Synopsis:

```
ClearReferenceLines()
```

Description:

The ClearReferenceLines function removes reference lines from the active visualization window. Reference lines are the lines that are drawn on a plot to show where you have performed lineouts.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "d")
Lineout((-3.0, 2.0), (2.0, 4.0), ("default", "u", "v"))
ClearReferenceLines()
```

5.4.18 ClearViewKeyframes

Synopsis:

```
ClearViewKeyframes() -> integer
```

return type [CLI_return_t] The ClearViewKeyframes function returns 1 on success and 0 on failure.

Description:

The ClearViewKeyframes function clears any view keyframes that may have been set. View keyframes are used to create complex view behavior such as fly-throughs when VisIt is in keyframing mode.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
k = KeyframeAttributes()
k.enabled, k.nFrames, k.nFramesWasUserSet = 1,10,1
SetKeyframeAttributes(k)
DrawPlots()
SetViewKeyframe()
v1 = GetView3D()
v1.viewNormal = (-0.66609, 0.337227, 0.665283)
v1.viewUp = (0.157431, 0.935425, -0.316537)
SetView3D(v1)
SetTimeSliderState(9)
SetViewKeyframe()
ToggleCameraViewMode()
for i in range(10):
    SetTimeSliderState(i)
ClearViewKeyframes()
```

5.4.19 ClearWindow

Synopsis:

```
ClearWindow() -> integer
```

return type [CLI_return_t] 1 on success, 0 on failure.

Description:

The ClearWindow function is used to clear out the plots from the active visualization window. The plots are removed from the visualization window but are left in the plot list so that subsequent calls to the

DrawPlots function regenerate the plots in the plot list. The ClearAllWindows function preforms the same work as the ClearWindow function except that all windows are cleared of their plots.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
AddWindow()
SetActiveWindow(2) # Make window 2 active
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Subset", "mat1")
DrawPlots()
ClearWindow() # Clear the plots in window 2.
DrawPlots() # Redraw the plots in window 2.
ClearAllWindows() # Clear the plots from all windows.
```

5.4.20 CloneWindow

Synopsis:

```
CloneWindow() -> integer
```

return type [CLI_return_t] The CloneWindow function returns an integer value of 1 for success and 0 for failure.

Description:

The CloneWindow function tells the viewer to create a new window, based on the active window, that contains the same plots, annotations, lights, and view as the active window. This function is useful for when you have a window set up like you want and then want to do the same thing in another window using a different database. You can first clone the window and then replace the database.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
v = ViewAttributes()
v.camera = (-0.505893, 0.32034, 0.800909)
v.viewUp = (0.1314, 0.946269, -0.295482)
v.parallelScale = 14.5472
v.nearPlane = -34.641
v.farPlane = 34.641
v.perspective = 1
SetView3D() # Set the view
a = AnnotationAttributes()
a.backgroundColor = (0, 0, 255, 255)
SetAnnotationAttributes(a) # Set the annotation properties
CloneWindow() # Create a clone of the active window
DrawPlots() # Make the new window draw its plots
```

5.4.21 Close

Synopsis:

```
Close()
```

Description:

The Close function terminates VisIt's viewer. This is useful for Python scripts that only need access to VisIt's capabilties for a short time before closing VisIt.

Example:

```
import visit
visit.Launch()
visit.Close() # Close the viewer
```

5.4.22 CloseComputeEngine

Synopsis:

```
CloseComputeEngine() -> integer
CloseComputeEngine(hostName) -> integer
CloseComputeEngine(hostName, simulation) -> integer
```

hostName [string] Optional name of the computer on which the compute engine is running.

simulation [string] Optional name of a simulation.

return type [CLI_return_t] The CloseComputeEngine function returns an integer value of 1 for success and 0 for failure.

Description:

The CloseComputeEngine function tells the viewer to close the compute engine running a specified host. The hostName argument is a string that contains the name of the computer where the compute engine is running. The hostName argument can also be the name "localhost" if you want to close the compute engine on the local machine without having to specify its name. It is not necessary to provide the hostName argument. If the argument is omitted, the first compute engine in the engine list will be closed. The simulation argument can be provided if you want to close a connection to a simulation that is acting as a VisIt compute engine. A compute engine can be launched again by creating a plot or by calling the OpenComputeEngine function.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo") # Launches an engine
AddPlot("Pseudocolor", "u")
DrawPlots()
CloseComputeEngine() # Close the compute engine
```

5.4.23 CloseDatabase

Synopsis:

```
CloseDatabase(name) -> integer
```

name [string] The name of the database to close.

return type [CLI_return_t] The CloseDatabase function returns 1 on success and 0 on failure.

Description:

The CloseDatabase function is used to close a specified database and free all resources that were devoted to keeping the database open. This function has an effect similar to ClearCache but it does more in that in addition to clearing the compute engine's cache, which it only does for the specified database, it also removes all references to the specified database from tables of cached metadata, etc. Note that the CloseDatabase function will fail and the database will not be closed if any plots reference the specified database.

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/globe.silo"
OpenDatabase(db)
AddPlot("Pseudocolor", "u")
DrawPlots()
print("This won't work: retval = %d" % CloseDatabase(db))
DeleteAllPlots()
print("Now it works: retval = %d" % CloseDatabase(db))
```

5.4.24 ColorTableNames

Synopsis:

```
ColorTableNames() -> tuple
```

return type [tuple] The ColorTableNames function returns a tuple of strings containing the names of the color tables that have been defined.

Description:

The ColorTableNames function returns a tuple of strings containing the names of the color tables that have been defined. This method can be used in case you want to iterate over several color tables.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
for ct in ColorTableNames():
    p = PseudocolorAttributes()
    p.colorTableName = ct
    SetPlotOptions(p)
```

5.4.25 ConstructDataBinning

Synopsis:

```
ConstructDataBinning(options) -> integer
```

options [ConstructDataBinningAttributes object] An object of type ConstructDataBinningAttributes. This object specifies the options for constructing a data binning.

return type [CLI_return_t] Returns 1 on success, 0 on failure.

Description:

The ConstructDataBinning function creates a data binning function for the active plot. Data Binnings place data from a data set into bins and reduce that data. They are used to either be incorporated with expressions to make new derived quantities or to be directly visualized.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv3d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Set the construct data binning attributes.
i = ConstructDataBinningAttributes()
i.name = "db1"
i.binningScheme = i.Uniform
i.varnames = ("u", "w")
i.binBoundaries = (-1, 1, -1, 1) # minu, maxu, minw, maxw
i.numSamples = (25, 25)
i.reductionOperator = i.Average
i.varForReductionOperator = "v"
ConstructDataBinning(i)
# Example of binning using spatial coordinates
i.varnames = ("X", "u") # X is added as a placeholder to maintain indexing
i.binType = (1, 0) # 1 = X, 2 = Y, 3 = Z, 0 = variable
```

5.4.26 CopyAnnotationsToWindow

Synopsis:

```
CopyAnnotationsToWindow(source, dest) -> integer
```

source [integer] The index (an integer from 1 to 16) of the source window.

dest [integer] The index (an integer from 1 to 16) of the destination window.

return type [CLI_return_t] 1 for success and 0 for failure.

Description:

The Copy functions copy attributes from one visualization window to another visualization window. The CopyAnnotationsToWindow function copies the annotations from a source visualization window to a destination visualization window.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
AddWindow()
SetActiveWindow(2)
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Mesh", "mesh1")
# Copy window 1's Pseudocolor plot to window 2.
CopyPlotsToWindow(1, 2)
DrawPlots() # Window 2 will have 2 plots
# Spin the plots around in window 2 using the mouse.
CopyViewToWindow(2, 1) # Copy window 2's view to window 1.
```

5.4.27 CopyLightingToWindow

Synopsis:

```
CopyLightingToWindow(source, dest) -> integer
```

source [integer] The index (an integer from 1 to 16) of the source window.

dest [integer] The index (an integer from 1 to 16) of the destination window.

return type [CLI_return_t] 1 for success and 0 for failure.

Description:

The Copy functions copy attributes from one visualization window to another visualization window. The CopyLightingAttributes function copies lighting.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
AddWindow()
SetActiveWindow(2)
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Mesh", "mesh1")
# Copy window 1's Pseudocolor plot to window 2.
CopyPlotsToWindow(1, 2)
DrawPlots() # Window 2 will have 2 plots
# Spin the plots around in window 2 using the mouse.
CopyViewToWindow(2, 1) # Copy window 2's view to window 1.
```

5.4.28 CopyPlotsToWindow

Synopsis:

```
CopyPlotsToWindow(source, dest) -> integer
```

source [integer] The index (an integer from 1 to 16) of the source window.

dest [integer] The index (an integer from 1 to 16) of the destination window.

return type [CLI_return_t] 1 for success and 0 for failure.

Description:

The Copy functions copy attributes from one visualization window to another visualization window. The CopyPlotsToWindow function copies the plots from one visualization window to another visualization window but does not also force plots to generate so after copying plots with the CopyPlotsToWindow function, you should also call the DrawPlots function.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
AddWindow()
```

(continues on next page)

(continued from previous page)

```
SetActiveWindow(2)
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Mesh", "mesh1")
# Copy window 1's Pseudocolor plot to window 2.
CopyPlotsToWindow(1, 2)
DrawPlots() # Window 2 will have 2 plots
# Spin the plots around in window 2 using the mouse.
CopyViewToWindow(2, 1) # Copy window 2's view to window 1.
```

5.4.29 CopyViewToWindow

Synopsis:

```
CopyViewToWindow(source, dest) -> integer
```

source [integer] The index (an integer from 1 to 16) of the source window.

dest [integer] The index (an integer from 1 to 16) of the destination window.

return type [CLI_return_t] The Copy functions return an integer value of 1 for success and 0 for failure.

Description:

The Copy functions copy attributes from one visualization window to another visualization window. The CopyViewToWindow function copies the view.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
AddWindow()
SetActiveWindow(2)
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Mesh", "mesh1")
# Copy window 1's Pseudocolor plot to window 2.
CopyPlotsToWindow(1, 2)
DrawPlots() # Window 2 will have 2 plots
# Spin the plots around in window 2 using the mouse.
CopyViewToWindow(2, 1) # Copy window 2's view to window 1.
```

5.4.30 CreateAnnotationObject

Synopsis:

```
CreateAnnotationObject(annotType[,annotName,visibleFlag]) -> annotation object
```

annotType [string] The name of the type of annotation object to create.

Annotation type	String
2D text annotation	Text2D
3D text annotation	Text3D
Time slider annotation	TimeSlider
Image annotation	Image
Line/arrow annotation	Line2D

annotName [string] A user-defined name of the annotation object to create. By default, VisIt creates names like 'newObject0', 'newObject1', ...

visibleFlag [integer] An optional integer to indicate if the annotation object should be created with initial visibility on or off. Pass 0 for off and non-zero for on. By default, VisIt creates annotation objects with visibility on. If you wish only to pass the visibleFlag argument, there is no need to also pass the annotName argument.

return type [annotation object] CreateAnnotationObject is a factory function that creates annotation objects of different types. The return value, if a valid annotation type is provided, is an annotation object. If the function fails, VisItException is raised.

Description:

CreateAnnotationObject is a factory function that creates different kinds of annotation objects. The annotType argument is a string containing the name of the type of annotation object to create. Each type of annotation object has different properties that can be set. Setting the different properties of an Annotation objects directly modifes annotations in the vis window. Currently there are 5 types of annotation objects:

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit", 17)
AddPlot("Pseudocolor", "pressure")
DrawPlots()
slider = CreateAnnotationObject("TimeSlider")
print(slider)
slider.startColor = (255,0,0,255)
slider.endColor = (255,255,0,255)
```

5.4.31 CreateDatabaseCorrelation

Synopsis:

```
CreateDatabaseCorrelation(name, databases, method) -> integer
```

name [string] The name of the database correlation to be created.

databases [tuple or list of strings] Tuple or list of strings containing the names of the databases to involve in the database correlation.

method [integer] An integer in the range [0,3] that determines the correlation method.

Correlation method	Value
IndexForIndexCorrelation	0
StretchedIndexCorrelation	1
TimeCorrelation	2
CycleCorrelation	3

return type [CLI_return_t] The CreateDatabaseCorrelation function returns 1 on success and 0 on failure.

Description:

The CreateDatabaseCorrelation function creates a database correlation, which is a VisIt construct that relates the time states for two or more databases in some way. You would use the CreateDatabaseCorrelation function if you wanted to put plots from more than one time-varying database in the same vis window and then move them both through time in some synchronized way. The name argument is a string that is the name of the database correlation to be created. You will use the name of the database correlation to set the active time slider later so that you can change time states. The databases argument is a list or tuple of string objects containing the fully-qualified (host:/path/filename) names of the databases to be involved in the database query. The method argument allows you to specify a database correlation method. Each database correlation has its own time slider that can be used to set the time state of databases that are part of a database correlation. Individual time-varying databases have their own trivial database correlation, consisting of only 1 database. When you call the CreateDatabaseCorrelation function, VisIt creates a new time slider with the same name as the database correlation and makes it be the active time slider.

Example:

```
#% visit -cli
dbs = ("/usr/gapps/visit/data/dbA00.pdb",
"/usr/gapps/visit/data/dbB00.pdb")
OpenDatabase(dbs[0])
AddPlot("FilledBoundary", "material(mesh)")
OpenDatabase(dbs[1])
AddPlot("FilledBoundary", "material(mesh)")
DrawPlots()
CreateDatabaseCorrelation("common", dbs, 1)
# Creating a new database correlation also creates a new time
# slider and makes it be active.
w = GetWindowInformation()
print("Active time slider: %s" % w.timeSliders[w.activeTimeSlider])
# Animate through time using the "common" database correlation's
# time slider.
for i in range(TimeSliderGetNStates()):
    SetTimeSliderState(i)
```

5.4.32 CreateNamedSelection

Synopsis:

```
CreateNamedSelection(name) -> integer
CreateNamedSelection(name, properties) -> integer
```

name [string] The name of a named selection.

properties [SelectionProperties object] This optional argument lets you pass a SelectionProperties object containing the properties that will be used to create the named selection. When this argument is omitted, the named selection will always be associated with the active plot. You can use this argument to set up more complex named selections that may be associated with plots or databases.

return type [CLI_return_t] The CreateNamedSelection function returns 1 for success and 0 for failure.

Description:

Named Selections allow you to select a group of elements (or particles). One typically creates a named selection from a group of elements and then later applies the named selection to another plot (thus reducing the set of elements displayed to the ones from when the named selection was created).

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/wave*.silo database"
OpenDatabase(db)
AddPlot("Pseudocolor", "pressure")
AddOperator("Clip")
c = ClipAttributes()
c.plane1Origin = (0,0.6,0)
c.plane1Normal = (0,-1,0)
SetOperatorOption(c)
DrawPlots()
CreateNamedSelection("els_above_at_time_0")
SetTimeSliderState(40)
RemoveLastOperator()
ApplyNamedSelection("els_above_at_time_0")
```

5.4.33 DatabasePlugins

Synopsis:

```
DatabasePlugins() -> dictionary
DatabasePlugins(host) -> dictionary
```

host [string] The name of the host for which we want database plugins.

return type [dictionary] The DatabasePlugins functions returns a dictionary.

Description:

The DatabasePlugins function returns a dictionary containing the names of the database plugins for the specified host. If no host is given, localhost is assumed. The dictionary contains two keys: "host" and "plugins".

Example:

```
#% visit -cli
dbp = DatabasePlugins("localhost")
print(dbp["host"])
print(dbp["plugins"])
```

5.4.34 DeIconifyAllWindows

Synopsis:

```
DeIconifyAllWindows()
```

Description:

The DeIconifyAllWindows function unhides all of the hidden visualization windows. This function is usually called after IconifyAllWindows as a way of making all of the hidden visualization windows visible.

Example:

```
#% visit -cli
SetWindowLayout(4) # Have 4 windows
```

(continues on next page)

(continued from previous page)

```
IconifyAllWindows()
DeIconifyAllWindows()
```

5.4.35 DefineArrayExpression

Synopsis:

```
DefineArrayExpression(variableName, expression) -> integer
```

variableName [string] The name of the variable to be created.

expression [string] The expression definition as a string.

return type [CLI_return_t] The DefineArrayExpression function returns 1 on success and 0 on failure.

Description:

DefineArrayExpression creates new array variables. Array variables are a collection of scalar variables that are grouped together. All the variables must have the same centering and only scalar variables are supported, for example, no vector, tensor or material variables. Array variables are used in the Label plot.

The variableName argument is a string that contains the name of the new variable. You can pass the name of an existing expression if you want to provide a new expression definition. The expression argument is a string that contains the definition of the new variable in terms of *built-in expressions* and pre-existing variable names using VisIt's *expression grammar*. If you run into problems defining your expression you might want to read the section on *expression compatibility gotchas*.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv3d.silo")
DefineScalarExpression("d1", 'recenter(d, "zonal")')
DefineScalarExpression("p1", 'recenter(p, "zonal")')
# Define 2 array variables, each from 2 scalars. Here, we
# reuse the same scalars twice for illustrative purposes
# only. Normally, the scalars are different.
DefineArrayExpression("da", "array_compose(d1, d1)")
DefineArrayExpression("pa", "array_compose(p1, p1)")
# Create a plot to use for performing an XRay Image query.
AddPlot("Pseudocolor", "d1")
DrawPlots()
# Do the query.
params = GetQueryParameters("XRay Image")
params['output_type'] ="png"
params['divide_emis_by_absorb'] = 1
params['origin'] = (0.0, 2.5, 10.0)
params['up_vector'] = (0, 1, 0)
params['theta'] = 0
params['phi'] = 0
params['width'] = 10.
params['height'] = 10.
params['image_size'] = (300, 300)
params['vars'] = ("da", "pa")
Query("XRay Image", params)
```

5.4.36 DefineCurveExpression

Synopsis:

```
DefineCurveExpression(variableName, expression) -> integer
```

variableName [string] The name of the variable to be created.

expression [string] The expression definition as a string.

return type [CLI_return_t] The DefineCurveExpression function returns 1 on success and 0 on failure.

Description:

DefineCurveExpression creates new curve variables. Curve variables are a collection of X - Y coordinates that form a curve. Curve variables are used in the Curve, Parallel Coordinates, Scatter and Spreadsheet plots.

The variableName argument is a string that contains the name of the new variable. You can pass the name of an existing expression if you want to provide a new expression definition. The expression argument is a string that contains the definition of the new variable in terms of *built-in expressions* and pre-existing variable names using VisIt's *expression grammar*. If you run into problems defining your expression you might want to read the section on *expression compatibility gotchas*.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/lines.curve")
# Define an expression to multiply the y value of the curve by 2.
DefineCurveExpression("myvar", "2 * curve1")
# Plot the curve variable.
AddPlot("Curve", "myvar")
DrawPlots()
```

5.4.37 DefineMaterialExpression

Synopsis:

```
DefineMaterialExpression(variableName, expression) -> integer
```

variableName [string] The name of the variable to be created.

expression [string] The expression definition as a string.

return type [CLI_return_t] The DefineMaterialExpression function returns 1 on success and 0 on failure.

Description:

DefineMaterialExpression creates new material variables. Material variables are special variables that store material information for mesh and scalar variables. Material variables are used by the Boundary and Filled Boundary plots. Currently there are no built-in expressions that create material variables.

5.4.38 DefineMeshExpression

Synopsis:

```
DefineMeshExpression(variableName, expression) -> integer
```

variableName [string] The name of the variable to be created.

expression [string] The expression definition as a string.

return type [CLI_return_t] The DefineMeshExpression function returns 1 on success and 0 on failure.

Description:

DefineMeshExpression creates new mesh variables. Mesh variables define the coordinates and connectivity of a mesh. Mesh variables are used by the Label, Mesh and Subset plots. Currently there are no built-in expressions that create mesh variables.

5.4.39 DefinePythonExpression

Synopsis:

```
DefinePythonExpression(myvar,args,source)
DefinePythonExpression(myvar,args,source,type)
DefinePythonExpression(myvar,args,file)
```

myvar [string] The name of the variable to be created.

args [tuple] A tuple (or list) of strings providing the variable names of the arguments to the Python Expression.

source [string] A string containing the source code for a Python Expression Filter .

file [string] A string containing the path to a Python Expression Filter script file.

type [string] An optional string defining the output type of the expression. Default type - 'scalar' Avalaible types - 'scalar','vector','tensor','array','curve' Note - Use only one of the 'source' or 'file' arguments. If both are used the 'source' argument overrides 'file'.

Description:

Used to define a Python Filter Expression.

5.4.40 DefineScalarExpression

Synopsis:

```
DefineScalarExpression(variableName, expression) -> integer
```

variableName [string] The name of the variable to be created.

expression [string] The expression definition as a string.

return type [CLI_return_t] The DefineScalarExpression function returns 1 on success and 0 on failure.

Description:

DefineScalarExpression creates new scalar variables. Scalar variables define a scalar field over a mesh and are used by plots that take scalar variables.

The variableName argument is a string that contains the name of the new variable. You can pass the name of an existing expression if you want to provide a new expression definition. The expression argument is a string that contains the definition of the new variable in terms of *built-in expressions* and pre-existing variable names using VisIt's *expression grammar*. If you run into problems defining your expression you might want to read the section on

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
# Plot the scalar variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()
```

5.4.41 DefineSpeciesExpression

Synopsis:

```
DefineSpeciesExpression(variableName, expression) -> integer
```

variableName [string] The name of the variable to be created.

expression [string] The expression definition as a string.

return type [CLI_return_t] The DefineSpeciesExpression function returns 1 on success and 0 on failure.

Description:

DefineSpeciesExpression creates new species variables. Species variables are special variables that are associated with material variables that store species information for scalar variables. Currently there are no built-in expressions that create species variables.

5.4.42 DefineTensorExpression

Synopsis:

```
DefineTensorExpression(variableName, expression) -> integer
```

variableName [string] The name of the variable to be created.

expression [string] The expression definition as a string.

return type [CLI_return_t] The DefineTensorExpression function returns 1 on success and 0 on failure.

Description:

DefineTensorExpression creates new tensor variables. Tensor variables define a tensor field over a mesh and are used by the Tensor plot. A 2D tensor would consist of a vector of 2 2-component vectors. A 3D tensor would consist of a vector of 3 3-component vectors. A symmetric tensor would need to provide 4 or 9 components even though a 2D tensor has 3 unique values and a 3D tensor has 6 unique values. For a 2D symmetric tensor, the components would be supplied as {{Sxx, Syx}, {Syx, Syy}}. For a 3D symmetric tensor, the components would be supplied as {{Sxx, Syx, Szx}, {Syx, Syy, Szy}, {Szx, Szy, Szz}}.

The variableName argument is a string that contains the name of the new variable. You can pass the name of an existing expression if you want to provide a new expression definition. The expression argument is a string that contains the definition of the new variable in terms of *built-in expressions* and pre-existing variable names using VisIt's *expression grammar*. If you run into problems defining your expression you might want to read the section on

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
# Plot a tensor variable.
DefineTensorExpression("myten", "{{u,v,w},{u,v,w},{u,v,w}}")
AddPlot("Tensor", "myten")
DrawPlots()
```

5.4.43 DefineVectorExpression

Synopsis:

```
DefineVectorExpression(variableName, expression) -> integer
```

variableName [string] The name of the variable to be created.

expression [string] The expression definition as a string.

return type [CLI_return_t] The DefineVectorExpression function returns 1 on success and 0 on failure.

Description:

DefineVectorExpression creates new vector variables. Vector variables define a vector field over a mesh and are used by the Vector plot. A 2D vector would consist of 2 components. A 3D vector would consist of 3 components.

The variableName argument is a string that contains the name of the new variable. You can pass the name of an existing expression if you want to provide a new expression definition. The expression argument is a string that contains the definition of the new variable in terms of *built-in expressions* and pre-existing variable names using VisIt's *expression grammar*. If you run into problems defining your expression you might want to read the section on *expression compatibility gotchas*.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
# Plot a vector variable.
DefineVectorExpression("myvec", "{u,v,w}")
AddPlot("Vector", "myvec")
DrawPlots()
```

5.4.44 DeleteActivePlots

Synopsis:

```
DeleteActivePlots() -> integer
```

return type [CLI_return_t] The Delete functions return an integer value of 1 for success and 0 for failure.

Description:

The Delete functions delete plots from the active window's plot list. The DeleteActivePlots function deletes all of the active plots from the plot list. There is no way to retrieve a plot once it has been deleted from the plot list. The active plots are set using the SetActivePlots function. The DeleteAllPlots function deletes all plots from the active window's plot list regardless of whether or not they are active.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "d")
AddPlot("Contour", "u")
AddPlot("Mesh", "curvmesh2d")
DrawPlots()
DeleteActivePlots() # Delete the mesh plot
DeleteAllPlots() # Delete the pseudocolor and contour plots.
```

5.4.45 DeleteAllPlots

Synopsis:

```
DeleteAllPlots() -> integer
```

return type [CLI_return_t] The Delete functions return an integer value of 1 for success and 0 for failure.

Description:

The Delete functions delete plots from the active window's plot list. The DeleteActivePlots function deletes all of the active plots from the plot list. There is no way to retrieve a plot once it has been deleted from the plot list. The active plots are set using the SetActivePlots function. The DeleteAllPlots function deletes all plots from the active window's plot list regardless of whether or not they are active.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "d")
AddPlot("Contour", "u")
AddPlot("Mesh", "curvmesh2d")
DrawPlots()
DeleteActivePlots() # Delete the mesh plot
DeleteAllPlots() # Delete the pseudocolor and contour plots.
```

5.4.46 DeleteDatabaseCorrelation

Synopsis:

```
DeleteDatabaseCorrelation(name) -> integer
```

name [string] The name of the database correlation to delete.

return type [CLI_return_t] The DeleteDatabaseCorrelation function returns 1 on success and 0 on failure.

Description:

The DeleteDatabaseCorrelation function deletes a specific database correlation and its associated time slider. If you delete a database correlation whose time slider is being used for the current time slider, the time slider will be reset to the time slider of the next best suited database correlation. You can use the DeleteDatabaseCorrelation function to remove database correlations that you no longer need such as when you choose to examine databases that have nothing to do with your current databases.

Example:

```
#% visit -cli
dbs = ("dbA00.pdb", "dbB00.pdb")
OpenDatabase(dbs[0])
AddPlot("FilledBoundary", "material(mesh)")
OpenDatabase(dbs[1])
AddPlot("FilledBoundary", "material(mesh)")
DrawPlots()
CreateDatabaseCorrelation("common", dbs, 1)
SetTimeSliderState(10)
DeleteAllPlots()
DeleteDatabaseCorrelation("common")
CloseDatabase(dbs[0])
CloseDatabase(dbs[1])
```

5.4.47 DeleteExpression

Synopsis:

```
DeleteExpression(variableName) -> integer
```

variableName [string] The name of the expression variable to be deleted.

return type [CLI_return_t] The DeleteExpression function returns 1 on success and 0 on failure.

Description:

The DeleteExpression function deletes the definition of an expression. The variableName argument is a string containing the name of the variable expression to be deleted. Any plot that uses an expression that has been deleted will fail to regenerate if its attributes are changed.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
AddPlot("Pseudocolor", "myvar") # Plot the expression variable.
DrawPlots()
DeleteExpression("myvar") # Delete the expression variable myvar.
```

5.4.48 DeleteNamedSelection

Synopsis:

```
DeleteNamedSelection(name) -> integer
```

name [string] The name of a named selection.

return type [CLI_return_t] The DeleteNamedSelection function returns 1 for success and 0 for failure.

Description:

Named Selections allow you to select a group of elements (or particles). One typically creates a named selection from a group of elements and then later applies the named selection to another plot (thus reducing the set of elements displayed to the ones from when the named selection was created). If you have created a named selection that you are no longer interested in, you can delete it with the DeleteNamedSelection function.

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/wave*.silo database"
OpenDatabase(db)
AddPlot("Pseudocolor", "pressure")
AddOperator("Clip")
c = ClipAttributes()
c.plane1Origin = (0,0.6,0)
c.plane1Normal = (0,-1,0)
SetOperatorOption(c)
DrawPlots()
CreateNamedSelection("els_above_y")
SetTimeSliderState(40)
DeleteNamedSelection("els_above_y")
CreateNamedSelection("els_above_y")
```

5.4.49 DeleteOperatorKeyframe

Synopsis:

```
DeleteOperatorKeyframe(plotIndex, operatorIndex, frame)
```

plotIndex [integer] A zero-based integer value corresponding to a plot's index in the plot list.

operatorIndex [integer] A zero-based integer value corresponding to an operators's index in the plot.

frame [integer] A zero-based integer value corresponding to a plot keyframe at a particular animation frame.

Description:

The DeleteOperatorKeyframe function removes an operator keyframe from a specific operator and plot. An operator keyframe is the set of operator attributes at a specified frame. Operator keyframes are used to determine what operator attributes will be used at a given animation frame when VisIt's keyframing mode is enabled. The plotIndex argument is a zero-based integer that is used to identify a plot in the plot list. The operatorIndex is a zero-based integer that is used to identify an operator of a plot. The frame argument is a zero-based integer that is used to identify the frame at which a keyframe is to be removed.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
k = GetKeyframeAttributes()
k.enabled,k.nFrames,k.nFramesWasUserSet = 1,11,1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "hardyglobal")
AddOperator("Slice")
# Set up operator keyframes so the Slice operator's percent will change
# over time.
s0 = SliceAttributes()
s0.originType = s0.Percent
s0.originPercent = 0
s1 = SliceAttributes()
s1.originType = s1.Percent
s1.originPercent = 100
SetOperatorOptions(s0)
SetTimeSliderState(10)
SetOperatorOptions(s1)
```

(continues on next page)

(continued from previous page)

```
SetTimeSliderState(0)
DrawPlots()
ListPlots()
# Iterate over all animation frames and wrap around to the first one.
for i in list(range(TimeSliderGetNStates())) + [0]:
    SetTimeSliderState(i)
# Delete the operator keyframe at frame 10 so the slice won't
# change anymore.
DeleteOperatorKeyframe(0, 0, 10)
ListPlots()
SetTimeSliderState(10)
```

5.4.50 DeletePlotDatabaseKeyframe

Synopsis:

```
DeletePlotDatabaseKeyframe(plotIndex, frame)
```

plotIndex [integer] A zero-based integer value corresponding to a plot's index in the plot list.

frame [integer] A zero-based integer value corresponding to a database keyframe at a particular animation frame.

Description:

The DeletePlotDatabaseKeyframe function removes a database keyframe from a specific plot. A database keyframe represents the database time state that will be used at a given animation frame when VisIt's keyframing mode is enabled. The plotIndex argument is a zero-based integer that is used to identify a plot in the plot list. The frame argument is a zero-based integer that is used to identify the frame at which a database keyframe is to be removed for the specified plot.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
k = GetKeyframeAttributes()
k.enabled,k.nFrames,k.nFramesWasUserSet = 1,20,1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "pressure")
SetPlotDatabaseState(0, 0, 60)
# Repeat time state 60 for the first few animation frames by adding a
# keyframe at frame 3.
SetPlotDatabaseState(0, 3, 60)
SetPlotDatabaseState(0, 19, 0)
DrawPlots()
ListPlots()
# Delete the database keyframe at frame 3.
DeletePlotDatabaseKeyframe(0, 3)
ListPlots()
```

5.4.51 DeletePlotKeyframe

Synopsis:

```
DeletePlotKeyframe(plotIndex, frame)
```

plotIndex [integer] A zero-based integer value corresponding to a plot's index in the plot list.

frame [integer] A zero-based integer value corresponding to a plot keyframe at a particular animation frame.

Description:

The DeletePlotKeyframe function removes a plot keyframe from a specific plot. A plot keyframe is the set of plot attributes at a specified frame. Plot keyframes are used to determine what plot attributes will be used at a given animation frame when VisIt's keyframing mode is enabled. The plotIndex argument is a zero-based integer that is used to identify a plot in the plot list. The frame argument is a zero-based integer that is used to identify the frame at which a keyframe is to be removed.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
k = GetKeyframeAttributes()
k.enabled,k.nFrames,k.nFramesWasUserSet = 1,20,1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "pressure")
# Set up plot keyframes so the Pseudocolor plot's min will change
# over time.
p0 = PseudocolorAttributes()
p0.minFlag,p0.min = 1,0.0
p1 = PseudocolorAttributes()
p1.minFlag,p1.min = 1, 0.5
SetPlotOptions(p0)
SetTimeSliderState(19)
SetPlotOptions(p1)
SetTimeSliderState(0)
DrawPlots()
ListPlots()
# Iterate over all animation frames and wrap around to the first one.
for i in list(range(TimeSliderGetNStates())) + [0]:
    SetTimeSliderState(i)
# Delete the plot keyframe at frame 19 so the min won't
# change anymore.
DeletePlotKeyframe(19)
ListPlots()
SetTimeSliderState(10)
```

5.4.52 DeleteViewKeyframe

Synopsis:

```
DeleteViewKeyframe(frame)
```

frame [integer] A zero-based integer value corresponding to a view keyframe at a particular animation frame.

Description:

The DeleteViewKeyframe function removes a view keyframe at a specified frame. View keyframes are used to determine what view will be used at a given animation frame when VisIt's keyframing mode is enabled. The frame argument is a zero-based integer that is used to identify the frame at which a keyframe is to be removed.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
k = KeyframeAttributes()
k.enabled, k.nFrames, k.nFramesWasUserSet = 1,10,1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "u")
DrawPlots()
# Set some view keyframes
SetViewKeyframe()
v1 = GetView3D()
v1.viewNormal = (-0.66609, 0.337227, 0.665283)
v1.viewUp = (0.157431, 0.935425, -0.316537)
SetView3D(v1)
SetTimeSliderState(9)
SetViewKeyframe()
ToggleCameraViewMode()
# Iterate over the animation frames to watch the view change.
for i in list(range(10)) + [0]:
    SetTimeSliderState(i)
# Delete the last view keyframe, which is on frame 9.
DeleteViewKeyframe(9)
# Iterate over the animation frames again. The view should stay
# the same.
for i in range(10):
    SetTimeSliderState(i)
```

5.4.53 DeleteWindow

Synopsis:

```
DeleteWindow() -> integer
```

return type [CLI_return_t] The DeleteWindow function returns an integer value of 1 for success and 0 for failure.

Description:

The DeleteWindow function deletes the active visualization window and makes the visualization window with the smallest window index the new active window. This function has no effect when there is only one remaining visualization window.

Example:

```
#% visit -cli
DeleteWindow() # Does nothing since there is only one window
AddWindow()
DeleteWindow() # Deletes the new window.
```

5.4.54 DemoteOperator

Synopsis:

```
DemoteOperator(opIndex) -> integer
DemoteOperator(opIndex, applyToAllPlots) -> integer
```

opIndex [integer] A zero-based integer corresponding to the operator that should be demoted.

applyToAllPlots [integer] An integer flag that causes all plots in the plot list to be affected when it is non-zero.

return type [CLI_return_t] DemoteOperator returns 1 on success and 0 on failure.

Description:

The DemoteOperator function moves an operator closer to the database in the visualization pipeline. This allows you to change the order of operators that have been applied to a plot without having to remove them from the plot. For example, consider moving a Slice to before a Reflect operator when it had been the other way around. Changing the order of operators can result in vastly different results for a plot. The opposite function is PromoteOperator.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
AddOperator("Slice")
s = SliceAttributes()
s.project2d = 0
s.originPoint = (0,5,0)
s.originType=s.Point
s.normal = (0,1,0)
s.upAxis = (-1,0,0)
SetOperatorOptions(s)
AddOperator("Reflect")
DrawPlots()
# Now reflect before slicing. We'll only get 1 slice plane
# instead of 2.
DemoteOperator(1)
DrawPlots()
```

5.4.55 DisableRedraw

Synopsis:

```
DisableRedraw()
```

Description:

The DisableRedraw function prevents the active visualization window from ever redrawing itself. This is a useful function to call when performing many operations that would cause unnecessary redraws in the visualization window. The effects of this function are undone by calling the RedrawWindow function.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Contour", "u")
AddPlot("Pseudocolor", "w")
DrawPlots()
DisableRedraw()
AddOperator("Slice")
# Set the slice operator attributes
# Redraw now that thw operator attributes are set. This will
# prevent 1 redraw.
RedrawWindow()
```

5.4.56 DrawPlots

Synopsis:

```
DrawPlots() -> integer
```

return type [CLI_return_t] The DrawPlots function returns an integer value of 1 for success and 0 for failure.

Description:

The DrawPlots function forces all new plots in the plot list to be drawn. Plots are added and then their attributes are modified. Finally, the DrawPlots function is called to make sure all of the new plots draw themselves in the visualization window. This function has no effect if all of the plots in the plot list are already drawn.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots() # Draw the new pseudocolor plot.
```

5.4.57 EnableTool

Synopsis:

```
EnableTool(toolIndex, activeFlag)
```

toolIndex [integer] This is an integer that corresponds to an interactive tool. (Plane tool = 0, Line tool = 1, Plane tool = 2, Box tool = 3, Sphere tool = 4, Axis Restriction tool = 5)

activeFlag [integer] An integer value of 1 enables the tool while a value of 0 disables the tool.

return [CLI_return_t] The EnableToole function returns 1 on success and 0 on failure.

Description:

The EnableTool function is used to set the enabled state of an interactive tool in the active visualization window. The toolIndex argument is an integer index that corresponds to a certain tool. The activeFlag argument is an integer value (0 or 1) that indicates whether to turn the tool on or off.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
EnableTool(0, 1) # Turn on the line tool.
EnableTool(1,1) # Turn on the plane tool.
EnableTool(2,1) # Turn on the sphere tool.
EnableTool(2,0) # Turn off the sphere tool.
```

5.4.58 EvalCubic

Synopsis:

```
EvalCubic(t, c0, c1, c2, c3) -> f(t)
```

t [double] A floating point number in the range [0., 1.] that represents the distance from c0 to c3.

c0 [arithmetic expression object] The first control point. f(0) = c0. Any object that can be used in an arithmetic expression can be passed for c0.

c1 [arithmetic expression object] The second control point. Any object that can be used in an arithmetic expression can be passed for c1.

c2 [arithmetic expression object] The third control point. Any object that can be used in an arithmetic expression can be passed for c2.

c3 [arithmetic expression object] The last control point. f(1) = c3. Any object that can be used in an arithmetic expression can be passed for c3.

return [double] The EvalCubic function returns the interpolated value for t taking into account the control points that were passed in.

Description:

The EvalCubic function takes in four objects and blends them using a cubic polynomial and returns the blended value.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
v0 = GetView3D()
# rotate the plots
v1 = GetView3D()
# rotate the plots again.
v2 = GetView3D()
# rotate the plots one last time.
v3 = GetView3D()
# Fly around the plots using the views that have been specified.
nSteps = 100
for i in range(nSteps):
    t = float(i) / float(nSteps - 1)
    newView = EvalCubic(t, v0, v1, v2, v3)
    SetView3D(newView)
```

5.4.59 EvalCubicSpline

Synopsis:

```
EvalCubicSpline(t, weights, values) -> f(t)
```

t [double] A floating point number in the range [0., 1.] that represents the distance from the first control point to the last control point.

weights [tuple of doubles] A tuple of N floating point values in the range [0., 1.] that represent how far along in parameterized space, the values will be located.

values [tuple of arithmetic expression object] A tuple of N objects to be blended. Any objects that can be used in arithmetic expressions can be passed in.

return [double] The EvalCubicSpline function returns the interpolated value for t considering the objects that were passed in.

Description: The EvalCubicSpline function takes in N objects to be blended and blends them using piece-wise cubic polynomials and returns the blended value.

5.4.60 EvalLinear

Synopsis:

```
EvalLinear(t, value1, value2) -> f(t)
```

t [double] A floating point value in the range [0., 1.] that represents the distance between the first and last control point in parameterized space.

value1 [arithmetic expression object] Any object that can be used in an arithmetic expression. f(0) = value1.

value2 [arithmetic expression object] Any object that can be used in an arithmetic expression. f(1) = value2.

return [double] The EvalLinear function returns an interpolated value for t based on the objects that were passed in.

Description: The EvalLinear function linearly interpolates between two values and returns the result.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
c0 = GetView3D()
c1 = GetView3D()
c1.viewNormal = (-0.499159, 0.475135, 0.724629)
c1.viewUp = (0.196284, 0.876524, -0.439521)
nSteps = 100
for i in range(nSteps):
    t = float(i) / float(nSteps - 1)
    v = EvalLinear(t, c0, c1)
    SetView3D(v)
```

5.4.61 EvalQuadratic

Synopsis:

```
EvalQuadratic(t, c0, c1, c2) -> f(t)
```

t [double] A floating point number in the range [0., 1.] that represents the distance from c0 to c3.

c0 [arithmetic expression object] The first control point. f(0) = c0. Any object that can be used in an arithmetic expression can be passed for c0.

c1 [arithmetic expression object] The second control point. Any object that can be used in an arithmetic expression can be passed for c1.

c2 [arithmetic expression object] The last control point. f(1) = c2. Any object that can be used in an arithmetic expression can be passed for c2.

return [double] The EvalQuadratic function returns the interpolated value for t taking into account the control points that were passed in.

Description: The EvalQuadratic function takes in four objects and blends them using a cubic polynomial and returns the blended value.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
v0 = GetView3D()
# rotate the plots
v1 = GetView3D()
# rotate the plots one last time.
v2 = GetView3D()
# Fly around the plots using the views that have been specified.
nSteps = 100
for i in range(nSteps):
    t = float(i) / float(nSteps - 1)
    newView = EvalQuadratic(t, v0, v1, v2)
    SetView3D(newView)
```

5.4.62 ExecuteMacro

Synopsis:

```
ExecuteMacro(name) -> value
```

name [string] The name of the macro to execute.

return type [value] The ExecuteMacro function returns the value returned from the user's macro function.

Description:

The ExecuteMacro function lets you call a macro function that was previously registered using the RegisterMacro method. Once macros are registered with a name, this function can be called whenever the macro function associated with that name needs to be called. The VisIt gui uses this function to tell the Python interface when macros need to be executed in response to user button clicks.

Example:

```
def SetupMyPlots():
    OpenDatabase('noise.silo')
    AddPlot('Pseudocolor', 'hardyglobal')
    DrawPlots()

RegisterMacro('Setup My Plots', SetupMyPlots)
ExecuteMacro('Setup My Plots')
```

5.4.63 ExportDatabase

Synopsis:

```
ExportDatabase(e) -> integer
ExportDatabase(e, o) -> integer
```

e [ExportDBAttributes object] An object of type ExportDBAttributes. This object specifies the options for exporting the database.

o [dictionary] A dictionary containing a key/value mapping to set options needed by the database exporter. The default values can be obtained in the appropriate format using GetExportOptions('plugin').

return type [CLI_return_t] Returns 1 on success, 0 on failure.

Description:

The ExportDatabase function exports the active plot for the current window to a file. The format of the file, name, and variables to be saved are specified using the ExportDBAttributes argument. Note that this functionality is distinct from the geometric formats of SaveWindow, such as STL. SaveWindow can only save surfaces (triangle meshes), while ExportDatabase can export an entire three dimensional data set.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv3d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Set the export database attributes.
e = ExportDBAttributes()
e.db_type = "Silo"
e.variables = ("u", "v")
e.filename = "test_ex_db"
# Set the export directory
e.dirname = "."
ExportDatabase(e)
```

5.4.64 Expressions

Synopsis:

```
Expressions() -> tuple of expression tuples
```

return type [tuple of expression tuples] The Expressions function returns a tuple of tuples that contain two strings that give the expression name and definition.

Description:

The Expressions function returns a tuple of tuples that contain two strings that give the expression name and definition. This function is useful for listing the available expressions or for iterating through a list of expressions in order to create plots.

Example:

```
#% visit -cli
SetWindowLayout(4)
DefineScalarExpression("sin_u", "sin(u)")
DefineScalarExpression("cos_u", "cos(u)")
DefineScalarExpression("neg_u", "-u")
DefineScalarExpression("bob", "sin_u + cos_u")
for i in range(1,5):
    SetActiveWindow(i)
    OpenDatabase("/usr/gapps/visit/data/globe.silo")
    exprName = Expressions()[i-1][0]
    AddPlot("Pseudocolor", exprName)
    DrawPlots()
```

5.4.65 Flatten

Synopsis:

```
Flatten(vars) -> dictionary
Flatten(vars, fillValue, nodeIds, zoneIds, nodeIJK, zoneIJK, zoneCenters,
          maxDataSize, forceNoSharedMemory) -> dictionary
```

return type [dictionary] Flatten returns a dictionary that contains different keys depending on the data. If the output contains node centered data then there will be 'nodeColumnNames' and 'nodeTable' entries. If the output contains zone centered data then there will be 'zoneColumnNames' and 'zoneTable' entries. If the query results in no output data, then an empty dictionary is returned. The 'Table' entries are compatible with numpy via the 'numpy.asarray()' function.

vars: The names of the desired variables (tuple of strings).

fillValue: The default value for a column if no data is present (float, default = 0.)

nodeIds: Whether or not the nodeIds should be included in the output table. (bool, default = True)

zoneIds: Whether or not the zoneIds should be included in the output table. (bool, default = True)

nodeIJK: Whether or not the nodeIJK should be included in the output table. (bool, default = True)

zoneIJK: Whether or not the zoneIJK should be included in the output table. (bool, default = True)

zoneCenters: Whether or not to add the central coordinates of each zone. (bool, default = False)

maxDataSize: The maximum output data size when not using shared memory, expressed in GB. This parameters exists because the default method of returning query results does not scale well up to large sizes. (float, default=1.024)

forceNoSharedMemory: An override that makes sure the function will NOT use shared memory to transport the output data to the VisIt CLI, even if the environment seems to support it. (bool, default = False)

Description:

Query the active plot for the data at each node/zone for the given variables. Data is returned as numpy compatible 2D arrays using numpy.asarray().

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/rect2d.silo"
OpenDatabase(db)
AddPlot("Pseudocolor", "d")
DrawPlots()
data = Flatten(("p", "d"))
if "nodeTable" in data:
  print(numpy.asarray(data["nodeTable"]))
if "zoneTable" in data:
  print(numpy.asarray(data["zoneTable"]))
```

5.4.66 GetDefaultContinuousColorTable

Synopsis:

```
GetDefaultContinuousColorTable() -> string
```

return type [string] Returns a string object containing the name of a color table.

Description:

A color table is a set of color values that are used as the colors for plots. VisIt supports two flavors of color table: continuous and discrete. A continuous color table is defined by a small set of color control points and the colors specified by the color control points are interpolated smoothly to fill in any gaps. Continuous color tables are used for plots that need to be colored smoothly by a variable (e.g. Pseudocolor plot). A discrete color table is a set of color control points that are used to color distinct regions of a plot (e.g. Subset plot). VisIt supports the notion of default continuous and default discrete color tables so plots can just use the "default" color table. This lets you change the color table used by many plots by just changing the "default" color table. The GetDefaultContinuousColorTable function returns the name of the default continuous color table. The GetDefaultDiscreteColorTable function returns the name of the default discrete color table.

Example:

```
#% visit -cli
print("Default continuous color table: %s" % GetDefaultContinuousColorTable())
print("Default discrete color table: %s" % GetDefaultDiscreteColorTable())
```

5.4.67 GetDefaultDiscreteColorTable

Synopsis:

```
GetDefaultDiscreteColorTable() -> string
```

return type [string] Returns a string object containing the name of a color table.

Description:

A color table is a set of color values that are used as the colors for plots. VisIt supports two flavors of color table: continuous and discrete. A continuous color table is defined by a small set of color control points and the colors specified by the color control points are interpolated smoothly to fill in any gaps. Continuous color tables are used for plots that need to be colored smoothly by a variable (e.g. Pseudocolor plot). A discrete color table is a set of color control points that are used to color distinct regions of a plot (e.g. Subset plot). VisIt supports the notion of default continuous and default discrete color tables so plots can just use the "default" color table. This lets you change the color table used by many plots by just changing the "default" color table. The GetDefaultContinuousColorTable function returns the name of the default continuous color table. The GetDefaultDiscreteColorTable function returns the name of the default discrete color table.

Example:

```
#% visit -cli
print("Default continuous color table: %s" % GetDefaultContinuousColorTable())
print("Default discrete color table: %s" % GetDefaultDiscreteColorTable())
```

5.4.68 GetActiveTimeSlider

Synopsis:

```
GetActiveTimeSlider() -> string
```

return type [string] The GetActiveTimeSlider function returns a string containing the name of the active time slider.

Description:

VisIt can support having multiple time sliders when you have opened more than one time-varying database. You can then use each time slider to independently change time states for each database or you can use a database correlation to change time states for all databases simultaneously. Every time-varying database has a database correlation and every database correlation has its own time slider. If you want to query to determine which time slider is currently the active time slider, you can use the GetActiveTimeSlider function.

Example:

```
#% visit -cli
OpenDatabase("dbA00.pdb")
AddPlot("FilledBoundary", "material(mesh)")
OpenDatabase("dbB00.pdb")
AddPlot("FilledBoundary", "materials(mesh)")
print("Active time slider: %s" % GetActiveTimeSlider())
CreateDatabaseCorrelation("common", ("dbA00.pdb", "dbB00.pdb"), 2)
print("Active time slider: %s" % GetActiveTimeSlider())
```

5.4.69 GetAnimationAttributes

Synopsis:

```
GetAnimationAttributes() -> AnimationAttributes object
```

return type [AnimationAttributes object] The GetAnimationAttributes function returns an AnimationAttributes object.

Description:

This function returns the current animation attributes, which contain the animation mode, increment, and playback speed.

Example:

```
a = GetAnimationAttributes()
print(a)
```

5.4.70 GetAnimationTimeout

Synopsis:

```
GetAnimationTimeout() -> integer
```

return type [CLI_return_t] The GetAnimationTimeout function returns an integer that contains the time interval, measured in milliseconds, between the rendering of animation frames.

Description:

The GetAnimationTimeout returns an integer that contains the time interval, measured in milliseconds, between the rendering of animation frames.

Example:

```
#% visit -cli
print("Animation timeout = %d" % GetAnimationTimeout())
```

5.4.71 GetAnnotationAttributes

Synopsis:

```
GetAnnotationAttributes() -> AnnotationAttributes object
```

return type [AnnotationAttributes object] The GetAnnotationAttributes function returns an AnnotationAttributes object that contains the annotation settings for the active visualization window.

Description:

The GetAnnotationAttributes function returns an AnnotationAttributes object that contains the annotation settings for the active visualization window. It is often useful to retrieve the annotation settings and modify them to suit the visualization.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
a = GetAnnotationAttributes()
print(a)
a.backgroundMode = a.BACKGROUNDMODE_GRADIENT
a.gradientColor1 = (0, 0, 255)
SetAnnotationAttributes(a)
```

5.4.72 GetAnnotationObject

Synopsis:

```
GetAnnotationObject(objectName) -> Annotation object
```

objectName [string] The name of the annotation object as returned by GetAnnotationObjectNames.

return type [Annotation object] GetAnnotationObject returns a reference to an annotation object that was created using the CreateAnnotationObject function, or a legend object created when a plot is added.

Description:

GetAnnotationObject returns a reference to an annotation object that was created using the CreateAnnotationObject function. The string argument specifies the name of the desired annotation object. It must be one of the names returned by GetAnnotationObjectNames. This function is not currently necessary unless the annotation object that you used to create an annotation has gone out of scope and you need to create another reference to the object to set its properties.

GetAnnotationObject can also return a reference to a legend, which is automatically created when a plot is added. It is associated with the name of the plot. While the plot's name can be seen in the list obtained from GetAnnotationObjectNames, it is better to get the plot's name from the PlotList, especially when multiple plots are present.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Mesh", "quadmesh")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
```

(continues on next page)

(continued from previous page)

```
a = CreateAnnotationObject("TimeSlider")
GetAnnotationObjectNames()
["Plot0000", "Plot0001", "TimeSlider1"]
ref = GetAnnotationObject("TimeSlider1")
print(ref)
# Get the name of the Pseudocolor plot for legend retrieval.
# It is the second plot in the plot list (which is 0-indexed)
plotName = GetPlotList().GetPlots(1).plotName
legend = GetAnnotationObject(plotName)
```

5.4.73 GetAnnotationObjectNames

Synopsis:

```
GetAnnotationObjectNames() -> tuple of strings
```

return type [tuple of strings] GetAnnotationObjectNames returns a tuple of strings of the names of all annotation objects defined for the currently active window.

Example:

```
names = GetAnnotationObjectNames()
names
["plot0000", "Line2D1", "TimeSlider1"]
```

5.4.74 GetCallbackArgumentCount

Synopsis:

```
GetCallbackArgumentCount(callbackName) -> integer
```

callbackName [string] The name of a callback function. This name is a member of the tuple returned by GetCallbackNames().

return type [CLI_return_t] The GetCallbackArgumentCount function returns the number of arguments associated with a particular callback function.

Example:

```
cbName = 'OpenDatabaseRPC'
count = GetCallbackArgumentCount(cbName)
print('The number of arguments for %s is: %d' % (cbName, count))
```

5.4.75 GetCallbackNames

Synopsis:

```
GetCallbackNames() -> tuple of string objects
```

return type [tuple of string objects] GetCallbackNames returns a tuple containing the names of valid callback function identifiers for use in RegisterCallback().

Description:

The GetCallbackNames function returns a tuple containing the names of valid callback function identifiers for use in RegisterCallback().

Example:

```
import visit
print(visit.GetCallbackNames())
```

5.4.76 GetDatabaseNStates

Synopsis:

```
GetDatabaseNStates() -> integer
```

return type [CLI_return_t] Returns the number of time states in the active database or 0 if there is no active database.

Description:

GetDatabaseNStates returns the number of time states in the active database, which is not the same as the number of states in the active time slider. Time sliders can have different lengths due to database correlations and keyframing. Use this function when you need the actual number of time states in the active database.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave*.silo database")
print("Number of time states: %d" % GetDatabaseNStates())
```

5.4.77 GetDebugLevel

Synopsis:

```
GetDebugLevel() -> integer
```

return type [CLI_return_t] The GetDebugLevel function returns the debug level of the VisIt module.

Description:

The GetDebugLevel and SetDebugLevel functions are used when debugging VisIt Python scripts. The GetDebugLevel function can be used in Python scripts to alter the behavior of the script. For instance, the debug level can be used to selectively print values to the console.

Example:

```
#% visit -cli -debug 2
print("VisIt's debug level is: %d" % GetDebugLevel())
```

5.4.78 GetDefaultFileOpenOptions

Synopsis:

```
GetDefaultFileOpenOptions(pluginName) -> dictionary
```

pluginName [string] The name of a plugin.

return type [dictionary] Returns a dictionary containing the options.

Description:

GetDefaultFileOpenOptions returns the current options used to open new files when a specific plugin is triggered.

Example:

```
#% visit -cli
OpenMDServer()
opts = GetDefaultFileOpenOptions("VASP")
opts["Allow multiple timesteps"] = 1
SetDefaultFileOpenOptions("VASP", opts)
OpenDatabase("CHGCAR")
```

5.4.79 GetDomains

Synopsis:

```
GetDomains() -> tuple of strings
```

return type [tuple of strings] GetDomains returns a tuple of strings.

Description:

GetDomains returns a tuple containing the names of all of the domain subsets for a plot that was created using a database with multiple domains. This function can be used in specialized logic that iterates over domains to turn them on or off in some programmed way.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/multi_ucd3d.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
doms = GetDomains()
print(doms)
# Turn off all but the last domain, one after the other.
for d in doms[:-1]:
    TurnDomainsOff(d)
```

5.4.80 GetEngineList

Synopsis:

```
GetEngineList() -> tuple of strings
GetEngineList(flag) -> tuple of tuples of strings
```

flag [integer] If flag is a non-zero integer then the function returns a tuple of tuples with information about simulations.

return type [tuple of strings] GetEngineList returns a tuple of strings that contain the names of the computers on which compute engines are running. If flag is a non-zero integer argument then the function returns a tuple of tuples where each tuple is of length 2. Element 0 contains the names of the computers where the engines are running. Element 1 contains the names of the simulations being run.

Description:

The GetEngineList function returns a tuple of strings containing the names of the computers on which compute engines are running. This function can be useful if engines are going to be closed and opened explicitly in the Python script. The contents of the tuple can be used to help determine which compute engines should be closed or they can be used to determine if a compute engine was successfully launched.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
OpenDatabase("mcr:/usr/gapps/visit/data/globe.silo")
AddPlot("Mesh", "mesh1")
DrawPlots()
for name in GetEngineList():
    print("VisIt has a compute engine running on %s" % name)
    CloseComputeEngine(GetEngineList()[1])
```

5.4.81 GetEngineProperties

Synopsis:

```
GetEngineProperties()             -> EngineProperties object
GetEngineProperties(engine)       -> EngineProperties object
GetEngineProperties(engine, sim) -> EngineProperties object
```

engine When engine is passed and it matches one of the computer names returned from GetEngineList() then the EngineProperties object for that engine is returned.

sim When both engine and sim arguments are passed, then the EngineProperties object for the simulation is returned.

return type [EngineProperties object] The EngineProperties object for the specified compute engine/sim.

Description:

GetEngineProperties returns an EngineProperties object containing the properties for the specified compute engine/sim. The EngineProperties let you discover information such as number of processors, etc for a compute engine/sim.

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/globe.silo"
OpenDatabase(db)
props = GetEngineProperties(GetEngineList()[0])
```

5.4.82 GetFlattenOutput

Synopsis:

```
GetFlattenOutput()            -> dictionary
```

Description:

GetFlattenOutput is used by the Flatten() CLI function to retrieve the output table from the 'Flatten' query. Prefer using the Flatten() CLI function to using the 'Flatten' query directly. Returns a dictionary containing numpy compatible 2D arrays and associated column names.

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/rect2d.silo"
OpenDatabase(db)
AddPlot("Pseudocolor", "d")
DrawPlots()
flattenOpts = dict()
flattenOpts["vars"] = ("p", "d")
Query("Flatten", flattenOpts)
data = GetFlattenOutput()
if "nodeTable" in data:
  print(numpy.asarray(data["nodeTable"]))
if "zoneTable" in data:
  print(numpy.asarray(data["zoneTable"]))
```

5.4.83 GetGlobalAttributes

Synopsis:

```
GetGlobalAttributes() -> GlobalAttributes object
```

return type [GlobalAttributes object] Returns a GlobalAttributes object that has been initialized.

Description:

The GetGlobalAttributes function returns a GlobalAttributes object that has been initialized with the current state of the viewer proxy's GlobalAttributes object. The GlobalAttributes object contains read-only information about the list of sources, the list of windows, and various flags that can be queried.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
g = GetGlobalAttributes()
print(g)
```

5.4.84 GetGlobalLineoutAttributes

Synopsis:

```
GetGlobalLineoutAttributes() -> GlobalLineoutAttributes object
```

return type [GlobalLineoutAttributes object] Returns an initialized GlobalLineoutAttributes object.

Description:

The GetGlobalLineoutAttributes function returns an initialized GlobalLineoutAttributes object. The GlobalLineoutAttributes, as suggested by its name, contains global properties that apply to all lineouts. You can use the GlobalLineoutAttributes object to turn on lineout sampling, specify the destination window, etc. for curve plots created as a result of performing lineouts. Once you make changes to the object by setting its properties, use the SetGlobalLineoutAttributes function to make VisIt use the modified global lineout attributes.

Example:

```
#% visit -cli
SetWindowLayout(4)
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
g = GetGlobalLineoutAttributes()
print(g)
g.samplingOn = 1
g.windowId = 4
g.createWindow = 0
g.numSamples = 100
SetGlobalLineoutAttributes(g)
Lineout((-3,2),(3,3),("default"))
```

5.4.85 GetInteractorAttributes

Synopsis:

```
GetInteractorAttributes() -> InteractorAttributes object
```

return type [InteractorAttributes object] Returns an initialized InteractorAttributes object.

Description:

The GetInteractorAttributes function returns an initialized InteractorAttributes object. The InteractorAttributes object can be used to set certain interactor properties. Interactors, can be thought of as how mouse clicks and movements are translated into actions in the vis window. To set the interactor attributes, first get the interactor attributes using the GetInteractorAttributes function. Once you've set the object's properties, call the SetInteractorAttributes function to make VisIt use the new interactor attributes.

Example:

```
#% visit -cli
ia = GetInteractorAttributes()
print(ia)
ia.showGuidelines = 0
SetInteractorAttributes(ia)
```

5.4.86 GetKeyframeAttributes

Synopsis:

```
GetKeyframeAttributes() -> KeyframeAttributes object
```

return type [KeyframeAttributes object] GetKeyframeAttributes returns an initialized KeyframeAttributes object.

Description:

Use the GetKeyframeAttributes function when you want to examine a KeyframeAttributes object so you can determine VisIt's state when it is in keyframing mode. The KeyframeAttributes object allows you to see whether VisIt is in keyframing mode and, if so, how many animation frames are in the current keyframe animation.

Example:

```
#% visit -cli
k = GetKeyframeAttributes()
print(k)
k.enabled,k.nFrames,k.nFramesWasUserSet = 1, 100, 1
SetKeyframeAttributes(k)
```

5.4.87 GetLastError

Synopsis:

```
GetLastError() -> string
GetLastError(int-clr) -> string
```

return type [string] GetLastError returns a string containing the last error message that VisIt issued since being cleared. If int-clr is present and is non-zero, once the message is retrieved it is also cleared.

Description:

The GetLastError function returns a string containing the last error message that VisIt issued since being cleared. If int-clr is present and is non-zero, once the message is retrieved it is also cleared.

Example:

```
#% visit -cli
OpenDatabase("/this/database/does/not/exist")
print("VisIt Error: %s" % GetLastError())
# Get last message into msg and then clear last error message to ""
msg = GetLastError(1)
```

5.4.88 GetLight

Synopsis:

```
GetLight(index) -> LightAttributes object
```

index [integer] A zero-based integer index into the light list. Index can be in the range [0,7].

return type [LightAttributes object] GetLight returns a LightAttributes object.

Description:

The GetLight function returns a LightAttributes object containing the attributes for a specific light. You can use the LightAttributes object that GetLight returns to set light properties and then you can pass the object to SetLight to make VisIt use the light properties that you've set.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "w")
p = PseudocolorAttributes()
p.colorTableName = "xray"
SetPlotOptions(p)
DrawPlots()
InvertBackgroundColor()
light = GetLight(0)
```

(continues on next page)

(continued from previous page)

```
print(light)
light.enabledFlag = 1
light.direction = (0,-1,0)
light.color = (255,0,0,255)
SetLight(0, light)
light.color,light.direction = (0,255,0,255), (-1,0,0)
SetLight(1, light)
```

5.4.89 GetLocalHostName

Synopsis:

```
GetLocalHostName() -> string
```

return type [string] Both functions return a string.

Description:

The GetLocalHostName function returns a string that contains the name of the local computer.

Example:

```
## visit -cli
print("Local machine name is: %s" % GetLocalHostName())
print("My username: %s" % GetLocalUserName())
```

5.4.90 GetLocalUserName

Synopsis:

```
GetLocalUserName() -> string
```

return type [string] Both functions return a string.

Description:

The GetLocalUserName function returns a string containing the name of the user running VisIt.

Example:

```
## visit -cli
print("Local machine name is: %s" % GetLocalHostName())
print("My username: %s" % GetLocalUserName())
```

5.4.91 GetMachineProfile

Synopsis:

```
GetMachineProfile(hostname) -> MachineProfile object
```

hostname : string

return type [MachineProfile object] MachineProfile for hostname.

Description:

Gets the MachineProfile for a given hostname

5.4.92 GetMachineProfileNames

Synopsis:

```
GetMachineProfileNames() -> [hostname1, hostname2, ...]
```

return type [list of strings] A list of MachineProfile hostnames

Description:

Returns a list of hostnames that can be used to get a specific MachineProfile

5.4.93 GetMaterialAttributes

Synopsis:

```
GetMaterialAttributes() -> MaterialAttributes object
```

return type [MaterialAttributes object] Returns a MaterialAttributes object.

Description:

The GetMaterialAttributes function returns a MaterialAttributes object that contains VisIt's current material interface reconstruction settings. You can set properties on the MaterialAttributes object and then pass it to SetMaterialAttributes to make VisIt use the new material attributes that you've specified:

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/allinone00.pdb")
AddPlot("Pseudocolor", "mesh/mixvar")
p = PseudocolorAttributes()
p.min,p.minFlag = 4.0, 1
p.max,p.maxFlag = 13.0, 1
SetPlotOptions(p)
DrawPlots()
# Tell VisIt to always do material interface reconstruction.
m = GetMaterialAttributes()
m.forceMIR = 1
SetMaterialAttributes(m)
ClearWindow()
# Redraw the plot forcing VisIt to use the mixed variable information.
DrawPlots()
```

5.4.94 GetMaterials

Synopsis:

```
GetMaterials() -> tuple of strings
```

return type [tuple of strings] The GetMaterials function returns a tuple of strings.

Description:

The GetMaterials function returns a tuple of strings containing the names of the available materials for the current plot's database. Note that the active plot's database must have materials for this function to return a tuple that has any string objects in it. Also, you must have at least one plot. You can use the materials returned by the GetMaterials function for a variety of purposes including turning materials on or off.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/allinone00.pdb")
AddPlot("Pseudocolor", "mesh/mixvar")
DrawPlots()
mats = GetMaterials()
for m in mats[:-1]:
    TurnMaterialOff(m)
```

5.4.95 GetMeshManagementAttributes

Synopsis:

```
GetMeshManagementAttributes() -> MeshmanagementAttributes object
```

return type [MeshmanagementAttributes object] Returns a MeshmanagementAttributes object.

Description:

The GetMeshmanagementAttributes function returns a MeshmanagementAttributes object that contains VisIt's current mesh discretization settings. You can set properties on the MeshManagementAttributes object and then pass it to SetMeshManagementAttributes to make VisIt use the new material attributes that you've specified:

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/csg.silo")
AddPlot("Mesh", "csgmesh")
DrawPlots()
# Tell VisIt to always do material interface reconstruction.
mma = GetMeshManagementAttributes()
mma.discretizationTolernace = (0.01, 0.025)
SetMeshManagementAttributes(mma)
ClearWindow()
# Redraw the plot forcing VisIt to use the mixed variable information.
DrawPlots()
```

5.4.96 GetMetaData

Synopsis:

```
GetMetaData(db) -> avtDatabaseMetaData object
GetMetaData(db, ts) -> avtDatabaseMetaData object
```

db [string] The name of the database for which to return metadata.

ts [integer] An optional integer indicating the time state at which to open the database.

return type [avtDatabaseMetaData object] The GetMetaData function returns an avtDatabaseMetaData object.

Description:

VisIt relies on metadata to populate its variable menus and make important decisions. Metadata can be used to create complex scripts whose behavior adapts based on the contents of the database.

Example:

```
md = GetMetaData('noise.silo')
for i in range(md.GetNumScalars()):
    AddPlot('Pseudocolor', md.GetScalars(i).name)
DrawPlots()
```

5.4.97 GetNumPlots

Synopsis:

```
GetNumPlots() -> integer
```

return type [CLI_return_t] Returns the number of plots in the active window.

Description:

The GetNumPlots function returns the number of plots in the active window.

Example:

```
#% visit -cli
print("Number of plots", GetNumPlots())
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "d")
print("Number of plots", GetNumPlots())
AddPlot("Mesh", "curvmesh2d")
DrawPlots()
print("Number of plots", GetNumPlots())
```

5.4.98 GetOperatorOptions

Synopsis:

```
GetOperatorOptions(index) -> operator attributes object
```

index [integer] The integer index of the operator within the plot's list of operators.

return type [operator attributes object] The GetOperatorOptions function returns an operator attributes object.

Description:

This function is provided to make it easy to probe the current attributes for a specific operator on the active plot.

Example:

```
AddPlot('Pseudocolor', 'temperature')
AddOperator('Transform')
AddOperator('Transform')
t = GetOperatorOptions(1)
print('Attributes for the 2nd Transform operator:', t)
```

5.4.99 GetPickAttributes

Synopsis:

```
GetPickAttributes() -> PickAttributes object
```

return type [PickAttributes object] GetPickAttributes returns a PickAttributes object.

Description:

The GetPickAttributes object returns the pick settings that VisIt is currently using when it performs picks. These settings mainly determine which pick information is displayed when pick results are printed out but they can also be used to select auxiliary variables and generate time curves. You can examing the settings and you can set properties on the returned object. Once you've changed pick settings by setting properties on the object, you can pass the altered object to the SetPickAttributes function to force VisIt to use the new pick settings.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/allinone00.pdb")
AddPlot("Pseudocolor", "mesh/ireg")
DrawPlots()
p = GetPickAttributes()
print(p)
p.variables = ("default", "mesh/a", "mesh/mixvar")
SetPickAttributes(p)
# Now do some interactive picks and you'll see pick information
# for more than 1 variable.
p.doTimeCurve = 1
SetPickAttributes(p)
# Now do some interactive picks and you'll get time-curves in
# a new window.
```

5.4.100 GetPickOutput

Synopsis:

```
GetPickOutput() -> string
```

return type [string] GetPickOutput returns a string containing the output from the last pick.

Description:

The GetPickOutput returns a string object that contains the output from the last pick.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
ZonePick(coord=(0.4, 0.6, 0), vars=("default", "u", "v"))
s = GetPickOutput()
print(s)
```

5.4.101 GetPickOutputObject

Synopsis:

```
GetPickOutputObject() -> dictionary
```

return type [dictionary] GetPickOutputObject returns a dictionary produced by the last pick.

Description:

GetPickOutputObject returns a dictionary object containing output from the last pick.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
ZonePick(coord=(0.4, 0.6, 0), vars=("default", "u", "v"))
o = GetPickOutputObject()
print(o)
```

5.4.102 GetPipelineCachingMode

Synopsis:

```
GetPipelineCachingMode() -> integer
```

return type [CLI_return_t] The GetPipelineCachingMode function returns 1 if pipelines are being cached and 0 otherwise.

Description:

The GetPipelineCachingMode function returns whether or not pipelines are being cached in the viewer. For animations of long time sequences, it is often useful to turn off pipeline caching so the viewer does not run out of memory.

Example:

```
#%visit -cli
offon = ("off", "on")
print("Pipeline caching is %s" % offon[GetPipelineCachingMode()])
```

5.4.103 GetPlotInformation

Synopsis:

```
GetPlotInformation() -> dictionary
```

return type [dictionary] GetPlotInformation returns a dictionary.

Description:

The GetPlotInformation function returns information about the active plot. For example, a Curve plot will return the xy pairs that comprise the curve. The tuple is arranged <x1, y1, x2, y2, . . . , xn, yn>.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
Lineout((0, 0), (1, 1))
SetActiveWindow(2)
info = GetPlotInformation()
lineout = info["Curve"]
print("The first lineout point is: [%g, %g] " % lineout[0], lineout[1])
```

5.4.104 GetPlotList

Synopsis:

```
GetPlotList() -> PlotList object
```

return type [PlotList object] The GetPlotList function returns a PlotList object.

Description:

The GetPlotList function returns a copy of the plot list that gets exchanged between VisIt's viewer and its clients. The plot list object contains the list of plots, along with the databases, and any operators that are applied to each plot. Changing this object has NO EFFECT but it can be useful when writing complex functions that need to know about the plots and operators that exist within a visualization window

Example:

```
# Copy plots (without operators to window 2)
pL = GetPlotList()
AddWindow()
for i in range(pL.GetNumPlots()):
    AddPlot(PlotPlugins()[pL.GetPlots(i).plotType], pL.GetPlots(i).plotVar)
DrawPlots()
```

5.4.105 GetPlotOptions

Synopsis:

```
GetPlotOptions() -> plot attributes object
```

return type [plot attributes object] The GetPlotOptions function returns a plot attributes object whose type varies depending the selected plots.

Description:

This function is provided to make it easy to probe the current attributes for the selected plot.

Example:

```
pc = GetPlotOptions()
pc.legend = 0
SetPlotOptions(pc)
```

5.4.106 GetPreferredFileFormats

Synopsis:

```
GetPreferredFileFormats() -> tuple of strings
```

return type [tuple of strings] The GetPreferredFileFormats returns the current list of preferred plugins.

Description:

The GetPreferredFileFormats method is a way to get the list of file format reader plugins which are tried before any others. These IDs are full IDs, not just names, and are tried in order.

Example:

```
GetPreferredFileFormats()
# returns ('Silo_1.0',)
```

5.4.107 GetQueryOutputObject

Synopsis:

```
GetQueryOutputObject() -> dictionary or value
```

return type [dictionary or value] GetQueryOutputObject returns a dictionary or value.

Description:

GetQueryOutputObject, GetQueryOutputString, GetQueryOutputValue and GetQueryOutputXML all return output from the last query. GetQueryOutputObject returns a dictionary of the output of the last query.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
Query("MinMax")
obj = GetQueryOutputObject()
print("The min is: %g and the max is: %g" % (obj["min"], obj["max"]))
```

5.4.108 GetQueryOutputString

Synopsis:

```
GetQueryOutputString() -> string
```

return type [string] GetQueryOutputString returns a string.

Description:

GetQueryOutputObject, GetQueryOutputString, GetQueryOutputValue and GetQueryOutputXML all return output from the last query. GetQueryOutputString returns a string containing the output of the last query.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
Query("MinMax")
print(GetQueryOutputString())
```

5.4.109 GetQueryOutputValue

Synopsis:

```
GetQueryOutputValue() -> double, tuple of doubles
```

return type [double, tuple of doubles] GetQueryOutputValue returns a single double precision number, a tuple of double precision numbers, or None if an error occured.

Description:

GetQueryOutputObject, GetQueryOutputString, GetQueryOutputValue and GetQueryOutputXML all return output from the last query. GetQueryOutputValue returns a single number or tuple of numbers, depending on the nature of the last query to be executed. If an error occurs, GetQueryOutputValue returns None.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
Query("MinMax")
min_max = GetQueryOutputValue()
if min_max != None:
    print("The min is: %g and the max is: %g" % min_max)
```

5.4.110 GetQueryOutputXML

Synopsis:

```
GetQueryOutputXML() -> string
```

return type [string] GetQueryOutputXML returns an XML string.

Description:

GetQueryOutputObject, GetQueryOutputString, GetQueryOutputValue and GetQueryOutputXML all return output from the last query. GetQueryOutputXML returns a XML string containing the output of the last query.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
Query("MinMax")
print(GetQueryOutputXML())
```

5.4.111 GetQueryOverTimeAttributes

Synopsis:

```
GetQueryOverTimeAttributes() -> QueryOverTimeAttributes object
```

return type [QueryOverTimeAttributes object] GetQueryOverTimeAttributes returns a QueryOverTimeAttributes object.

Description:

The GetQueryOverTimeAttributes function returns a QueryOverTimeAttributes object containing the settings that VisIt currently uses for query over time. You can use the returned object to change those settings by first setting object properties and then by passing the modified object to the SetQueryOverTimeAttributes function.

Example:

```
#% visit -cli
SetWindowLayout(4)
OpenDatabase("/usr/gapps/visit/data/allinone00.pdb")
AddPlot("Pseudocolor", "mesh/mixvar")
DrawPlots()
qot = GetQueryOverTimeAttributes()
print(qot)
# Make queries over time go to window 4.
qot.createWindow,q.windowId = 0, 4
SetQueryOverTimeAttributes(qot)
QueryOverTime("Min")
# Make queries over time only use half of the number of time states.
endTime = GetDatabaseNStates() / 2
QueryOverTime("Min", end_time=endTime)
ResetView()
```

5.4.112 GetQueryParameters

Synopsis:

```
GetQueryParameters(name) -> dictionary
```

name [string] The named query.

return type [dictionary] A python dictionary.

Description:

The GetQueryParameters function returns a Python dictionary containing the default parameters for the named query, or None if the query does not accept additional parameters. The returned dictionary (if any) can then be modified if necessary and passed back as an argument to the Query function.

Example:

```
#% visit -cli
minMaxInput = GetQueryParameters("MinMax")
minMaxInput["use_actual_data"] = 1
Query("MinMax", minMaxInput)
xrayInput = GetQueryParameters("XRay Image")
xrayInput["origin"]=(0.5, 2.5, 0.)
```

(continues on next page)

(continued from previous page)

```
xrayInput["image_size"]=(300,300)
xrayInput["vars"]=("p", "d")
Query("XRay Image", xrayInput)
```

5.4.113 GetRenderingAttributes

Synopsis:

```
GetRenderingAttributes() -> RenderingAttributes object
```

return type [RenderingAttributes object] Returns a RenderingAttributes object.

Description:

The GetRenderingAttributes function returns a RenderingAttributes object that contains the rendering settings that VisIt currently uses. The RenderingAttributes object contains information related to rendering such as whether or not specular highlights or shadows are enabled. The RenderingAttributes object also contains information scalable rendering such as whether or not it is currently in use and the scalable rendering threshold.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Surface", "hgslice")
DrawPlots()
v = GetView3D()
v.viewNormal = (-0.215934, -0.454611, 0.864119)
v.viewUp = (0.973938, -0.163188, 0.157523)
v.imageZoom = 1.64765
SetView3D(v)
light = GetLight(0)
light.direction = (0,1,-1)
SetLight(0, light)
r = GetRenderingAttributes()
r.scalableActivationMode = r.Always
r.doShadowing = 1
SetRenderingAttributes(r)
```

5.4.114 GetSaveWindowAttributes

Synopsis:

```
GetSaveWindowAttributes() -> SaveWindowAttributes object
```

return type [SaveWindowAttributes object] This function returns a VisIt SaveWindowAttributes object that contains the attributes used in saving windows.

Description:

The GetSaveWindowAttributes function returns a SaveWindowAttributes object that is a structure containing several fields which determine how windows are saved to files. The object that us returned can be modified and used to set the save window attributes.

Example:

```
#% visit -cli
s = GetSaveWindowAttributes()
print(s)
s.width = 600
s.height = 600
s.format = s.RGB
print(s)
```

5.4.115 GetSelection

Synopsis:

```
GetSelection(name) -> SelectionProperties object
```

name [string] The name of the selection whose properties we want to retrieve.

return type [SelectionProperties object] The GetSelection function returns a SelectionProperties object.

Description:

Named selections have properties that describe how the selection is defined. This function lets you query those selection properties.

Example:

```
CreateNamedSelection('selection1')
s = GetSelection('selection1')
s.selectionType = s.CumulativeQuerySelection
s.histogramType = s.HistogramMatches
s.combineRule = s.CombineOr
s.variables = ('temperature',)
s.variableMins = (2.9,)
s.variableMaxs = (3.1,)
UpdateNamedSelection('selection1', s)
```

5.4.116 GetSelectionList

Synopsis:

```
GetSelectionList() -> SelectionList object
```

return type [SelectionList object] The GetSelectionList function returns a SelectionList object.

Description:

VisIt maintains a list of named selections, which are sets of cells that are used to restrict the cells processed by other plots. This function returns a list of the selections that VisIt knows about, including their properties.

Example:

```
s = GetSelectionList()
```

5.4.117 GetSelectionSummary

Synopsis:

```
GetSelectionSummary(name) -> SelectionSummary object
```

name [string] The name of the selection whose summary we want to retrieve.

return type [SelectionSummary object] The GetSelectionSummary function returns a SelectionSummary object.

Description:

Named selections have both properties, which describe how the selection is defined, and a summary that desribes the data that was processed while creating the selection. The selection summary object contains some statistics about the selection such as how many cells it contains and histograms of the various variables that were used in creating the selection.

Example:

```
print(GetSelectionSummary('selection1'))
```

5.4.118 GetTimeSliders

Synopsis:

```
GetTimeSliders() -> tuple of strings
```

return type [tuple of strings] GetTimeSliders returns a tuple of strings.

Description:

The GetTimeSliders function returns a tuple of strings containing the names of each of the available time sliders. The list of time sliders contains the names of any open time-varying database, all database correlations, and the keyframing time slider if VisIt is in keyframing mode.

Example:

```
#% visit -cli
path = "/usr/gapps/visit/data/"
dbs = (path + "/dbA00.pdb", path + "dbB00.pdb", path + "dbC00.pdb")
for db in dbs:
    OpenDatabase(db)
    AddPlot("FilledBoundary", "material(mesh)")
    DrawPlots()
CreateDatabaseCorrelation("common", dbs, 1)
print("The list of time sliders is: ", GetTimeSliders())
```

5.4.119 GetUltraScript

Synopsis:

```
GetUltraScript() -> string
```

return type [string] The GetUltraScript function returns a filename.

Description:

Return the name of the file in use by the LoadUltra function. Normal users do not need to use this function.

5.4.120 GetView2D

Synopsis:

```
GetView2D() -> View2DAttributes object
```

return type [View2DAttributes object] Object that represents the 2D view information.

Description:

The GetView functions return ViewAttributes objects which describe the current camera location. The GetView2D function should be called if the active visualization window contains 2D plots.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Change the view interactively using the mouse.
v0 = GetView3D()
# Change the view again using the mouse
v1 = GetView3D()
print(v0)
for i in range(0,20):
    t = float(i) / 19.
    v2 = (1. - t) * v1 + t * v0
    SetView3D(v2) # Animate the view back to the first view.
```

5.4.121 GetView3D

Synopsis:

```
GetView3D() -> View3DAttributes object
```

return type [View3DAttributes object] Object that represents the 3D view information.

Description:

The GetView functions return ViewAttributes objects which describe the current camera location. The GetView3D function should be called to get the view if the active visualization window contains 3D plots.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Change the view interactively using the mouse.
v0 = GetView3D()
# Change the view again using the mouse
v1 = GetView3D()
print(v0)
for i in range(0,20):
    t = float(i) / 19.
    v2 = (1. - t) * v1 + t * v0
    SetView3D(v2) # Animate the view back to the first view.
```

5.4.122 GetViewAxisArray

Synopsis:

```
GetViewAxisArray() -> ViewAxisArrayAttributes object
```

return type [ViewAxisArrayAttributes object] Object that represents the AxisArray view information.

Description:

The GetView functions return ViewAttributes objects which describe the current camera location. The GetViewAxisArray function should be called if the active visualization window contains axis-array plots.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Change the view interactively using the mouse.
v0 = GetView3D()
# Change the view again using the mouse
v1 = GetView3D()
print(v0)
for i in range(0,20):
    t = float(i) / 19.
    v2 = (1. - t) * v1 + t * v0
    SetView3D(v2) # Animate the view back to the first view.
```

5.4.123 GetViewCurve

Synopsis:

```
GetViewCurve() -> ViewCurveAttributes object
```

return type [ViewCurveAttributes object] Object that represents the curve view information.

Description:

The GetView functions return ViewAttributes objects which describe the current camera location. The GetViewCurve function should be called if the active visualization window contains 1D curve plots.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Change the view interactively using the mouse.
v0 = GetView3D()
# Change the view again using the mouse
v1 = GetView3D()
print(v0)
for i in range(0,20):
    t = float(i) / 19.
    v2 = (1. - t) * v1 + t * v0
    SetView3D(v2) # Animate the view back to the first view.
```

5.4.124 GetWindowInformation

Synopsis:

```
GetWindowInformation() -> WindowInformation object
```

return type [WindowInformation object] The GetWindowInformation object returns a WindowInformation object.

Description:

The GetWindowInformation object returns a WindowInformation object that contains information about the active visualization window. The WindowInformation object contains the name of the active source, the active time slider index, the list of available time sliders and their current states, as well as certain window flags that determine whether a window's view is locked, etc. Use the WindowInformation object if you need to query any of these types of information in your script to influence how it behaves.

Example:

```
path = "/usr/gapps/visit/data/"
dbs = (path + "dbA00.pdb", path + "dbB00.pdb", path + "dbC00.pdb")
for db in dbs:
    OpenDatabase(db)
    AddPlot("FilledBoundary", "material(mesh)")
    DrawPlots()
CreateDatabaseCorrelation("common", dbs, 1)
# Get the list of available time sliders.
tsList = GetWindowInformation().timeSliders
# Iterate through "time" on each time slider.
for ts in tsList:
    SetActiveTimeSlider(ts)
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
# Print the window information to examine the other attributes
# that are available.
GetWindowInformation()
```

5.4.125 HideActivePlots

Synopsis:

```
HideActivePlots() -> integer
```

return type [CLI_return_t] The HideActivePlots function returns an integer value of 1 for success and 0 for failure.

Description:

The HideActivePlots function tells the viewer to hide the active plots in the active visualization window.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddPlot("Mesh", "mesh1")
DrawPlots()
SetActivePlots(0)
HideActivePlots()
```

(continues on next page)

(continued from previous page)

```
AddPlot("FilledBoundary", "mat1")
DrawPlots()
```

5.4.126 HideToolbars

Synopsis:

```
HideToolbars() -> integer
HideToolbars(allWindows) -> integer
```

allWindows [integer] An optional integer value that tells VisIt to hide the toolbars for all windows when it is non-zero.

return type [CLI_return_t] The HideToolbars function returns 1 on success and 0 on failure.

Description:

The HideToolbars function tells VisIt to hide the toolbars for the active visualization window or for all visualization windows when the optional allWindows argument is provided and is set to a non-zero value.

Example:

```
#% visit -cli
SetWindowLayout(4)
HideToolbars()
ShowToolbars()
# Hide the toolbars for all windows.
HideToolbars(1)
```

5.4.127 IconifyAllWindows

Synopsis:

```
IconifyAllWindows()
```

Description:

The IconifyAllWindows function minimizes all of the hidden visualization windows to get them out of the way.

Example:

```
#% visit -cli
SetWindowLayout(4) # Have 4 windows
IconifyAllWindows()
DeIconifyAllWindows()
```

5.4.128 InitializeNamedSelectionVariables

Synopsis:

```
InitializeNamedSelectionVariables(name) -> integer
```

name [string] The name of the named selection to initialize.

return type [CLI_return_t] The InitializeNamedSelectionVariables function returns 1 on success and 0 on failure.

Description:

Complex thresholds are often defined using the Parallel Coordinates plot or the Threshold operator. This function can copy variable ranges from compatible plots and operators into the specified named selection's properties. This can be useful when setting up Cumulative Query selections.

Example:

```
InitializeNamedSelectionVariables('selection1')
```

5.4.129 InvertBackgroundColor

Synopsis:

```
InvertBackgroundColor()
```

Description:

The InvertBackgroundColor function swaps the background and foreground colors in the active visualization window. This function is a cheap alternative to setting the foreground and background colors though the AnnotationAttributes in that it is a simple no-argument function call. It is not adequate to set new colors for the background and foreground, but in the event where the two colors can be exchanged favorably, it is a good function to use. An example of when this function is used is after the creation of a Volume plot.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Volume", "u")
DrawPlots()
InvertBackgroundColor()
```

5.4.130 Launch

Synopsis:

```
Launch() -> integer
Launch(program) -> integer
```

program [string] The complete path as a string to the top level 'visit' script.

return type [CLI_return_t] The Launch functions return 1 for success and 0 for failure

Description:

The Launch function is used to launch VisIt's viewer when the VisIt module is imported into a standalone Python interpreter. The Launch function has no effect when a viewer already exists. The difference between Launch and LaunchNowin is that LaunchNowin prevents the viewer from ever creating onscreen visualization windows. The LaunchNowin function is primarily used in Python scripts that want to generate visualizations using VisIt without the use of a display such as when generating movies.

Example:

```
import visit
import visit
visit.AddArgument("-nowin")
visit.Launch()
```

5.4.131 LaunchNowin

Synopsis:

```
LaunchNowin() -> integer
LaunchNowin(program) -> integer
```

program [string] The complete path as a string to the top level 'visit' script.

return type [CLI_return_t] The LaunchNowin functions return 1 for success and 0 for failure

Description:

The Launch function is used to launch VisIt's viewer when the VisIt module is imported into a standalone Python interpreter. The Launch function has no effect when a viewer already exists. The difference between Launch and LaunchNowin is that LaunchNowin prevents the viewer from ever creating onscreen visualization windows. The LaunchNowin function is primarily used in Python scripts that want to generate visualizations using VisIt without the use of a display such as when generating movies.

Example:

```
import visit
visit.AddArgument("-geometry")
visit.AddArgument("1024x1024")
visit.LaunchNowin()
```

5.4.132 Lineout

Synopsis:

```
Lineout(start, end) -> integer
Lineout(start, end, variables) -> integer
Lineout(start, end, samples) -> integer
Lineout(start, end, variables, samples) -> integer
Lineout(keywordarg1=arg1, keywrdarg2=arg2,...,keywordargn=argn ) -> integer
```

start [tuple of doubles] A 2 or 3 item tuple containing the coordinates of the starting point. keyword arg - start_point

end [tuple of doubles] A 2 or 3 item tuple containing the coordinates of the end point. keyword arg - end_point

variables [tuple of strings] A tuple of strings containing the names of the variables for which lineouts should be created. keyword arg - vars

samples [integer] An integer value containing the number of sample points along the lineout. keyword arg - num_samples keyword arg - use_sampling

return type [CLI_return_t] The Lineout function returns 1 on success and 0 on failure.

Description:

The Lineout function extracts data along a given line segment and creates curves from it in a new visualization window. The start argument is a tuple of numbers that make up the coordinate of the lineout's

starting location. The end argument is a tuple of numbers that make up the coordinate of the lineout's ending location. The optional variables argument is a tuple of strings that contain the variables that should be sampled to create lineouts. The optional samples argument is used to determine the number of sample points that should be taken along the specified line. If the samples argument is not provided then VisIt will sample the mesh where it intersects the specified line instead of using the number of samples to compute a list of points to sample.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "ascii")
DrawPlots()
Lineout((0.2,0.2), (0.8,1.2))
Lineout((0.2,1.2), (0.8,0.2), ("default", "d", "u"))
Lineout((0.6, 0.1), (0.6, 1.2), 100)
Lineout(start_point=(0.6, 0.1), end_point=(0.6, 1.2), use_sampling=1, num_samples=100)
```

5.4.133 ListDomains

Synopsis:

```
ListDomains()
```

Description:

ListDomains prints a list of the domains for the active plots, which indicates which domains are on and off. The list functions are used mostly to print the results of restricting the SIL.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
TurnMaterialsOff("4") # Turn off material 4
ListMaterials() # List the materials in the SIL restriction
```

5.4.134 ListMaterials

Synopsis:

```
ListMaterials()
```

Description:

ListMaterials prints a list of the materials for the active plots, which indicates which materials are on and off. The list functions are used mostly to print the results of restricting the SIL.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
```

(continues on next page)

(continued from previous page)

```
TurnMaterialsOff("4") # Turn off material 4
ListMaterials() # List the materials in the SIL restriction
```

5.4.135 ListPlots

Synopsis:

```
ListPlots() -> string
ListPlots(stringOnly) -> string
```

return type [string] The ListPlots function returns a string containing a representation of the. plot list.

Description:

Sometimes it is difficult to remember the order of the plots in the active visualization window's plot list. The ListPlots function prints the contents of the plot list to the output console and returns that string as well.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "u")
AddPlot("Contour", "d")
DrawPlots()
ListPlots()
```

5.4.136 LoadAttribute

Synopsis:

```
LoadAttribute(filename, object)
```

filename [string] The name of the XML file to load the attribute from or save the attribute to.

object The object to load or save.

Description:

The LoadAttribute and SaveAttribute methods save a single attribute, such as a current plot or operator python object, to a standalone XML file. Note that LoadAttribute requires that the target attribute already be created by other means; it fills, but does not create, the attribute.

Example:

```
#% visit -cli
a = MeshPlotAttributes()
SaveAttribute('mesh.xml', a)
b = MeshPlotAttributes()
LoadAttribute('mesh.xml', b)
```

5.4.137 LoadNamedSelection

Synopsis:

```
LoadNamedSelection(name) -> integer
LoadNamedSelection(name, engineName) -> integer
LoadNamedSelection(name, engineName, simName) -> integer
```

name [string] The name of a named selection.

engineName [string] (optional) The name of the engine where the selection was saved.

simName [string] (optional) The name of the simulation that saved the selection.

return type [CLI_return_t] The LoadNamedSelection function returns 1 for success and 0 for failure.

Description:

Named Selections allow you to select a group of elements (or particles). One typically creates a named selection from a group of elements and then later applies the named selection to another plot (thus reducing the set of elements displayed to the ones from when the named selection was created). Named selections only last for the current session. However, if you find a named selection that is particularly interesting, you can save it to a file for use in later sessions. You would use LoadNamedSelection to do the loading.

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/wave*.silo database"
OpenDatabase(db)
AddPlot("Pseudocolor", "pressure")
LoadNamedSelection("selection_from_previous_session")
ApplyNamedSelection("selection_from_previous_session")
```

5.4.138 LoadUltra

Synopsis:

```
LoadUltra()
```

Description:

LoadUltra launches the Ultra command parser, allowing you to enter Ultra commands and have VisIt process them. A new command prompt is presented, and only Ultra commands will be allowed until 'end' or 'quit' is entered, at which time, you will be returned to VisIt's cli prompt. For information on currently supported commands, type 'help' at the Ultra prompt Please note that filenames/paths must be surrounded by quotes, unlike with Ultra.

Example:

```
#% visit -cli
#>>> LoadUltra()
#U-> rd "../../data/distribution.ultra"
#U-> select 1
#U-> end
#>>>
```

5.4.139 LocalNameSpace

Synopsis:

```
LocalNameSpace()
```

Description:

The LocalNameSpace function tells the VisIt module to add plugin functions to the global namespace when the VisIt module is imported into a stand-alone Python interpreter. This is the default behavior when using VisIt's cli program.

Example:

```
import visit
visit.LocalNameSpace()
visit.Launch()
```

5.4.140 LongFileName

Synopsis:

```
LongFileName(filename) -> string
```

filename [string] A string object containing the short filename to expand.

return type [string] The LongFileName function returns a string. This function returns the input argument unless you are on the Windows platform.

Description:

On Windows, filenames can have two different sizes: traditional 8.3 format, and long format. The long format, which lets you name files whatever you want, is implemented using the traditional 8.3 format under the covers. Sometimes filenames are given to VisIt in the traditional 8.3 format and must be expanded to long format before it is possible to open them. If you ever find that you need to do this conversion, such as when you process command line arguments, then you can use the LongFileName function to return the longer filename.

5.4.141 MoveAndResizeWindow

Synopsis:

```
MoveAndResizeWindow(win, x, y, w, h) -> integer
```

win [integer] The integer id of the window to be moved [1..16].

x [integer] The new integer x location for the window being moved.

y [integer] The new integer y location for the window being moved.

w [integer] The new integer width for the window being moved.

h [integer] The new integer height for the window being moved.

return type [CLI_return_t] MoveAndResizeWindow returns 1 on success and 0 on failure.

Description:

MoveAndResizeWindow moves and resizes a visualization window.

Example:

```
#% visit -cli
MoveAndResizeWindow(1, 100, 100, 300, 600)
```

5.4.142 MoveOperatorKeyframe

Synopsis:

```
MoveOperatorKeyframe(plotIndex, operatorIndex, oldFrame, newFrame)
```

plotIndex [integer] An integer representing the index of the plot in the plot list.

operatorIndex [integer] An integer representing the index of the operator in the plot.

oldFrame [integer] An integer that is the old animation frame where the keyframe is located.

newFrame [integer] An integer that is the new animation frame where the keyframe will be moved.

Description:

MoveOperatorKeyframe moves a keyframe for an operator to a new animation frame, which changes the operator attributes that are used for each animation frame when VisIt is in keyframing mode.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
k = GetKeyframeAttributes()
k.enabled,k.nFrames,k.nFramesWasUserSet = 1,11,1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "hardyglobal")
AddOperator("Slice")
# Set up operator keyframes so the Slice operator's percent will change
# over time.
s0 = SliceAttributes()
s0.originType = s0.Percent
s0.originPercent = 0
s1 = SliceAttributes()
s0.originType = s1.Percent
s1.originPercent = 100
SetOperatorOptions(s0)
SetTimeSliderState(10)
SetOperatorOptions(s1)
SetTimeSliderState(0)
DrawPlots()
ListPlots()
# Iterate over all animation frames and wrap around to the first one.
for i in list(range(TimeSliderGetNStates())) + [0]:
    SetTimeSliderState(i)
# Move the operator keyframe at frame 10 to frame 5
MoveOperatorKeyframe(0, 0, 10, 5)
ListPlots()
SetTimeSliderState(5)
```

5.4.143 MovePlotDatabaseKeyframe

Synopsis:

```
MovePlotDatabaseKeyframe(index, oldFrame, newFrame)
```

index [integer] An integer representing the index of the plot in the plot list.

oldFrame [integer] An integer that is thhe old animation frame where the keyframe is located.

newFrame [integer] An integer that is the new animation frame where the keyframe will be moved.

Description:

MovePlotDatabaseKeyframe moves a database keyframe for a specified plot to a new animation frame, which changes the list of database time states that are used for each animation frame when VisIt is in keyframing mode.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
AddPlot("Pseudocolor", "pressure")
SetPlotFrameRange(0, 0, nFrames-1)
SetPlotDatabaseKeyframe(0, 0, 70)
SetPlotDatabaseKeyframe(0, nFrames/2, 35)
SetPlotDatabaseKeyframe(0, nFrames-1, 0)
DrawPlots()
for state in list(range(TimeSliderGetNStates())) + [0]:
    SetTimeSliderState(state)
MovePlotDatabaseKeyframe(0, nFrames/2, nFrames/4)
for state in list(range(TimeSliderGetNStates())) + [0]:
    SetTimeSliderState(state)
```

5.4.144 MovePlotKeyframe

Synopsis:

```
MovePlotKeyframe(plotIndex, oldFrame, newFrame)
```

plotIndex [integer] An integer representing the index of the plot in the plot list.

oldFrame [integer] An integer that is the old animation frame where the keyframe is located.

newFrame [integer] An integer that is the new animation frame where the keyframe will be moved.

Description:

MovePlotKeyframe moves a keyframe for a specified plot to a new animation frame, which changes the plot attributes that are used for each animation frame when VisIt is in keyframing mode.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Contour", "hgslice")
DrawPlots()
k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
```

(continues on next page)

(continued from previous page)

```
SetKeyframeAttributes(k)
SetPlotFrameRange(0, 0, nFrames-1)
c = ContourAttributes()
c.contourNLevels = 5
SetPlotOptions(c)
SetTimeSliderState(nFrames/2)
c.contourNLevels = 10
SetPlotOptions(c)
c.contourLevels = 25
SetTimeSliderState(nFrames-1)
SetPlotOptions(c)
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
    SaveWindow()
temp = nFrames-2
MovePlotKeyframe(0, nFrames/2, temp)
MovePlotKeyframe(0, nFrames-1, nFrames/2)
MovePlotKeyframe(0, temp, nFrames-1)
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
    SaveWindow()
```

5.4.145 MovePlotOrderTowardFirst

Synopsis:

```
MovePlotOrderTowardFirst(index) -> integer
```

index [integer] The integer index of the plot that will be moved within the plot list.

return type [CLI_return_t] The MovePlotOrderTowardFirst function returns 1 on success and 0 on failure.

Description:

This function shifts the specified plot one slot towards the start of the plot list.

Example:

```
MovePlotOrderTowardFirst(2)
```

5.4.146 MovePlotOrderTowardLast

Synopsis:

```
MovePlotOrderTowardLast(index) -> integer
```

index [integer] The integer index of the plot that will be moved within the plot list.

return type [CLI_return_t] The MovePlotOrderTowardLast function returns 1 on success and 0 on failure.

Description:

This function shifts the specified plot one slot towards the end of the plot list.

Example:

```
MovePlotOrderTowardLast(0)
```

5.4.147 MoveViewKeyframe

Synopsis:

```
MoveViewKeyframe(oldFrame, newFrame) -> integer
```

oldFrame [integer] An integer that is the old animation frame where the keyframe is located.

newFrame [integer] An integer that is the new animation frame where the keyframe will be moved.

return type [CLI_return_t] MoveViewKeyframe returns 1 on success and 0 on failure.

Description:

MoveViewKeyframe moves a view keyframe to a new animation frame, which changes the view that is used for each animation frame when VisIt is in keyframing mode.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Contour", "hardyglobal")
DrawPlots()
k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
SetViewKeyframe()
SetTimeSliderState(nFrames/2)
v = GetView3d()
v.viewNormal = (-0.616518, 0.676972, 0.402014)
v.viewUp = (0.49808, 0.730785, -0.466764)
SetViewKeyframe()
SetTimeSliderState(0)
# Move the view keyframe to the last animation frame.
MoveViewKeyframe(nFrames/2, nFrames-1)
```

5.4.148 MoveWindow

Synopsis:

```
MoveWindow(win, x, y) -> integer
```

win [integer] The integer id of the window to be moved [1..16].

x [integer] The new integer x location for the window being moved.

y [integer] The new integer y location for the window being moved.

return type [CLI_return_t] MoveWindow returns 1 on success and 0 on failure.

Description:

MoveWindow moves a visualization window.

Example:

```
#% visit -cli
MoveWindow(1, 100, 100)
```

5.4.149 NodePick

Synopsis:

```
NodePick(namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

coord [tuple] A tuple of doubles containing the spatial coordinate (x, y, z).

x [integer] An integer containing the screen X location (in pixels) offset from the left side of the visualization window.

y [integer] An integer containing the screen Y location (in pixels) offset from the bottom of the visualization window.

vars (optional) [tuple] A tuple of strings with the variable names for which to return results. Default is the currently plotted variable.

do_time (optional) [integer] An integer indicating whether to do a time pick. 1 -> do a time pick, 0 (default) -> do not do a time pick.

start_time (optional) [integer] An integer with the starting frame index. Default is 0.

end_time (optional) [integer] An integer with the ending frame index. Default is num_timestates-1.

stride (optional) [integer] An integer with the stride for advancing in time. Default is 1.

preserve_coord (optional) [integer] An integer indicating whether to pick an element or a coordinate. 0 -> used picked element (default), 1-> used picked coordinate. Note: enabling this option may substantially slow down the speed with which the query can be performed.

curve_plot_type (optional) [integer] An integer indicating whether the output should be on a single axis or with multiple axes. 0 -> single Y axis (default), 1 -> multiple Y Axes.

return type [dictionary] NodePick returns a python dictionary of the pick results, unless do_time is specified, then a time curve is created in a new window.

Description:

The NodePick function prints pick information for the node closest to the specified point. The point can be specified as a 2D or 3D point in world space or it can be specified as a pixel location in screen space. If the point is specified as a pixel location then VisIt finds the node closest to a ray that is projected into the mesh. Once the nodal pick has been calculated, you can use the GetPickOutput function to retrieve the printed pick output as a string which can be used for other purposes.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()
# Perform node pick in screen space
pick_out = NodePick(x=200,y=200)
# Perform node pick in world space.
pick_out = NodePick(coord=(-5.0, 5.0, 0))
```

5.4.150 NumColorTableNames

Synopsis:

```
NumColorTableNames() -> integer
```

return type [CLI_return_t] The NumColorTableNames function return an integer.

Description:

The NumColorTableNames function returns the number of color tables that have been defined.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
p = PseudocolorAttributes()
p.colorTableName = "default"
SetPlotOptions(p)
DrawPlots()
print("There are %d color tables." % NumColorTableNames())
for ct in ColorTableNames():
    SetDefaultContinuousColorTable(ct)
    SaveWindow()
```

5.4.151 NumOperatorPlugins

Synopsis:

```
NumOperatorPlugins() -> integer
```

return type [CLI_return_t] The NumOperatorPlugins function returns an integer.

Description:

The NumOperatorPlugins function returns the number of available operator plugins.

Example:

```
#% visit -cli
print("The number of operator plugins is: ", NumOperatorPlugins())
print("The names of the plugins are: ", OperatorPlugins())
```

5.4.152 NumPlotPlugins

Synopsis:

```
NumPlotPlugins() -> integer
```

return type [CLI_return_t] The NumPlotPlugins function returns an integer.

Description:

The NumPlotPlugins function returns the number of available plot plugins.

Example:

```
#% visit -cli
print("The number of plot plugins is: ", NumPlotPlugins())
print("The names of the plugins are: ", PlotPlugins())
```

5.4.153 OpenCLI

Synopsis:

```
OpenCLI() -> integer
OpenCLI(args) -> integer
```

args: list of strings The list of arguments to pass to the CLI.

Description:

The OpenCLI function is used to launch the CLI in an Xterm window with the specified arguments.

Example:

```
#% visit -cli -nowin
OpenCLI("-debug", "5")
```

5.4.154 OpenComputeEngine

Synopsis:

```
OpenComputeEngine() -> integer
OpenComputeEngine(hostName) -> integer
OpenComputeEngine(hostName, simulation) -> integer
OpenComputeEngine(hostName, args) -> integer
OpenComputeEngine(MachineProfile) -> integer
```

hostName [string] The name of the computer on which to start the engine.

args [tuple] Optional tuple of command line arguments for the engine. Alternative arguments - MachineProfile object to load with OpenComputeEngine call

MachineProfile [MachineProfile object] The Machine Profile of the computer on which to start the engine.

return type [CLI_return_t] The OpenComputeEngine function returns an integer value of 1 for success and 0 for failure.

Description:

The OpenComputeEngine function is used to explicitly open a compute engine with certain properties. When a compute engine is opened implicitly, the viewer relies on sets of attributes called host profiles. Host profiles determine how compute engines are launched. This allows compute engines to be easily launched in parallel. Since the VisIt Python Interface does not expose VisIt's host profiles, it provides the OpenComputeEngine function to allow users to launch compute engines. The OpenComputeEngine function must be called before opening a database in order to prevent any latent host profiles from taking precedence.

Example:

```
#% visit -cli
# Launch parallel compute engine remotely.
args = ("-np", "16", "-nn", "4")
OpenComputeEngine("thunder", args)
OpenDatabase("thunder:/usr/gapps/visit/data/multi_ucd3d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
```

5.4.155 OpenDatabase

Synopsis:

```
OpenDatabase(databaseName) -> integer
OpenDatabase(databaseName, timeIndex) -> integer
OpenDatabase(databaseName, timeIndex, dbPluginName) -> integer
```

databaseName [string] The name of the database to open.

timeIndex [integer] This is an optional integer argument indicating the time index at which to open the database. If it is not specified, a time index of zero is assumed.

dbPluginIndex [string] An optional string containing the name of the plugin to use. Note that this string must also include the plugin's version number (with few exceptions, almost all plugins' version numbers are 1.0). Note also that you must capitalize the spelling identically to what the plugin's GetName() method returns. For example, "XYZ_1.0" is the string you would use for the XYZ plugin.

return type [CLI_return_t] The OpenDatabase function returns an integer value of 1 for success and 0 for failure.

Description:

The OpenDatabase function is one of the most important functions in the VisIt Python Interface because it opens a database so it can be plotted. The databaseName argument is a string containing the full name of the database to be opened. The database name is of the form: computer:/path/filename. The computer part of the filename can be omitted if the database to be opened resides on the local computer.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
OpenDatabase("mcr:/usr/gapps/visit/data/multi_ucd3d.silo")
OpenDatabase("file.visit")
OpenDatabase("file.visit", 4)
OpenDatabase("mcr:/usr/gapps/visit/data/multi_ucd3d.silo",0,"Silo_1.0")
```

5.4.156 OpenGUI

Synopsis:

```
OpenGUI() -> integer
OpenGUI(args) -> integer
```

args: list of strings The list of arguments to pass to the GUI.

Description:

The OpenGUI function is used to launch the GUI with the specified arguments.

Example:

```
#% visit -cli -nowin
OpenGUI("-debug", "5")
```

5.4.157 OpenMDServer

Synopsis:

```
OpenMDServer() -> integer
OpenMDServer(host) -> integer
OpenMDServer(host, args) -> integer
OpenMDServer(MachineProfile) -> integer
```

host [string] The optional host argument determines the host on which the metadata server is to be launched. If this argument is not provided, "localhost" is assumed.

args [tuple] A tuple of strings containing command line flags for the metadata server.

Argument	Description
-debug #	The -debug argument allows you to specify a debug level.
-dir visitdir	The -dir argument allows you to specify where VisIt is.

MachineProfile [MachineProfile object] MachineProfile object to load with OpenMDServer call

return type [CLI_return_t] The OpenMDServer function returns 1 on success and 0 on failure.

Description:

The OpenMDServer explicitly launches a metadata server on a specified host. This allows you to provide command line options that influence how the metadata server will run. range [1,5] that VisIt uses to write debug logs to disk. located on a remote computer. This allows you to successfully connect to a remote computer in the absence of host profiles. It also allows you to debug VisIt in distributed mode. -fallback_format <format> The -fallback_format argument allows you to specify the database plugin that will be used to open files if all other guessing failed. This is useful when the files that you want to open do not have file extensions. -assume_format <format> The -assume_format argument allows you to specify the database plugin that will be used FIRST when attempting to open files. This is useful when the files that you want to open have a file extension which may match multiple file format readers.

Example:

```
#% visit -cli -assume_format PDB
#args = ("-dir", "/my/private/visit/version/", "-assume_format", "PDB", "-debug", "4")
# Open a metadata server before the call to OpenDatabase so we
# can launch it how we want.
OpenMDServer("thunder", args)
OpenDatabase("thunder:/usr/gapps/visit/data/allinone00.pdb")
# Open a metadata server on localhost too.
OpenMDServer()
```

5.4.158 OperatorPlugins

Synopsis:

```
OperatorPlugins() -> tuple of strings
```

return type [tuple of strings] The OperatorPlugins function returns a tuple of strings.

Description:

The OperatorPlugins function returns a tuple of strings that contain the names of the loaded operator plugins. This can be useful for the creation of scripts that alter their behavior based on the available operator plugins.

Example:

```
#% visit -cli
for plugin in OperatorPlugins():
    print("The %s operator plugin is loaded." % plugin)
```

5.4.159 OverlayDatabase

Synopsis:

```
OverlayDatabase(databaseName) -> integer
OverlayDatabase(databaseName, state) -> integer
```

databaseName [string] The name of the new plot database.

state [integer] The time state at which to open the database.

return type [CLI_return_t] The OverlayDatabase function returns an integer value of 1 for success and 0 for failure.

Description:

VisIt has the concept of overlaying plots which, in the nutshell, means that the entire plot list is copied and a new set of plots with exactly the same attributes but a different database is appended to the plot list of the active window. The OverlayDatabase function allows the VisIt Python Interface to overlay plots. OverlayDatabase takes a single string argument which contains the name of the database. After calling the OverlayDatabase function, the plot list is larger and contains plots of the specified overlay database.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
OverlayDatabase("riptide:/usr/gapps/visit/data/curv3d.silo")
```

5.4.160 PickByGlobalNode

Synopsis:

```
PickByGlobalNode(namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

element [integer] An integer with the global node id.

vars (optional) [tuple] A tuple of strings with the variable names for which to return results. Default is the currently plotted variable.

do_time (optional) [integer] An integer indicating whether to do a time pick. 1 -> do a time pick, 0 (default) -> do not do a time pick.

start_time (optional) [integer] An integer with the starting frame index. Default is 0.

end_time (optional) [integer] An integer with the ending frame index. Default is num_timestates-1.

stride (optional) [integer] An integer with the stride for advancing in time. Default is 1.

preserve_coord (optional) [integer] An integer indicating whether to pick an element or a coordinate. 0 -> used picked element (default), 1-> used picked coordinate. Note: enabling this option may substantially slow down the speed with which the query can be performed.

curve_plot_type (optional) [integer] An integer indicating whether the output should be on a single axis or with multiple axes. 0 -> single Y axis (default), 1 -> multiple Y Axes.

return type [dictionary] PickByGlobalNode returns a python dictionary of pick results.

Description:

The PickByGlobalNode function tells VisIt to perform pick using a specific global node index for the entire problem. Some meshes are broken up into smaller "domains" and then these smaller domains can employ a global indexing scheme to make it appear as though the mesh was still one large mesh. Not all meshes that have been decomposed into domains provide sufficient information to allow global node indexing. You can use the GetPickOutput function to retrieve a string containing the pick information once you've called PickByGlobalNode.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/global_node.silo")
AddPlot("Pseudocolor", "dist")
DrawPlots()
# Pick on global node 236827
pick_out = PickByGlobalNode(element=246827)
# examine output
print('value of dist at global node 246827: %g' % pick_out['dist'])
print('local domain/node: %d/%d' % (pick_out['domain_id'], pick_out['node_id']))
# get last pick output as string
print('Last pick = ', GetPickOutput())
```

5.4.161 PickByGlobalZone

Synopsis:

```
PickByGlobalZone(namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

element [integer] An integer with the global zone id.

vars (optional) [tuple] A tuple of strings with the variable names for which to return results. Default is the currently plotted variable.

do_time (optional) [integer] An integer indicating whether to do a time pick. 1 -> do a time pick, 0 (default) -> do not do a time pick.

start_time (optional) [integer] An integer with the starting frame index. Default is 0.

end_time (optional) [integer] An integer with the ending frame index. Default is num_timestates-1.

stride (optional) [integer] An integer with the stride for advancing in time. Default is 1.

preserve_coord (optional) [integer] An integer indicating whether to pick an element or a coordinate. 0 -> used picked element (default), 1-> used picked coordinate. Note: enabling this option may substantially slow down the speed with which the query can be performed.

curve_plot_type (optional) [integer] An integer indicating whether the output should be on a single axis or with multiple axes. 0 -> single Y axis (default), 1 -> multiple Y Axes.

return type [dictionary] PickByGlobalZone returns a python dictionary of pick results.

Description:

The PickByGlobalZone function tells VisIt to perform pick using a specific global cell index for the entire problem. Some meshes are broken up into smaller "domains" and then these smaller domains can employ

a global indexing scheme to make it appear as though the mesh was still one large mesh. Not all meshes that have been decomposed into domains provide sufficient information to allow global cell indexing. You can use the GetPickOutput function to retrieve a string containing the pick information once you've called PickByGlobalZone.

Example:

```
OpenDatabase("/usr/gapps/visit/data/global_node.silo")
AddPlot("Pseudocolor", "p")
DrawPlots()
# Pick on global zone 237394
pick_out = PickByGlobalZone(element=237394)
# examine output
print('value of p at global zone 237394: %g' % pick_out['p'])
print('local domain/zone: %d/%d' % (pick_out['domain_id'], pick_out['zone_id']))
# get last pick output as string
print('Last pick = ', GetPickOutput())
```

5.4.162 PickByNode

Synopsis:

```
PickByNode(namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

domain [integer] An integer with the domain id.

element [integer] An integer with the node id.

vars (optional) [tuple] A tuple of strings with the variable names for which to return results. Default is the currently plotted variable.

do_time (optional) [integer] An integer indicating whether to do a time pick. 1 -> do a time pick, 0 (default) -> do not do a time pick.

start_time (optional) [integer] An integer with the starting frame index. Default is 0.

end_time (optional) [integer] An integer with the ending frame index. Default is num_timestates-1.

stride (optional) [integer] An integer with the stride for advancing in time. Default is 1.

preserve_coord (optional) [integer] An integer indicating whether to pick an element or a coordinate. 0 -> used picked element (default), 1-> used picked coordinate. Note: enabling this option may substantially slow down the speed with which the query can be performed.

curve_plot_type (optional) [integer] An integer indicating whether the output should be on a single axis or with multiple axes. 0 -> single Y axis (default), 1 -> multiple Y Axes. Currently, this is only available when performing a pick range.

return type [dictionary] PickByNode returns a python dictionary of the pick results, unless do_time is specified, then a time curve is created in a new window. If the picked variable is zone centered, the variable values are grouped according to incident zone ids.

Description:

The PickByNode function tells VisIt to perform pick using a specific node index in a given domain. Other pick by node variants first determine the node that is closest to some user-specified 3D point but the PickByNode functions cuts out this step and allows you to directly pick on the node of your choice. You can use the GetPickOutput function to retrieve a string containing the pick information once you've called PickByNode.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/multi_curv2d.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Pick on node 200 in the first domain.
pick_out = PickByNode(element=200, domain=1)
# examine output
print('value of u at node 200: %g' % pick_out['u'])
# Pick on node 100 in domain 5 and return information for two additional
# variables.
pick_out = PickByNode(domain=5, element=100, vars=("u", "v", "d"))
# examine output
print('incident zones for node 100: ', pick_out['incident_zones'])
print('value of d at incident zone %d: %g' % (pick_out['incident_zones'][0], pick_out[
↪'d'][str(pick_out['incident_zones'][0])]))
# print results formatted as string
print("Last pick = ", GetPickOutput())
```

5.4.163 PickByNodeLabel

Synopsis:

```
PickByNodeLabel(namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

element_label [string] An string with the label of the node to pick.

vars (optional) [tuple] A tuple of strings with the variable names for which to return results. Default is the currently plotted variable.

do_time (optional) [integer] An integer indicating whether to do a time pick. 1 -> do a time pick, 0 (default) -> do not do a time pick.

start_time (optional) [integer] An integer with the starting frame index. Default is 0.

end_time (optional) [integer] An integer with the ending frame index. Default is num_timestates-1.

stride (optional) [integer] An integer with the stride for advancing in time. Default is 1.

preserve_coord (optional) [integer] An integer indicating whether to pick an element or a coordinate. 0 -> used picked element (default), 1-> used picked coordinate. Note: enabling this option may substantially slow down the speed with which the query can be performed.

curve_plot_type (optional) [integer] An integer indicating whether the output should be on a single axis or with multiple axes. 0 -> single Y axis (default), 1 -> multiple Y Axes.

return type [dictionary] PickByNodeLabel returns a python dictionary of the pick results, unless do_time is specified, then a time curve is created in a new window. If the picked variable is node centered, the variable values are grouped according to incident node ids.

Description:

The PickByNodeLabel function tells VisIt to perform pick using a specific cell label. Other pick by zone variants first determine the cell that contains some user-specified 3D point but the PickByZone functions cuts out this step and allows you to directly pick on the cell of your choice. You can use the GetPickOutput function to retrieve a string containing the pick information once you've called PickByZone.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/multi_curv2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Pick on node labeled "node 4".
pick_out = PickByNodeLabel(element_label="node 4")
# Pick on cell labeled "node 4" using a python dictionary.
opts = {}
opts["element_label"] ="node 4"
pick_out = PickByNodeLabel(opts)
# examine output
print('value of d at "node 4": %g' % pick_out['d'])
# Pick on node labeled "node 12" return information for two additional
# variables.
pick_out = PickByNodeLabel(element_label="node 12", vars=("d", "u", "v"))
# examine output
print('incident nodes for "node 12": ', pick_out['incident_nodes'])
print('values of u at incident node %d: %g' % (pick_out['incident_nodes'][0], pick_
↪out['u'][str(pick_out['incident_zones'][0])]))
# print results formatted as string
print("Last pick = ", GetPickOutput())
```

5.4.164 PickByZone

Synopsis:

```
PickByZone(namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

domain [integer] An integer with the domain id.

element [integer] An integer with the zone id.

vars (optional) [tuple] A tuple of strings with the variable names for which to return results. Default is the currently plotted variable.

do_time (optional) [integer] An integer indicating whether to do a time pick. 1 -> do a time pick, 0 (default) -> do not do a time pick.

start_time (optional) [integer] An integer with the starting frame index. Default is 0.

end_time (optional) [integer] An integer with the ending frame index. Default is num_timestates-1.

stride (optional) [integer] An integer with the stride for advancing in time. Default is 1.

preserve_coord (optional) [integer] An integer indicating whether to pick an element or a coordinate. 0 -> used picked element (default), 1-> used picked coordinate. Note: enabling this option may substantially slow down the speed with which the query can be performed.

curve_plot_type (optional) [integer] An integer indicating whether the output should be on a single axis or with multiple axes. 0 -> single Y axis (default), 1 -> multiple Y Axes. Currently, this is only available when performing a pick range.

return type [dictionary] PickByZone returns a python dictionary of the pick results, unless do_time is specified, then a time curve is created in a new window. If the picked variable is node centered, the variable values are grouped according to incident node ids.

Description:

The PickByZone function tells VisIt to perform pick using a specific cell index in a given domain. Other pick by zone variants first determine the cell that contains some user-specified 3D point but the PickByZone functions cuts out this step and allows you to directly pick on the cell of your choice. You can use the GetPickOutput function to retrieve a string containing the pick information once you've called PickByZone.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/multi_curv2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Pick on cell 200 in the second domain.
pick_out = PickByZone(element=200, domain=2)
# examine output
print('value of d at zone 200: %g' % pick_out['d'])
# Pick on cell 100 in domain 5 and return information for two additional
# variables.
pick_out = PickByZone(element=100, domain=5, vars=("d", "u", "v"))
# examine output
print('incident nodes for zone 100: ', pick_out['incident_nodes'])
print('values of u at incident zone %d: %g' % (pick_out['incident_nodes'][0], pick_
↪out['u'][str(pick_out['incident_zones'][0])]))
# print results formatted as string
print("Last pick = ", GetPickOutput())
```

5.4.165 PickByZoneLabel

Synopsis:

```
PickByZoneLabel(namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

element_label [string] An string with the label of the zone to pick.

vars (optional) [tuple] A tuple of strings with the variable names for which to return results. Default is the currently plotted variable.

do_time (optional) [integer] An integer indicating whether to do a time pick. 1 -> do a time pick, 0 (default) -> do not do a time pick.

start_time (optional) [integer] An integer with the starting frame index. Default is 0.

end_time (optional) [integer] An integer with the ending frame index. Default is num_timestates-1.

stride (optional) [integer] An integer with the stride for advancing in time. Default is 1.

preserve_coord (optional) [integer] An integer indicating whether to pick an element or a coordinate. 0 -> used picked element (default), 1-> used picked coordinate. Note: enabling this option may substantially slow down the speed with which the query can be performed.

curve_plot_type (optional) [integer] An integer indicating whether the output should be on a single axis or with multiple axes. 0 -> single Y axis (default), 1 -> multiple Y Axes.

return type [dictionary] PickByZoneLabel returns a python dictionary of the pick results, unless do_time is specified, then a time curve is created in a new window. If the picked variable is node centered, the variable values are grouped according to incident node ids.

Description:

The PickByZoneLabel function tells VisIt to perform pick using a specific cell label. Other pick by zone variants first determine the cell that contains some user-specified 3D point but the PickByZone functions cuts out this step and allows you to directly pick on the cell of your choice. You can use the GetPickOutput function to retrieve a string containing the pick information once you've called PickByZone.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/multi_curv2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Pick on cell labeled "brick 4".
pick_out = PickByZoneLabel(element_label="brick 4")
# Pick on cell labeled "brick 4" using a python dictionary.
opts = {}
opts["element_label"] ="brick 4"
pick_out = PickByZoneLabel(opts)
# examine output
print('value of d at "brick 4": %g' % pick_out['d'])
# Pick on cell labeled "shell 12" return information for two additional
# variables.
pick_out = PickByZoneLabel(element_label="shell 12", vars=("d", "u", "v"))
# examine output
print('incident nodes for "shell 12": ', pick_out['incident_nodes'])
print('values of u at incident zone %d: %g' % (pick_out['incident_nodes'][0], pick_
↪out['u'][str(pick_out['incident_zones'][0])]))
# print results formatted as string
print("Last pick = ", GetPickOutput())
```

5.4.166 PlotPlugins

Synopsis:

```
PlotPlugins() -> tuple of strings
```

return type [tuple of strings] The PlotPlugins function returns a tuple of strings.

Description:

The PlotPlugins function returns a tuple of strings that contain the names of the loaded plot plugins. This can be useful for the creation of scripts that alter their behavior based on the available plot plugins.

Example:

```
#% visit -cli
for plugin in PluginPlugins():
    print("The %s plot plugin is loaded." % plugin)
```

5.4.167 PointPick

Synopsis:

```
PointPick(namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

coord [tuple] A tuple of doubles containing the spatial coordinate (x, y, z).

x [integer] An integer containing the screen X location (in pixels) offset from the left side of the visualization window.

y [integer] An integer containing the screen Y location (in pixels) offset from the bottom of the visualization window.

vars (optional) [tuple] A tuple of strings with the variable names for which to return results. Default is the currently plotted variable.

do_time (optional) [integer] An integer indicating whether to do a time pick. 1 -> do a time pick, 0 (default) -> do not do a time pick.

start_time (optional) [integer] An integer with the starting frame index. Default is 0.

end_time (optional) [integer] An integer with the ending frame index. Default is num_timestates-1.

stride (optional) [integer] An integer with the stride for advancing in time. Default is 1.

preserve_coord (optional) [integer] An integer indicating whether to pick an element or a coordinate. 0 -> used picked element (default), 1-> used picked coordinate. Note: enabling this option may substantially slow down the speed with which the query can be performed.

curve_plot_type (optional) [integer] An integer indicating whether the output should be on a single axis or with multiple axes. 0 -> single Y axis (default), 1 -> multiple Y Axes.

return type [dictionary] PointPick returns a python dictionary of the pick results, unless do_time is specified, then a time curve is created in a new window.

Description:

The PointPick function prints pick information for the node closest to the specified point. The point can be specified as a 2D or 3D point in world space or it can be specified as a pixel location in screen space. If the point is specified as a pixel location then VisIt finds the node closest to a ray that is projected into the mesh. Once the nodal pick has been calculated, you can use the GetPickOutput function to retrieve the printed pick output as a string which can be used for other purposes.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()
# Perform node pick in screen space
pick_out = PointPick(x=200,y=200)
# Perform node pick in world space.
pick_out = PointPick(coord=(-5.0, 5.0, 0))
```

5.4.168 PrintWindow

Synopsis:

```
PrintWindow() -> integer
```

return type [CLI_return_t] The PrintWindow function returns an integer value of 1 for success and 0 for failure.

Description:

The PrintWindow function tells the viewer to print the image in the active visualization window using the current printer settings.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "d")
AddPlot("Contour", "u")
DrawPlots()
PrintWindow()
```

5.4.169 PromoteOperator

Synopsis:

```
PromoteOperator(opIndex) -> integer
PromoteOperator(opIndex, applyToAllPlots) -> integer
```

opIndex [integer] A zero-based integer corresponding to the operator that should be promoted.

applyToAllPlots [integer] An integer flag that causes all plots in the plot list to be affected when it is non-zero.

return type [CLI_return_t] PromoteOperator returns 1 on success and 0 on failure.

Description:

The PromoteOperator function moves an operator closer to the end of the visualization pipeline. This allows you to change the order of operators that have been applied to a plot without having to remove them from the plot. For example, consider moving a Slice to after a Reflect operator when it had been the other way around. Changing the order of operators can result in vastly different results for a plot. The opposite function is DemoteOperator.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
AddOperator("Slice")
s = SliceAttributes()
s.project2d = 0
s.originPoint = (0,5,0)
s.originType=s.Point
s.normal = (0,1,0)
s.upAxis = (-1,0,0)
SetOperatorOptions(s)
AddOperator("Reflect")
DrawPlots()
# Now slice after reflect. We'll only get 1 slice plane instead of 2.
PromoteOperator(0)
DrawPlots()
```

5.4.170 PythonQuery

Synopsis:

```
PythonQuery(source='python filter source ...') -> integer
PythonQuery(file='path/to/python_filter_script.py') -> integer
```

source [string] A string containing the source code for a Python Query Filter .

file [string] A string containing the path to a Python Query Filter script file. Note - Use only one of the 'source' or 'file' arguments. If both are used the 'source' argument overrides 'file'.

return type [CLI_return_t] The PythonQuery function returns 1 on success and 0 on failure.

Description:

Used to execute a Python Filter Query.

5.4.171 Queries

Synopsis:

```
Queries() -> tuple of strings
```

return type [tuple of strings] The Queries function returns a tuple of strings.

Description:

The Queries function returns a tuple of strings that contain the names of all of VisIt's supported queries.

Example:

```
#% visit -cli
print("supported queries: ", Queries())
```

5.4.172 QueriesOverTime

Synopsis:

```
QueriesOverTime() -> tuple of strings
```

return type [tuple of strings] Returns a tuple of strings.

Description:

The QueriesOverTime function returns a tuple of strings that contains the names of all of the VisIt queries that can be executed over time.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/allineone00.pdb")
AddPlot("Pseudocolor", "mesh/mixvar")
DrawPlots()
# Execute each of the queries over time on the plots.
for q in QueriesOverTime():
    QueryOverTime(q)
# You can control timestates used in the query via start_time,
# end_time, and stride as follows:
QueryOverTime("Volume", start_time=5, end_time=250, stride=5)
# (Defaults used if not specified are 0, nStates, 1)
```

5.4.173 Query

Synopsis:

```
Query(name) -> string
Query(name, dict) -> string
Query(name, namedarg1=arg1, namedarg2=arg2, ...) -> string
Query(name) -> double, tuple of double
Query(name, dict) -> double, tuple of double
Query(name, namedarg1=arg1, namedarg2=arg2, ...) -> double, tuple of double
Query(name) -> dictionary
Query(name, dict) -> dictionary
Query(name, namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

name [string] The name of the query to execute.

dict [dictionary] An optional dictionary containing additional query arguments. namedarg1, namedarg2,... An optional list of named arguments supplying additional query parameters.

return type [see SetQueryOutputToXXX() functions] The Query function returns either a String (default), Value(s), or Object. The return type can be customized via calls to SetQueryOutputToXXX(), where 'XXX' is 'String', 'Value', or 'Object'. For more information on these return types, see 'GetQueryOutput'.

Description:

The Query function is used to execute any of VisIt's predefined queries. The list of queries can be found in theVisIt User's Manual in the Quantitative Analysis chapter. You can get also get a list of queries using 'Queries' function. Since queries can take a wide array of arguments, the Query function takes either a python dictorary or a list of named arguments specific to the given query. To obtain the possible options for a given query, use the GetQueryParameters(name) function. If the query accepts additional arguments beyond its name, this function will return a python dictionary containing the needed variables and their default values. This can be modified and passed back to the Query method, or named arguments can be used instead.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
Query("Volume")
Query("MinMax")
Query("MinMax", use_actual_data=1)
hohlraumArgs = GetQueryParameters("Hohlraum Flux")
hohlraumArgs["ray_center"]=(0.5,0.5,0)
hohlraumArgs["vars"]=("a1", "e1")
Query("Hohlraum Flux", hohlraumArgs)
```

5.4.174 QueryOverTime

Synopsis:

```
QueryOverTime(name) -> integer
QueryOverTime(name, dict) -> integer
QueryOverTime(name, namedarg1=val1, namedarg2=val2, ...) -> integer
```

name [string] The name of the query to execute.

dict [dictionary] An optional dictionary containing additional query arguments. namedarg1, namedarg2, ... An optional list of named arguments supplying additional query parameters.

return type [CLI_return_t] The QueryOverTime function returns 1 on success and 0 on failure.

Description:

The QueryOverTime function is used to execute any of VisIt's predefined queries. The list of queries can be found in the VisIt User's Manual in the Quantitative Analysis chapter. You can get also get a list of queries that can be executed over time using 'QueriesOverTime' function. Since queries can take a wide array of arguments, the Query function takes either a python dictionary or a list of named arguments specific to the given query. To obtain the possible options for a given query, use the GetQueryParameters(name) function. If the query accepts additional arguments beyond its name, this function will return a python dictionary containing the needed variables and their default values. This can be modified and passed back to the QueryOverTime method, or named arguments can be used instead.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
for q in QueriesOverTime():
    QueryOverTime(q)
```

5.4.175 ReOpenDatabase

Synopsis:

```
ReOpenDatabase(databaseName) -> integer
```

databaseName [string] The name of the database to open.

return type [CLI_return_t] The ReOpenDatabase function returns an integer value of 1 for success and 0 for failure.

Description:

The ReOpenDatabase function reopens a database that has been opened previously with the OpenDatabase function. The ReOpenDatabase function is primarily used for regenerating plots whose database has been rewritten on disk. ReOpenDatabase allows VisIt to access new variables and new time states that have been added since the database was opened using the OpenDatabase function. Note that ReOpenDatabase is expensive since it causes all plots that use the specified database to be regenerated. If you want to ensure that a time-varying database has all of its time states as they are being created by a simulation, try the CheckForNewStates function instead. The databaseName argument is a string containing the full name of the database to be opened. The database name is of the form: host:/path/filename. The host part of the filename can be omitted if the database to be reopened resides on the local computer.

Example:

```
#% visit -cli
OpenDatabase("edge:/usr/gapps/visit/data/wave*.silo database")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
last = TimeSliderGetNStates()
for state in range(last):
    SetTimeSliderState(state)
    SaveWindow()
ReOpenDatabase("edge:/usr/gapps/visit/data/wave*.silo database")
for state in range(last, TimeSliderGetNStates()):
    SetTimeSliderState(state)
    SaveWindow()
```

5.4.176 ReadHostProfilesFromDirectory

Synopsis:

```
ReadHostProfilesFromDirectory(directory, clear) -> integer
```

directory [string] The name of the directory that contains the host profile XML files.

clear [integer] An integer flag indicating whether the host profile list should cleared first.

return type [CLI_return_t] The ReadHostProfilesFromDirectory function returns an integer value of 1 for success and 0 for failure.

Description:

The ReadHostProfilesFromDirectory provides a way to tell VisIt to load host profiles from the XML files in a specified directory. This is needed because the machine profile for host profiles contains client/server options that sometimes cannot be specified via the VisIt command line.

Example:

```
ReadHostProfilesFromDirectory("/usr/gapps/visit/2.8.2/linux-x86_64/resources/hosts/
↪llnl", 1)
```

5.4.177 RecenterView

Synopsis:

```
RecenterView() -> integer
```

return type [CLI_return_t] The RecenterView function returns 1 on success and 0 on failure.

Description:

After adding plots to a visualization window or applying operators to those plots, it is sometimes necessary to recenter the view. When the view is recentered, the orientation does not change but the view is shifted to make better use of the screen.

Example:

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
OpenDatabase("/usr/gapps/visit/data/curv3d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
RecenterView()
```

5.4.178 RedoView

Synopsis:

```
RedoView() -> integer
```

return type [CLI_return_t] The RedoView function returns 1 on success and 0 on failure.

Description:

When the view changes in the visualization window, it puts the old view on a stack of views. VisIt provides the UndoView function that lets you undo view changes. The RedoView function re-applies any views that have been undone by the UndoView function.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Subset", "mat1")
DrawPlots()
v = GetView2D()
v.windowCoords = (-2.3,2.4,0.2,4.9)
SetView2D(v)
UndoView()
RedoView()
```

5.4.179 RedrawWindow

Synopsis:

```
RedrawWindow() -> integer
```

return type [CLI_return_t] The RedrawWindow function returns 1 on success and 0 on failure.

Description:

The RedrawWindow function allows a visualization window to redraw itself and then forces the window to redraw. This function does the opposite of the DisableRedraw function and is used to recover from it.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Contour", "u")
AddPlot("Pseudocolor", "w")
DrawPlots()
DisableRedraw()
AddOperator("Slice")
# Set the slice operator attributes
# Redraw now that the operator attributes are set. This will
# prevent 1 redraw.
RedrawWindow()
```

5.4.180 RegisterCallback

Synopsis:

```
RegisterCallback(callbackname, callback) --> integer
```

callbackname [string] A string object designating the callback that we're installing. Allowable values are returned by the GetCallbackNames() function.

callback [python function] A Python function, typically with one argument by which VisIt passes the object that caused the callback to be called.

return type [CLI_return_t] RegisterCallback returns 1 on success.

Description:

The RegisterCallback function is used to associate a user-defined callback function with the updating of a state object or execution of a particular rpc

Example:

```
import visit
def print_sliceatts(atts):
    print("SLICEATTS=", atts)

visit.RegisterCallback("SliceAttributes", print_sliceatts)
```

5.4.181 RegisterMacro

Synopsis:

```
RegisterMacro(name, callable)
```

name [string] The name of the macro.

callable [python function] A Python function that will be associated with the macro name.

Description:

The RegisterMacro function lets you associate a Python function with a name so when VisIt's gui calls down into Python to execute a macro, it ends up executing the registered Python function. Macros let users define complex new behaviors using Python functions yet still call them simply by clicking a button within VisIt's gui. When a new macro function is registered, a message is sent to the gui that adds the known macros as buttons in the Macros window.

Example:

```
def SetupMyPlots():
    OpenDatabase('noise.silo')
    AddPlot('Pseudocolor', 'hardyglobal')
    DrawPlots()

RegisterMacro('Setup My Plots', SetupMyPlots)
```

5.4.182 RemoveAllOperators

Synopsis:

```
RemoveAllOperators() -> integer
RemoveAllOperators(all) -> integer
```

all [integer] An optional integer argument that tells the function to ignore the active plots and use all plots in the plot list if the value of the argument is non-zero.

return type [CLI_return_t] All functions return an integer value of 1 for success and 0 for failure.

Description:

The RemoveAllOperators function removes all operators from the active plots in the active visualization window. If the all argument is provided and contains a non-zero value, all plots in the active visualization window are affected. If the value is zero or if the argument is not provided, only the active plots are affected.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddOperator("Threshold")
AddOperator("Slice")
AddOperator("SphereSlice")
DrawPlots()
RemoveLastOperator() # Remove SphereSlice
RemoveOperator(0) # Remove Threshold
RemoveAllOperators() # Remove the rest of the operators
```

5.4.183 RemoveLastOperator

Synopsis:

```
RemoveLastOperator() -> integer
RemoveLastOperator(all) -> integer
```

all [integer] An optional integer argument that tells the function to ignore the active plots and use all plots in the plot list if the value of the argument is non-zero.

return type [CLI_return_t] All functions return an integer value of 1 for success and 0 for failure.

Description:

The RemoveLastOperator function removes the operator that was last applied to the active plots. If the all argument is provided and contains a non-zero value, all plots in the active visualization window are affected. If the value is zero or if the argument is not provided, only the active plots are affected.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddOperator("Threshold")
AddOperator("Slice")
AddOperator("SphereSlice")
DrawPlots()
RemoveLastOperator() # Remove SphereSlice
RemoveOperator(0) # Remove Threshold
RemoveAllOperators() # Remove the rest of the operators
```

5.4.184 RemoveMachineProfile

Synopsis:

```
RemoveMachineProfile(hostname) -> integer
```

hostname : string

Description:

Removes machine profile with hostname from HostProfileList

5.4.185 RemoveOperator

Synopsis:

```
RemoveOperator(index) -> integer
RemoveOperator(index, all) -> integer
```

all [integer] An optional integer argument that tells the function to ignore the active plots and use all plots in the plot list if the value of the argument is non-zero.

index [integer] The zero-based integer index into a plot's operator list that specifies which operator is to be deleted.

return type [CLI_return_t] All functions return an integer value of 1 for success and 0 for failure.

Description:

The RemoveOperator functions allow operators to be removed from plots. If the all argument is provided and contains a non-zero value, all plots in the active visualization window are affected. If the value is zero or if the argument is not provided, only the active plots are affected.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddOperator("Threshold")
AddOperator("Slice")
AddOperator("SphereSlice")
DrawPlots()
RemoveLastOperator() # Remove SphereSlice
RemoveOperator(0) # Remove Threshold
RemoveAllOperators() # Remove the rest of the operators
```

5.4.186 RemovePicks

Synopsis:

```
RemovePicks()
```

Description:

The RemovePicks function removes a list of pick points from the active visualization window. Pick points are the letters that are added to the visualization window where the mouse is clicked when the visualization window is in pick mode.

Example:

```
#% visit -cli
# Put the visualization window into pick mode using the popup
# menu and add some pick points (let's say A -> G).
# Clear the pick points.
RemovePicks('A, B, D')
```

5.4.187 RenamePickLabel

Synopsis:

```
RenamePickLabel(oldLabel, newLabel) -> integer
```

oldLabel [string] A string that is the old pick label to replace. (e.g. 'A', 'B').

newLabel [string] A string that is the new label to display in place of the old label.

return type [CLI_return_t] The RenamePickLabel function returns 1 on success and 0 on failure.

Description:

The RenamePickLabel function can be used to replace an automatically generated pick label such as 'A' with a user-defined string.

Example:

```
RenamePickLabel('A', 'Point of interest')
```

5.4.188 ReplaceDatabase

Synopsis:

```
ReplaceDatabase(databaseName) -> integer
ReplaceDatabase(databaseName, timeState) -> integer
```

databaseName [string] The name of the new database.

timeState [integer] A zero-based integer containing the time state that should be made active once the database has been replaced.

return type [CLI_return_t] The ReplaceDatabase function returns an integer value of 1 for success and 0 for failure.

Description:

The ReplaceDatabase function replaces the database in the current plots with a new database. This is one way of switching timesteps if no ".visit" file was ever created. If two databases have the same variable name then replace is usually a success. In the case where the new database does not have the desired variable, the plot with the variable not contained in the new database does not get regenerated with the new database.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
ReplaceDatabase("/usr/gapps/visit/data/curv3d.silo")
SaveWindow()
# Replace with a time-varying database and change the time
# state to 17.
ReplaceDatabase("/usr/gapps/visit/data/wave.visit", 17)
```

5.4.189 ResetLineoutColor

Synopsis:

```
ResetLineoutColor() -> integer
```

return type [CLI_return_t] ResetLineoutColor returns 1 on success and 0 on failure.

Description:

Lineouts on VisIt cause reference lines to be drawn over the plot where the lineout was being extracted. Each reference line uses a different color in a discrete color table. Once the colors in the discrete color table are used up, the reference lines start using the color from the start of the discrete color table and so on. ResetLineoutColor forces reference lines to start using the color at the start of the discrete color table again thus resetting the lineout color.

5.4.190 ResetOperatorOptions

Synopsis:

```
ResetOperatorOptions(operatorType) -> integer
ResetOperatorOptions(operatorType, all) -> integer
```

operatorType [string] The name of a valid operator type.

all [integer] An optional integer argument that tells the function to reset the operator options for all plots regardless of whether or not they are active.

return type [CLI_return_t] The ResetOperatorOptions function returns an integer value of 1 for success and 0 for failure.

Description:

The ResetOperatorOptions function resets the operator attributes of the specified operator type for the active plots back to the default values. The operatorType argument is a string containing the name of the type of operator whose attributes are to be reset. The all argument is an optional flag that tells the function to reset the operator attributes for the indicated operator in all plots regardless of whether the plots are active. When non-zero values are passed for the all argument, all plots are reset. When the all argument is zero or not provided, only the operators on active plots are modified.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
AddOperator("Slice")
a = SliceAttributes()
a.normal,a.upAxis = (0,0,1),(0,1,0)
SetOperatorOptions(a)
ResetOperatorOptions("Slice")
```

5.4.191 ResetPickLetter

Synopsis:

```
ResetPickLetter() -> integer
```

return type [CLI_return_t] ResetPickLetter returns 1 on success and 0 on failure.

Description:

The ResetPickLetter function resets the pick marker back to “A” so that the next pick will use “A” as the pick letter and then “B” and so on.

5.4.192 ResetPlotOptions

Synopsis:

```
ResetPlotOptions(plotType) -> integer
```

plotType [string] The name of the plot type.

return type [CLI_return_t] The ResetPlotOptions function returns an integer value of 1 for success and 0 for failure.

Description:

The ResetPlotOptions function resets the plot attributes of the specified plot type for the active plots back to the default values. The plotType argument is a string containing the name of the type of plot whose attributes are to be reset.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
p = PseudocolorAttributes()
p.colorTableName = "calewhite"
p.minFlag,p.maxFlag = 1,1
p.min,p.max = -5.0, 8.0
SetPlotOptions(p)
ResetPlotOptions("Pseudocolor")
```

5.4.193 ResetView

Synopsis:

```
ResetView() -> integer
```

return type [CLI_return_t] The ResetView function returns 1 on success and 0 on failure.

Description:

The ResetView function resets the camera to the initial view.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv3d.silo")
AddPlot("Mesh", "curvmesh3d")
v = ViewAttributes()
v.camera = (-0.45396, 0.401908, 0.79523)
v.focus = (0, 2.5, 15)
v.viewUp = (0.109387, 0.910879, -0.397913)
v.viewAngle = 30
v.setScale = 1
v.parallelScale = 16.0078
v.nearPlane = -32.0156
v.farPlane = 32.0156
v.perspective = 1
SetView3D(v) # Set the 3D view
DrawPlots()
ResetView()
```

5.4.194 ResizeWindow

Synopsis:

```
ResizeWindow(win, w, h) -> integer
```

win [integer] The integer id of the window to be moved [1..16].

w [integer] The new integer width for the window.

h [integer] The new integer height for the window.

return type [CLI_return_t] ResizeWindow returns 1 on success and 0 on failure.

Description:

ResizeWindow resizes a visualization window.

Example:

```
#% visit -cli
ResizeWindow(1, 300, 600)
```

5.4.195 RestoreSession

Synopsis:

```
RestoreSession(filename, visitDir) -> integer
```

filename [string] The name of the session file to restore.

visitDir [integer] An integer flag that indicates whether the filename to be restored is located in the user's VisIt directory. If the flag is set to 1 then the session file is assumed to be located in the user's VisIt directory otherwise the filename must contain an absolute path.

return type [CLI_return_t] RestoreSession returns 1 on success and 0 on failure.

Description:

The RestoreSession function is important for setting up complex visualizations because you can design a VisIt session file, which is an XML file that describes exactly how plots are set up, using the VisIt GUI and then use that same session file in the CLI to generate movies in batch. The RestoreSession function takes 2 arguments. The first argument specifies the filename that contains the VisIt session to be restored. The second argument determines whether the session file is assumed to be in the user's VisIt directory. If the visitDir argument is set to 0 then the filename argument must contain the absolute path to the session file.

Example:

```
#% visit -cli
# Restore my session file for a time-varying database from
# my .visit directory.
RestoreSessionFile("visit.session", 1)
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
    SaveWindow()
```

5.4.196 RestoreSessionWithDifferentSources

Synopsis:

```
RestoreSessionWithDifferentSources(filename, visitDir, mapping) -> integer
```

filename [string] The name of the session file to restore.

visitDir [integer] An integer flag that indicates whether the filename to be restored is located in the user's VisIt directory. If the flag is set to 1 then the session file is assumed to be located in the user's VisIt directory otherwise the filename must contain an absolute path.

mapping [tuple] A tuple of strings representing the maping from sources as specified in the original session file to new sources. Sources in the original session file are numbered starting from 0. So, this tuple of strings simply contains the new names for each of the sources, in order.

return type [CLI_return_t] RestoreSession returns 1 on success and 0 on failure.

Description:

The RestoreSession function is important for setting up complex visualizations because you can design a VisIt session file, which is an XML file that describes exactly how plots are set up, using the VisIt GUI and then use that same session file in the CLI to generate movies in batch. The RestoreSession function takes 2 arguments. The first argument specifies the filename that contains the VisIt session to be restored. The second argument determines whether the session file is assumed to be in the user's VisIt directory. If the visitDir argument is set to 0 then the filename argument must contain the absolute path to the session file.

Example:

```
#% visit -cli
# Restore my session file for a time-varying database from
# my .visit directory.
RestoreSessionFile("visit.session", 1)
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
    SaveWindow()
```

5.4.197 SaveAttribute

Synopsis:

```
SaveAttribute(filename, object)
```

filename [string] The name of the XML file to load the attribute from or save the attribute to.

object The object to load or save.

Description:

The LoadAttribute and SaveAttribute methods save a single attribute, such as a current plot or operator python object, to a standalone XML file. Note that LoadAttribute requires that the target attribute already be created by other means; it fills, but does not create, the attribute.

Example:

```
#% visit -cli
a = MeshPlotAttributes()
```

(continues on next page)

(continued from previous page)

```
SaveAttribute('mesh.xml', a)
b = MeshPlotAttributes()
LoadAttribute('mesh.xml', b)
```

5.4.198 SaveNamedSelection

Synopsis:

```
SaveNamedSelection(name) -> integer
```

name [string] The name of a named selection.

return type [CLI_return_t] The SaveNamedSelection function returns 1 for success and 0 for failure.

Description:

Named Selections allow you to select a group of elements (or particles). One typically creates a named selection from a group of elements and then later applies the named selection to another plot (thus reducing the set of elements displayed to the ones from when the named selection was created). Named selections only last for the current session. If you create a named selection that you want to use over and over, you can save it to a file with the SaveNamedSelection function.

Example:

```
#% visit -cli
db = "/usr/gapps/visit/data/wave*.silo database"
OpenDatabase(db)
AddPlot("Pseudocolor", "pressure")
AddOperator("Clip")
c = ClipAttributes()
c.plane1Origin = (0,0.6,0)
c.plane1Normal = (0,-1,0)
SetOperatorOption(c)
DrawPlots()
CreateNamedSelection("els_above_at_time_0")
SaveNamedSelection("els_above_at_time_0")
```

5.4.199 SaveSession

Synopsis:

```
SaveSession(filename) -> integer
```

filename [string] The filename argument is the filename that is used to save the session file. The filename is relative to the current working directory.

return type [CLI_return_t] The SaveSession function returns 1 on success and 0 on failure.

Description:

The SaveSession function tells VisIt to save an XML session file that describes everything about the current visualization. Session files are very useful for creating movies and also as shortcuts for setting up complex visualizations.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
# Set up a keyframe animation of view and save a session file of it.
k = GetKeyframeAttributes()
k.enabled,k.nFrames,k.nFramesWasUserSet = 1,20,1
SetKeyframeAttributes(k)
AddPlot("Surface", "hgslice")
DrawPlots()
v = GetView3D()
v.viewNormal = (0.40823, -0.826468, 0.387684)
v.viewUp, v.imageZoom = (-0.261942, 0.300775, 0.917017), 1.60684
SetView3D(v)
SetViewKeyframe()
SetTimeSliderState(TimeSliderGetNStates() - 1)
v.viewNormal = (-0.291901, -0.435608, 0.851492)
v.viewUp = (0.516969, 0.677156, 0.523644)
SetView3D(v)
SetViewKeyframe()
ToggleCameraViewMode()
SaveSession("~/.visit/keyframe.session")
```

5.4.200 SaveWindow

Synopsis:

```
SaveWindow() -> string
```

return type [string] The SaveWindow function returns a string containing the name of the file that was saved.

Description:

The SaveWindow function saves the contents of the active visualization window. The format of the saved window is dictated by the SaveWindowAttributes which can be set using the SetSaveWindowAttributes function. The contents of the active visualization window can be saved as TIFF, JPEG, RGB, PPM, PNG images or they can be saved as curve, Alias Wavefront Obj, or VTK geometry files.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv3d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Set the save window attributes.
s = SaveWindowAttributes()
s.fileName = "test"
s.format = s.JPEG
s.progressive = 1
s.fileName = "test"
SetSaveWindowAttributes(s)
name = SaveWindow()
print("name = %s" % name)
```

5.4.201 SendSimulationCommand

Synopsis:

```
SendSimulationCommand(host, simulation, command)
SendSimulationCommand(host, simulation, command, argument)
```

host [string] The name of the computer where the simulation is running.

simulation [string] The name of the simulation being processed at the specified host.

command [string] A string that is the command to send to the simulation.

argument An argument to the command.

Description:

The SendSimulationCommand method tells the viewer to send a command to a simulation that is running on the specified host. The host argument is a string that contains the name of the computer where the simulation is running. The simulation argument is a string that contains the name of the simulation to send the command to.

5.4.202 SetDefaultContinuousColorTable

Synopsis:

```
SetDefaultContinuousColorTable(name) -> integer
```

name [string] The name of the color table to use for the active color table. The name must be present in the tuple returned by the ColorTableNames function.

return type [CLI_return_t] Both functions return 1 on success and 0 on failure.

Description:

VisIt supports two flavors of color tables: continuous and discrete. Both types of color tables have the same underlying representation but each type of color table is used a slightly different way. Continuous color tables are made of a small number of color control points and the gaps in the color table between two color control points are filled by interpolating the colors of the color control points. Discrete color tables do not use any kind of interpolation and like continuous color tables, they are made up of control points. The color control points in a discrete color table repeat infinitely such that if we have 4 color control points: A, B, C, D then the pattern of repetition is: ABCDABCDABCD... Discrete color tables are mainly used for plots that have a discrete set of items to display (e.g. Subset plot). Continuous color tables are used in plots that display a continuous range of values (e.g. Pseudocolor).

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Contour", "hgslice")
DrawPlots()
SetDefaultDiscreteColorTable("levels")
```

5.4.203 SetDefaultDiscreteColorTable

Synopsis:

```
SetDefaultDiscreteColorTable(name) -> integer
```

name [string] The name of the color table to use for the active color table. The name must be present in the tuple returned by the ColorTableNames function.

return type [CLI_return_t] Both functions return 1 on success and 0 on failure.

Description:

VisIt supports two flavors of color tables: continuous and discrete. Both types of color tables have the same underlying representation but each type of color table is used a slightly different way. Continuous color tables are made of a small number of color control points and the gaps in the color table between two color control points are filled by interpolating the colors of the color control points. Discrete color tables do not use any kind of interpolation and like continuous color tables, they are made up of control points. The color control points in a discrete color table repeat infinitely such that if we have 4 color control points: A, B, C, D then the pattern of repetition is: ABCDABCDABCD... Discrete color tables are mainly used for plots that have a discrete set of items to display (e.g. Subset plot). Continuous color tables are used in plots that display a continuous range of values (e.g. Pseudocolor).

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Contour", "hgslice")
DrawPlots()
SetDefaultDiscreteColorTable("levels")
```

5.4.204 SetActivePlots

Synopsis:

```
SetActivePlots(plots) -> integer
```

plots [tuple of integers] A tuple of integer plot indices starting at zero. A single integer is also accepted

return type [CLI_return_t] The SetActivePlots function returns an integer value of 1 for success and 0 for failure.

Description:

Any time VisIt sets the attributes for a plot, it only sets the attributes for plots which are active. The SetActivePlots function must be called to set the active plots. The function takes one argument which is a tuple of integer plot indices that start at zero. If only one plot is being selected, the plots argument can be an integer instead of a tuple.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Subset", "mat1")
AddPlot("Mesh", "mesh1")
AddPlot("Contour", "u")
DrawPlots()
SetActivePlots((0,1,2)) # Make all plots active
SetActivePlots(0) # Make only the Subset plot active
```

5.4.205 SetActiveTimeSlider

Synopsis:

```
SetActiveTimeSlider(tsName) -> integer
```

tsName [string] The name of the time slider that should be made active.

return type [CLI_return_t] SetActiveTimeSlider returns 1 on success and 0 on failure.

Description:

Sets the active time slider, which is the time slider that is used to change time states.

Example:

```
#% visit -cli
path = "/usr/gapps/visit/data/"
dbs = (path + "dbA00.pdb", path + "dbB00.pdb", path + "dbC00.pdb")
for db in dbs:
    OpenDatabase(db)
    AddPlot("FilledBoundary", "material(mesh)")
    DrawPlots()
CreateDatabaseCorrelation("common", dbs, 1)
tsNames = GetWindowInformation().timeSliders
for ts in tsNames:
    SetActiveTimeSlider(ts)
for state in list(range(TimeSliderGetNStates())) + [0]:
    SetTimeSliderState(state)
```

5.4.206 SetActiveWindow

Synopsis:

```
SetActiveWindow(windowIndex) -> integer
SetActiveWindow(windowIndex, raiseWindow) -> integer
```

windowIndex [integer] An integer window index starting at 1.

raiseWindow [integer] This is an optional integer argument that raises and activates the window if set to 1. If omitted, the default behavior is to raise and activate the window.

return type [CLI_return_t] The SetActiveWindow function returns an integer value of 1 for success and 0 for failure.

Description:

Most of the functions in the VisIt Python Interface operate on the contents of the active window. If there is more than one window, it is very important to be able to set the active window. To set the active window, use the SetActiveWindow function. The SetActiveWindow function takes a single integer argument which is the index of the new active window. The new window index must be an integer greater than zero and less than or equal to the number of open windows.

Example:

```
#% visit -cli
SetWindowLayout(2)
SetActiveWindow(2)
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Mesh", "mesh1")
DrawPlots()
```

5.4.207 SetAnimationTimeout

Synopsis:

```
SetAnimationTimeout(milliseconds) -> integer
```

milliseconds [integer] A positive integer to specify the number of milliseconds.

return type [integer] The SetAnimationTimeout function returns 1 for success and 0 for failure.

Description:

The SetAnimationTimeout function sets the animation timeout which is a value that governs how fast animations play. The timeout is specified in milliseconds and has a default value of 1 millisecond. Larger timeout values decrease the speed at which animations play.

Example:

```
#%visit -cli
# Play a new frame every 5 seconds.
SetAnimationTimeout(5000)
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
# Click the play button in the toolbar
```

5.4.208 SetAnnotationAttributes

Synopsis:

```
SetAnnotationAttributes(atts) -> integer
```

atts [AnnotationAttributes object] An AnnotationAttributes object containing the annotation settings.

return type [CLI_return_t] Both functions return 1 on success and 0 on failure.

Description:

The annotation settings control what bits of text are drawn in the visualization window. Among the annotations are the plot legends, database information, user information, plot axes, triad, and the background style and colors. Setting the annotation attributes is important for producing quality visualizations. The annotation settings are stored in AnnotationAttributes objects. To set the annotation attributes, first create an AnnotationAttributes object using the AnnotationAttributes function and then pass the object to the SetAnnotationAttributes function. To set the default annotation attributes, also pass the object to the SetDefaultAnnotationAttributes function.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
a = AnnotationAttributes()
a.gradientBackgroundStyle = a.GRADIENTSTYLE_RADIAL
a.gradientColor1 = (0,255,255)
a.gradientColor2 = (0,0,0)
a.backgroundMode = a.BACKGROUNDMODE_GRADIENT
SetAnnotationAttributes(a)
```

5.4.209 SetBackendType

Synopsis:

```
SetBackendType(name) -> integer
```

name [string] VTK, VTKM.

return type [CLI_return_t] Both functions return 1 on success and 0 on failure.

Description:

The compute back end determines the compute library that is used for processing plots in VisIt. The default is VTK, which supports all VisIt operations. VTKm can be used too but it only supports a fraction of VisIt's functionality. Filters that support VTKm will use those libraries when their compute back end is selected using this function.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
SetBackendType("VTKm")
AddPlot("Contour", "radial")
DrawPlots()
```

5.4.210 SetCenterOfRotation

Synopsis:

```
SetCenterOfRotation(x,y,z) -> integer
```

x [double] A double that is the x component of the center of rotation.

y [double] A double that is the y component of the center of rotation.

z [double] A double that is the z component of the center of rotation.

return type [CLI_return_t] The SetCenterOfRotation function returns 1 on success and 0 on failure.

Description:

The SetCenterOfRotation function sets the center of rotation for plots in a 3D visualization window. The center of rotation, is the point about which plots are rotated when you interactively spin the plots using the mouse. It is useful to set the center of rotation if you've zoomed in on any 3D plots so in the event that you rotate the plots, the point of interest remains fixed on the screen.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddPlot("Mesh", "mesh1")
DrawPlots()
v = GetView3D()
v.viewNormal = (-0.409139, 0.631025, 0.6591)
v.viewUp = (0.320232, 0.775678, -0.543851)
v.imageZoom = 4.8006
SetCenterOfRotation(-4.755280, 6.545080, 5.877850)
# Rotate the plots interactively.
```

5.4.211 SetColorTexturingEnabled

Synopsis:

```
SetColorTexturingEnabled(enabled) -> integer
```

enabled [integer] A integer value. Non-zero values enable color texturing and zero disables it.

return type [CLI_return_t] The SetColorTexturingEnabled function returns 1 on success and 0 on failure.

Description:

Node-centered variables are drawn on plots such as the Pseudocolor plot such that the nodal value looks interpolated throughout the zone. This can be done by interpolating colors, which can produce some colors that do not appear in a color table. Alternatively, the nodal values can be mapped to a texture coordinate in a 1D texture and those values can be interpolated, with colors being selected after interpolating the texture coordinate. This method always uses colors that are defined in the color table.

Example:

```
SetColorTexturingEnabled(1)
```

5.4.212 SetCreateMeshQualityExpressions

Synopsis:

```
SetCreateMeshQualityExpressions(val) -> integer
```

val [integer] Either a zero (false) or non-zero (true) integer value to indicate if Mesh Quality expressions should be automatically created when a database is opened.

return type [CLI_return_t] The SetCreateMeshQualityExpressions function returns 1 on success and 0 on failure.

Description:

The SetCreateMeshQualityExpressions function sets a boolean in the global attributes indicating whether or not Mesh Quality expressions should be automatically created. The default behavior is for the expressions to be created, which may slow down VisIt's performance if there is an extraordinary large number of meshes. Turning this feature off tells VisIt to skip automatic creation of the Mesh Quality expressions.

Example:

```
#% visit -cli
SetCreateMeshQualityExpressions(1) # turn this feature on
SetCreateMeshQualityExpressions(0) # turn this feature off
```

5.4.213 SetCreateTimeDerivativeExpressions

Synopsis:

```
SetCreateTimeDerivativeExpressions(val) -> integer
```

val [integer] Either a zero (false) or non-zero (true) integer value to indicate if Time Derivative expressions should be automatically created when a database is opened.

return type [CLI_return_t] The SetCreateTimeDerivativeExpressions function returns 1 on success and 0 on failure.

Description:

The SetCreateTimeDerivativeExpressions function sets a boolean in the global attributes indicating whether or not Time Derivative expressions should be automatically created. The default behavior is for the expressions to be created, which may slow down VisIt's performance if there is an extraordinary large number of variables. Turning this feature off tells VisIt to skip automatic creation of the Time Derivative expressions.

Example:

```
#% visit -cli
SetCreateTimeDerivativeExpressions(1) # turn this feature on
SetCreateTimeDerivativeExpressions(0) # turn this feature off
```

5.4.214 SetCreateVectorMagnitudeExpressions

Synopsis:

```
SetCreateVectorMagnitudeExpressions(val) -> integer
```

val [integer] Either a zero (false) or non-zero (true) integer value to indicate if Vector magnitude expressions should be automatically created when a database is opened.

return type [CLI_return_t] The SetCreateVectorMagnitudeExpressions function returns 1 on success and 0 on failure.

Description:

The SetCreateVectorMagnitudeExpressions function sets a boolean in the global attributes indicating whether or not vector magnitude expressions should be automatically created. The default behavior is for the expressions to be created, which may slow down VisIt's performance if there is an extraordinary large number of vector variables. Turning this feature off tells VisIt to skip automatic creation of the vector magnitude expressions.

Example:

```
#% visit -cli
SetCreateVectorMagnitudeExpressions(1) # turn this feature on
SetCreateVectorMagnitudeExpressions(0) # turn this feature off
```

5.4.215 SetDatabaseCorrelationOptions

Synopsis:

```
SetDatabaseCorrelationOptions(method, whenToCreate) -> integer
```

method [integer] An integer that tells VisIt what default method to use when automatically creating a database correlation. The value must be in the range [0,3].

method	Description
0	IndexForIndexCorrelation
1	StretchedIndexCorrelation
2	TimeCorrelation
3	CycleCorrelation

whenToCreate [integer] An integer that tells VisIt when to automatically create database correlations.

whenToCreate	Description
0	Always create database correlation
1	Never create database correlation
2	Create database correlation only if the new time-varying database has

return type [CLI_return_t] SetDatabaseCorrelationOptions returns 1 on success and 0 on failure.

Description:

VisIt provides functions to explicitly create and alter database correlations but there are also a number of occasions where VisIt can automatically create a database correlation. The SetDatabaseCorrelationOptions function allows you to tell VisIt the default correlation method to use when automatically creating a new database correlation and it also allows you to tell VisIt when database correlations can be automatically created. the same length as another time-varying database already being used in a plot.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/dbA00.pdb")
AddPlot("FilledBoundary", "material(mesh)")
DrawPlots()
# Always create a stretched index correlation.
SetDatabaseCorrelationOptions(1, 0)
OpenDatabase("/usr/gapps/visit/data/dbB00.pdb")
AddPlot("FilledBoundary", "material(mesh)")
# The AddPlot caused a database correlation to be created.
DrawPlots()
wi = GetWindowInformation()
print("Active time slider: " % wi.timeSliders[wi.activeTimeSlider])
# This will set time for both databases since the database correlation is
# the active time slider.
SetTimeSliderState(5)
```

5.4.216 SetDebugLevel

Synopsis:

```
SetDebugLevel(level)
```

level [string] A string '1', '2', '3', '4', '5' with an optional 'b' suffix to indicate whether the output should be buffered. A value of '1' is a low debug level, which should be used to produce little output while a value of 5 should produce a lot of debug output.

Description:

The GetDebugLevel and SetDebugLevel functions are used when debugging VisIt Python scripts. The SetDebugLevel function sets the debug level for VisIt's viewer thus it must be called before a Launch method. The debug level determines how much detail is written to VisIt's execution logs when it executes.

Example:

```
#% visit -cli -debug 2
print("VisIt's debug level is: %d" % GetDebugLevel())
```

5.4.217 SetDefaultAnnotationAttributes

Synopsis:

```
SetDefaultAnnotationAttributes(atts) -> integer
```

atts [AnnotationAttributes object] An AnnotationAttributes object containing the annotation settings.

return type [CLI_return_t] Both functions return 1 on success and 0 on failure.

Description:

The annotation settings control what bits of text are drawn in the visualization window. Among the annotations are the plot legends, database information, user information, plot axes, triad, and the background style and colors. Setting the annotation attributes is important for producing quality visualizations. The annotation settings are stored in AnnotationAttributes objects. To set the annotation attributes, first create an AnnotationAttributes object using the AnnotationAttributes function and then pass the object to the SetAnnotationAttributes function. To set the default annotation attributes, also pass the object to the SetDefaultAnnotationAttributes function.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
a = AnnotationAttributes()
a.gradientBackgroundStyle = a.GRADIENTSTYLE_RADIAL
a.gradientColor1 = (0,255,255)
a.gradientColor2 = (0,0,0)
a.backgroundMode = a.BACKGROUNDMODE_GRADIENT
SetAnnotationAttributes(a)
```

5.4.218 SetDefaultFileOpenOptions

Synopsis:

```
SetDefaultFileOpenOptions(pluginName, options) -> integer
```

pluginName [string] The name of a plugin.

options [dictionary] A dictionary containing the new default options for that plugin.

return type [CLI_return_t] The SetDefaultFileOpenOptions function returns 1 on success and 0 on failure.

Description:

SetDefaultFileOpenOptions sets the current options used to open new files when a specific plugin is triggered.

Example:

```
#% visit -cli
OpenMDServer()
opts = GetDefaultFileOpenOptions("VASP")
opts["Allow multiple timesteps"] = 1
SetDefaultFileOpenOptions("VASP", opts)
OpenDatabase("CHGCAR")
```

5.4.219 SetDefaultInteractorAttributes

Synopsis:

```
SetDefaultInteractorAttributes(atts) -> integer
```

atts [InteractorAttributes object] An InteractorAttributes object that contains the new interactor attributes that you want to use.

return type [CLI_return_t] SetInteractorAttributes returns 1 on success and 0 on failure.

Description:

The SetInteractorAttributes function is used to set certain interactor properties. Interactors, can be thought of as how mouse clicks and movements are translated into actions in the vis window. To set the interactor attributes, first get the interactor attributes using the GetInteractorAttributes function. Once you've set the object's properties, call the SetInteractorAttributes function to make VisIt use the new interactor attributes. The SetDefaultInteractorAttributes function sets the default interactor attributes, which are used for new visualization windows. The default interactor attributes can also be saved to the VisIt configuration file to ensure that future VisIt sessions have the right default interactor attributes.

Example:

```
#% visit -cli
ia = GetInteractorAttributes()
print(ia)
ia.showGuidelines = 0
SetInteractorAttributes(ia)
```

5.4.220 SetDefaultMaterialAttributes

Synopsis:

```
SetDefaultMaterialAttributes(atts) -> integer
```

atts [MaterialAttributes object] A MaterialAttributes object containing the new settings.

return type [CLI_return_t] Both functions return 1 on success and 0 on failure.

Description:

The SetMaterialAttributes function takes a MaterialAttributes object and makes VisIt use the material settings that it contains. You use the SetMaterialAttributes function when you want to change how VisIt performs material interface reconstruction. The SetDefaultMaterialAttributes function sets the default material attributes, which are saved to the config file and are also used by new visualization windows.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/allinone00.pdb")
AddPlot("Pseudocolor", "mesh/mixvar")
p = PseudocolorAttributes()
p.min,p.minFlag = 4.0, 1
p.max,p.maxFlag = 13.0, 1
SetPlotOptions(p)
DrawPlots()
# Tell VisIt to always do material interface reconstruction.
m = GetMaterialAttributes()
```

(continues on next page)

(continued from previous page)

```
m.forceMIR = 1
SetMaterialAttributes(m)
ClearWindow()
# Redraw the plot forcing VisIt to use the mixed variable information.
DrawPlots()
```

5.4.221 SetDefaultMeshManagementAttributes

Synopsis:

```
SetMeshManagementAttributes() -> MeshmanagementAttributes object
```

return type [MeshmanagementAttributes object] Returns a MeshmanagementAttributes object.

Description:

The GetMeshmanagementAttributes function returns a MeshmanagementAttributes object that contains VisIt's current mesh discretization settings. You can set properties on the MeshManagementAttributes object and then pass it to SetMeshManagementAttributes to make VisIt use the new material attributes that you've specified:

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/csg.silo")
AddPlot("Mesh", "csgmesh")
DrawPlots()
# Tell VisIt to always do material interface reconstruction.
mma = GetMeshManagementAttributes()
mma.discretizationTolernace = (0.01, 0.025)
SetMeshManagementAttributes(mma)
ClearWindow()
# Redraw the plot forcing VisIt to use the mixed variable information.
DrawPlots()
```

5.4.222 SetDefaultOperatorOptions

Synopsis:

```
SetDefaultOperatorOptions(atts) -> integer
```

atts [operator attributes object] Any type of operator attributes object.

return type [CLI_return_t] All functions return an integer value of 1 for success and 0 for failure.

Description:

Each operator in VisIt has a group of attributes that controls the operator. To set the attributes for an operator, first create an operator attributes object. This is done by calling a function which is the name of the operator plus the word "Attributes". For example, a Slice operator's operator attributes object is created and returned by the SliceAttributes function. Assign the new operator attributes object into a variable and set its fields. After setting the desired fields in the operator attributes object, pass the object to the SetOperatorOptions function. The SetOperatorOptions function determines the type of operator to which the operator attributes object applies and sets the attributes for that operator type. To set the default plot attributes, use the SetDefaultOperatorOptions function. Setting the default attributes ensures

that all future instances of a certain operator are initialized with the new default values. Note that there is no SetOperatorOptions(atts, all) variant of this call. To set operator options for all plots that have a an instance of the associated operator, you must first make all plots active with SetActivePlots() and then use the SetOperatorOptions(atts) variant.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddPlot("Mesh", "mesh1")
AddOperator("Slice", 1) # Add the operator to both plots
a = SliceAttributes()
a.normal, a.upAxis = (0,0,1), (0,1,0)
# Only set the attributes for the active plot.
SetOperatorOptions(a)
DrawPlots()
```

5.4.223 SetDefaultPickAttributes

Synopsis:

```
SetDefaultPickAttributes(atts) -> integer
```

atts [PickAttributes object] A PickAttributes object containing the new pick settings.

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The SetPickAttributes function changes the pick attributes that are used when VisIt picks on plots. The pick attributes allow you to format your pick output in various ways and also allows you to select auxiliary pick variables.

Example:

```
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()
ZonePick(coord=(-5,5,0))
p = GetPickAttributes()
p.showTimeStep = 0
p.showMeshName = 0
p.showZoneId = 0
SetPickAttributes(p)
ZonePick(coord=(0,5,0))
```

5.4.224 SetDefaultPlotOptions

Synopsis:

```
SetDefaultPlotOptions(atts) -> integer
```

atts [plot attributes object] Any type of plot attributes object.

return type [CLI_return_t] All functions return an integer value of 1 for success and 0 for failure.

Description:

Each plot in VisIt has a group of attributes that controls the appearance of the plot. To set the attributes for a plot, first create a plot attributes object. This is done by calling a function which is the name of the plot plus the word "Attributes". For example, a Pseudocolor plot's plotattributes object is created and returned by the PseudocolorAttributes function. Assign the new plot attributes object into a variable and set its fields. After setting the desired fields in the plot attributes object, pass the object to the SetPlotOptions function. The SetPlotOptions function determines the type of plot to which the plot attributes object applies and sets the attributes for that plot type. To set the default plot attributes, use the SetDefaultPlotOptions function. Setting the default attributes ensures that all future instances of a certain plot are initialized with the new default values.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
p = PseudocolorAttributes()
p.colorTableName = "calewhite"
p.minFlag,p.maxFlag = 1,1
p.min,p.max = -5.0, 8.0
SetPlotOptions(p)
DrawPlots()
```

5.4.225 SetGlobalLineoutAttributes

Synopsis:

```
SetGlobalLineoutAttributes(atts) -> integer
```

atts [GlobalLineoutAttributes object] A GlobalLineoutAttributes object that contains the new settings.

return type [CLI_return_t] The SetGlobalLineoutAttributes function returns 1 on success and 0 on failure.

Description:

The SetGlobalLineoutAttributes function allows you to set global lineout options that are used in the creation of all lineouts. You can, for example, specify the destination window and the number of sample points for lineouts.

Example:

```
#% visit -cli
SetWindowLayout(4)
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()
gla = GetGlobalLineoutAttributes()
gla.createWindow = 0
gla.windowId = 4
gla.samplingOn = 1
gla.numSamples = 150
SetGlobalLineoutAttributes(gla)
Lineout((-5,-8), (-3.5, 8))
```

5.4.226 SetInteractorAttributes

Synopsis:

```
SetInteractorAttributes(atts) -> integer
```

atts [InteractorAttributes object] An InteractorAttributes object that contains the new interactor attributes that you want to use.

return type [CLI_return_t] SetInteractorAttributes returns 1 on success and 0 on failure.

Description:

The SetInteractorAttributes function is used to set certain interactor properties. Interactors, can be thought of as how mouse clicks and movements are translated into actions in the vis window. To set the interactor attributes, first get the interactor attributes using the GetInteractorAttributes function. Once you've set the object's properties, call the SetInteractorAttributes function to make VisIt use the new interactor attributes. The SetDefaultInteractorAttributes function sets the default interactor attributes, which are used for new visualization windows. The default interactor attributes can also be saved to the VisIt configuration file to ensure that future VisIt sessions have the right default interactor attributes.

Example:

```
#% visit -cli
ia = GetInteractorAttributes()
print(ia)
ia.showGuidelines = 0
SetInteractorAttributes(ia)
```

5.4.227 SetKeyframeAttributes

Synopsis:

```
SetKeyframeAttributes(kfAtts) -> integer
```

kfAtts [KeyframeAttributes object] A KeyframeAttributes object that contains the new keyframing attributes to use.

return type [CLI_return_t] SetKeyframeAttributes returns 1 on success and 0 on failure.

Description:

Use the SetKeyframeAttributes function when you want to change VisIt's keyframing settings. You must pass a KeyframeAttributes object, which you can create using the GetKeyframeAttributes function. The KeyframeAttributes object must contain the keyframing settings that you want VisIt to use. For example, you would use the SetKeyframeAttributes function if you wanted to turn on keyframing mode and set the number of animation frames.

Example:

```
#% visit -cli
k = GetKeyframeAttributes()
print(k)
k.enabled,k.nFrames,k.nFramesWasUserSet = 1, 100, 1
SetKeyframeAttributes(k)
```

5.4.228 SetLight

Synopsis:

```
SetLight(index, light) -> integer
```

index [integer] A zero-based integer index into the light list. Index can be in the range [0,7].

light [LightAttributes object] A LightAttributes object containing the properties to use for the specified light.

return type [CLI_return_t] SetLight returns 1 on success and 0 on failure.

Description:

The SetLight function sets the attributes for a specific light.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "w")
p = PseudocolorAttributes()
p.colorTableName = "xray"
SetPlotOptions(p)
DrawPlots()
InvertBackgroundColor()
light = GetLight(0)
print(light)
light.enabledFlag = 1
light.direction = (0,-1,0)
light.color = (255,0,0,255)
SetLight(0, light)
light.color,light.direction = (0,255,0,255), (-1,0,0)
SetLight(1, light)
```

5.4.229 SetMachineProfile

Synopsis:

```
SetMachineProfile(MachineProfile) -> integer
```

MachineProfile [MachineProfile object] A MachineProfile object containing the new settings.

Description:

Sets the input machine profile in the HostProfileList, replaces if one already exists. Otherwise adds to the list

5.4.230 SetMaterialAttributes

Synopsis:

```
SetMaterialAttributes(atts) -> integer
```

atts [MaterialAttributes object] A MaterialAttributes object containing the new settings.

return type [CLI_return_t] Both functions return 1 on success and 0 on failure.

Description:

The SetMaterialAttributes function takes a MaterialAttributes object and makes VisIt use the material settings that it contains. You use the SetMaterialAttributes function when you want to change how VisIt performs material interface reconstruction. The SetDefaultMaterialAttributes function sets the default material attributes, which are saved to the config file and are also used by new visualization windows.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/allinone00.pdb")
AddPlot("Pseudocolor", "mesh/mixvar")
p = PseudocolorAttributes()
p.min,p.minFlag = 4.0, 1
p.max,p.maxFlag = 13.0, 1
SetPlotOptions(p)
DrawPlots()
# Tell VisIt to always do material interface reconstruction.
m = GetMaterialAttributes()
m.forceMIR = 1
SetMaterialAttributes(m)
ClearWindow()
# Redraw the plot forcing VisIt to use the mixed variable information.
DrawPlots()
```

5.4.231 SetMeshManagementAttributes

Synopsis:

```
GetMeshManagementAttributes() -> MeshmanagementAttributes object
```

return type [MeshmanagementAttributes object] Returns a MeshmanagementAttributes object.

Description:

The GetMeshmanagementAttributes function returns a MeshmanagementAttributes object that contains VisIt's current mesh discretization settings. You can set properties on the MeshManagementAttributes object and then pass it to SetMeshManagementAttributes to make VisIt use the new material attributes that you've specified:

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/csg.silo")
AddPlot("Mesh", "csgmesh")
DrawPlots()
# Tell VisIt to always do material interface reconstruction.
mma = GetMeshManagementAttributes()
mma.discretizationTolernace = (0.01, 0.025)
SetMeshManagementAttributes(mma)
ClearWindow()
# Redraw the plot forcing VisIt to use the mixed variable information.
DrawPlots()
```

5.4.232 SetNamedSelectionAutoApply

Synopsis:

```
SetNamedSelectionAutoApply(flag) -> integer
```

flag [integer] An integer flag. Non-zero values turn on selection auto apply mode.

return type [CLI_return_t] The SetNamedSelectionAutoApply function returns 1 on success and 0 on failure.

Description:

Named selections are often associated with plots for their data source. When those plots update, their named selections can be updated, which in turn will update any plots that use the named selection. When this mode is enabled, changes to a named selection's originating plot will cause the selection to be updated automatically.

Example:

```
SetNamedSelectionAutoApply(1)
```

5.4.233 SetOperatorOptions

Synopsis:

```
SetOperatorOptions(atts) -> integer
SetOperatorOptions(atts, operatorIndex) -> integer
SetOperatorOptions(atts, operatorIndex, all) -> integer
```

atts [operator attributes object] Any type of operator attributes object.

operatorIndex [integer] An optional zero-based integer that serves as an index into the active plot's operator list. Use this argument if you want to set the operator attributes for a plot that has multiple instances of the same type of operator. For example, if the active plot had a Transform operator followed by a Slice operator followed by another Transform operator and you wanted to adjust the attributes of the second Transform operator, you would pass an operatorIndex value of 2.

all [integer] An optional integer argument that tells the function to apply theoperator attributes to all plots containing the specified operator if the value of the argument is non-zero.

return type [CLI_return_t] All functions return an integer value of 1 for success and 0 for failure.

Description:

Each operator in VisIt has a group of attributes that controls the operator. To set the attributes for an operator, first create an operator attributes object. This is done by calling a function which is the name of the operator plus the word "Attributes". For example, a Slice operator's operator attributes object is created and returned by the SliceAttributes function. Assign the new operator attributes object into a variable and set its fields. After setting the desired fields in the operator attributes object, pass the object to the SetOperatorOptions function. The SetOperatorOptions function determines the type of operator to which the operator attributes object applies and sets the attributes for that operator type. To set the default plot attributes, use the SetDefaultOperatorOptions function. Setting the default attributes ensures that all future instances of a certain operator are initialized with the new default values. Note that there is no SetOperatorOptions(atts, all) variant of this call. To set operator options for all plots that have a an instance of the associated operator, you must first make all plots active with SetActivePlots() and then use the SetOperatorOptions(atts) variant.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
```

(continues on next page)

(continued from previous page)

```
AddPlot("Pseudocolor", "u")
AddPlot("Mesh", "mesh1")
AddOperator("Slice", 1) # Add the operator to both plots
a = SliceAttributes()
a.normal, a.upAxis = (0,0,1), (0,1,0)
# Only set the attributes for the active plot.
SetOperatorOptions(a)
DrawPlots()
```

5.4.234 SetPickAttributes

Synopsis:

```
SetPickAttributes(atts) -> integer
```

atts [PickAttributes object] A PickAttributes object containing the new pick settings.

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The SetPickAttributes function changes the pick attributes that are used when VisIt picks on plots. The pick attributes allow you to format your pick output in various ways and also allows you to select auxiliary pick variables.

Example:

```
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()
ZonePick(coord=(-5,5,0))
p = GetPickAttributes()
p.showTimeStep = 0
p.showMeshName = 0
p.showZoneId = 0
SetPickAttributes(p)
ZonePick(coord=(0,5,0))
```

5.4.235 SetPipelineCachingMode

Synopsis:

```
SetPipelineCachingMode(mode) -> integer
```

mode [boolean] A boolean value to turn pipeline caching on or off.

return type [CLI_return_t] The SetPipelineCachingMode function returns 1 for success and 0 for failure.

Description:

The SetPipelineCachingMode function turns pipeline caching on or off in the viewer. When pipeline caching is enabled, animation timesteps are cached for fast playback. This can be a disadvantage for large databases or for plots with many timesteps because it increases memory consumption. In those cases, it is often useful to disable pipeline caching so the viewer does not use as much memory. When the viewer does not cache pipelines, each plot for a timestep must be recalculated each time the timestep is visited.

Example:

```
#% visit -cli
SetPipelineCachingMode(0) # Disable caching
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
AddPlot("Mesh", "quadmesh")
DrawPlots()
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
```

5.4.236 SetPlotDatabaseState

Synopsis:

```
SetPlotDatabaseState(index, frame, state)
```

index [integer] A zero-based integer index that is the plot's location in the plot list.

frame [integer] A zero-based integer index representing the animation frame for which we're going to add a database keyframe.

state [integer] A zero-based integer index representing the database time state that we're going to use at the specified animation frame.

Description:

The SetPlotDatabaseState function is used when VisIt is in keyframing mode to add a database keyframe for a specific plot. VisIt uses database keyframes to determine which database state is to be used for a given animation frame. Database keyframes can be used to stop "database time" while "animation time" continues forward and they can also be used to make "database time" go in reverse, etc.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "pressure")
AddPlot("Mesh", "quadmesh")
DrawPlots()
# Make "database time" for the Pseudocolor plot go in reverse
SetPlotDatabaseState(0, 0, 70)
SetPlotDatabaseState(0, nFrames-1, 0)
# Animate through the animation frames since the "Keyframe animation"
# time slider is active.
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
```

5.4.237 SetPlotDescription

Synopsis:

```
SetPlotDescription(index, description) -> integer
```

index [integer] The integer index of the plot within the plot list.

description [list] A new description srting that will be shown in the plot list so the plot can be identified readily.

return type [CLI_return_t] The SetPlotDescription function returns 1 on success and 0 on failure.

Description:

Managing many related plots can be a complex task. This function lets users provide meaningful descriptions for each plot so they can more easily be identified in the plot list.

Example:

```
SetPlotDescription(0, 'Mesh for reflected pressure plot')
```

5.4.238 SetPlotFollowsTime

Synopsis:

```
SetPlotFollowsTime(val) -> integer
```

val [integer] An optional integer flag indicating whether the plot should follow the time slider. The default behavior is for the plot to follow the time slider.

return type [CLI_return_t] The function returns 1 on success and 0 on failure.

Description:

SetPlotFollowsTime can let you set whether the active plot follows the time slider.

Example:

```
SetPlotFollowsTime()
```

5.4.239 SetPlotFrameRange

Synopsis:

```
SetPlotFrameRange(index, start, end)
```

index [integer] A zero-based integer representing an index into the plot list.

start [integer] A zero-based integer representing the animation frame where the plot first appears in the visualization.

end [integer] A zero-based integer representing the animation frame where the plot disappears from the visualization.

Description:

The SetPlotFrameRange function sets the start and end frames for a plot when VisIt is in keyframing mode. Outside of this frame range, the plot does not appear in the visualization. By default, plots are valid over the entire range of animation frames when they are first created. Frame ranges allow you to construct complex animations where plots appear and disappear dynamically.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
k = GetKeyframeAttributes()
nFrames = 20
```

(continues on next page)

(continued from previous page)

```
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "pressure")
AddPlot("Mesh", "quadmesh")
DrawPlots()
# Make the Pseudocolor plot take up the first half of the animation frames
# before it disappears.
SetPlotFrameRange(0, 0, nFrames/2-1)
# Make the Mesh plot take up the second half of the animation frames.
SetPlotFrameRange(1, nFrames/2, nFrames-1)
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
    SaveWindow()
```

5.4.240 SetPlotOptions

Synopsis:

```
SetPlotOptions(atts) -> integer
```

atts [plot attributes object] Any type of plot attributes object.

return type [CLI_return_t] All functions return an integer value of 1 for success and 0 for failure.

Description:

Each plot in VisIt has a group of attributes that controls the appearance of the plot. To set the attributes for a plot, first create a plot attributes object. This is done by calling a function which is the name of the plot plus the word "Attributes". For example, a Pseudocolor plot's plotattributes object is created and returned by the PseudocolorAttributes function. Assign the new plot attributes object into a variable and set its fields. After setting the desired fields in the plot attributes object, pass the object to the SetPlotOptions function. The SetPlotOptions function determines the type of plot to which the plot attributes object applies and sets the attributes for that plot type. To set the default plot attributes, use the SetDefaultPlotOptions function. Setting the default attributes ensures that all future instances of a certain plot are initialized with the new default values.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
p = PseudocolorAttributes()
p.colorTableName = "calewhite"
p.minFlag,p.maxFlag = 1,1
p.min,p.max = -5.0, 8.0
SetPlotOptions(p)
DrawPlots()
```

5.4.241 SetPlotOrderToFirst

Synopsis:

```
SetPlotOrderToFirst(index) -> integer
```

index [integer] The integer index of the plot within the plot list.

return type [CLI_return_t] The SetPlotOrderToFirst function returns 1 on success and 0 on failure.

Description:

Move the i'th plot in the plot list to the start of the plot list.

Example:

```
AddPlot('Mesh', 'mesh')
AddPlot('Pseudocolor', 'pressure')
# Make the Pseudocolor plot first in the plot list
SetPlotOrderToFirst(1)
```

5.4.242 SetPlotOrderToLast

Synopsis:

```
SetPlotOrderToLast(index) -> integer
```

index [integer] The integer index of the plot within the plot list.

return type [CLI_return_t] The SetPlotOrderToLast function returns 1 on success and 0 on failure.

Description:

Move the i'th plot in the plot list to the end of the plot list.

Example:

```
AddPlot('Mesh', 'mesh')
AddPlot('Pseudocolor', 'pressure')
# Make the Mesh plot last in the plot list
SetPlotOrderToLast(0)
```

5.4.243 SetPlotSILRestriction

Synopsis:

```
SetPlotSILRestriction(silr) -> integer
SetPlotSILRestriction(silr, all) -> integer
```

silr [SIL restriction object] A SIL restriction object.

all [integer] An optional argument that tells the function if the SIL restriction should be applied to all plots in the plot list (set all = 1) or not (set all = 0).

return type [CLI_return_t] The SetPlotSILRestriction function returns an integer value of 1 for success and 0 for failure.

Description:

VisIt allows the user to select subsets of databases. The description of the subset is called a Subset Inclusion Lattice Restriction, or SIL restriction. The SIL restriction allows databases to be subselected in several different ways. The VisIt Python Interface provides the SetPlotSILRestriction function to allow Python scripts to turn off portions of the plotted database. The SetPlotSILRestriction function accepts a SILRestriction object that contains the SIL restriction for the active plots. The optional all argument is an integer that tells the function to apply the SIL restriction to all plots when the value of the argument is non-zero. If the all argument is not supplied, then the SIL restriction is only applied to the active plots.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/multi_curv2d.silo")
AddPlot("Subset", "mat1")
silr = SILRestriction()
silr.TurnOffSet(silr.SetsInCategory('mat1')[1])
SetPlotSILRestriction(silr)
DrawPlots()
```

5.4.244 SetPrecisionType

Synopsis:

```
SetPrecisionType(typeAsInt)
SetPrecisionType(typeAsString)
```

typeAsInt [integer] Precision type specified as an integer. Options are 0 for Float, 1 for Native, and 2 for Double. The default is 1.

typeAsString [string] Precision type specified as a string. Options are "Float", "Native", and "Double". The default option is "Native."

Description:

The SetPrecisionType function sets the floating point precision used by VisIt's pipeline. The function accepts a single argument either an integer or string representing the precision desired. 0 = "float", 1 = "native", 2 = "double"

Example:

```
SetPrecisionType("double")
SetPrecisionType(2)
```

5.4.245 SetPreferredFileFormats

Synopsis:

```
SetPreferredFileFormats(pluginIDs) -> integer
```

pluginIDs [tuple] A tuple of plugin IDs to be attempted first when opening files.

return type [CLI_return_t] The SetPreferredFileFormats method does not return a value.

Description:

The SetPreferredFileFormats method is a way to set the list of file format reader plugins which are tried before any others. These IDs must be full IDs, not just names, and are tried in order.

Example:

```
SetPreferredFileFormats('Silo_1.0')
SetPreferredFileFormats(('Silo_1.0','PDB_1.0'))
```

5.4.246 SetPrinterAttributes

Synopsis:

```
SetPrinterAttributes(atts)
```

atts [PrinterAttributes object] A PrinterAttributes object.

Description:

The SetPrinterAttributes function sets the printer attributes. VisIt uses the printer attributes to determine how the active visualization window should be printed. The function accepts a single argument which is a PrinterAttributes object containing the printer attributes to use for future printing. VisIt allows images to be printed to a network printer or to a PostScript file that can be printed later by other applications.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Surface", "v")
DrawPlots()
# Make it print to a file.
p = PrinterAttributes()
p.outputToFile = 1
p.outputToFileName = "printfile"
SetPrinterAttributes(p)
PrintWindow()
```

5.4.247 SetQueryFloatFormat

Synopsis:

```
SetQueryFloatFormat(format_string)
```

format_string [string] A string object that provides a printf style floating point format.

Description:

The SetQueryFloatFormat method sets a printf style format string that isused by VisIt's querys to produce textual output.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Set floating point format string.
SetQueryFloatFormat("%.1f")
Query("MinMax")
# Set format back to default "%g".
SetQueryFloatFormat("%g")
Query("MinMax")
```

5.4.248 SetQueryOutputToObject

Synopsis:

```
SetQueryOutputToObject()
```

Description:

SetQueryOutputToObject changes the return type of future Queries to the 'object' or Python dictionary form. This is the same object that would be returned by calling 'GetQueryOutputObject()' after a Query call. All other output modes are still available after the Query call (eg GetQueryOutputValue(),GetQueryOutputObject(), GetQueryOutputString()).

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Set query output type.
SetQueryOutputToObject()
query_output = Query("MinMax")
print(query_output)
```

5.4.249 SetQueryOutputToString

Synopsis:

```
SetQueryOutputToString()
```

Description:

SetQueryOutputToString changes the return type of future Queries to the 'string' form. This is the same as what would be returned by calling 'GetQueryOutputString' after a Query call. All other output modes are still available after the Query call (eg GetQueryOutputValue(), GetQueryOutputObject(), GetQueryOutputString()).

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Set query output type.
SetQueryOutputToString()
query_output = Query("MinMax")
print(query_output)
'''
d -- Min = 0.0235702 (zone 434 at coord <0.483333, 0.483333>)
d -- Max = 0.948976 (zone 1170 at coord <0.0166667, 1.31667>)
'''
```

5.4.250 SetQueryOutputToValue

Synopsis:

```
SetQueryOutputToValue()
```

Description:

SetQueryOutputToValue changes the return type of future Queries to the 'value' form. This is the same as what would be returned by calling 'GetQueryOutputValue()' after a Query call. All other output modes are still available after the Query call (eg GetQueryOutputValue(), GetQueryOutputObject(), GetQueryOutputString()).

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Set query output type.
SetQueryOutputToValue()
query_output = Query("MinMax")
print(query_output)
(0.02357020415365696, 0.9489759802818298)
```

5.4.251 SetQueryOverTimeAttributes

Synopsis:

```
SetQueryOverTimeAttributes(atts) -> integer
```

atts [QueryOverTimeAttributes object] A QueryOverTimeAttributes object containing the new settings to use for queries over time.

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The SetQueryOverTimeAttributes function changes the settings that VisIt uses for query over time. The SetDefaultQueryOverTimeAttributes function changes the settings that new visualization windows inherit for doing query over time. Finally, the ResetQueryOverTimeAttributes function forces VisIt to use the stored default query over time attributes instead of the previous settings.

Example:

```
#% visit -cli
SetWindowLayout(4)
OpenDatabase("/usr/gapps/visit/data/allinone00.pdb")
AddPlot("Pseudocolor", "mesh/mixvar")
DrawPlots()
qot = GetQueryOverTimeAttributes()
# Make queries over time go to window 4.
qot.createWindow,q.windowId = 0, 4
SetQueryOverTimeAttributes(qot)
QueryOverTime("Min")
# Make queries over time only use half of the number of time states.
qot.endTimeFlag,qot.endTime = 1, GetDatabaseNStates() / 2
SetQueryOverTimeAttributes(qot)
QueryOverTime("Min")
ResetView()
```

5.4.252 SetRemoveDuplicateNodes

Synopsis:

```
SetRemoveDuplicateNodes(val) -> integer
```

val [integer] Either a zero (false) or non-zero (true) integer value to indicate if duplicate nodes in fully disconnected unstructured grids should be automatically removed by visit.

return type [CLI_return_t] The SetRemoveDuplicateNodes function returns 1 on success and 0 on failure.

Description:

The SetRemoveDuplicateNodes function sets a boolean in the global attributes indicating whether or not duplicate nodes in fully disconnected unstructured grids should be automatically removed. The default behavior is for the original grid to be left as read, which may slow down VisIt's performance for extraordinary large meshes. Turning this feature off tells VisIt to remove the duplicate nodes after the mesh is read, but before further processing in VisIt.

Example:

```
#% visit -cli
SetRemoveDuplicateNodes(1) # turn this feature on
SetRemoveDuplicateNodes(0) # turn this feature off
```

5.4.253 SetRenderingAttributes

Synopsis:

```
SetRenderingAttributes(atts) -> integer
```

atts [RenderingAttributes object] A RenderingAttributes object that contains the rendering attributes that we want to make VisIt use.

return type [CLI_return_t] The SetRenderingAttributes function returns 1 on success and 0 on failure.

Description:

The SetRenderingAttributes makes VisIt use the rendering attributes stored in the specified RenderingAttributes object. The RenderingAttributes object stores rendering attributes such as: scalable rendering options, shadows, specular highlights, display lists, etc.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Surface", "hgslice")
DrawPlots()
v = GetView2D()
v.viewNormal = (-0.215934, -0.454611, 0.864119)
v.viewUp = (0.973938, -0.163188, 0.157523)
v.imageZoom = 1.64765
SetView3D(v)
light = GetLight(0)
light.direction = (0,1,-1)
SetLight(0, light)
r = GetRenderingAttributes()
print(r)
r.scalableActivationMode = r.Always
r.doShadowing = 1
SetRenderingAttributes(r)
```

5.4.254 SetSaveWindowAttributes

Synopsis:

```
SetSaveWindowAttributes(atts)
```

atts [SaveWindowAttributes object] A SaveWindowAttributes object.

Description:

The SetSaveWindowAttributes function sets the format and filename that are used to save windows when the SaveWindow function is called. The contents of the active visualization window can be saved as TIFF, JPEG, RGB, PPM, PNG images or they can be saved as curve, Alias Wavefront Obj, or VTK geometry files. To set the SaveWindowAttributes, create a SaveWindowAttributes object using the SaveWindowAttributes function and assign it into a variable. Set the fields in the object and pass it to the SetSaveWindowAttributes function.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv3d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Set the save window attributes
s = SaveWindowAttributes()
s.fileName = "test"
s.format = s.JPEG
s.progressive = 1
s.fileName = "test"
SetSaveWindowAttributes(s)
# Save the window
SaveWindow()
```

5.4.255 SetTimeSliderState

Synopsis:

```
SetTimeSliderState(state) -> integer
```

state [integer] A zero-based integer containing the time state that we want to make active.

return type [CLI_return_t] The SetTimeSliderState function returns 1 on success and 0 on failure.

Description:

The SetTimeSliderState function sets the time state for the active time slider. This is the function to use if you want to animate through time or change the current keyframe frame.

Example:

```
#% visit -cli
path = "/usr/gapps/visit/data/"
dbs = (path + "dbA00.pdb", path + "dbB00.pdb", path + "dbC00.pdb")
for db in dbs:
    OpenDatabase(db)
    AddPlot("FilledBoundary", "material(mesh)")
    DrawPlots()
CreateDatabaseCorrelation("common", dbs, 1)
```

(continues on next page)

(continued from previous page)

```
tsNames = GetWindowInformation().timeSliders
for ts in tsNames:
    SetActiveTimeSlider(ts)
for state in list(range(TimeSliderGetNStates())) + [0]:
    SetTimeSliderState(state)
```

5.4.256 SetTreatAllDBsAsTimeVarying

Synopsis:

```
SetTreatAllDBsAsTimeVarying(val) -> integer
```

val [integer] Either a zero (false) or non-zero (true) integer value to indicate if all databases should be treated as time varying (true) or not (false).

return type [CLI_return_t] The SetTreatAllDBsAsTimeVarying function returns 1 on success and 0 on failure.

Description:

The SetTreatAllDBsAsTimeVarying function sets a boolean in the global attributes indicating if all databases should be treated as time varying or not. Ordinarily, VisIt tries to minimize file I/O and database interaction by avoiding re-reading metadata that is 'time-invariant' and, therefore, assumed to be the same in a database from one time step to the next. However, sometimes, portions of the metadata, such as the list of variable names and/or number of domains, does in fact vary. In this case, VisIt can actually fail to acknowledge the existence of new variables in the file. Turning this feature on forces VisIt to re-read metadata each time the time-state is changed.

Example:

```
#% visit -cli
SetTreatAllDBsAsTimeVarying(1) # turn this feature on
SetTreatAllDBsAsTimeVarying(0) # turn this feature off
```

5.4.257 SetTryHarderCyclesTimes

Synopsis:

```
SetTryHarderCyclesTimes(val) -> integer
```

val [integer] Either a zero (false) or non-zero (true) integer value to indicate if VisIt read cycle/time information for all timestates when opening a database.

return type [CLI_return_t] The SetTryHarderCyclesTimes function returns 1 on success and 0 on failure.

Description:

For certain classes of databases, obtaining cycle/time information for all time states in the database is an expensive operation, requiring each file to be opened and queried. The cost of the operation gets worse the more time states there are in the database. Ordinarily, VisIt does not bother to query each time state for precise cycle/time information. In fact, often VisIt can guess this information from the filename(s) comprising the databse. However, turning this feature on will force VisIt to obtain accurate cycle/time information for all time states by opening and querying all file(s) in the database.

Example:

```
#% visit -cli
SetTryHarderCyclesTimes(1) # Turn this feature on
SetTryHarderCyclesTimes(0) # Turn this feature off
```

5.4.258 SetUltraScript

Synopsis:

```
SetUltraScript(filename) -> integer
```

filename [string] The name of the file to be used as the ultra script when LoadUltra is called.

return type [CLI_return_t] The SetUltraScript function returns 1.

Description:

Set the path to the script to be used by the LoadUltra command. Normal users do not need to use this function.

5.4.259 SetView2D

Synopsis:

```
SetView2D(View2DAttributes) -> integer
```

view [ViewAttributes object] A ViewAttributes object containing the view.

return type [CLI_return_t] All functions returns 1 on success and 0 on failure.

Description:

The view is a crucial part of a visualization since it determines which parts of the database are examined. The VisIt Python Interface provides four functions for setting the view: SetView2D, SetView3D, SetViewCurve, and SetViewAxisArray. If the visualization window contains 2D plots, use the SetView2D function. To set the view, first create the appropriate ViewAttributes object and set the object's fields to set a new view. After setting the fields, pass the object to the matching SetView function. A common use of the SetView functions is to animate the view to produce simple animations where the camera appears to fly around the plots in the visualization window.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "v")
DrawPlots()
va = GetView3D()
va.RotateAxis(1,30.0) # rotate around the y axis 30 degrees.
SetView3D(va)
v0 = GetView3D()
v1 = GetView3D()
v1.camera,v1.viewUp = (1,1,1),(-1,1,-1)
v1.parallelScale = 10.
for i in range(0,20):
    t = float(i) / 19.
    v2 = (1. - t) * v0 + t * v1
    SetView3D(v2) # Animate the view.
```

5.4.260 SetView3D

Synopsis:

```
SetView3D(View3DAttributes) -> integer
```

view [ViewAttributes object] A ViewAttributes object containing the view.

return type [CLI_return_t] All functions returns 1 on success and 0 on failure.

Description:

The view is a crucial part of a visualization since it determines which parts of the database are examined. The VisIt Python Interface provides four functions for setting the view: SetView2D, SetView3D, SetViewCurve, and SetViewAxisArray. Use the SetView3D function when the visualization window contains 3D plots. To set the view, first create the appropriate ViewAttributes object and set the object's fields to set a new view. After setting the fields, pass the object to the matching SetView function. A common use of the SetView functions is to animate the view to produce simple animations where the camera appears to fly around the plots in the visualization window. A View3D object also supports the RotateAxis(int axis, double deg) method which mimics the 'rotx', 'roty' and 'rotz' view commands in the GUI.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "v")
DrawPlots()
va = GetView3D()
va.RotateAxis(1,30.0) # rotate around the y axis 30 degrees.
SetView3D(va)
v0 = GetView3D()
v1 = GetView3D()
v1.camera,v1.viewUp = (1,1,1),(-1,1,-1)
v1.parallelScale = 10.
for i in range(0,20):
    t = float(i) / 19.
    v2 = (1. - t) * v0 + t * v1
    SetView3D(v2) # Animate the view.
```

5.4.261 SetViewAxisArray

Synopsis:

```
SetViewAxisArray(ViewAxisArrayAttributes) -> integer
```

view [ViewAttributes object] A ViewAttributes object containing the view.

return type [CLI_return_t] All functions returns 1 on success and 0 on failure.

Description:

The view is a crucial part of a visualization since it determines which parts of the database are examined. The VisIt Python Interface provides four functions for setting the view: SetView2D, SetView3D, SetViewCurve, and SetViewAxisArray. To set the view, first create the appropriate ViewAttributes object and set the object's fields to set a new view. After setting the fields, pass the object to the matching SetView function. A common use of the SetView functions is to animate the view to produce simple animations where the camera appears to fly around the plots in the visualization window.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "v")
DrawPlots()
va = GetView3D()
va.RotateAxis(1,30.0) # rotate around the y axis 30 degrees.
SetView3D(va)
v0 = GetView3D()
v1 = GetView3D()
v1.camera,v1.viewUp = (1,1,1),(-1,1,-1)
v1.parallelScale = 10.
for i in range(0,20):
    t = float(i) / 19.
    v2 = (1. - t) * v0 + t * v1
    SetView3D(v2) # Animate the view.
```

5.4.262 SetViewCurve

Synopsis:

```
SetViewCurve(ViewCurveAttributes) -> integer
```

view [ViewAttributes object] A ViewAttributes object containing the view.

return type [CLI_return_t] All functions returns 1 on success and 0 on failure.

Description:

The view is a crucial part of a visualization since it determines which parts of the database are examined. The VisIt Python Interface provides four functions for setting the view: SetView2D, SetView3D, SetViewCurve, and SetViewAxisArray. To set the view, first create the appropriate ViewAttributes object and set the object's fields to set a new view. After setting the fields, pass the object to the matching SetView function. A common use of the SetView functions is to animate the view to produce simple animations where the camera appears to fly around the plots in the visualization window.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "v")
DrawPlots()
va = GetView3D()
va.RotateAxis(1,30.0) # rotate around the y axis 30 degrees.
SetView3D(va)
v0 = GetView3D()
v1 = GetView3D()
v1.camera,v1.viewUp = (1,1,1),(-1,1,-1)
v1.parallelScale = 10.
for i in range(0,20):
    t = float(i) / 19.
    v2 = (1. - t) * v0 + t * v1
    SetView3D(v2) # Animate the view.
```

5.4.263 SetViewExtentsType

Synopsis:

```
SetViewExtentsType(type) -> integer
```

type [integer] An integer or a string. Options are 0, 1 and 'original', 'actual', respectively.

return type [CLI_return_t] SetViewExtentsType returns 1 on success and 0 on failure.

Description:

VisIt can use a plot's spatial extents in two ways when computing the view. The first way of using the extents is to use the "original" extents, which are the spatial extents before any modifications, such as subset selection, have been made to the plot. This ensures that the view will remain relatively constant for a plot. Alternatively, you can use the "actual" extents, which are the spatial extents of the pieces of the plot that remain after operations such as subset selection.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
SetViewExtentsType("actual")
AddPlot("FilledBoundary", "mat1")
DrawPlots()
v = GetView3D()
v.viewNormal = (-0.618945, 0.450655, 0.643286)
v.viewUp = (0.276106, 0.891586, -0.358943)
SetView3D(v)
mats = GetMaterials()
nmats = len(mats)
# Turn off all but the last material in sequence and watch
# the view update each time.
for i in range(nmats-1):
    index = nmats-1-i
    TurnMaterialsOff(mats[index])
    SaveWindow()
    SetViewExtentsType("original")
```

5.4.264 SetViewKeyframe

Synopsis:

```
SetViewKeyframe() -> integer
```

return type [CLI_return_t] The SetViewKeyframe function returns 1 on success and 0 on failure.

Description:

The SetViewKeyframe function adds a view keyframe when VisIt is in keyframing mode. View keyframes are used to set the view at crucial points during an animation. Frames that lie between view keyframes have an interpolated view that is based on the view keyframes. You can use the SetViewKeyframe function to create complex camera animations that allow you to fly around (or through) your visualization.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Contour", "hardyglobal")
DrawPlots()
k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
SetPlotFrameRange(0, 0, nFrames-1)
SetViewKeyframe()
SetTimeSliderState(10)
v = GetView3D()
v.viewNormal = (-0.721721, 0.40829, 0.558944)
v.viewUp = (0.294696, 0.911913, -0.285604)
SetView3D(v)
SetViewKeyframe()
SetTimeSliderState(nFrames-1)
v.viewNormal = (-0.74872, 0.423588, -0.509894)
v.viewUp = (0.369095, 0.905328, 0.210117)
SetView3D()
SetViewKeyframe()
ToggleCameraViewMode()
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
    SaveWindow()
```

5.4.265 SetWindowArea

Synopsis:

```
SetWindowArea(x, y, width, height) -> integer
```

x [integer] An integer that is the left X coordinate in screen pixels.

y [integer] An integer that is the top Y coordinate in screen pixels.

width [integer] An integer that is the width of the window area in pixels.

height [integer] An integer that is the height of the window area in pixels.

return type [CLI_return_t] The SetWindowArea function returns 1 on success and 0 on failure.

Description:

The SetWindowArea method sets the area of the screen that can be used by VisIt's visualization windows. This is useful for making sure windows are a certain size when running a Python script.

Example:

```
import visit
visit.Launch()
visit.SetWindowArea(0, 0, 600, 600)
visit.SetWindowLayout(4)
```

5.4.266 SetWindowLayout

Synopsis:

```
SetWindowLayout(layout) -> integer
```

layout [integer] An integer that specifies the window layout. (1,2,4,8,9,16 are valid)

return type [CLI_return_t] The SetWindowLayout function returns an integer value of 1 for success and 0 for failure.

Description:

VisIt's visualization windows can be arranged in various tiled patterns that allow VisIt to make good use of the screen while displaying several visualization windows. The window layout determines how windows are shown on the screen. The SetWindowLayout function sets the window layout. The layout argument is an integer value equal to 1,2,4,8,9, or 16.

Example:

```
#% visit -cli
SetWindowLayout(2) # switch to 1x2 layout
SetWindowLayout(4) # switch to 2x2 layout
SetWindowLayout(8) # switch to 2x4 layout
```

5.4.267 SetWindowMode

Synopsis:

```
SetWindowMode(mode) -> integer
```

mode [string] A string containing the new mode. Options are 'navigate', 'zoom', 'lineout', 'pick', 'zone pick', 'node pick', 'spreadsheet pick'.

return type [CLI_return_t] The SetWindowMode function returns 1 on success and 0 on failure.

Description:

VisIt's visualization windows have various window modes that alter their behavior. Most of the time a visualization window is in "navigate" mode which changes the view when the mouse is moved in the window. The "zoom" mode allows a zoom rectangle to be drawn in the window for changing the view. The "pick" mode retrieves information about the plots when the mouse is clicked in the window. The "lineout" mode allows the user to draw lines which produce curve plots.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
SetWindowMode("zoom")
# Draw a rectangle in the visualization window to zoom the plots
```

5.4.268 ShowAllWindows

Synopsis:

```
ShowAllWindows() -> integer
```

return type [CLI_return_t] The ShowAllWindows function returns 1 on success and 0 on failure.

Description:

The ShowAllWindows function tells VisIt's viewer to show all of its visualization windows. The command line interface calls ShowAllWindows before giving control to any user-supplied script to ensure that the visualization windows appear as expected. Call the ShowAllWindows function when using the VisIt module inside another Python interpreter so the visualization windows are made visible.

Example:

```
#% python
import visit
visit.Launch()
visit.ShowAllWindows()
```

5.4.269 ShowToolbars

Synopsis:

```
ShowToolbars() -> integer
ShowToolbars(allWindows) -> integer
```

allWindows [integer] An integer value that tells VisIt to show the toolbars for all windows when it is non-zero.

return type [CLI_return_t] The ShowToolbars function returns 1 on success and 0 on failure.

Description:

The ShowToolbars function tells VisIt to show the toolbars for the active visualization window or for all visualization windows when the optional allWindows argument is provided and is set to a non-zero value.

Example:

```
#% visit -cli
SetWindowLayout(4)
HideToolbars(1)
ShowToolbars()
# Show the toolbars for all windows.
ShowToolbars(1)
```

5.4.270 Source

Synopsis:

```
Source(filename)
```

Description:

The Source function reads in the contents of a text file and interprets it with the Python interpreter. This is a simple mechanism that allows simple scripts to be included in larger scripts. The Source function takes a single string argument that contains the name of the script to execute.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# include another script that does some animation.
Source("Animate.py")
```

5.4.271 SuppressMessages

Synopsis:

```
SuppressMessages(level) -> integer
```

level [integer] An integer value of 1,2,3 or 4

return type [CLI_return_t] The SuppressMessages function returns the previous suppression level on success and 0 on failure.

Description:

The SuppressMessage function sets the suppression level for status messages generated by VisIt. A value of 1 suppresses all types of messages. A value of 2 suppresses Warnings and Messages but does NOT suppress Errors. A value of 3 suppresses Messages but does not suppress Warnings or Errors. A value of 4 does not suppress any messages. The default setting is 4.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Turn off Warning and Message messages.
SuppressMessages(2)
SaveWindow()
```

5.4.272 SuppressQueryOutputOff

Synopsis:

```
SuppressQueryOutputOff() -> integer
```

return type [CLI_return_t] The SuppressQueryOutput function returns 1 on success and 0 on failure.

Description:

The SuppressQueryOutput function tells VisIt to turn on/off the automatic printing of query output. Query output will still be available via GetQueryOutputString and GetQueryOutputValue.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Turn off automatic printing of Query output.
SuppressQueryOutputOn()
Query("MinMax")
print("The min is: %g and the max is: %g" % GetQueryOutputValue())
# Turn on automatic printing of Query output.
SuppressQueryOutputOff()
Query("MinMax")
```

5.4.273 SuppressQueryOutputOn

Synopsis:

```
SuppressQueryOutputOn() -> integer
```

return type [CLI_return_t] The SuppressQueryOutput function returns 1 on success and 0 on failure.

Description:

The SuppressQueryOutput function tells VisIt to turn on/off the automatic printing of query output. Query output will still be available via GetQueryOutputString and GetQueryOutputValue.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Turn off automatic printing of Query output.
SuppressQueryOutputOn()
Query("MinMax")
print("The min is: %g and the max is: %g" % GetQueryOutputValue())
# Turn on automatic printing of Query output.
SuppressQueryOutputOff()
Query("MinMax")
```

5.4.274 TimeSliderGetNStates

Synopsis:

```
TimeSliderGetNStates() -> integer
```

return type [CLI_return_t] Returns an integer containing the number of time states for the current time slider.

Description:

The TimeSliderGetNStates function returns the number of time states for the active time slider. Remember that the length of the time slider does not have to be equal to the number of time states in a time-varying database because of database correlations and keyframing. If you want to iterate through time, use this function to determine the number of iterations that are required to reach the end of the active time slider.

Example:

```
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
for state in range(TimeSliderGetNStates()):
    SetTimeSliderState(state)
    SaveWindow()
```

5.4.275 TimeSliderNextState

Synopsis:

```
TimeSliderNextState() -> integer
```

return type [CLI_return_t] The TimeSliderNextState function returns 1 on success and 0 on failure.

Description:

The TimeSliderNextState function advances the active time slider to the next time slider state.

Example:

```
# Assume that files are being written to the disk.
#% visit -cli
OpenDatabase("dynamic*.silo database")
AddPlot("Pseudocolor", "var")
AddPlot("Mesh", "mesh")
DrawPlots()
SetTimeSliderState(TimeSliderGetNStates() - 1)
while 1:
    SaveWindow()
    TimeSliderPreviousState()
```

5.4.276 TimeSliderPreviousState

Synopsis:

```
TimeSliderPreviousState() -> integer
```

return type [CLI_return_t] The TimeSliderPreviousState function returns 1 on success and 0 on failure.

Description:

The TimeSliderPreviousState function moves the active time slider to the previous time slider state.

Example:

```
# Assume that files are being written to the disk.
#% visit -cli
OpenDatabase("dynamic*.silo database")
AddPlot("Pseudocolor", "var")
AddPlot("Mesh", "mesh")
DrawPlots()
while 1:
    TimeSliderNextState()
    SaveWindow()
```

5.4.277 TimeSliderSetState

Synopsis:

```
TimeSliderSetState(state) -> integer
```

state [integer] A zero-based integer containing the time state that we want to make active.

return type [CLI_return_t] The TimeSliderSetState function returns 1 on success and 0 on failure.

Description:

The TimeSliderSetState function sets the time state for the active time slider. This is the function to use if you want to animate through time or change the current keyframe frame.

Example:

```
#% visit -cli
path = "/usr/gapps/visit/data/"
dbs = (path + "dbA00.pdb", path + "dbB00.pdb", path + "dbC00.pdb")
for db in dbs:
    OpenDatabase(db)
    AddPlot("FilledBoundary", "material(mesh)")
    DrawPlots()
CreateDatabaseCorrelation("common", dbs, 1)
tsNames = GetWindowInformation().timeSliders
for ts in tsNames:
    SetActiveTimeSlider(ts)
for state in list(range(TimeSliderGetNStates())) + [0]:
    TimeSliderSetState(state)
```

5.4.278 ToggleBoundingBoxMode

Synopsis:

```
ToggleBoundingBoxMode() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The visualization window has various modes that affect its behavior and the VisIt Python Interface provides a few functions to toggle some of those modes. The ToggleBoundingBoxMode function toggles bounding box mode on and off. When the visualization window is in bounding box mode, any plots it contains are hidden while the view is being changed so the window redraws faster.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Turn on spin mode.
ToggleSpinMode()
# Rotate the plot interactively using the mouse and watch it keep spinning
# after the mouse release.
# Turn off spin mode.
ToggleSpinMode()
```

5.4.279 ToggleCameraViewMode

Synopsis:

```
ToggleCameraViewMode() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The visualization window has various modes that affect its behavior and the VisIt Python Interface provides a few functions to toggle some of those modes. The ToggleCameraViewMode function toggles camera view mode on and off. When the visualization window is in camera view mode, the view is updated using any view keyframes that have been defined when VisIt is in keyframing mode.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Turn on spin mode.
ToggleSpinMode()
# Rotate the plot interactively using the mouse and watch it keep spinning
# after the mouse release.
# Turn off spin mode.
ToggleSpinMode()
```

5.4.280 ToggleFullFrameMode

Synopsis:

```
ToggleFullFrameMode() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The visualization window has various modes that affect its behavior and the VisIt Python Interface provides a few functions to toggle some of those modes. The ToggleFullFrameMode function toggles full-frame mode on and off. When the visualization window is in fullframe mode, the viewport is stretched non-uniformly so that it covers most of the visualization window. While not maintaining a 1:1 aspect ratio, it does make better use of the visualization window.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Turn on spin mode.
ToggleSpinMode()
# Rotate the plot interactively using the mouse and watch it keep spinning
# after the mouse release.
# Turn off spin mode.
ToggleSpinMode()
```

5.4.281 ToggleLockTime

Synopsis:

```
ToggleLockTime() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The visualization window has various modes that affect its behavior and the VisIt Python Interface provides a few functions to toggle some of those modes. The ToggleLockTime function turns time locking on and off in a visualization window. When time locking is on in a visualization window, VisIt creates a database correlation that works for the databases in all visualization windows that are time-locked. When

you change the time state using the time slider for the the afore-mentioned database correlation, it has the effect of updating time in all time-locked visualization windows.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Turn on spin mode.
ToggleSpinMode()
# Rotate the plot interactively using the mouse and watch it keep spinning
# after the mouse release.
# Turn off spin mode.
ToggleSpinMode()
```

5.4.282 ToggleLockTools

Synopsis:

```
ToggleBoundingBoxMode() -> integer
ToggleCameraViewMode() -> integer
ToggleFullFrameMode() -> integer
ToggleLockTime() -> integer
ToggleLockViewMode() -> integer
ToggleMaintainViewMode() -> integer
ToggleSpinMode() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The visualization window has various modes that affect its behavior and the VisIt Python Interface provides a few functions to toggle some of those modes. The ToggleBoundingBoxMode function toggles bounding box mode on and off. When the visualization window is in bounding box mode, any plots it contains are hidden while the view is being changed so the window redraws faster. The ToggleCameraViewMode function toggles camera view mode on and off. When the visualization window is in camera view mode, the view is updated using any view keyframes that have been defined when VisIt is in keyframing mode. The ToggleFullFrameMode function toggles fullframe mode on and off. When the visualization window is in fullframe mode, the viewport is stretched non-uniformly so that it covers most of the visualization window. While not maintaining a 1:1 aspect ratio, it does make better use of the visualization window. The ToggleLockTime function turns time locking on and off in a visualization window. When time locking is on in a visualization window, VisIt creates a database correlation that works for the databases in all visualization windows that are time-locked. When you change the time state using the time slider for the the afore-mentioned database correlation, it has the effect of updating time in all time-locked visualization windows. The ToggleLockViewMode function turns lock view mode on and off. When windows are in lock view mode, each view change is broadcast to other windows that are also in lock view mode. This allows windows containing similar plots to be compared easily. The ToggleMaintainViewMode function forces the view, that was in effect when the mode was toggled to be used for all subsequent time states. The ToggleSpinMode function turns spin mode on and off. When the visualization window is in spin mode, it continues to spin along the axis of rotation when the view is changed interactively.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Turn on spin mode.
ToggleSpinMode()
# Rotate the plot interactively using the mouse and watch it keep spinning
# after the mouse release.
# Turn off spin mode.
ToggleSpinMode()
```

5.4.283 ToggleLockViewMode

Synopsis:

```
ToggleLockViewMode() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The visualization window has various modes that affect its behavior and the VisIt Python Interface provides a few functions to toggle some of those modes. The ToggleLockViewMode function turns lock view mode on and off. When windows are in lock view mode, each view change is broadcast to other windows that are also in lock view mode. This allows windows containing similar plots to be compared easily.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Turn on spin mode.
ToggleSpinMode()
# Rotate the plot interactively using the mouse and watch it keep spinning
# after the mouse release.
# Turn off spin mode.
ToggleSpinMode()
```

5.4.284 ToggleMaintainViewMode

Synopsis:

```
ToggleMaintainViewMode() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The visualization window has various modes that affect its behavior and the VisIt Python Interface provides a few functions to toggle some of those modes. The ToggleMaintainViewMode functions forces the view that was in effect when the mode was toggled to be used for all subsequent time states.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Turn on spin mode.
ToggleSpinMode()
# Rotate the plot interactively using the mouse and watch it keep spinning
# after the mouse release.
# Turn off spin mode.
ToggleSpinMode()
```

5.4.285 ToggleSpinMode

Synopsis:

```
ToggleSpinMode() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

Description:

The visualization window has various modes that affect its behavior and the VisIt Python Interface provides a few functions to toggle some of those modes. The ToggleSpinMode function turns spin mode on and off. When the visualization window is in spin mode, it continues to spin along the axis of rotation when the view is changed interactively.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
# Turn on spin mode.
ToggleSpinMode()
# Rotate the plot interactively using the mouse and watch it keep spinning
# after the mouse release.
# Turn off spin mode.
ToggleSpinMode()
```

5.4.286 TurnDomainsOff

Synopsis:

```
TurnDomainsOff() -> integer
TurnDomainsOff(set_name) -> integer
TurnDomainsOff(tuple_set_name) -> integer
```

set_name [string] The name of the set to modify.

tuple_set_name [tuple of strings] A tuple of strings for the sets to modify.

return type [CLI_return_t] The Turn functions return an integer with a value of 1 for success or 0 for failure.

Description:

The TurnXXXOn|Off functions are provided to simplify the inclusion or exclusion of material or domain subsets. Instead of manipulating a SILRestriction object, you can use the TurnXXXOn|Off functions to

turn materials or domains on or off. The TurnXXXOn function turns materials or domains on. All of the TurnXXXOn|Off functions have three possible argument lists. When you do not provide any arguments, the function applies to all subsets. For example, TurnMaterialsOn() with no arguments, turns all materials on. All TurnXXXOn|Off functions can also take a single string as an argument, which is the name of the set to modify. All of the TurnXXXOn|Off functions can also be used to modify more than one set by providing a tuple of set names. After you use the TurnXXXOn|Off functions, it might be useful to call the ListMaterials or ListDomains functions to confirm the functions had the intended effect.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
TurnMaterialsOff("4") # Turn off material 4
TurnMaterialsOff(("1", "2")) # Turn off materials 1 and 2
TurnMaterialsOn() # Turn on all materials
```

5.4.287 TurnDomainsOn

Synopsis:

```
TurnDomainsOn() -> integer
TurnDomainsOn(set_name) -> integer
TurnDomainsOn(tuple_set_name) -> integer
```

set_name [string] The name of the set to modify.

tuple_set_name [tuple of strings] A tuple of strings for the sets to modify.

return type [CLI_return_t] The Turn functions return an integer with a value of 1 for success or 0 for failure.

Description:

The TurnXXXOn|Off functions are provided to simplify the inclusion or exclusion of material or domain subsets. Instead of manipulating a SILRestriction object, you can use the TurnXXXOn|Off functions to turn materials or domains on or off. The TurnXXXOn function turns materials or domains on. All of the TurnXXXOn|Off functions have three possible argument lists. When you do not provide any arguments, the function applies to all subsets. For example, TurnMaterialsOn() with no arguments, turns all materials on. All TurnXXXOn|Off functions can also take a single string as an argument, which is the name of the set to modify. All of the TurnXXXOn|Off functions can also be used to modify more than one set by providing a tuple of set names. After you use the TurnXXXOn|Off functions, it might be useful to call the ListMaterials or ListDomains functions to confirm the functions had the intended effect.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
TurnMaterialsOff("4") # Turn off material 4
TurnMaterialsOff(("1", "2")) # Turn off materials 1 and 2
TurnMaterialsOn() # Turn on all materials
```

5.4.288 TurnMaterialsOff

Synopsis:

```
TurnMaterialsOff() -> integer
TurnMaterialsOff(set_name) -> integer
TurnMaterialsOff(tuple_set_name) -> integer
```

set_name [string] The name of the set to modify.

tuple_set_name [tuple of strings] A tuple of strings for the sets to modify.

return type [CLI_return_t] The Turn functions return an integer with a value of 1 for success or 0 for failure.

Description:

The TurnXXXOn|Off functions are provided to simplify the inclusion or exclusion of material or domain subsets. Instead of manipulating a SILRestriction object, you can use the TurnXXXOn|Off functions to turn materials or domains on or off. The TurnXXXOn function turns materials or domains on. All of the TurnXXXOn|Off functions have three possible argument lists. When you do not provide any arguments, the function applies to all subsets. For example, TurnMaterialsOn() with no arguments, turns all materials on. All TurnXXXOn|Off functions can also take a single string as an argument, which is the name of the set to modify. All of the TurnXXXOn|Off functions can also be used to modify more than one set by providing a tuple of set names. After you use the TurnXXXOn|Off functions, it might be useful to call the ListMaterials or ListDomains functions to confirm the functions had the intended effect.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
TurnMaterialsOff("4") # Turn off material 4
TurnMaterialsOff(("1", "2")) # Turn off materials 1 and 2
TurnMaterialsOn() # Turn on all materials
```

5.4.289 TurnMaterialsOn

Synopsis:

```
TurnMaterialsOn() -> integer
TurnMaterialsOn(string) -> integer
TurnMaterialsOn(tuple of strings) -> integer
```

set_name [string] The name of the set to modify.

tuple_set_name [tuple of strings] A tuple of strings for the sets to modify.

return type [CLI_return_t] The Turn functions return an integer with a value of 1 for success or 0 for failure.

Description:

The TurnXXXOn|Off functions are provided to simplify the inclusion or exclusion of material or domain subsets. Instead of manipulating a SILRestriction object, you can use the TurnXXXOn|Off functions to turn materials or domains on or off. The TurnXXXOn function turns materials or domains on. All of the TurnXXXOn|Off functions have three possible argument lists. When you do not provide any arguments, the function applies to all subsets. For example, TurnMaterialsOn() with no arguments, turns all materials on. All TurnXXXOn|Off functions can also take a single string as an argument, which is the name of the set to modify. All of the TurnXXXOn|Off functions can also be used to modify more than one set by

providing a tuple of set names. After you use the TurnXXXOn|Off functions, it might be useful to call the ListMaterials or ListDomains functions to confirm the functions had the intended effect.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
TurnMaterialsOff("4") # Turn off material 4
TurnMaterialsOff(("1", "2")) # Turn off materials 1 and 2
TurnMaterialsOn() # Turn on all materials
```

5.4.290 UndoView

Synopsis:

```
UndoView()
```

Description:

When the view changes in the visualization window, it puts the old view on a stack of views. The UndoView function restores the view on top of the stack and removes it. This allows the user to undo up to ten view changes.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Subset", "mat1")
DrawPlots()
v = GetView2D()
v.windowCoords = (-2.3,2.4,0.2,4.9)
SetView2D(v)
UndoView()
```

5.4.291 UpdateNamedSelection

Synopsis:

```
UpdateNamedSelection(name) -> integer
UpdateNamedSelection(name, properties) -> integer
```

name [string] The name of the selection to update.

properties [SelectionProperties object] An optional SelectionProperties object that contains the selection properties to use when reevaluating the selection.

return type [CLI_return_t] The UpdateNamedSelection function returns 1 on success and 0 on failure.

Description:

This function causes VisIt to reevaluate a named selection using new selection properties. If no selection properties are provided then the selection will be reevaluated using data for the plot that was associated with the selection when it was created. This is useful if you want to change a plot in several ways before causing its associated named selection to update using the changes.

Example:

```
s = GetSelection('selection1')
s.selectionType = s.CumulativeQuerySelection
s.histogramType = s.HistogramMatches
s.combineRule = s.CombineOr
s.variables = ('temperature',)
s.variableMins = (2.9,)
s.variableMaxs = (3.1,)
UpdateNamedSelection('selection1', s)
```

5.4.292 Version

Synopsis:

```
Version() -> string
```

return type [string] The Version function return a string that represents VisIt's version.

Description:

The Version function returns a string that represents VisIt's version. The version string can be used in Python scripts to make sure that the VisIt module is a certain version before processing the rest of the Python script.

Example:

```
#% visit -cli
print("We are running VisIt version %s" % Version())
```

5.4.293 WriteConfigFile

Synopsis:

```
WriteConfigFile()
```

Description:

The viewer maintains internal settings which determine the default values for objects like plots and operators. The viewer can save out the default values so they can be used in future VisIt sessions. The WriteConfig function tells the viewer to write out the settings to the VisIt configuration file.

Example:

```
#% visit -cli
p = PseudocolorAttributes()
p.minFlag, p.min = 1, 5.0
p.maxFlag, p.max = 1, 20.0
SetDefaultPlotOptions(p)
# Save the new default Pseudocolor settings to the config file.
WriteConfig()
```

5.4.294 WriteScript

Synopsis:

```
WriteScript(f)
```

f [file] The python file object that will be written to.

Description:

`WriteScript()` saves the current state of VisIt as a Python script that can be used later to reproduce a visualization. This is like saving a session file. But, the output of WriteScript can be further customized. The resulting script will contain commands to set up plots in any visualization window that contained plots when WriteScript was called. It may be more verbose than necessary, so users may find it useful to delete portions of the script that are not needed. This will depend on how many plots there are or the complexity of the data. For example, it might useful to remove code related to setting a plot's SIL restriction. Once the script is edited to satisfaction, it can be replayed it in VisIt. See below.

Example:

```
#
# First, create the script.
#
#% visit -cli
OpenDatabase("foo.silo")
AddPlot("Pseudocolor","dx")
DrawPlots()
ChangeActivePlotsVar("dy")
WriteScript("plot_dx_and_dy.py")
#
# or
#
#% visit -cli
OpenDatabase("foo.silo")
AddPlot("Pseudocolor","dx")
DrawPlots()
ChangeActivePlotsVar("dy")
f = open("plot_dx_and_dy.py", "wt")
WriteScript(f)
f.close()
#
# Now run the script in a terminal to replay it in VisIt.
#
# visit -cli -s script.py
#
# Or, the script can be used with VisIt's movie making scripts as a
# basis to set up the initial visualization:
#
# visit -movie -format mpeg -geometry 800x800 -scriptfile script.py -output␣
↪scriptmovie
```

5.4.295 ZonePick

Synopsis:

```
ZonePick(namedarg1=arg1, namedarg2=arg2, ...) -> dictionary
```

coord [tuple] A tuple of doubles containing the spatial coordinate (x, y, z).

x [integer] An integer containing the screen X location (in pixels) offset from the left side of the visualization window.

y [integer] An integer containing the screen Y location (in pixels) offset from the bottom of the visualization window.

vars (optional) [tuple] A tuple of strings with the variable names for which to return results. Default is the currently plotted variable.

do_time (optional) [integer] An integer indicating whether to do a time pick. 1 -> do a time pick, 0 (default) -> do not do a time pick.

start_time (optional) [integer] An integer with the starting frame index. Default is 0.

end_time (optional) [integer] An integer with the ending frame index. Default is num_timestates-1.

stride (optional) [integer] An integer with the stride for advancing in time. Default is 1.

preserve_coord (optional) [integer] An integer indicating whether to pick an element or a coordinate. 0 -> used picked element (default), 1-> used picked coordinate. Note: enabling this option may substantially slow down the speed with which the query can be performed.

curve_plot_type (optional) [integer] An integer indicating whether the output should be on a single axis or with multiple axes. 0 -> single Y axis (default), 1 -> multiple Y Axes.

return type [dictionary] ZonePick returns a python dictionary of the pick results, unless do_time is specified, then a time curve is created in a new window. If the picked variable is node centered, the variable values are grouped according to incident node ids.

Description:

The ZonePick function prints pick information for the cell (a.k.a zone) that contains the specified point. The point can be specified as a 2D or 3D point in world space or it can be specified as a pixel location in screen space. If the point is specified as a pixel location then VisIt finds the zone that contains the intersection of a cell and a ray that is projected into the mesh. Once the zonal pick has been calculated, you can use the GetPickOutput function to retrieve the printed pick output as a string which can be used for other purposes.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()
# Perform zone pick in screen space
pick_out = ZonePick(x=200,y=200)
# Perform zone pick in world space.
pick_out = ZonePick(coord = (-5.0, 5.0, 0))
```

5.5 Attribute Reference

This chapter is an alphabetical listing of all the VisIt python *attribute* objects used to control VisIt's behavior. By and large, VisIt's python attribute objects resemble C++ classes with constructors, public members and setter/getter methods.

For example, for the attribute object controlling the **Coordinate Swap** operator, the function `CoordSwapAttributes()` serves as the constructor (or *instantiator*). The object it creates has members, `newCoord1`, `newCoord2`, and `newCoord3` each of which is an enumeration type that can take one of the three values, `Coord1`, `Coord2`, `Coord3`. In addition, it has setter/getter methods, `SetNewCoord1()`, `GetNewCoord1()` and so forth.

For each attribute object, the constructor function is given in italics followed by a table of the object's member names. The members themselves are not documented here, but in most cases their names are self explanatory. The setter/getter *methods* are not documented here either. When a member can take values only from a given list of options (e.g. an

enumeration), the default option is printed first in italic followed by a comma separated list of the other available options.

From within the running CLI, printing an attribute object with Python's `print()` method will list the members whereas diring an attribute object with Python's `dir()` method will list the setter/getter methods. For more information on finding things and getting help from within the running CLI, be sure to read *the section on apropos*.

Many of the **Plot** and **Operator** attribute methods accept an optional `1` argument to indicate whether or not to return the *default* (`0`) or *current* (`1`) attributes. For example, `CurveAttributes()` returns the default attributes for a **Curve** plot whereas `CurveAttributes(1)` returns the attributes of either the currently active **Curve** plot or the *first* **Curve** plot in the plot list regardless of whether it is selected or hidden.

Many functions return an integer where `1` means success and `0` means failure. This behavior is represented by the type `CLI_return_t` in an attempt to distinguish it from functions that may utilize the full range of integers.

5.5.1 AMRStitchCell: *AMRStitchCellAttributes()*

Attribute	**Default**/Allowed Values
CreateCellsOfType	**DualGridAndStitchCells**, DualGrid, StitchCells

5.5.2 Animation: *AnimationAttributes()*

Attribute	**Default**/Allowed Values
animationMode	**StopMode**, ReversePlayMode, PlayMode
pipelineCachingMode	0
frameIncrement	1
timeout	1
playbackMode	**PlayOnce**, Looping, Swing

5.5.3 Annotation: *AnnotationAttributes()*

Attribute	**Default**/Allowed Values
axes2D.visible	1
axes2D.autoSetTicks	1
axes2D.autoSetScaling	1
axes2D.lineWidth	0
axes2D.tickLocation	**Outside**, Inside, Both
axes2D.tickAxes	**BottomLeft**, Off, Bottom, Left, All
axes2D.xAxis.title.visible	1
axes2D.xAxis.title.font.font	**Courier**, Arial, Times
axes2D.xAxis.title.font.scale	1
axes2D.xAxis.title.font.useForegroundColor	1
axes2D.xAxis.title.font.color	(0, 0, 0, 255)
axes2D.xAxis.title.font.bold	1
axes2D.xAxis.title.font.italic	1
axes2D.xAxis.title.userTitle	0
axes2D.xAxis.title.userUnits	0
axes2D.xAxis.title.title	"X-Axis"

Continued on next page

Table 5.1 – continued from previous page

axes2D.xAxis.title.units	""
axes2D.xAxis.label.visible	1
axes2D.xAxis.label.font.font	**Courier**, Arial, Times
axes2D.xAxis.label.font.scale	1
axes2D.xAxis.label.font.useForegroundColor	1
axes2D.xAxis.label.font.color	(0, 0, 0, 255)
axes2D.xAxis.label.font.bold	1
axes2D.xAxis.label.font.italic	1
axes2D.xAxis.label.scaling	0
axes2D.xAxis.tickMarks.visible	1
axes2D.xAxis.tickMarks.majorMinimum	0
axes2D.xAxis.tickMarks.majorMaximum	1
axes2D.xAxis.tickMarks.minorSpacing	0.02
axes2D.xAxis.tickMarks.majorSpacing	0.2
axes2D.xAxis.grid	0
axes2D.yAxis.title.visible	1
axes2D.yAxis.title.font.font	**Courier**, Arial, Times
axes2D.yAxis.title.font.scale	1
axes2D.yAxis.title.font.useForegroundColor	1
axes2D.yAxis.title.font.color	(0, 0, 0, 255)
axes2D.yAxis.title.font.bold	1
axes2D.yAxis.title.font.italic	1
axes2D.yAxis.title.userTitle	0
axes2D.yAxis.title.userUnits	0
axes2D.yAxis.title.title	"Y-Axis"
axes2D.yAxis.title.units	""
axes2D.yAxis.label.visible	1
axes2D.yAxis.label.font.font	**Courier**, Arial, Times
axes2D.yAxis.label.font.scale	1
axes2D.yAxis.label.font.useForegroundColor	1
axes2D.yAxis.label.font.color	(0, 0, 0, 255)
axes2D.yAxis.label.font.bold	1
axes2D.yAxis.label.font.italic	1
axes2D.yAxis.label.scaling	0
axes2D.yAxis.tickMarks.visible	1
axes2D.yAxis.tickMarks.majorMinimum	0
axes2D.yAxis.tickMarks.majorMaximum	1
axes2D.yAxis.tickMarks.minorSpacing	0.02
axes2D.yAxis.tickMarks.majorSpacing	0.2
axes2D.yAxis.grid	0
axes3D.visible	1
axes3D.autoSetTicks	1
axes3D.autoSetScaling	1
axes3D.lineWidth	0
axes3D.tickLocation	**Inside**, Outside, Both
axes3D.axesType	**ClosestTriad**, FurthestTriad, OutsideEdges, StaticTriad, StaticEdges
axes3D.triadFlag	1
axes3D.bboxFlag	1
axes3D.xAxis.title.visible	1
axes3D.xAxis.title.font.font	**Arial**, Courier, Times

Continued on next page

Table 5.1 – continued from previous page

axes3D.xAxis.title.font.scale	1
axes3D.xAxis.title.font.useForegroundColor	1
axes3D.xAxis.title.font.color	(0, 0, 0, 255)
axes3D.xAxis.title.font.bold	0
axes3D.xAxis.title.font.italic	0
axes3D.xAxis.title.userTitle	0
axes3D.xAxis.title.userUnits	0
axes3D.xAxis.title.title	"X-Axis"
axes3D.xAxis.title.units	""
axes3D.xAxis.label.visible	1
axes3D.xAxis.label.font.font	**Arial**, Courier, Times
axes3D.xAxis.label.font.scale	1
axes3D.xAxis.label.font.useForegroundColor	1
axes3D.xAxis.label.font.color	(0, 0, 0, 255)
axes3D.xAxis.label.font.bold	0
axes3D.xAxis.label.font.italic	0
axes3D.xAxis.label.scaling	0
axes3D.xAxis.tickMarks.visible	1
axes3D.xAxis.tickMarks.majorMinimum	0
axes3D.xAxis.tickMarks.majorMaximum	1
axes3D.xAxis.tickMarks.minorSpacing	0.02
axes3D.xAxis.tickMarks.majorSpacing	0.2
axes3D.xAxis.grid	0
axes3D.yAxis.title.visible	1
axes3D.yAxis.title.font.font	**Arial**, Courier, Times
axes3D.yAxis.title.font.scale	1
axes3D.yAxis.title.font.useForegroundColor	1
axes3D.yAxis.title.font.color	(0, 0, 0, 255)
axes3D.yAxis.title.font.bold	0
axes3D.yAxis.title.font.italic	0
axes3D.yAxis.title.userTitle	0
axes3D.yAxis.title.userUnits	0
axes3D.yAxis.title.title	"Y-Axis"
axes3D.yAxis.title.units	""
axes3D.yAxis.label.visible	1
axes3D.yAxis.label.font.font	**Arial**, Courier, Times
axes3D.yAxis.label.font.scale	1
axes3D.yAxis.label.font.useForegroundColor	1
axes3D.yAxis.label.font.color	(0, 0, 0, 255)
axes3D.yAxis.label.font.bold	0
axes3D.yAxis.label.font.italic	0
axes3D.yAxis.label.scaling	0
axes3D.yAxis.tickMarks.visible	1
axes3D.yAxis.tickMarks.majorMinimum	0
axes3D.yAxis.tickMarks.majorMaximum	1
axes3D.yAxis.tickMarks.minorSpacing	0.02
axes3D.yAxis.tickMarks.majorSpacing	0.2
axes3D.yAxis.grid	0
axes3D.zAxis.title.visible	1
axes3D.zAxis.title.font.font	**Arial**, Courier, Times

Continued on next page

Table 5.1 – continued from previous page

axes3D.zAxis.title.font.scale	1
axes3D.zAxis.title.font.useForegroundColor	1
axes3D.zAxis.title.font.color	(0, 0, 0, 255)
axes3D.zAxis.title.font.bold	0
axes3D.zAxis.title.font.italic	0
axes3D.zAxis.title.userTitle	0
axes3D.zAxis.title.userUnits	0
axes3D.zAxis.title.title	"Z-Axis"
axes3D.zAxis.title.units	""
axes3D.zAxis.label.visible	1
axes3D.zAxis.label.font.font	**Arial**, Courier, Times
axes3D.zAxis.label.font.scale	1
axes3D.zAxis.label.font.useForegroundColor	1
axes3D.zAxis.label.font.color	(0, 0, 0, 255)
axes3D.zAxis.label.font.bold	0
axes3D.zAxis.label.font.italic	0
axes3D.zAxis.label.scaling	0
axes3D.zAxis.tickMarks.visible	1
axes3D.zAxis.tickMarks.majorMinimum	0
axes3D.zAxis.tickMarks.majorMaximum	1
axes3D.zAxis.tickMarks.minorSpacing	0.02
axes3D.zAxis.tickMarks.majorSpacing	0.2
axes3D.zAxis.grid	0
axes3D.setBBoxLocation	0
axes3D.bboxLocation	(0, 1, 0, 1, 0, 1)
axes3D.triadColor	(0, 0, 0)
axes3D.triadLineWidth	1
axes3D.triadFont	0
axes3D.triadBold	1
axes3D.triadItalic	1
axes3D.triadSetManually	0
userInfoFlag	1
userInfoFont.font	**Arial**, Courier, Times
userInfoFont.scale	1
userInfoFont.useForegroundColor	1
userInfoFont.color	(0, 0, 0, 255)
userInfoFont.bold	0
userInfoFont.italic	0
databaseInfoFlag	1
timeInfoFlag	1
databaseInfoFont.font	**Arial**, Courier, Times
databaseInfoFont.scale	1
databaseInfoFont.useForegroundColor	1
databaseInfoFont.color	(0, 0, 0, 255)
databaseInfoFont.bold	0
databaseInfoFont.italic	0
databaseInfoExpansionMode	**File**, Directory, Full, Smart, SmartDirectory
databaseInfoTimeScale	1
databaseInfoTimeOffset	0
legendInfoFlag	1

Continued on next page

Table 5.1 – continued from previous page

backgroundColor	(255, 255, 255, 255)
foregroundColor	(0, 0, 0, 255)
gradientBackgroundStyle	**Radial**, TopToBottom, BottomToTop, LeftToRight, RightToLeft
gradientColor1	(0, 0, 255, 255)
gradientColor2	(0, 0, 0, 255)
backgroundMode	**Solid**, Gradient, Image, ImageSphere
backgroundImage	“”
imageRepeatX	1
imageRepeatY	1
axesArray.visible	1
axesArray.ticksVisible	1
axesArray.autoSetTicks	1
axesArray.autoSetScaling	1
axesArray.lineWidth	0
axesArray.axes.title.visible	1
axesArray.axes.title.font.font	**Arial**, Courier, Times
axesArray.axes.title.font.scale	1
axesArray.axes.title.font.useForegroundColor	1
axesArray.axes.title.font.color	(0, 0, 0, 255)
axesArray.axes.title.font.bold	0
axesArray.axes.title.font.italic	0
axesArray.axes.title.userTitle	0
axesArray.axes.title.userUnits	0
axesArray.axes.title.title	“”
axesArray.axes.title.units	“”
axesArray.axes.label.visible	1
axesArray.axes.label.font.font	**Arial**, Courier, Times
axesArray.axes.label.font.scale	1
axesArray.axes.label.font.useForegroundColor	1
axesArray.axes.label.font.color	(0, 0, 0, 255)
axesArray.axes.label.font.bold	0
axesArray.axes.label.font.italic	0
axesArray.axes.label.scaling	0
axesArray.axes.tickMarks.visible	1
axesArray.axes.tickMarks.majorMinimum	0
axesArray.axes.tickMarks.majorMaximum	1
axesArray.axes.tickMarks.minorSpacing	0.02
axesArray.axes.tickMarks.majorSpacing	0.2
axesArray.axes.grid	0

5.5.4 Axis: *AxisAttributes()*

Attribute	**Default**/Allowed Values
title.visible	1
title.font.font	**Arial**, Courier, Times
title.font.scale	1
title.font.useForegroundColor	1
title.font.color	(0, 0, 0, 255)
title.font.bold	0
title.font.italic	0
title.userTitle	0
title.userUnits	0
title.title	“”
title.units	“”
label.visible	1
label.font.font	**Arial**, Courier, Times
label.font.scale	1
label.font.useForegroundColor	1
label.font.color	(0, 0, 0, 255)
label.font.bold	0
label.font.italic	0
label.scaling	0
tickMarks.visible	1
tickMarks.majorMinimum	0
tickMarks.majorMaximum	1
tickMarks.minorSpacing	0.02
tickMarks.majorSpacing	0.2
grid	0

5.5.5 AxisAlignedSlice4D: *AxisAlignedSlice4DAttributes()*

Attribute	**Default**/Allowed Values
I	()
J	()
K	()
L	()

5.5.6 Boundary: *BoundaryAttributes()*

Attribute	**Default**/Allowed Values
colorType	**ColorByMultipleColors**, ColorBySingleColor, ColorByColorTable
colorTableName	"Default"
invertColorTable	0
legendFlag	1
lineWidth	0
singleColor	(0, 0, 0, 255)
boundaryNames	()
opacity	1
wireframe	0
smoothingLevel	0

5.5.7 BoundaryOp: *BoundaryOpAttributes()*

Attribute	**Default**/Allowed Values
smoothingLevel	0

5.5.8 Box: *BoxAttributes()*

Attribute	**Default**/Allowed Values
amount	**Some**, All
minx	0
maxx	1
miny	0
maxy	1
minz	0
maxz	1
inverse	0

5.5.9 CartographicProjection: *CartographicProjectionAttributes()*

Attribute	**Default**/Allowed Values
projectionID	**aitoff**, eck4, eqdc, hammer, laea, lcc, merc, mill, moll, ortho, wink2
centralMeridian	0

5.5.10 Clip: *ClipAttributes()*

Attribute	**Default**/Allowed Values
quality	**Fast**, Accurate
funcType	**Plane**, Sphere
plane1Status	1
plane2Status	0
plane3Status	0
plane1Origin	(0, 0, 0)
plane2Origin	(0, 0, 0)
plane3Origin	(0, 0, 0)
plane1Normal	(1, 0, 0)
plane2Normal	(0, 1, 0)
plane3Normal	(0, 0, 1)
planeInverse	0
planeToolControlledClipPlane	**Plane1**, NONE, Plane2, Plane3
center	(0, 0, 0)
radius	1
sphereInverse	0
crinkleClip	0

5.5.11 Cone: *ConeAttributes()*

Attribute	**Default**/Allowed Values
angle	45
origin	(0, 0, 0)
normal	(0, 0, 1)
representation	**Flattened**, ThreeD, R_Theta
upAxis	(0, 1, 0)
cutByLength	0
length	1

5.5.12 ConnectedComponents: *ConnectedComponentsAttributes()*

Attribute	**Default**/Allowed Values
EnableGhostNeighborsOptimization	1

5.5.13 ConstructDataBinning: *ConstructDataBinningAttributes()*

Attribute	**Default**/Allowed Values
name	“”
varnames	()
binType	()
binBoundaries	()
reductionOperator	**Average**, Minimum, Maximum, StandardDeviation, Variance, Sum, Count, RMS, PDF
varForReductionOperator	“”
undefinedValue	0
binningScheme	**Uniform**, Unknown
numBins	()
overTime	0
timeStart	0
timeEnd	1
timeStride	1
outOfBoundsBehavior	**Clamp**, Discard

5.5.14 Contour: *ContourAttributes()*

Attribute	**Default**/Allowed Values
defaultPalette.GetControlPoints(0).colors	(255, 0, 0, 255)
defaultPalette.GetControlPoints(0).position	0
defaultPalette.GetControlPoints(1).colors	(0, 255, 0, 255)
defaultPalette.GetControlPoints(1).position	0.034
defaultPalette.GetControlPoints(2).colors	(0, 0, 255, 255)
defaultPalette.GetControlPoints(2).position	0.069
defaultPalette.GetControlPoints(3).colors	(0, 255, 255, 255)
defaultPalette.GetControlPoints(3).position	0.103
defaultPalette.GetControlPoints(4).colors	(255, 0, 255, 255)
defaultPalette.GetControlPoints(4).position	0.138
defaultPalette.GetControlPoints(5).colors	(255, 255, 0, 255)
defaultPalette.GetControlPoints(5).position	0.172
defaultPalette.GetControlPoints(6).colors	(255, 135, 0, 255)
defaultPalette.GetControlPoints(6).position	0.207
defaultPalette.GetControlPoints(7).colors	(255, 0, 135, 255)
defaultPalette.GetControlPoints(7).position	0.241
defaultPalette.GetControlPoints(8).colors	(168, 168, 168, 255)
defaultPalette.GetControlPoints(8).position	0.276
defaultPalette.GetControlPoints(9).colors	(255, 68, 68, 255)
defaultPalette.GetControlPoints(9).position	0.31
defaultPalette.GetControlPoints(10).colors	(99, 255, 99, 255)
defaultPalette.GetControlPoints(10).position	0.345
defaultPalette.GetControlPoints(11).colors	(99, 99, 255, 255)
defaultPalette.GetControlPoints(11).position	0.379
defaultPalette.GetControlPoints(12).colors	(40, 165, 165, 255)
defaultPalette.GetControlPoints(12).position	0.414
defaultPalette.GetControlPoints(13).colors	(255, 99, 255, 255)

Continued on next page

Table 5.2 – continued from previous page

defaultPalette.GetControlPoints(13).position	0.448
defaultPalette.GetControlPoints(14).colors	(255, 255, 99, 255)
defaultPalette.GetControlPoints(14).position	0.483
defaultPalette.GetControlPoints(15).colors	(255, 170, 99, 255)
defaultPalette.GetControlPoints(15).position	0.517
defaultPalette.GetControlPoints(16).colors	(170, 79, 255, 255)
defaultPalette.GetControlPoints(16).position	0.552
defaultPalette.GetControlPoints(17).colors	(150, 0, 0, 255)
defaultPalette.GetControlPoints(17).position	0.586
defaultPalette.GetControlPoints(18).colors	(0, 150, 0, 255)
defaultPalette.GetControlPoints(18).position	0.621
defaultPalette.GetControlPoints(19).colors	(0, 0, 150, 255)
defaultPalette.GetControlPoints(19).position	0.655
defaultPalette.GetControlPoints(20).colors	(0, 109, 109, 255)
defaultPalette.GetControlPoints(20).position	0.69
defaultPalette.GetControlPoints(21).colors	(150, 0, 150, 255)
defaultPalette.GetControlPoints(21).position	0.724
defaultPalette.GetControlPoints(22).colors	(150, 150, 0, 255)
defaultPalette.GetControlPoints(22).position	0.759
defaultPalette.GetControlPoints(23).colors	(150, 84, 0, 255)
defaultPalette.GetControlPoints(23).position	0.793
defaultPalette.GetControlPoints(24).colors	(160, 0, 79, 255)
defaultPalette.GetControlPoints(24).position	0.828
defaultPalette.GetControlPoints(25).colors	(255, 104, 28, 255)
defaultPalette.GetControlPoints(25).position	0.862
defaultPalette.GetControlPoints(26).colors	(0, 170, 81, 255)
defaultPalette.GetControlPoints(26).position	0.897
defaultPalette.GetControlPoints(27).colors	(68, 255, 124, 255)
defaultPalette.GetControlPoints(27).position	0.931
defaultPalette.GetControlPoints(28).colors	(0, 130, 255, 255)
defaultPalette.GetControlPoints(28).position	0.966
defaultPalette.GetControlPoints(29).colors	(130, 0, 255, 255)
defaultPalette.GetControlPoints(29).position	1
defaultPalette.smoothing	**NONE**, Linear, CubicSpline
defaultPalette.equalSpacingFlag	1
defaultPalette.discreteFlag	1
defaultPalette.tagNames	(“Default”, “Discrete”)
changedColors	()
colorType	**ColorByMultipleColors**, ColorBySingleColor, ColorByColorTable
colorTableName	“Default”
invertColorTable	0
legendFlag	1
lineWidth	0
singleColor	(255, 0, 0, 255)
contourMethod	**Level**, Value, Percent
contourNLevels	10
contourValue	()
contourPercent	()
	SetMultiColor(0, (255, 0, 0, 255))
	SetMultiColor(1, (0, 255, 0, 255))

Continued on next page

Table 5.2 – continued from previous page

	SetMultiColor(2, (0, 0, 255, 255))
	SetMultiColor(3, (0, 255, 255, 255))
	SetMultiColor(4, (255, 0, 255, 255))
	SetMultiColor(5, (255, 255, 0, 255))
	SetMultiColor(6, (255, 135, 0, 255))
	SetMultiColor(7, (255, 0, 135, 255))
	SetMultiColor(8, (168, 168, 168, 255))
	SetMultiColor(9, (255, 68, 68, 255))
minFlag	0
maxFlag	0
min	0
max	1
scaling	**Linear**, Log
wireframe	0

5.5.15 CoordSwap: *CoordSwapAttributes()*

Attribute	**Default**/Allowed Values
newCoord1	**Coord1**, Coord2, Coord3
newCoord2	**Coord2**, Coord1, Coord3
newCoord3	**Coord3**, Coord1, Coord2

5.5.16 CreateBonds: *CreateBondsAttributes()*

Attribute	**Default**/Allowed Values
elementVariable	"element"
atomicNumber1	(1, -1)
atomicNumber2	(-1, -1)
minDist	(0.4, 0.4)
maxDist	(1.2, 1.9)
maxBondsClamp	10
addPeriodicBonds	0
useUnitCellVectors	1
periodicInX	1
periodicInY	1
periodicInZ	1
xVector	(1, 0, 0)
yVector	(0, 1, 0)
zVector	(0, 0, 1)

5.5.17 Curve: *CurveAttributes()*

Attribute	**Default**/Allowed Values
showLines	1
lineWidth	0
showPoints	0
symbol	**Point**, TriangleUp, TriangleDown, Square, Circle, Plus, X
pointSize	5
pointFillMode	**Static**, Dynamic
pointStride	1
symbolDensity	50
curveColorSource	**Cycle**, Custom
curveColor	(0, 0, 0, 255)
showLegend	1
showLabels	1
designator	“”
doBallTimeCue	0
ballTimeCueColor	(0, 0, 0, 255)
timeCueBallSize	0.01
doLineTimeCue	0
lineTimeCueColor	(0, 0, 0, 255)
lineTimeCueWidth	0
doCropTimeCue	0
timeForTimeCue	0
fillMode	**NoFill**, Solid, HorizontalGradient, VerticalGradient
fillColor1	(255, 0, 0, 255)
fillColor2	(255, 100, 100, 255)
polarToCartesian	0
polarCoordinateOrder	**R_Theta**, Theta_R
angleUnits	**Radians**, Degrees

5.5.18 Cylinder: *CylinderAttributes()*

Attribute	**Default**/Allowed Values
point1	(0, 0, 0)
point2	(1, 0, 0)
radius	1
inverse	0

5.5.19 DataBinning: *DataBinningAttributes()*

Attribute	**Default**/Allowed Values
numDimensions	**One**, Two, Three
dim1BinBasedOn	**Variable**, X, Y, Z
dim1Var	"default"
dim1SpecifyRange	0
dim1MinRange	0
dim1MaxRange	1
dim1NumBins	50
dim2BinBasedOn	**Variable**, X, Y, Z
dim2Var	"default"
dim2SpecifyRange	0
dim2MinRange	0
dim2MaxRange	1
dim2NumBins	50
dim3BinBasedOn	**Variable**, X, Y, Z
dim3Var	"default"
dim3SpecifyRange	0
dim3MinRange	0
dim3MaxRange	1
dim3NumBins	50
outOfBoundsBehavior	**Clamp**, Discard
reductionOperator	**Average**, Minimum, Maximum, StandardDeviation, Variance, Sum, Count, RMS, PDF
varForReduction	"default"
emptyVal	0
outputType	**OutputOnBins**, OutputOnInputMesh
removeEmptyValFrom-Curve	1

5.5.20 DeferExpression: *DeferExpressionAttributes()*

Attribute	**Default**/Allowed Values
exprs	()

5.5.21 Displace: *DisplaceAttributes()*

Attribute	**Default**/Allowed Values
factor	1
variable	"default"

5.5.22 DualMesh: *DualMeshAttributes()*

Attribute	**Default**/Allowed Values
mode	**Auto**, NodesToZones, ZonesToNodes

5.5.23 Edge: *EdgeAttributes()*

Attribute	**Default**/Allowed Values
dummy	1

5.5.24 Elevate: *ElevateAttributes()*

Attribute	**Default**/Allowed Values
useXYLimits	**Auto**, Never, Always
limitsMode	**OriginalData**, CurrentPlot
scaling	**Linear**, Log, Skew
skewFactor	1
minFlag	0
min	0
maxFlag	0
max	1
zeroFlag	0
variable	"default"

5.5.25 EllipsoidSlice: *EllipsoidSliceAttributes()*

Attribute	**Default**/Allowed Values
origin	(0, 0, 0)
radii	(1, 1, 1)
rotationAngle	(0, 0, 0)

5.5.26 Explode: *ExplodeAttributes()*

Attribute	**Default**/Allowed Values
explosionType	**Point**, Plane, Cylinder
explosionPoint	(0, 0, 0)
planePoint	(0, 0, 0)
planeNorm	(0, 0, 0)
cylinderPoint1	(0, 0, 0)
cylinderPoint2	(0, 0, 0)
materialExplosionFactor	1
material	""
cylinderRadius	0
explodeMaterialCells	0
cellExplosionFactor	1
explosionPattern	**Impact**, Scatter
explodeAllCells	0
boundaryNames	()
	explosions does not contain any ExplodeAttributes objects.

5.5.27 ExportDB: *ExportDBAttributes()*

Attribute	**Default**/Allowed Values
allTimes	0
dirname	"."
filename	"visit_ex_db"
timeStateFormat	"_%04d"
db_type	""
db_type_fullname	""
variables	()
writeUsingGroups	0
groupSize	48

5.5.28 ExternalSurface: *ExternalSurfaceAttributes()*

Attribute	**Default**/Allowed Values
removeGhosts	0
edgesIn2D	1

5.5.29 Extrude: *ExtrudeAttributes()*

Attribute	**Default**/Allowed Values
axis	(0, 0, 1)
byVariable	0
variable	"default"
length	1
steps	1
preserveOriginalCellNumbers	1

5.5.30 FFT: *FFTAttributes()*

Attribute	**Default**/Allowed Values
dummy	0

5.5.31 FilledBoundary: *FilledBoundaryAttributes()*

Attribute	**Default**/Allowed Values
colorType	**ColorByMultipleColors**, ColorBySingleColor, ColorByColorTable
colorTableName	"Default"
invertColorTable	0
legendFlag	1
lineWidth	0
singleColor	(0, 0, 0, 255)
boundaryNames	()
opacity	1
wireframe	0
drawInternal	0
smoothingLevel	0
cleanZonesOnly	0
mixedColor	(255, 255, 255, 255)
pointSize	0.05
pointType	**Point**, Box, Axis, Icosahedron, Octahedron, Tetrahedron, SphereGeometry, Sphere
pointSizeVarEnabled	0
pointSizeVar	"default"
pointSizePixels	2

5.5.32 Flux: *FluxAttributes()*

Attribute	**Default**/Allowed Values
flowField	"default"
weight	0
weightField	"default"

5.5.33 Font: *FontAttributes()*

Attribute	**Default**/Allowed Values
font	**Arial**, Courier, Times
scale	1
useForegroundColor	1
color	(0, 0, 0, 255)
bold	0
italic	0

5.5.34 Global: *GlobalAttributes()*

Attribute	**Default**/Allowed Values
sources	()
windows	(1)
activeWindow	0
iconifiedFlag	0
autoUpdateFlag	0
replacePlots	0
applyOperator	1
applySelection	1
applyWindow	0
executing	0
windowLayout	1
makeDefaultConfirm	1
cloneWindowOnFirstRef	0
automaticallyAddOperator	0
tryHarderCyclesTimes	0
treatAllDBsAsTimeVarying	0
createMeshQualityExpressions	1
createTimeDerivativeExpressions	1
createVectorMagnitudeExpressions	1
newPlotsInheritSILRestriction	1
userDirForSessionFiles	0
saveCrashRecoveryFile	1
ignoreExtentsFromDbs	0
expandNewPlots	0
userRestoreSessionFile	0
precisionType	**Native**, Float, Double
backendType	**VTK**, VTKM
removeDuplicateNodes	0

5.5.35 Histogram: *HistogramAttributes()*

Attribute	**Default**/Allowed Values
basedOn	**ManyZonesForSingleVar**, ManyVarsForSingleZone
histogramType	**Frequency**, Weighted, Variable
weightVariable	"default"
limitsMode	**OriginalData**, CurrentPlot
minFlag	0
maxFlag	0
min	0
max	1
numBins	32
domain	0
zone	0
useBinWidths	1
outputType	**Block**, Curve
lineWidth	0
color	(200, 80, 40, 255)
dataScale	**Linear**, Log, SquareRoot
binScale	**Linear**, Log, SquareRoot
normalizeHistogram	0
computeAsCDF	0

5.5.36 IndexSelect: *IndexSelectAttributes()*

Attribute	**Default**/Allowed Values
maxDim	**ThreeD**, OneD, TwoD
dim	**TwoD**, OneD, ThreeD
xAbsMax	-1
xMin	0
xMax	-1
xIncr	1
xWrap	0
yAbsMax	-1
yMin	0
yMax	-1
yIncr	1
yWrap	0
zAbsMax	-1
zMin	0
zMax	-1
zIncr	1
zWrap	0
useWholeCollection	1
categoryName	"Whole"
subsetName	"Whole"

5.5.37 IntegralCurve: *IntegralCurveAttributes()*

Attribute	**Default**/Allowed Values
sourceType	**SpecifiedPoint**, PointList, SpecifiedLine, Circle, SpecifiedPlane, SpecifiedSphere, SpecifiedB
pointSource	(0, 0, 0)
lineStart	(0, 0, 0)
lineEnd	(1, 0, 0)
planeOrigin	(0, 0, 0)
planeNormal	(0, 0, 1)
planeUpAxis	(0, 1, 0)
radius	1
sphereOrigin	(0, 0, 0)
boxExtents	(0, 1, 0, 1, 0, 1)
useWholeBox	1
pointList	(0, 0, 0, 1, 0, 0, 0, 1, 0)
fieldData	()
sampleDensity0	2
sampleDensity1	2
sampleDensity2	2
dataValue	**TimeAbsolute**, Solid, Random, SeedPointID, Speed, Vorticity, ArcLength, TimeRelative, Ave
dataVariable	“”
integrationDirection	**Forward**, Backward, Both, ForwardDirectionless, BackwardDirectionless, BothDirectionless
maxSteps	1000
terminateByDistance	0
termDistance	10
terminateByTime	0
termTime	10
maxStepLength	0.1
limitMaximumTimestep	0
maxTimeStep	0.1
relTol	0.0001
absTolSizeType	**FractionOfBBox**, Absolute
absTolAbsolute	1e-06
absTolBBox	1e-06
fieldType	**Default**, FlashField, M3DC12DField, M3DC13DField, Nek5000Field, NektarPPField
fieldConstant	1
velocitySource	(0, 0, 0)
integrationType	**DormandPrince**, Euler, Leapfrog, AdamsBashforth, RK4, M3DC12DIntegrator
parallelizationAlgorithmType	**VisItSelects**, LoadOnDemand, ParallelStaticDomains, ManagerWorker
maxProcessCount	10
maxDomainCacheSize	3
workGroupSize	32
pathlines	0
pathlinesOverrideStartingTimeFlag	0
pathlinesOverrideStartingTime	0
pathlinesPeriod	0
pathlinesCMFE	**POS_CMFE**, CONN_CMFE
displayGeometry	**Lines**, Tubes, Ribbons
cleanupMethod	**NoCleanup**, Merge, Before, After
cleanupThreshold	1e-08

Table 5.3 – continued from previous p

cropBeginFlag	0
cropBegin	0
cropEndFlag	0
cropEnd	0
cropValue	**Time**, Distance, StepNumber
sampleDistance0	10
sampleDistance1	10
sampleDistance2	10
fillInterior	1
randomSamples	0
randomSeed	0
numberOfRandomSamples	1
issueAdvectionWarnings	1
issueBoundaryWarnings	1
issueTerminationWarnings	1
issueStepsizeWarnings	1
issueStiffnessWarnings	1
issueCriticalPointsWarnings	1
criticalPointThreshold	0.001
correlationDistanceAngTol	5
correlationDistanceMinDistAbsolute	1
correlationDistanceMinDistBBox	0.005
correlationDistanceMinDistType	**FractionOfBBox**, Absolute
selection	“”

5.5.38 InverseGhostZone: *InverseGhostZoneAttributes()*

Attribute	**Default**/Allowed Values
requestGhostZones	1
showDuplicated	1
showEnhancedConnectivity	1
showReducedConnectivity	1
showAMRRefined	1
showExterior	1
showNotApplicable	1

5.5.39 Isosurface: *IsosurfaceAttributes()*

Attribute	**Default**/Allowed Values
contourNLevels	10
contourValue	()
contourPercent	()
contourMethod	**Level**, Value, Percent
minFlag	0
min	0
maxFlag	0
max	1
scaling	**Linear**, Log
variable	"default"

5.5.40 Isovolume: *IsovolumeAttributes()*

Attribute	**Default**/Allowed Values
lbound	-1e+37
ubound	1e+37
variable	"default"

5.5.41 Keyframe: *KeyframeAttributes()*

Attribute	**Default**/Allowed Values
enabled	0
nFrames	1
nFramesWasUserSet	0

5.5.42 LCS: *LCSAttributes()*

Attribute	**Default**/Allowed Values
sourceType	**NativeMesh**, RegularGrid
Resolution	(10, 10, 10)
UseDataSetStart	**Full**, Subset
StartPosition	(0, 0, 0)
UseDataSetEnd	**Full**, Subset
EndPosition	(1, 1, 1)
integrationDirection	**Forward**, Backward, Both
auxiliaryGrid	**NONE**, TwoDim, ThreeDim
auxiliaryGridSpacing	0.0001
maxSteps	1000
operationType	**Lyapunov**, IntegrationTime, ArcLength, AverageDistanceFromSeed, EigenValue, EigenVector
cauchyGreenTensor	**Right**, Left
eigenComponent	**Largest**, Smallest, Intermediate, PosShearVector, NegShearVector, PosLambdaShearVector, Ne
eigenWeight	1

Con

Table 5.4 – continued from previous page

operatorType	**BaseValue**, Gradient
terminationType	**Time**, Distance, Size
terminateBySize	0
termSize	10
terminateByDistance	0
termDistance	10
terminateByTime	0
termTime	10
maxStepLength	0.1
limitMaximumTimestep	0
maxTimeStep	0.1
relTol	0.0001
absTolSizeType	**FractionOfBBox**, Absolute
absTolAbsolute	1e-06
absTolBBox	1e-06
fieldType	**Default**, FlashField, M3DC12DField, M3DC13DField, Nek5000Field, NektarPPField
fieldConstant	1
velocitySource	(0, 0, 0)
integrationType	**DormandPrince**, Euler, Leapfrog, AdamsBashforth, RK4, M3DC12DIntegrator
clampLogValues	0
parallelizationAlgorithmType	**VisItSelects**, LoadOnDemand, ParallelStaticDomains, ManagerWorker
maxProcessCount	10
maxDomainCacheSize	3
workGroupSize	32
pathlines	0
pathlinesOverrideStartingTimeFlag	0
pathlinesOverrideStartingTime	0
pathlinesPeriod	0
pathlinesCMFE	**POS_CMFE**, CONN_CMFE
thresholdLimit	0.1
radialLimit	0.1
boundaryLimit	0.1
seedLimit	10
issueAdvectionWarnings	1
issueBoundaryWarnings	1
issueTerminationWarnings	1
issueStepsizeWarnings	1
issueStiffnessWarnings	1
issueCriticalPointsWarnings	1
criticalPointThreshold	0.001

5.5.43 Label: *LabelAttributes()*

Attribute	**Default**/Allowed Values
legendFlag	1
showNodes	0
showCells	1
restrictNumberOfLabels	1
drawLabelsFacing	**Front**, Back, FrontAndBack
labelDisplayFormat	**Natural**, LogicalIndex, Index
numberOfLabels	200
textFont1.font	**Arial**, Courier, Times
textFont1.scale	4
textFont1.useForegroundColor	1
textFont1.color	(255, 0, 0, 255)
textFont1.bold	0
textFont1.italic	0
textFont2.font	**Arial**, Courier, Times
textFont2.scale	4
textFont2.useForegroundColor	1
textFont2.color	(0, 0, 255, 255)
textFont2.bold	0
textFont2.italic	0
horizontalJustification	**HCenter**, Left, Right
verticalJustification	**VCenter**, Top, Bottom
depthTestMode	**LABEL_DT_AUTO**, LABEL_DT_ALWAYS, LABEL_DT_NEVER
formatTemplate	"%g"

5.5.44 Lagrangian: *LagrangianAttributes()*

Attribute	**Default**/Allowed Values
seedPoint	(0, 0, 0)
numSteps	1000
XAxisSample	**Step**, Time, ArcLength, Speed, Vorticity, Variable
YAxisSample	**Step**, Time, ArcLength, Speed, Vorticity, Variable
variable	"default"

5.5.45 Light: *LightAttributes()*

Attribute	**Default**/Allowed Values
enabledFlag	1
type	**Camera**, Ambient, Object
direction	(0, 0, -1)
color	(255, 255, 255, 255)
brightness	1

5.5.46 LimitCycle: *LimitCycleAttributes()*

Attribute	**Default**/Allowed Values
sourceType	**SpecifiedLine**, SpecifiedPlane
lineStart	(0, 0, 0)
lineEnd	(1, 0, 0)
planeOrigin	(0, 0, 0)
planeNormal	(0, 0, 1)
planeUpAxis	(0, 1, 0)
sampleDensity0	2
sampleDensity1	2
dataValue	**TimeAbsolute**, Solid, SeedPointID, Speed, Vorticity, ArcLength, TimeRelative, AverageDista
dataVariable	“”
integrationDirection	**Forward**, Backward, Both, ForwardDirectionless, BackwardDirectionless, BothDirectionless
maxSteps	1000
terminateByDistance	0
termDistance	10
terminateByTime	0
termTime	10
maxStepLength	0.1
limitMaximumTimestep	0
maxTimeStep	0.1
relTol	0.0001
absTolSizeType	**FractionOfBBox**, Absolute
absTolAbsolute	1e-06
absTolBBox	1e-06
fieldType	**Default**, FlashField, M3DC12DField, M3DC13DField, Nek5000Field, NektarPPField
fieldConstant	1
velocitySource	(0, 0, 0)
integrationType	**DormandPrince**, Euler, Leapfrog, AdamsBashforth, RK4, M3DC12DIntegrator
parallelizationAlgorithmType	**VisItSelects**, LoadOnDemand, ParallelStaticDomains, ManagerWorker
maxProcessCount	10
maxDomainCacheSize	3
workGroupSize	32
pathlines	0
pathlinesOverrideStartingTimeFlag	0
pathlinesOverrideStartingTime	0
pathlinesPeriod	0
pathlinesCMFE	**POS_CMFE**, CONN_CMFE
sampleDistance0	10
sampleDistance1	10
sampleDistance2	10
fillInterior	1
randomSamples	0
randomSeed	0
numberOfRandomSamples	1
forceNodeCenteredData	0
cycleTolerance	1e-06
maxIterations	10
showPartialResults	1

Table 5.5 – continued from previous page

showReturnDistances	0
issueAdvectionWarnings	1
issueBoundaryWarnings	1
issueTerminationWarnings	1
issueStepsizeWarnings	1
issueStiffnessWarnings	1
issueCriticalPointsWarnings	1
criticalPointThreshold	0.001
correlationDistanceAngTol	5
correlationDistanceMinDistAbsolute	1
correlationDistanceMinDistBBox	0.005
correlationDistanceMinDistType	**FractionOfBBox**, Absolute

5.5.47 Lineout: *LineoutAttributes()*

Attribute	**Default**/Allowed Values
point1	(0, 0, 0)
point2	(1, 1, 0)
interactive	0
ignoreGlobal	0
samplingOn	0
numberOfSamplePoints	50
reflineLabels	0

5.5.48 Material: *MaterialAttributes()*

Attribute	**Default**/Allowed Values
smoothing	0
forceMIR	0
cleanZonesOnly	0
needValidConnectivity	0
algorithm	**EquiZ**, EquiT, Isovolume, PLIC, Discrete
iterationEnabled	0
numIterations	5
iterationDamping	0.4
simplifyHeavilyMixedZones	0
maxMaterialsPerZone	3
isoVolumeFraction	0.5
annealingTime	10

5.5.49 Mesh: *MeshAttributes()*

Attribute	**Default**/Allowed Values
legendFlag	1
lineWidth	0
meshColor	(0, 0, 0, 255)
meshColorSource	**Foreground**, MeshCustom, MeshRandom
opaqueColorSource	**Background**, OpaqueCustom, OpaqueRandom
opaqueMode	**Auto**, On, Off
pointSize	0.05
opaqueColor	(255, 255, 255, 255)
smoothingLevel	**NONE**, Fast, High
pointSizeVarEnabled	0
pointSizeVar	"default"
pointType	**Point**, Box, Axis, Icosahedron, Octahedron, Tetrahedron, SphereGeometry, Sphere
showInternal	0
pointSizePixels	2
opacity	1

5.5.50 MeshManagement: *MeshManagementAttributes()*

Attribute	**Default**/Allowed Values
discretizationTolerance	(0.02, 0.025, 0.05)
discretizationToleranceX	()
discretizationToleranceY	()
discretizationToleranceZ	()
discretizationMode	**Uniform**, Adaptive, MultiPass
discretizeBoundaryOnly	0
passNativeCSG	0

5.5.51 Molecule: *MoleculeAttributes()*

Attribute	**Default**/Allowed Values
drawAtomsAs	**SphereAtoms**, NoAtoms, ImposterAtoms
scaleRadiusBy	**Fixed**, Covalent, Atomic, Variable
drawBondsAs	**CylinderBonds**, NoBonds, LineBonds
colorBonds	**ColorByAtom**, SingleColor
bondSingleColor	(128, 128, 128, 255)
radiusVariable	"default"
radiusScaleFactor	1
radiusFixed	0.3
atomSphereQuality	**Medium**, Low, High, Super
bondCylinderQuality	**Medium**, Low, High, Super
bondRadius	0.12
bondLineWidth	0
elementColorTable	"cpk_jmol"
residueTypeColorTable	"amino_shapely"
residueSequenceColorTable	"Default"
continuousColorTable	"Default"
legendFlag	1
minFlag	0
scalarMin	0
maxFlag	0
scalarMax	1

5.5.52 MultiCurve: *MultiCurveAttributes()*

Attribute	**Default**/Allowed Values
defaultPalette.GetControlPoints(0).colors	(255, 0, 0, 255)
defaultPalette.GetControlPoints(0).position	0
defaultPalette.GetControlPoints(1).colors	(0, 255, 0, 255)
defaultPalette.GetControlPoints(1).position	0.034
defaultPalette.GetControlPoints(2).colors	(0, 0, 255, 255)
defaultPalette.GetControlPoints(2).position	0.069
defaultPalette.GetControlPoints(3).colors	(0, 255, 255, 255)
defaultPalette.GetControlPoints(3).position	0.103
defaultPalette.GetControlPoints(4).colors	(255, 0, 255, 255)
defaultPalette.GetControlPoints(4).position	0.138
defaultPalette.GetControlPoints(5).colors	(255, 255, 0, 255)
defaultPalette.GetControlPoints(5).position	0.172
defaultPalette.GetControlPoints(6).colors	(255, 135, 0, 255)
defaultPalette.GetControlPoints(6).position	0.207
defaultPalette.GetControlPoints(7).colors	(255, 0, 135, 255)
defaultPalette.GetControlPoints(7).position	0.241
defaultPalette.GetControlPoints(8).colors	(168, 168, 168, 255)
defaultPalette.GetControlPoints(8).position	0.276
defaultPalette.GetControlPoints(9).colors	(255, 68, 68, 255)
defaultPalette.GetControlPoints(9).position	0.31
defaultPalette.GetControlPoints(10).colors	(99, 255, 99, 255)

Continued on next page

Table 5.6 – continued from previous page

defaultPalette.GetControlPoints(10).position	0.345
defaultPalette.GetControlPoints(11).colors	(99, 99, 255, 255)
defaultPalette.GetControlPoints(11).position	0.379
defaultPalette.GetControlPoints(12).colors	(40, 165, 165, 255)
defaultPalette.GetControlPoints(12).position	0.414
defaultPalette.GetControlPoints(13).colors	(255, 99, 255, 255)
defaultPalette.GetControlPoints(13).position	0.448
defaultPalette.GetControlPoints(14).colors	(255, 255, 99, 255)
defaultPalette.GetControlPoints(14).position	0.483
defaultPalette.GetControlPoints(15).colors	(255, 170, 99, 255)
defaultPalette.GetControlPoints(15).position	0.517
defaultPalette.GetControlPoints(16).colors	(170, 79, 255, 255)
defaultPalette.GetControlPoints(16).position	0.552
defaultPalette.GetControlPoints(17).colors	(150, 0, 0, 255)
defaultPalette.GetControlPoints(17).position	0.586
defaultPalette.GetControlPoints(18).colors	(0, 150, 0, 255)
defaultPalette.GetControlPoints(18).position	0.621
defaultPalette.GetControlPoints(19).colors	(0, 0, 150, 255)
defaultPalette.GetControlPoints(19).position	0.655
defaultPalette.GetControlPoints(20).colors	(0, 109, 109, 255)
defaultPalette.GetControlPoints(20).position	0.69
defaultPalette.GetControlPoints(21).colors	(150, 0, 150, 255)
defaultPalette.GetControlPoints(21).position	0.724
defaultPalette.GetControlPoints(22).colors	(150, 150, 0, 255)
defaultPalette.GetControlPoints(22).position	0.759
defaultPalette.GetControlPoints(23).colors	(150, 84, 0, 255)
defaultPalette.GetControlPoints(23).position	0.793
defaultPalette.GetControlPoints(24).colors	(160, 0, 79, 255)
defaultPalette.GetControlPoints(24).position	0.828
defaultPalette.GetControlPoints(25).colors	(255, 104, 28, 255)
defaultPalette.GetControlPoints(25).position	0.862
defaultPalette.GetControlPoints(26).colors	(0, 170, 81, 255)
defaultPalette.GetControlPoints(26).position	0.897
defaultPalette.GetControlPoints(27).colors	(68, 255, 124, 255)
defaultPalette.GetControlPoints(27).position	0.931
defaultPalette.GetControlPoints(28).colors	(0, 130, 255, 255)
defaultPalette.GetControlPoints(28).position	0.966
defaultPalette.GetControlPoints(29).colors	(130, 0, 255, 255)
defaultPalette.GetControlPoints(29).position	1
defaultPalette.smoothing	**NONE**, Linear, CubicSpline
defaultPalette.equalSpacingFlag	1
defaultPalette.discreteFlag	1
defaultPalette.tagNames	(“Default”, “Discrete”)
changedColors	()
colorType	**ColorByMultipleColors**, ColorBySingleColor
singleColor	(255, 0, 0, 255)
	SetMultiColor(0, (255, 0, 0, 255))
	SetMultiColor(1, (0, 255, 0, 255))
	SetMultiColor(2, (0, 0, 255, 255))
	SetMultiColor(3, (0, 255, 255, 255))

Continued on next page

Table 5.6 – continued from previous page

	SetMultiColor(4, (255, 0, 255, 255))
	SetMultiColor(5, (255, 255, 0, 255))
	SetMultiColor(6, (255, 135, 0, 255))
	SetMultiColor(7, (255, 0, 135, 255))
	SetMultiColor(8, (168, 168, 168, 255))
	SetMultiColor(9, (255, 68, 68, 255))
	SetMultiColor(10, (99, 255, 99, 255))
	SetMultiColor(11, (99, 99, 255, 255))
	SetMultiColor(12, (40, 165, 165, 255))
	SetMultiColor(13, (255, 99, 255, 255))
	SetMultiColor(14, (255, 255, 99, 255))
	SetMultiColor(15, (255, 170, 99, 255))
	SetMultiColor(16, (170, 79, 255, 255))
	SetMultiColor(17, (150, 0, 0, 255))
	SetMultiColor(18, (0, 150, 0, 255))
	SetMultiColor(19, (0, 0, 150, 255))
lineWidth	0
yAxisTitleFormat	"%g"
useYAxisTickSpacing	0
yAxisTickSpacing	1
displayMarkers	1
markerScale	1
markerLineWidth	0
markerVariable	"default"
displayIds	0
idVariable	"default"
legendFlag	1

5.5.53 MultiresControl: *MultiresControlAttributes()*

Attribute	**Default**/Allowed Values
resolution	0
maxResolution	1
info	""

5.5.54 OnionPeel: *OnionPeelAttributes()*

Attribute	**Default**/Allowed Values
adjacencyType	**Node**, Face
useGlobalId	0
categoryName	"Whole"
subsetName	"Whole"
index	(0)
logical	0
requestedLayer	0
seedType	**SeedCell**, SeedNode
honorOriginalMesh	1

5.5.55 ParallelCoordinates: *ParallelCoordinatesAttributes()*

Attribute	**Default**/Allowed Values
scalarAxisNames	()
visualAxisNames	()
extentMinima	()
extentMaxima	()
drawLines	1
linesColor	(128, 0, 0, 255)
drawContext	1
contextGamma	2
contextNumPartitions	128
contextColor	(0, 220, 0, 255)
drawLinesOnlyIfExtentsOn	1
unifyAxisExtents	0
linesNumPartitions	512
focusGamma	4
drawFocusAs	**BinsOfConstantColor**, IndividualLines, BinsColoredByPopulation

5.5.56 PersistentParticles: *PersistentParticlesAttributes()*

Attribute	**Default**/Allowed Values
startIndex	0
stopIndex	1
stride	1
startPathType	**Absolute**, Relative
stopPathType	**Absolute**, Relative
traceVariableX	"default"
traceVariableY	"default"
traceVariableZ	"default"
connectParticles	0
showPoints	0
indexVariable	"default"

5.5.57 Poincare: *PoincareAttributes()*

Attribute	**Default**/Allowed Values
opacityType	**Explicit**, ColorTable
opacity	1
minPunctures	50
maxPunctures	500
puncturePlotType	**Single**, Double
maxSteps	1000
terminateByTime	0
termTime	10
puncturePeriodTolerance	0.01
puncturePlane	**Poloidal**, Toroidal, Arbitrary
sourceType	**SpecifiedPoint**, PointList, SpecifiedLine
pointSource	(0, 0, 0)
pointList	(0, 0, 0, 1, 0, 0, 0, 1, 0)
lineStart	(0, 0, 0)
lineEnd	(1, 0, 0)
pointDensity	1
fieldType	**Default**, FlashField, M3DC12DField, M3DC13DField, Nek5000Field, NektarPPField
forceNodeCenteredData	0
fieldConstant	1
velocitySource	(0, 0, 0)
integrationType	**AdamsBashforth**, Euler, Leapfrog, DormandPrince, RK4, M3DC12DIntegrator
coordinateSystem	**Cartesian**, Cylindrical
maxStepLength	0.1
limitMaximumTimestep	0
maxTimeStep	0.1
relTol	0.0001
absTolSizeType	**FractionOfBBox**, Absolute
absTolAbsolute	1e-05
absTolBBox	1e-06
analysis	**Normal**, NONE
maximumToroidalWinding	0
overrideToroidalWinding	0
overridePoloidalWinding	0
windingPairConfidence	0.9
rationalSurfaceFactor	0.1
overlaps	**Remove**, Raw, Merge, Smooth
meshType	**Curves**, Surfaces
numberPlanes	1
singlePlane	0
min	0
max	0
minFlag	0
maxFlag	0
colorType	**ColorByColorTable**, ColorBySingleColor
singleColor	(0, 0, 0, 255)
colorTableName	“Default”
dataValue	**SafetyFactorQ**, Solid, SafetyFactorP, SafetyFactorQ_NotP, SafetyFactorP_NotQ, ToroidalWin

Table 5.7

showRationalSurfaces	0
RationalSurfaceMaxIterations	2
showOPoints	0
OPointMaxIterations	2
showXPoints	0
XPointMaxIterations	2
performOLineAnalysis	0
OLineToroidalWinding	1
OLineAxisFileName	""
showChaotic	0
showIslands	0
SummaryFlag	1
verboseFlag	0
show1DPlots	0
showLines	1
showPoints	0
parallelizationAlgorithmType	**VisItSelects**, LoadOnDemand, ParallelStaticDomains, ManagerWorker
maxProcessCount	10
maxDomainCacheSize	3
workGroupSize	32
pathlines	0
pathlinesOverrideStartingTimeFlag	0
pathlinesOverrideStartingTime	0
pathlinesPeriod	0
pathlinesCMFE	**POS_CMFE**, CONN_CMFE
issueTerminationWarnings	1
issueStepsizeWarnings	1
issueStiffnessWarnings	1
issueCriticalPointsWarnings	1
criticalPointThreshold	0.001

5.5.58 Printer: *PrinterAttributes()*

Attribute	**Default**/Allowed Values
printerName	""
printProgram	"lpr"
documentName	"untitled"
creator	""
numCopies	1
portrait	1
printColor	1
outputToFile	0
outputToFileName	"untitled"
pageSize	2

5.5.59 Process: *ProcessAttributes()*

Attribute	**Default**/Allowed Values
pids	()
ppids	()
hosts	()
isParallel	0
memory	()
times	()

5.5.60 Project: *ProjectAttributes()*

Attribute	**Default**/Allowed Values
projectionType	**XYCartesian**, ZYCartesian, XZCartesian, XRCylindrical, YRCylindrical, ZRCylindrical
vectorTransform-Method	**AsDirection**, NONE, AsPoint, AsDisplacement

5.5.61 Pseudocolor: *PseudocolorAttributes()*

Attribute	**Default**/Allowed Values
scaling	**Linear**, Log, Skew
skewFactor	1
limitsMode	**OriginalData**, ActualData
minFlag	0
min	0
useBelowMinColor	0
belowMinColor	(0, 0, 0, 255)
maxFlag	0
max	1
useAboveMaxColor	0
aboveMaxColor	(0, 0, 0, 255)
centering	**Natural**, Nodal, Zonal
colorTableName	“Default”
invertColorTable	0
opacityType	**FullyOpaque**, ColorTable, Constant, Ramp, VariableRange
opacityVariable	“”
opacity	1
opacityVarMin	0
opacityVarMax	1
opacityVarMinFlag	0
opacityVarMaxFlag	0
pointSize	0.05
pointType	**Point**, Box, Axis, Icosahedron, Octahedron, Tetrahedron, SphereGeometry, Sphere
pointSizeVarEnabled	0
pointSizeVar	“default”
pointSizePixels	2

Continued on next page

Table 5.8 – continued from previous page

lineType	**Line**, Tube, Ribbon
lineWidth	0
tubeResolution	10
tubeRadiusSizeType	**FractionOfBBox**, Absolute
tubeRadiusAbsolute	0.125
tubeRadiusBBox	0.005
tubeRadiusVarEnabled	0
tubeRadiusVar	“”
tubeRadiusVarRatio	10
tailStyle	**NONE**, Spheres, Cones
headStyle	**NONE**, Spheres, Cones
endPointRadiusSizeType	**FractionOfBBox**, Absolute
endPointRadiusAbsolute	0.125
endPointRadiusBBox	0.05
endPointResolution	10
endPointRatio	5
endPointRadiusVarEnabled	0
endPointRadiusVar	“”
endPointRadiusVarRatio	10
renderSurfaces	1
renderWireframe	0
renderPoints	0
smoothingLevel	0
legendFlag	1
lightingFlag	1
wireframeColor	(0, 0, 0, 0)
pointColor	(0, 0, 0, 0)

5.5.62 RadialResample: *RadialResampleAttributes()*

Attribute	**Default**/Allowed Values
isFast	0
minTheta	0
maxTheta	90
deltaTheta	5
radius	0.5
deltaRadius	0.05
center	(0.5, 0.5, 0.5)
is3D	1
minAzimuth	0
maxAzimuth	180
deltaAzimuth	5

5.5.63 Reflect: *ReflectAttributes()*

Attribute	**Default**/Allowed Values
octant	**PXPYPZ**, NXPYPZ, PXNYPZ, NXNYPZ, PXPYNZ, NXPYNZ, PXNYNZ, NXNYNZ
useXBoundary	1
specifiedX	0
useYBoundary	1
specifiedY	0
useZBoundary	1
specifiedZ	0
reflections	(1, 0, 1, 0, 0, 0, 0, 0)
planePoint	(0, 0, 0)
planeNormal	(0, 0, 0)
reflectType	**Axis**, Plane

5.5.64 Remap: *RemapAttributes()*

Attribute	**Default**/Allowed Values
useExtents	1
startX	0
endX	1
cellsX	10
startY	0
endY	1
cellsY	10
is3D	1
startZ	0
endZ	1
cellsZ	10
variableType	**intrinsic**, extrinsic

5.5.65 Rendering: *RenderingAttributes()*

Attribute	**Default**/Allowed Values
antialiasing	0
orderComposite	1
depthCompositeThreads	2
depthCompositeBlocking	65536
alphaCompositeThreads	2
alphaCompositeBlocking	65536
depthPeeling	0
occlusionRatio	0
numberOfPeels	16
multiresolutionMode	0
multiresolutionCellSize	0.002

Continued on next page

Table 5.9 – continued from previous page

geometryRepresentation	**Surfaces**, Wireframe, Points
stereoRendering	0
stereoType	**CrystalEyes**, RedBlue, Interlaced, RedGreen
notifyForEachRender	0
scalableActivationMode	**Auto**, Never, Always
scalableAutoThreshold	2000000
specularFlag	0
specularCoeff	0.6
specularPower	10
specularColor	(255, 255, 255, 255)
doShadowing	0
shadowStrength	0.5
doDepthCueing	0
depthCueingAutomatic	1
startCuePoint	(-10, 0, 0)
endCuePoint	(10, 0, 0)
compressionActivationMode	**Never**, Always, Auto
colorTexturingFlag	1
compactDomainsActivationMode	**Never**, Always, Auto
compactDomainsAutoThreshold	256
osprayRendering	0
ospraySPP	1
osprayAO	0
osprayShadows	0

5.5.66 Replicate: *ReplicateAttributes()*

Attribute	**Default**/Allowed Values
useUnitCellVectors	0
xVector	(1, 0, 0)
yVector	(0, 1, 0)
zVector	(0, 0, 1)
xReplications	1
yReplications	1
zReplications	1
mergeResults	1
replicateUnitCellAtoms	0
shiftPeriodicAtomOrigin	0
newPeriodicOrigin	(0, 0, 0)

5.5.67 Resample: *ResampleAttributes()*

Attribute	**Default**/Allowed Values
useExtents	1
startX	0
endX	1
samplesX	10
startY	0
endY	1
samplesY	10
is3D	1
startZ	0
endZ	1
samplesZ	10
tieResolver	**random**, largest, smallest
tieResolverVariable	"default"
defaultValue	0
distributedResample	1
cellCenteredOutput	0

5.5.68 Revolve: *RevolveAttributes()*

Attribute	**Default**/Allowed Values
meshType	**Auto**, XY, RZ, ZR
autoAxis	1
axis	(1, 0, 0)
startAngle	0
stopAngle	360
steps	30

5.5.69 SPHResample: *SPHResampleAttributes()*

Attribute	**Default**/Allowed Values
minX	0
maxX	1
xnum	10
minY	0
maxY	1
ynum	10
minZ	0
maxZ	1
znum	10
tensorSupportVariable	"H"
weightVariable	"mass"
RK	1

5.5.70 SaveWindow: *SaveWindowAttributes()*

Attribute	**Default**/Allowed Values
outputToCurrentDirectory	1
outputDirectory	"."
fileName	"visit"
family	1
format	**PNG**, BMP, CURVE, JPEG, OBJ, POSTSCRIPT, POVRAY, PPM, RGB, STL, TIFF, ULTRA
width	1024
height	1024
screenCapture	0
saveTiled	0
quality	80
progressive	0
binary	0
stereo	0
compression	**NONE**, PackBits, Jpeg, Deflate, LZW
forceMerge	0
resConstraint	**ScreenProportions**, NoConstraint, EqualWidthHeight
pixelData	1
advancedMultiWindowSave	0
subWindowAtts.win1.position	(0, 0)
subWindowAtts.win1.size	(128, 128)
subWindowAtts.win1.layer	0
subWindowAtts.win1.transparency	0
subWindowAtts.win1.omitWindow	0
subWindowAtts.win2.position	(0, 0)
subWindowAtts.win2.size	(128, 128)
subWindowAtts.win2.layer	0
subWindowAtts.win2.transparency	0
subWindowAtts.win2.omitWindow	0
subWindowAtts.win3.position	(0, 0)
subWindowAtts.win3.size	(128, 128)
subWindowAtts.win3.layer	0
subWindowAtts.win3.transparency	0
subWindowAtts.win3.omitWindow	0
subWindowAtts.win4.position	(0, 0)
subWindowAtts.win4.size	(128, 128)
subWindowAtts.win4.layer	0
subWindowAtts.win4.transparency	0
subWindowAtts.win4.omitWindow	0
subWindowAtts.win5.position	(0, 0)
subWindowAtts.win5.size	(128, 128)
subWindowAtts.win5.layer	0
subWindowAtts.win5.transparency	0
subWindowAtts.win5.omitWindow	0
subWindowAtts.win6.position	(0, 0)
subWindowAtts.win6.size	(128, 128)
subWindowAtts.win6.layer	0
subWindowAtts.win6.transparency	0

Continu

Table 5.10 – continued from previous page

subWindowAtts.win6.omitWindow	0
subWindowAtts.win7.position	(0, 0)
subWindowAtts.win7.size	(128, 128)
subWindowAtts.win7.layer	0
subWindowAtts.win7.transparency	0
subWindowAtts.win7.omitWindow	0
subWindowAtts.win8.position	(0, 0)
subWindowAtts.win8.size	(128, 128)
subWindowAtts.win8.layer	0
subWindowAtts.win8.transparency	0
subWindowAtts.win8.omitWindow	0
subWindowAtts.win9.position	(0, 0)
subWindowAtts.win9.size	(128, 128)
subWindowAtts.win9.layer	0
subWindowAtts.win9.transparency	0
subWindowAtts.win9.omitWindow	0
subWindowAtts.win10.position	(0, 0)
subWindowAtts.win10.size	(128, 128)
subWindowAtts.win10.layer	0
subWindowAtts.win10.transparency	0
subWindowAtts.win10.omitWindow	0
subWindowAtts.win11.position	(0, 0)
subWindowAtts.win11.size	(128, 128)
subWindowAtts.win11.layer	0
subWindowAtts.win11.transparency	0
subWindowAtts.win11.omitWindow	0
subWindowAtts.win12.position	(0, 0)
subWindowAtts.win12.size	(128, 128)
subWindowAtts.win12.layer	0
subWindowAtts.win12.transparency	0
subWindowAtts.win12.omitWindow	0
subWindowAtts.win13.position	(0, 0)
subWindowAtts.win13.size	(128, 128)
subWindowAtts.win13.layer	0
subWindowAtts.win13.transparency	0
subWindowAtts.win13.omitWindow	0
subWindowAtts.win14.position	(0, 0)
subWindowAtts.win14.size	(128, 128)
subWindowAtts.win14.layer	0
subWindowAtts.win14.transparency	0
subWindowAtts.win14.omitWindow	0
subWindowAtts.win15.position	(0, 0)
subWindowAtts.win15.size	(128, 128)
subWindowAtts.win15.layer	0
subWindowAtts.win15.transparency	0
subWindowAtts.win15.omitWindow	0
subWindowAtts.win16.position	(0, 0)
subWindowAtts.win16.size	(128, 128)
subWindowAtts.win16.layer	0
subWindowAtts.win16.transparency	0

Continu

Table 5.10 – continued from previous page

subWindowAtts.win16.omitWindow	0
opts.types	()
opts.help	""

5.5.71 Scatter: *ScatterAttributes()*

Attribute	**Default**/Allowed Values
var1	"default"
var1Role	**Coordinate0**, Coordinate1, Coordinate2, Color, NONE
var1MinFlag	0
var1MaxFlag	0
var1Min	0
var1Max	1
var1Scaling	**Linear**, Log, Skew
var1SkewFactor	1
var2Role	**Coordinate1**, Coordinate0, Coordinate2, Color, NONE
var2	"default"
var2MinFlag	0
var2MaxFlag	0
var2Min	0
var2Max	1
var2Scaling	**Linear**, Log, Skew
var2SkewFactor	1
var3Role	**NONE**, Coordinate0, Coordinate1, Coordinate2, Color
var3	"default"
var3MinFlag	0
var3MaxFlag	0
var3Min	0
var3Max	1
var3Scaling	**Linear**, Log, Skew
var3SkewFactor	1
var4Role	**NONE**, Coordinate0, Coordinate1, Coordinate2, Color
var4	"default"
var4MinFlag	0
var4MaxFlag	0
var4Min	0
var4Max	1
var4Scaling	**Linear**, Log, Skew
var4SkewFactor	1
pointSize	0.05
pointSizePixels	1
pointType	**Point**, Box, Axis, Icosahedron, Octahedron, Tetrahedron, SphereGeometry, Sphere
scaleCube	1
colorType	**ColorByForegroundColor**, ColorBySingleColor, ColorByColorTable
singleColor	(255, 0, 0, 255)
colorTableName	"Default"
invertColorTable	0

Continued on next page

Table 5.11 – continued from previous page

legendFlag	1

5.5.72 Slice: *SliceAttributes()*

Attribute	**Default**/Allowed Values
originType	**Intercept**, Point, Percent, Zone, Node
originPoint	(0, 0, 0)
originIntercept	0
originPercent	0
originZone	0
originNode	0
normal	(0, -1, 0)
axisType	**YAxis**, XAxis, ZAxis, Arbitrary, ThetaPhi
upAxis	(0, 0, 1)
project2d	1
interactive	1
flip	0
originZoneDomain	0
originNodeDomain	0
meshName	“default”
theta	0
phi	0

5.5.73 SmoothOperator: *SmoothOperatorAttributes()*

Attribute	**Default**/Allowed Values
numIterations	20
relaxationFactor	0.01
convergence	0
maintainFeatures	1
featureAngle	45
edgeAngle	15
smoothBoundaries	0

5.5.74 SphereSlice: *SphereSliceAttributes()*

Attribute	**Default**/Allowed Values
origin	(0, 0, 0)
radius	1

5.5.75 Spreadsheet: *SpreadsheetAttributes()*

Attribute	**Default**/Allowed Values
subsetName	"Whole"
formatString	"%1.6f"
useColorTable	0
colorTableName	"Default"
showTracerPlane	1
tracerColor	(255, 0, 0, 150)
normal	**Z**, X, Y
sliceIndex	0
spreadsheetFont	"Courier,12,-1,5,50,0,0,0,0,0"
showPatchOutline	1
showCurrentCellOutline	0
currentPickType	0
currentPickLetter	""
pastPickLetters	()

5.5.76 StatisticalTrends: *StatisticalTrendsAttributes()*

Attribute	**Default**/Allowed Values
startIndex	0
stopIndex	1
stride	1
startTrendType	**Absolute**, Relative
stopTrendType	**Absolute**, Relative
statisticType	**Mean**, Sum, Variance, StandardDeviation, Slope, Residuals
trendAxis	**Step**, Time, Cycle
variableSource	**Default**, OperatorExpression

5.5.77 SubdivideQuads: *SubdivideQuadsAttributes()*

Attribute	**Default**/Allowed Values
threshold	0.500002
maxSubdivs	4
fanOutPoints	1
doTriangles	0
variable	"default"

5.5.78 Subset: *SubsetAttributes()*

Attribute	**Default**/Allowed Values
colorType	**ColorByMultipleColors**, ColorBySingleColor, ColorByColorTable
colorTableName	"Default"
invertColorTable	0
legendFlag	1
lineWidth	0
singleColor	(0, 0, 0, 255)
subsetNames	()
opacity	1
wireframe	0
drawInternal	0
smoothingLevel	0
pointSize	0.05
pointType	**Point**, Box, Axis, Icosahedron, Octahedron, Tetrahedron, SphereGeometry, Sphere
pointSizeVarEnabled	0
pointSizeVar	"default"
pointSizePixels	2

5.5.79 SurfaceNormal: *SurfaceNormalAttributes()*

Attribute	**Default**/Allowed Values
centering	**Point**, Cell

5.5.80 Tensor: *TensorAttributes()*

Attribute	**Default**/Allowed Values
glyphLocation	**AdaptsToMeshResolution**, UniformInSpace
useStride	0
nTensors	400
stride	1
origOnly	1
limitsMode	**OriginalData**, CurrentPlot
minFlag	0
min	0
maxFlag	0
max	1
colorByEigenValues	1
colorTableName	"Default"
invertColorTable	0
tensorColor	(0, 0, 0, 255)
useLegend	1
scale	0.25
scaleByMagnitude	1
autoScale	1
animationStep	0

5.5.81 Tessellate: *TessellateAttributes()*

Attribute	**Default**/Allowed Values
chordError	0.035
fieldCriterion	0.035
mergePoints	1

5.5.82 ThreeSlice: *ThreeSliceAttributes()*

Attribute	**Default**/Allowed Values
x	0
y	0
z	0
interactive	1

5.5.83 Threshold: *ThresholdAttributes()*

Attribute	**Default**/Allowed Values
outputMeshType	0
boundsInputType	0
listedVarNames	(“default”)
zonePortions	()
lowerBounds	()
upperBounds	()
defaultVarName	“default”
defaultVarIsScalar	0
boundsRange	()

5.5.84 Transform: *TransformAttributes()*

Attribute	**Default**/Allowed Values
doRotate	0
rotateOrigin	(0, 0, 0)
rotateAxis	(0, 0, 1)
rotateAmount	0
rotateType	**Deg**, Rad
doScale	0
scaleOrigin	(0, 0, 0)
scaleX	1
scaleY	1
scaleZ	1
doTranslate	0
translateX	0
translateY	0
translateZ	0

Continued on next page

Table 5.12 – continued from previous page

transformType	**Similarity**, Coordinate, Linear
inputCoordSys	**Cartesian**, Cylindrical, Spherical
outputCoordSys	**Spherical**, Cartesian, Cylindrical
continuousPhi	0
m00	1
m01	0
m02	0
m03	0
m10	0
m11	1
m12	0
m13	0
m20	0
m21	0
m22	1
m23	0
m30	0
m31	0
m32	0
m33	1
invertLinearTransform	0
vectorTransformMethod	**AsDirection**, NONE, AsPoint, AsDisplacement
transformVectors	1

5.5.85 TriangulateRegularPoints: *TriangulateRegularPointsAttributes()*

Attribute	**Default**/Allowed Values
useXGridSpacing	0
xGridSpacing	1
useYGridSpacing	0
yGridSpacing	1

5.5.86 Truecolor: *TruecolorAttributes()*

Attribute	**Default**/Allowed Values
opacity	1
lightingFlag	1

5.5.87 Tube: *TubeAttributes()*

Attribute	**Default**/Allowed Values
scaleByVarFlag	0
tubeRadiusType	**FractionOfBBox**, Absolute
radiusFractionBBox	0.01
radiusAbsolute	1
scaleVariable	"default"
fineness	5
capping	0

5.5.88 Vector: *VectorAttributes()*

Attribute	**Default**/Allowed Values
glyphLocation	**AdaptsToMeshResolution**, UniformInSpace
useStride	0
nVectors	400
stride	1
origOnly	1
limitsMode	**OriginalData**, CurrentPlot
minFlag	0
min	0
maxFlag	0
max	1
colorByMagnitude	1
colorTableName	"Default"
invertColorTable	0
vectorColor	(0, 0, 0, 255)
useLegend	1
scale	0.25
scaleByMagnitude	1
autoScale	1
glyphType	**Arrow**, Ellipsoid
headOn	1
headSize	0.25
lineStem	**Line**, Cylinder
lineWidth	0
stemWidth	0.08
vectorOrigin	**Tail**, Head, Middle
geometryQuality	**Fast**, High
animationStep	0

5.5.89 View: *ViewAttributes()*

Attribute	**Default**/Allowed Values
viewNormal	(0, 0, 1)
focus	(0, 0, 0)
viewUp	(0, 1, 0)
viewAngle	30
setScale	0
parallelScale	1
nearPlane	0.001
farPlane	100
imagePan	(0, 0)
imageZoom	1
perspective	1
windowCoords	(0, 0, 1, 1)
viewportCoords	(0.1, 0.1, 0.9, 0.9)
eyeAngle	2

5.5.90 View2D: *View2DAttributes()*

Attribute	**Default**/Allowed Values
windowCoords	(0, 1, 0, 1)
viewportCoords	(0.2, 0.95, 0.15, 0.95)
fullFrameActivationMode	**Auto**, On, Off
fullFrameAutoThreshold	100
xScale	**LINEAR**, LOG
yScale	**LINEAR**, LOG
windowValid	0

5.5.91 View3D: *View3DAttributes()*

Attribute	**Default**/Allowed Values
viewNormal	(0, 0, 1)
focus	(0, 0, 0)
viewUp	(0, 1, 0)
viewAngle	30
parallelScale	0.5
nearPlane	-0.5
farPlane	0.5
imagePan	(0, 0)
imageZoom	1
perspective	1
eyeAngle	2
centerOfRotationSet	0
centerOfRotation	(0, 0, 0)
axis3DScaleFlag	0
axis3DScales	(1, 1, 1)
shear	(0, 0, 1)
windowValid	0

5.5.92 ViewAxisArray: *ViewAxisArrayAttributes()*

Attribute	**Default**/Allowed Values
domainCoords	(0, 1)
rangeCoords	(0, 1)
viewportCoords	(0.15, 0.9, 0.1, 0.85)

5.5.93 ViewCurve: *ViewCurveAttributes()*

Attribute	**Default**/Allowed Values
domainCoords	(0, 1)
rangeCoords	(0, 1)
viewportCoords	(0.2, 0.95, 0.15, 0.95)
domainScale	**LINEAR**, LOG
rangeScale	**LINEAR**, LOG

5.5.94 Volume: *VolumeAttributes()*

Attribute	**Default**/Allowed Values
osprayShadowsEnabledFlag	0
osprayUseGridAcceleratorFlag	0
osprayPreIntegrationFlag	0
ospraySingleShadeFlag	0
osprayOneSidedLightingFlag	0

osprayAoTransparencyEnabledFlag	0
ospraySpp	1
osprayAoSamples	0
osprayAoDistance	100000
osprayMinContribution	0.001
legendFlag	1
lightingFlag	1
colorControlPoints.GetControlPoints(0).colors	(0, 0, 255, 255)
colorControlPoints.GetControlPoints(0).position	0
colorControlPoints.GetControlPoints(1).colors	(0, 255, 255, 255)
colorControlPoints.GetControlPoints(1).position	0.25
colorControlPoints.GetControlPoints(2).colors	(0, 255, 0, 255)
colorControlPoints.GetControlPoints(2).position	0.5
colorControlPoints.GetControlPoints(3).colors	(255, 255, 0, 255)
colorControlPoints.GetControlPoints(3).position	0.75
colorControlPoints.GetControlPoints(4).colors	(255, 0, 0, 255)
colorControlPoints.GetControlPoints(4).position	1
colorControlPoints.smoothing	**Linear**, NONE, CubicSpline
colorControlPoints.equalSpacingFlag	0
colorControlPoints.discreteFlag	0
colorControlPoints.tagNames	()
opacityAttenuation	1
opacityMode	**FreeformMode**, GaussianMode, ColorTableMode
	controlPoints does not contain any GaussianControlPoint objects.
resampleFlag	1
resampleTarget	1000000
opacityVariable	“default”
compactVariable	“default”
freeformOpacity	(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
useColorVarMin	0
colorVarMin	0
useColorVarMax	0
colorVarMax	0
useOpacityVarMin	0
opacityVarMin	0
useOpacityVarMax	0
opacityVarMax	0
smoothData	0
samplesPerRay	500
rendererType	**Default**, RayCasting, RayCastingIntegration, RayCastingSLIVR, RayCastingOSP
gradientType	**SobelOperator**, CenteredDifferences
scaling	**Linear**, Log, Skew
skewFactor	1
limitsMode	**OriginalData**, CurrentPlot
sampling	**Rasterization**, KernelBased, Trilinear
rendererSamples	3
lowGradientLightingReduction	**Lower**, Off, Lowest, Low, Medium, High, Higher, Highest
lowGradientLightingClampFlag	0
lowGradientLightingClampValue	1
materialProperties	(0.4, 0.75, 0, 15)

5.6 VisIt CLI Events

This chapter shows a table with all events that the VisIt GUI could potentially generate. Different plugins create different events, so the list will depend on the user configuration. The list in this section is generated from a call to the *GetCallbackNames()* function and will therefore list just the events that are applicable to the user that generates this documentation.

The list is alphabetically ordered. The left column, labeled *EventName* displays each event or callback name. The right column, labeled *ArgCount* displays the result of calling *GetCallbackArgumentCount(EventName)* for the corresponding event, which returns the number of arguments a callback function for that event should accept.

EventName	*ArgCount*
AMRStitchCellAttributes	1
ActivateDatabaseRPC	1
AddAnnotationObjectRPC	2
AddEmbeddedPlotRPC	1
AddInitializedOperatorRPC	1
AddOperatorRPC	2
AddPlotRPC	2
AddWindowRPC	0
AlterDatabaseCorrelationRPC	4
AnimationAttributes	1
AnimationPlayRPC	0
AnimationReversePlayRPC	0
AnimationSetNFramesRPC	1
AnimationStopRPC	0
AnnotationAttributes	1
ApplyNamedSelectionRPC	1
AxisAlignedSlice4DAttributes	1
BoundaryAttributes	1
BoundaryOpAttributes	1
BoxAttributes	1
CartographicProjectionAttributes	1
ChangeActivePlotsVarRPC	1
CheckForNewStatesRPC	1
ChooseCenterOfRotationRPC	2
ClearAllWindowsRPC	0
ClearCacheForAllEnginesRPC	0
ClearCacheRPC	2
ClearPickPointsRPC	0
ClearRefLinesRPC	0
ClearViewKeyframesRPC	0
ClearWindowRPC	1
ClipAttributes	1

Continued on next page

Table 5.14 – continued from previous page

CloneWindowRPC	0
CloseComputeEngineRPC	2
CloseDatabaseRPC	1
CloseRPC	0
ColorTableAttributes	1
ConeAttributes	1
ConnectToMetaDataServerRPC	2
ConnectedComponentsAttributes	1
ConstructDataBinningAttributes	1
ConstructDataBinningRPC	0
ContourAttributes	1
CoordSwapAttributes	1
CopyActivePlotsRPC	0
CopyAnnotationsToWindowRPC	2
CopyLightingToWindowRPC	2
CopyPlotsToWindowRPC	2
CopyViewToWindowRPC	2
CreateBondsAttributes	1
CreateDatabaseCorrelationRPC	4
CreateNamedSelectionRPC	1
CurveAttributes	1
CylinderAttributes	1
DDTConnectRPC	1
DDTFocusRPC	1
DataBinningAttributes	1
DatabaseMetaData	1
DeIconifyAllWindowsRPC	0
DeferExpressionAttributes	1
DeleteActiveAnnotationObjectsRPC	0
DeleteActivePlotsRPC	0
DeleteDatabaseCorrelationRPC	1
DeleteNamedSelectionRPC	1
DeletePlotDatabaseKeyframeRPC	2
DeletePlotKeyframeRPC	2
DeleteViewKeyframeRPC	1
DeleteWindowRPC	0
DemoteOperatorRPC	1
DetachRPC	0
DisableRedrawRPC	0
DisplaceAttributes	1
DrawPlotsRPC	1
DualMeshAttributes	1
EdgeAttributes	1
ElevateAttributes	1
EllipsoidSliceAttributes	1
EnableToolRPC	2
EnableToolbarRPC	2
ExplodeAttributes	1
ExportColorTableRPC	1
ExportDBAttributes	1

Continued on next page

Table 5.14 – continued from previous page

ExportDBRPC	0
ExportEntireStateRPC	1
ExportHostProfileRPC	1
ExportRPC	1
ExpressionList	1
ExternalSurfaceAttributes	1
ExtrudeAttributes	1
FFTAttributes	1
FileOpenOptions	1
FilledBoundaryAttributes	1
FluxAttributes	1
GetProcInfoRPC	3
GetQueryParametersRPC	1
GlobalAttributes	1
GlobalLineoutAttributes	1
HideActiveAnnotationObjectsRPC	0
HideActivePlotsRPC	0
HideAllWindowsRPC	0
HideToolbarsForAllWindowsRPC	0
HideToolbarsRPC	0
HistogramAttributes	1
IconifyAllWindowsRPC	0
ImportEntireStateRPC	2
ImportEntireStateWithDifferentSourcesRPC	3
IndexSelectAttributes	1
InitializeNamedSelectionVariablesRPC	1
IntegralCurveAttributes	1
InteractorAttributes	1
InverseGhostZoneAttributes	1
InvertBackgroundRPC	0
IsosurfaceAttributes	1
IsovolumeAttributes	1
KeyframeAttributes	1
LCSAttributes	1
LabelAttributes	1
LagrangianAttributes	1
LimitCycleAttributes	1
LineoutAttributes	1
LoadNamedSelectionRPC	1
LowerActiveAnnotationObjectsRPC	0
MaterialAttributes	1
MenuQuitRPC	1
MeshAttributes	1
MeshManagementAttributes	1
ModelFitAtts	1
MoleculeAttributes	1
MoveAndResizeWindowRPC	5
MovePlotDatabaseKeyframeRPC	3
MovePlotKeyframeRPC	3
MovePlotOrderTowardFirstRPC	1

Continued on next page

Table 5.14 – continued from previous page

MovePlotOrderTowardLastRPC	1
MoveViewKeyframeRPC	2
MoveWindowRPC	3
MultiCurveAttributes	1
MultiresControlAttributes	1
OnionPeelAttributes	1
OpenCLIClientRPC	1
OpenClientRPC	3
OpenComputeEngineRPC	2
OpenDatabaseRPC	4
OpenGUIClientRPC	1
OpenMDServerRPC	2
OverlayDatabaseRPC	1
ParallelCoordinatesAttributes	1
PersistentParticlesAttributes	1
PickAttributes	1
PlotDDTVispointVariablesRPC	1
PlotList	1
PoincareAttributes	1
PrintWindowRPC	0
PrinterAttributes	1
ProcessAttributes	1
ProcessExpressionsRPC	0
ProjectAttributes	1
PromoteOperatorRPC	1
PseudocolorAttributes	1
QueryAttributes	1
QueryOverTimeAttributes	1
QueryRPC	1
RadialResampleAttributes	1
RaiseActiveAnnotationObjectsRPC	0
ReOpenDatabaseRPC	2
ReadHostProfilesFromDirectoryRPC	1
RecenterViewRPC	0
RedoViewRPC	0
RedrawRPC	0
ReflectAttributes	1
ReleaseToDDTRPC	1
RemapAttributes	1
RemoveAllOperatorsRPC	0
RemoveLastOperatorRPC	0
RemoveOperatorRPC	1
RemovePicksRPC	1
RenamePickLabelRPC	1
RenderingAttributes	1
ReplaceDatabaseRPC	2
ReplicateAttributes	1
RequestMetaDataRPC	2
ResampleAttributes	1
ResetAnnotationAttributesRPC	0

Continued on next page

Table 5.14 – continued from previous page

ResetAnnotationObjectListRPC	0
ResetInteractorAttributesRPC	0
ResetLightListRPC	0
ResetLineoutColorRPC	0
ResetMaterialAttributesRPC	0
ResetMeshManagementAttributesRPC	0
ResetOperatorOptionsRPC	1
ResetPickAttributesRPC	0
ResetPickLetterRPC	0
ResetPlotOptionsRPC	1
ResetQueryOverTimeAttributesRPC	0
ResetViewRPC	0
ResizeWindowRPC	3
RevolveAttributes	1
SPHResampleAttributes	1
SaveNamedSelectionRPC	1
SaveViewRPC	0
SaveWindowAttributes	1
SaveWindowRPC	0
ScatterAttributes	1
SendSimulationCommandRPC	4
SetActivePlotsRPC	2
SetActiveTimeSliderRPC	1
SetActiveWindowRPC	1
SetAnimationAttributesRPC	0
SetAnnotationAttributesRPC	0
SetAnnotationObjectOptionsRPC	0
SetAppearanceRPC	0
SetBackendTypeRPC	1
SetCenterOfRotationRPC	1
SetCreateMeshQualityExpressionsRPC	1
SetCreateTimeDerivativeExpressionsRPC	1
SetCreateVectorMagnitudeExpressionsRPC	1
SetDefaultAnnotationAttributesRPC	0
SetDefaultAnnotationObjectListRPC	0
SetDefaultFileOpenOptionsRPC	0
SetDefaultInteractorAttributesRPC	0
SetDefaultLightListRPC	0
SetDefaultMaterialAttributesRPC	0
SetDefaultMeshManagementAttributesRPC	0
SetDefaultOperatorOptionsRPC	1
SetDefaultPickAttributesRPC	0
SetDefaultPlotOptionsRPC	1
SetDefaultQueryOverTimeAttributesRPC	0
SetGlobalLineoutAttributesRPC	0
SetInteractorAttributesRPC	0
SetKeyframeAttributesRPC	0
SetLightListRPC	0
SetMaterialAttributesRPC	0
SetMeshManagementAttributesRPC	0

Continued on next page

Table 5.14 – continued from previous page

SetNamedSelectionAutoApplyRPC	1
SetOperatorOptionsRPC	1
SetPickAttributesRPC	0
SetPlotDatabaseStateRPC	3
SetPlotDescriptionRPC	1
SetPlotFollowsTimeRPC	0
SetPlotFrameRangeRPC	3
SetPlotOptionsRPC	1
SetPlotOrderToFirstRPC	1
SetPlotOrderToLastRPC	1
SetPlotSILRestrictionRPC	0
SetPrecisionTypeRPC	1
SetQueryFloatFormatRPC	1
SetQueryOverTimeAttributesRPC	0
SetRemoveDuplicateNodesRPC	1
SetRenderingAttributesRPC	0
SetStateLoggingRPC	0
SetSuppressMessagesRPC	1
SetTimeSliderStateRPC	1
SetToolUpdateModeRPC	1
SetToolbarIconSizeRPC	0
SetTreatAllDBsAsTimeVaryingRPC	1
SetTryHarderCyclesTimesRPC	1
SetView2DRPC	0
SetView3DRPC	0
SetViewAxisArrayRPC	1
SetViewCurveRPC	0
SetViewExtentsTypeRPC	1
SetViewKeyframeRPC	0
SetWindowAreaRPC	1
SetWindowLayoutRPC	1
SetWindowModeRPC	1
ShowAllWindowsRPC	0
ShowToolbarsForAllWindowsRPC	0
ShowToolbarsRPC	0
SliceAttributes	1
SmoothOperatorAttributes	1
SphereSliceAttributes	1
SpreadsheetAttributes	1
StartPlotAnimationRPC	1
StatisticalTrendsAttributes	1
StopPlotAnimationRPC	1
SubdivideQuadsAttributes	1
SubsetAttributes	1
SuppressQueryOutputRPC	1
SurfaceNormalAttributes	1
TensorAttributes	1
TessellateAttributes	1
ThreeSliceAttributes	1
ThresholdAttributes	1

Continued on next page

Table 5.14 – continued from previous page

TimeSliderNextStateRPC	0
TimeSliderPreviousStateRPC	0
ToggleAllowPopupRPC	1
ToggleBoundingBoxModeRPC	0
ToggleCameraViewModeRPC	0
ToggleFullFrameRPC	0
ToggleLockTimeRPC	0
ToggleLockToolsRPC	0
ToggleLockViewModeRPC	0
ToggleMaintainViewModeRPC	0
TogglePerspectiveViewRPC	0
ToggleSpinModeRPC	0
TransformAttributes	1
TriangulateRegularPointsAttributes	1
TruecolorAttributes	1
TubeAttributes	1
TurnOffAllLocksRPC	0
UndoViewRPC	0
UpdateColorTableRPC	1
UpdateDBPluginInfoRPC	1
UpdateNamedSelectionRPC	1
VectorAttributes	1
View2DAttributes	1
View3DAttributes	1
ViewCurveAttributes	1
VolumeAttributes	1
WindowInformation	1
WriteConfigFileRPC	0

5.7 visit_utils

visit_utils is a pure python module distributed along with Visit's Python interface. It provides a simple interface to encode movie files and methods that wrap more complex VisIt Python command sequences to simplify a few common use cases. It also provides a stand alone PySide based annotation rendering API.

Here we provide details on the encoding and engine launching modules:

5.7.1 visit_utils.encoding

`visit_utils.encoding` provides methods that allow you to use movie encoders (e.g ffmpeg) to encode movies from sequences of image files and extract image files from movie files.

Methods:

visit_utils.encoding.**encode**(*ipattern*, *ofile*, *fdup=None*, *etype=None*, *stereo=False*, *input_frame_rate=None*, *output_frame_rate=None*)

Encodes a sequence of images into a movie.

Example Usage:

```
from visit_utils.encoding import *
encode("input.%04d.png","output.mpg")
encode("input.%04d.png","output.wmv",fdup=5)
encode("input.%04d.png","output.sm")
encode("input.%04d.png","output.sm",stereo=True)
```

Parameters

- **ipattern** – Input file pattern. Requires a printf style # format like "file%04d.png".
- **ofile** – Output file name
- **fdup** – Allows you to set an integer number of times to duplicate the input frames as they are passed to the encoder. (The duplication actually happens via symlinks) [Default = None]
- **etype** – Allows to select which encoder to use (If not passed the file extension is used to select an encoder) [Default = None]
- **input_frame_rate** – Allows you to set the input frame rate, in frames per second, that the encoder uses. [Default = None]
- **output_frame_rate** – Allows you to set the output frame rate, in frames per second, that the encoder uses. Note output formats typically only support a few output fps values. To obtain a perceived fps, the input_frame_rate is a better option to try. [Default = None]

visit_utils.encoding.**extract**(*ifile*, *opattern*)

Extracts a sequence of images from a a movie.

Example:

```
extract("movie.mpg","output%04d.png")
```

Parameters

- **ifile** – Input file.
- **opattern** – Output file pattern. Requires a printf style # format like "file%04d.png".

visit_utils.encoding.**encoders**()

Returns A list of strings of the available encoders.

5.7.2 visit_utils.engine

visit_utils.engine provides an interface to launch VisIt engines that uses installed host profiles.

Methods:

visit_utils.engine.**open**(*nprocs*, *method*, *ppn=1*, *part=None*, *bank=None*, *rtime=None*, *vdir=None*)

Launch VisIt compute engine on the current host.

Example usage:

Launch engine with 36 MPI tasks using default options for this host:

```
engine.open(nprocs=36)
```

Launch engine with 36 MPI tasks using a specific partition:

```
engine.open(nprocs=36, part="pbatch")
```

Launch engine with 36 MPI tasks, ask for 60 minute time limit:

```
engine.open(nprocs=36, rtime=60)
```

If you already have a slurm batch allocation, you can use:

```
engine.open(method="slurm")
```

This reads the `SLURM_JOB_NUM_NODES` and `SLURM_CPUS_ON_NODE` env vars and uses these values to launch with srun.

If you already have a lsf batch allocation, you can use:

```
engine.open(method="lsf")
```

This reads the `LSB_DJOB_NUMPROC` env var and uses it the to launch with mpirun.

Parameters

- **nprocs** – Number of MPI tasks
- **methods** – Launch Method (*srun*, etc)
- **ppn** – MPI tasks per node
- **part** – Partition
- **rtime** – Job time
- **vdir** – Path to VisIt install

visit_utils.engine.**close**(*ename=None*)

Closes VisIt's Compute Engine.

Parameters **ename** – Engine name to close (optional)

visit_utils.engine.**supported_hosts**()

Returns A list of the names of supported hosts.

5.8 Acknowledgments

This document is primarily based on the excellent manual put together by Brad Whitlock of Lawrence Livermore in 2005. Several years afterwards, the content from that manual was converted to serve as online help for the command line interpreter itself. As new routines were added, this online help was updated. In 2010, Jakob van Bethlehem of the University of Groningen wrote a wonderful script to convert the online help to manual form. In 2011, Hank Childs of Lawrence Berkeley merged the descriptions from Brad Whitlock's original manual with the function definitions produced by Jakob's conversion of the online help. In 2018, Alister Maguire of Lawrence Livermore wrote a script for converting this manual to restructuredText format to be used with Sphinx. The result is this manual.

CHAPTER

SIX

VISIT TUTORIALS

This manual contains a series of hands on tutorials that expose the user to the features in VisIt. The first three tutorials form a good basis for using VisIt, including the basics of using the Graphical User Interface (GUI), performing data analysis and using Python to script and automate tasks in VisIt. After that are a series of tutorials that cover advanced topics in detail.

The datasets for various tutorials can be found in VisIt's large data.

- Aneurysm tutorial data
- MRI tutorial data
- Potential Flow tutorial data
- VisIt Basics tutorial data

Contents:

6.1 VisIt Basics

6.1.1 Starting VisIt

The way you start VisIt depends on the platform you are on:

- On Windows, double click on the VisIt desktop icon
- On Mac, double click on the VisIt icon where you installed it (generally in the /Applications folder).
- On Unix, invoke: `/path/to/visit/bin/visit`
  - Most people ultimately put `/path/to/visit/bin` in their $PATH and then just say `visit`.

6.1.2 What you see

- The tall grey window on the left is called the **Graphical User Interface**, which will be refered to from here on as the *GUI*. It is the primary mechanism for driving VisIt.
- The window on the right is called the **visualization window**. It displays results.

6.1.3 Opening files

The first thing to do is to open files.

1. Download the tutorial data folder.

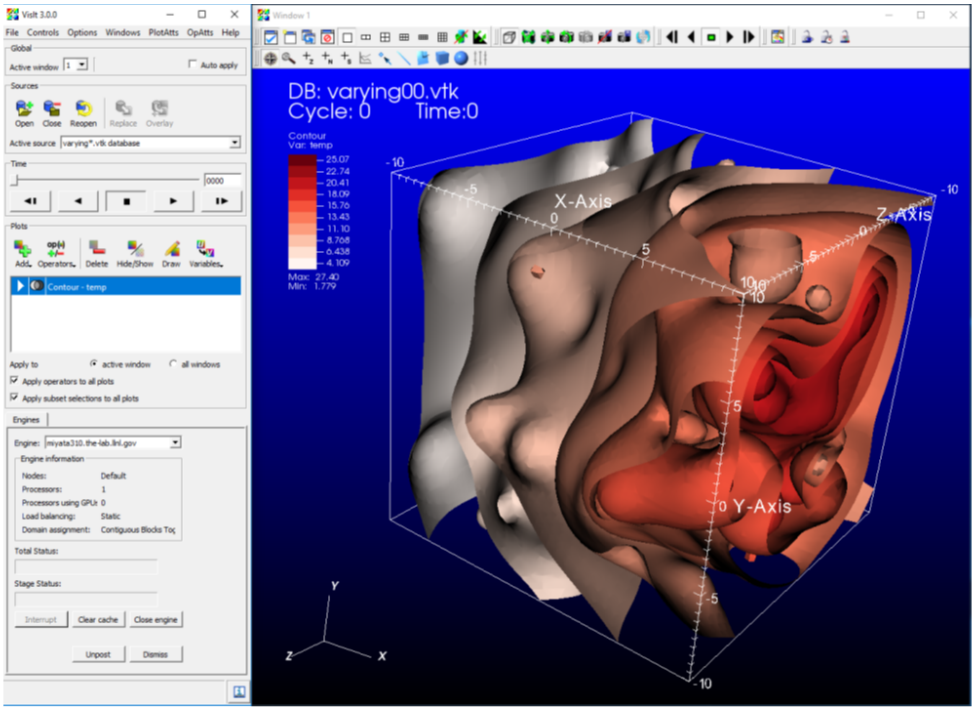


Fig. 6.1: The VisIt graphical user interface and visualization window

2. Go to the *GUI* and click on the *Open* icon.
3. This brings up the File open window.
4. Change the *Path* field to the "tutorial_data" folder.
5. Highlight the file "example.silo" and then click *OK*.

You've opened a file!

Advanced file opening features

1. In the File open window:
   - There is a field for *Host*. That is how you open a file on another system and run in client/server mode.
   - There is a *Filter*. That is provided to subset the file list to only the files VisIt may want.
     - Example filter: "*.silo *.vtk"
2. VisIt uses heuristics to determine the file type.
   - You can explicitly set the file type by setting the *Open file as type:* to the appropriate type.
3. You can also open files on the command line. For example, `visit -o file.ext` opens the file "file.ext".

6.1.4 Making a plot

1. Click on the *Add* icon to access various plots. This is located about half way down the Main window.
2. Select *Pseudocolor->temp* to add a Pseudocolor plot.
3. After adding a plot, you will see a green entry added to the "Plot list", which is located half way down the *GUI*.

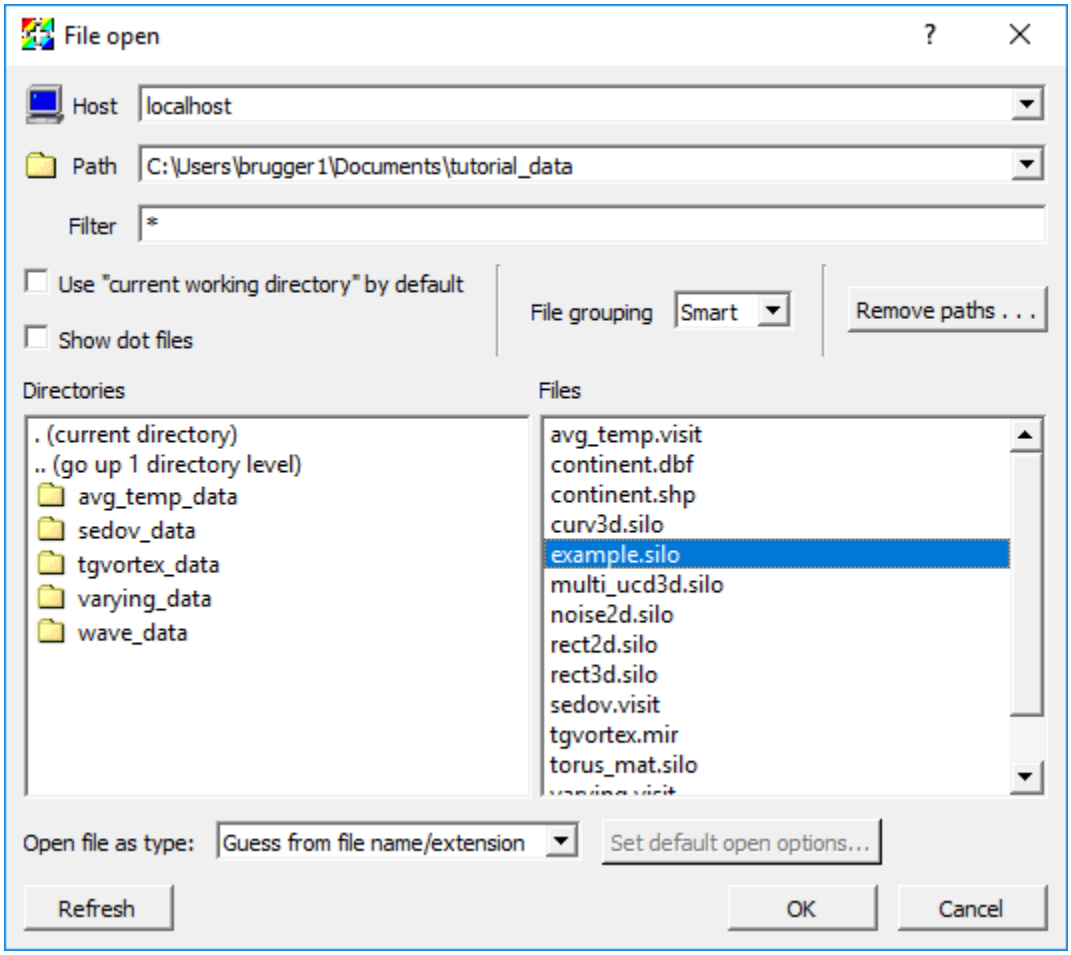


Fig. 6.2: The File open window

- This means VisIt will draw this plot after you click *Draw*.

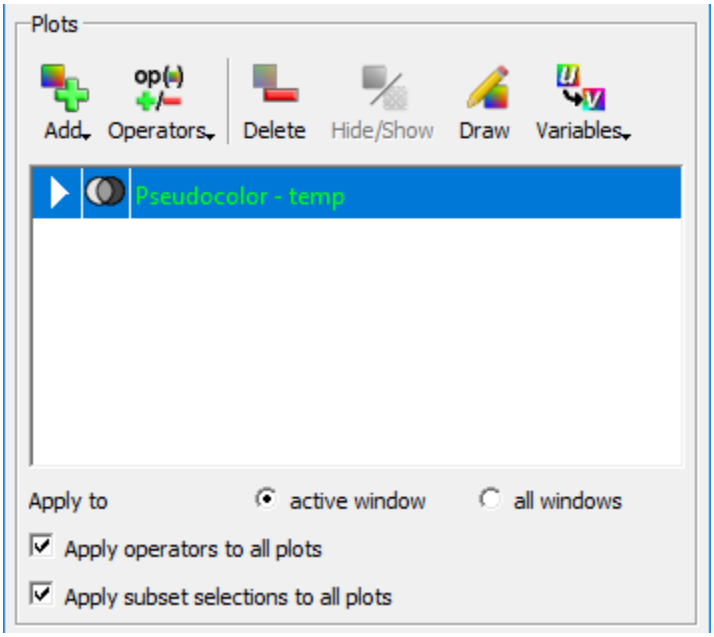


Fig. 6.3: The plot list with a Pseudocolor plot in it

4. Click *Draw*.
5. You should see a plot appear in the visualization window.
6. Go to *Add->Mesh->Mesh*.
7. Click *Draw*.
8. You should now see both a Pseudocolor and Mesh plot.
9. Highlight the Pseudocolor plot in the Plot list.
10. Click the *Hide/Show* button.
    - This will hide the Pseudocolor plot. You should now see only the Mesh plot.
11. Highlight the Mesh plot and click *Delete*.
    - You should now have an empty visualization window.
    - The Pseudocolor plot should now be selected.
12. Click *Hide/Show*.
    - The Pseudocolor plot should reappear.

6.1.5 Modifying the plot attributes

1. Go to *PlotAtts->Pseudocolor*. This is located in the menu bar at the top of the Main menu.
2. This brings up the Pseudocolor plot attributes window.
3. Change the *Scale* from *Linear* to *Log*.
4. Click *Apply*.

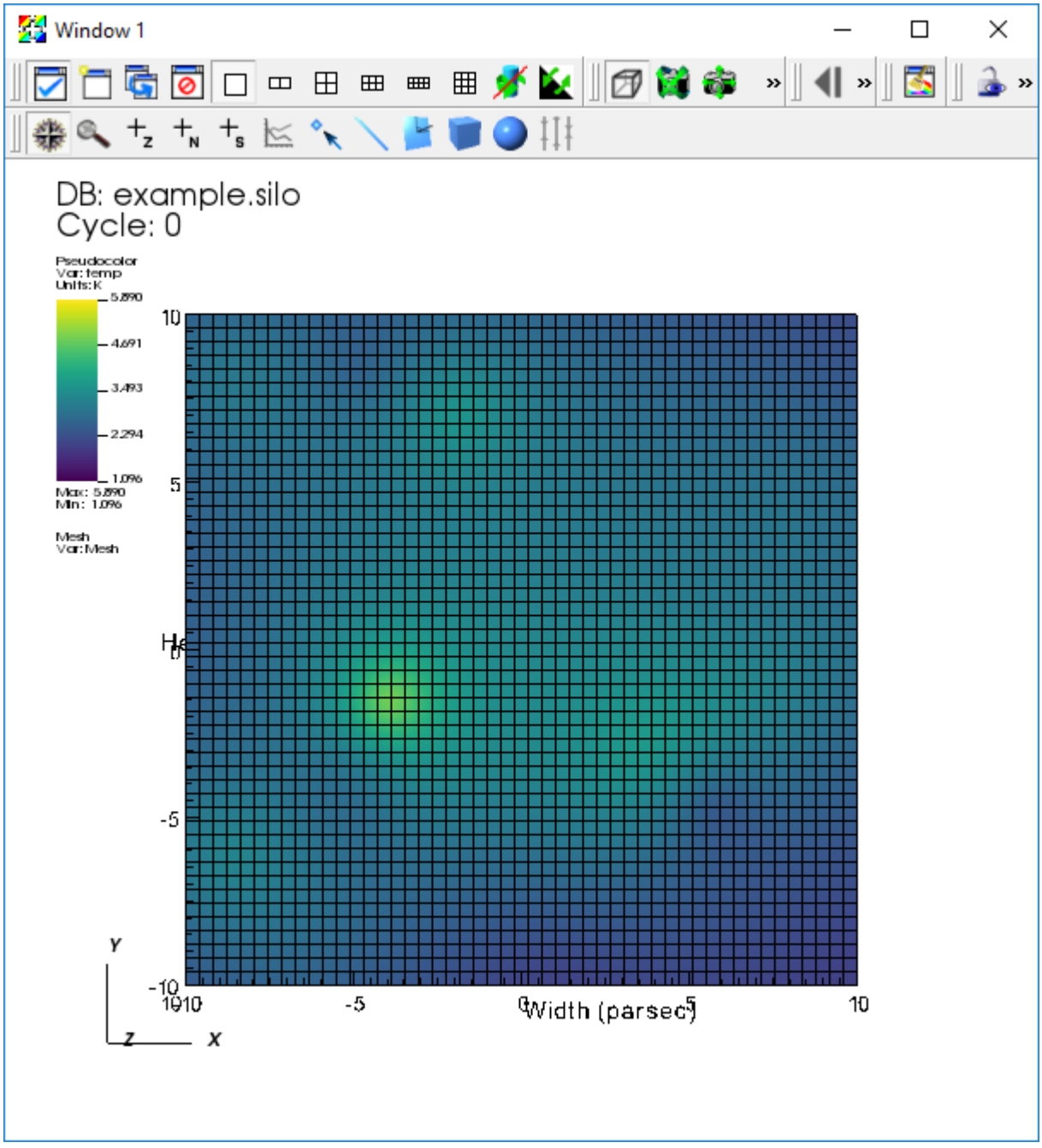


Fig. 6.4: A Pseudocolor and mesh plot displayed in a visualization window

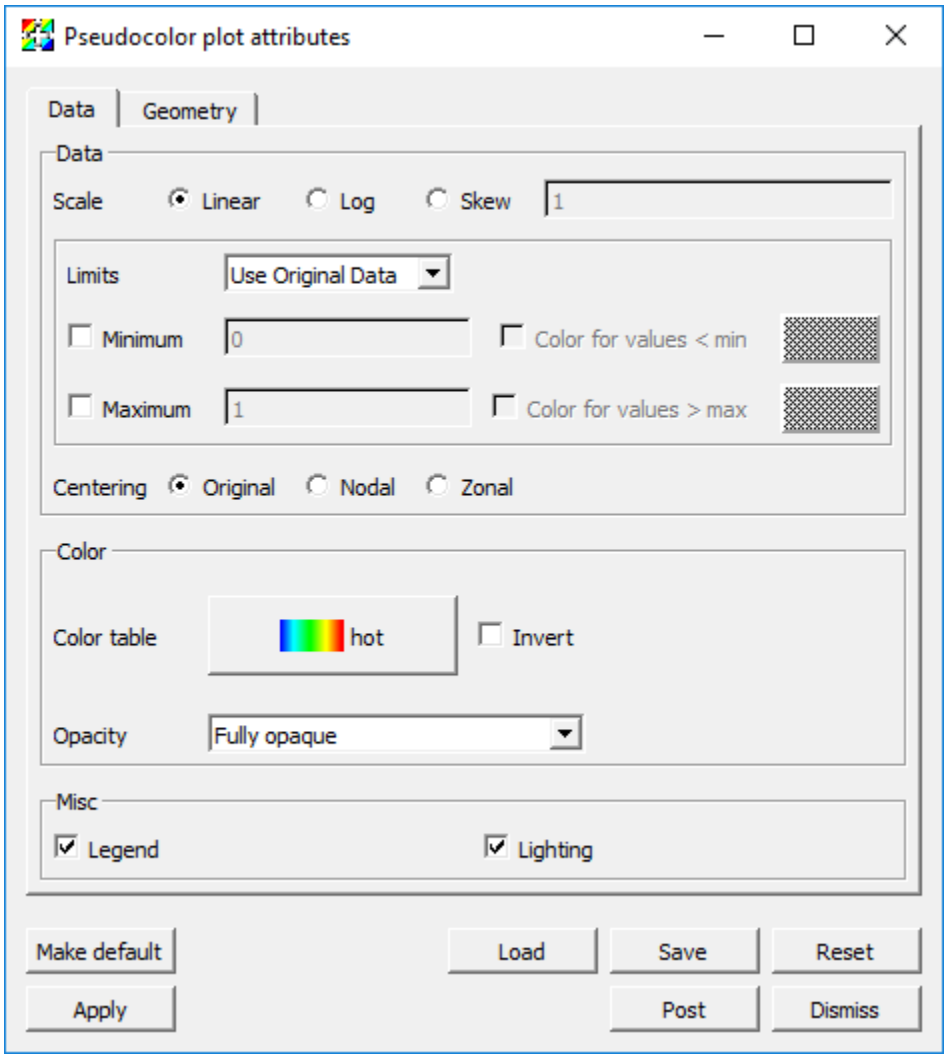


Fig. 6.5: The Pseudocolor plot attributes window

- The colors changed.

5. Click *Minimum* on and change the value to "3".
6. Click *Maximum* on and change the value to "4".
7. Click *Apply*.

- The colors change again.

8. Change the *Opacity* mode to *Constant*.

- Change the opacity slider to 50%.

9. Click *Apply*.

- You can now see through the plot. Note that you only see the external faces. If you want to see the data from the whole volume, that will be with the volume plot.

10. Change back the *Scale*, *Limits*, and *Opacity* back to their original settings and click *Apply*.
11. Dismiss the Pseudocolor plot attributes window.

6.1.6 Applying an operator

1. Click on the *Operators* button to access various operators. This is located next to the *Add* button.
2. Select *Slicing->Slice* to add a Slice operator.

- The visualization window will go blank and the Pseudocolor entry in the Plot list will turn green.
- This allows you to change the slice attributes before applying the Slice operator.
- We will apply the operator with the default attributes.

3. Click *Draw*.

- You are now looking at a 2D slice.

4. Go to *OpAtts->Slicing->Slice*.
5. This brings up the Slice operator attributes window.
6. There are many controls for setting the slice plane . . . play with them.
7. Operators can be removed by clicking on an expansion arrow in the Plot list, then clicking on the red X icon next to an operator.

6.1.7 VisIt interaction modes

There are six basic interaction modes:

1. Navigate
2. Zoom
3. Zone pick
4. Node pick
5. Spreadsheet pick
6. Lineout

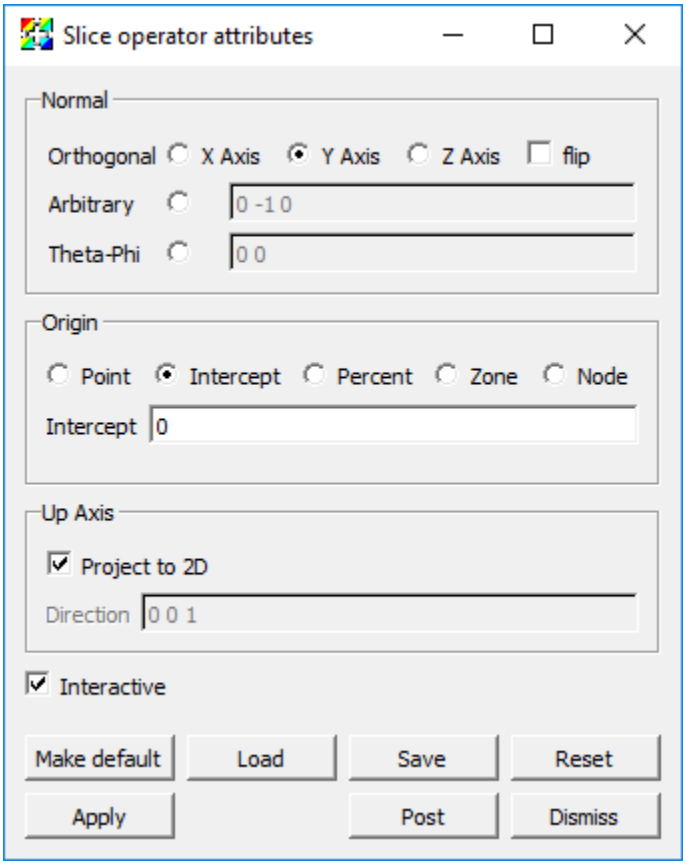


Fig. 6.6: The Slice operator attributes window

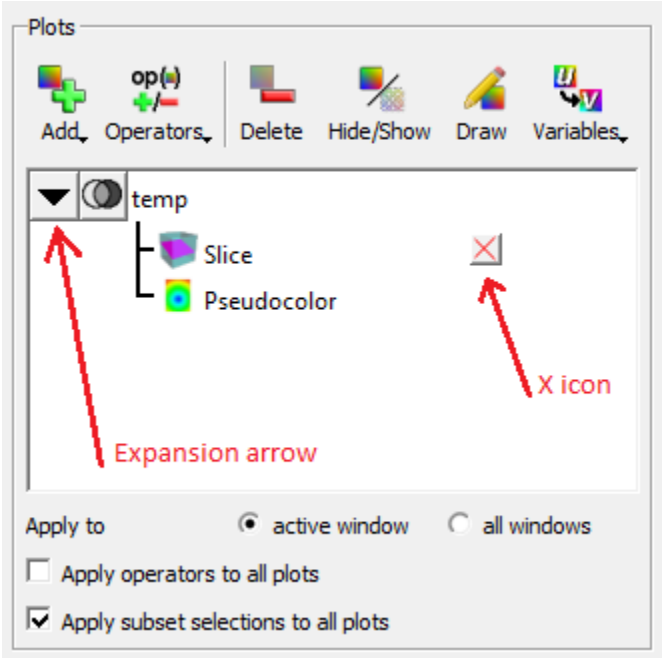


Fig. 6.7: The Expansion arrow and X icon in the Plot list

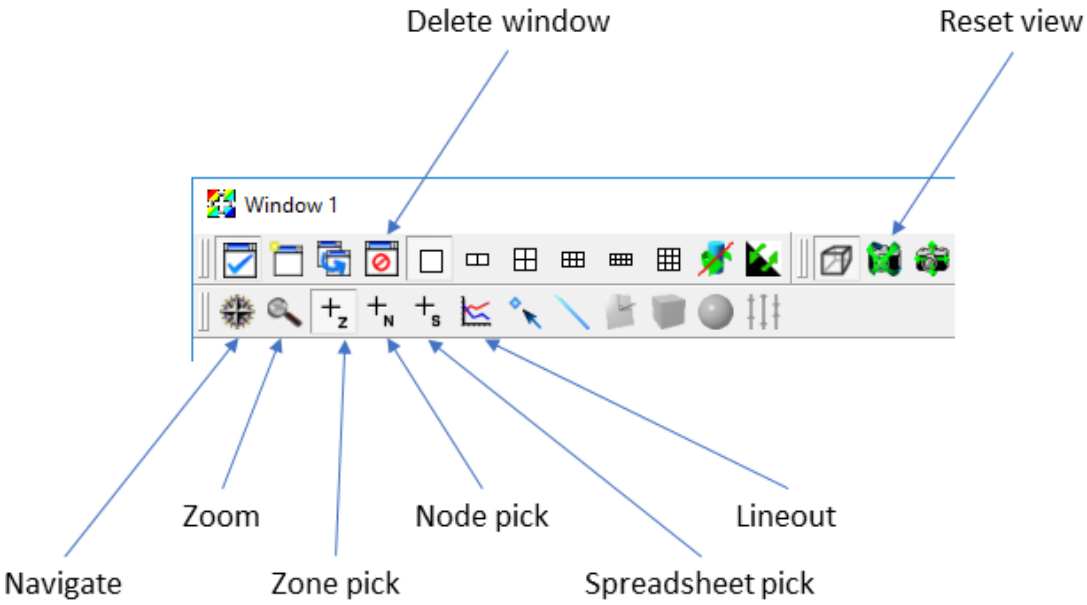


Fig. 6.8: The visualization tool bar with the icons for setting the interaction mode

The interaction mode is controlled by the toolbar, which is located at the top of the visualization window. The six interaction modes are all located together on the toolbar, towards the bottom.

The following descriptions apply to plots of 2D data. Before proceeding:

1. Select *Pseudocolor->temp* to add a Pseudocolor plot of *temp*.
2. Select *Operators->Slicing->Slice* to add a Slice operator.
3. Click *Draw*.
4. Click on the *Reset view* icon in the tool bar to reset the view. It is represented by a camera that has a green "X" around it (The camera is mostly obscured by the X).

Using navigate mode

You always start in Navigate mode. Navigate mode is indicated by the Navigate icon, represented by a compass, being indented. It allows you to pan and rotate the data set.

1. Put the cursor in the visualization window.
2. Left click (or single click if you do not have a 3 button mouse) and move the mouse.
3. The data set will pan with the mouse.

- In 3D, the data set rotates.

4. Middle click and move the mouse up and down. The data set will zoom in and out.

- In 3D, the data set will also zoom in and out.

Using zoom mode

Zoom mode is indicated by the Zoom icon, represented by a magnifying glass, being indented. It allows you to zoom the image by selecting a rectangular region.

1. Click on the Zoom icon.
2. Go to the visualization window and left click (single click) and HOLD IT DOWN.
3. Move the mouse a bit.

- You should see a rubber band.

4. Lift up the mouse button.

- You should now be zoomed in so that the viewport matches what was previously inside the rubber band.

Using lineout mode

Lineout mode is indicated by the Lineout icon, represented by a curve plot of red and blue curves, being indented. It allows the user to create a plot of a scalar variable as a function of distance along a line. Lineout is specific to 2D.

1. First, reset the view again.
2. Click on the Lineout icon.
3. Put the cursor over the data and left click (single click) and HOLD IT DOWN.
4. Move the mouse a bit.

- You should see a single line moving around.

5. Lift up the mouse button.
6. A new window will appear. The new window contains a "Lineout", which has *temp* as a function of distance over the line.
7. On the new window, find the Delete window icon, represented by a window with a red circle with a line through it.
8. Click this button.

- The new window will disappear and you should now have only one window.

Using pick mode

Pick mode is indicated by the Zone pick or Node pick icon, represented by a "+" with a small Z or a "+" with a small N, being indented. It allows the user to query a variable associated with a zone or node.

1. Click on the Zone pick icon.
2. Put the cursor over the data set and left click (single click).
3. This brings up the Pick window.

- The Pick window contains information about the zone (i.e. cell or element) that you just picked in.

Pick can return a lot more information than what it just did if you use the Pick window.

4. Go to the *Variables* drop down menu and select *Scalars/pressure*.
5. Turn on *Physical Coords* under *For Nodes*.
6. Turn on *Domain-Logical Coords* under *For Zones*.

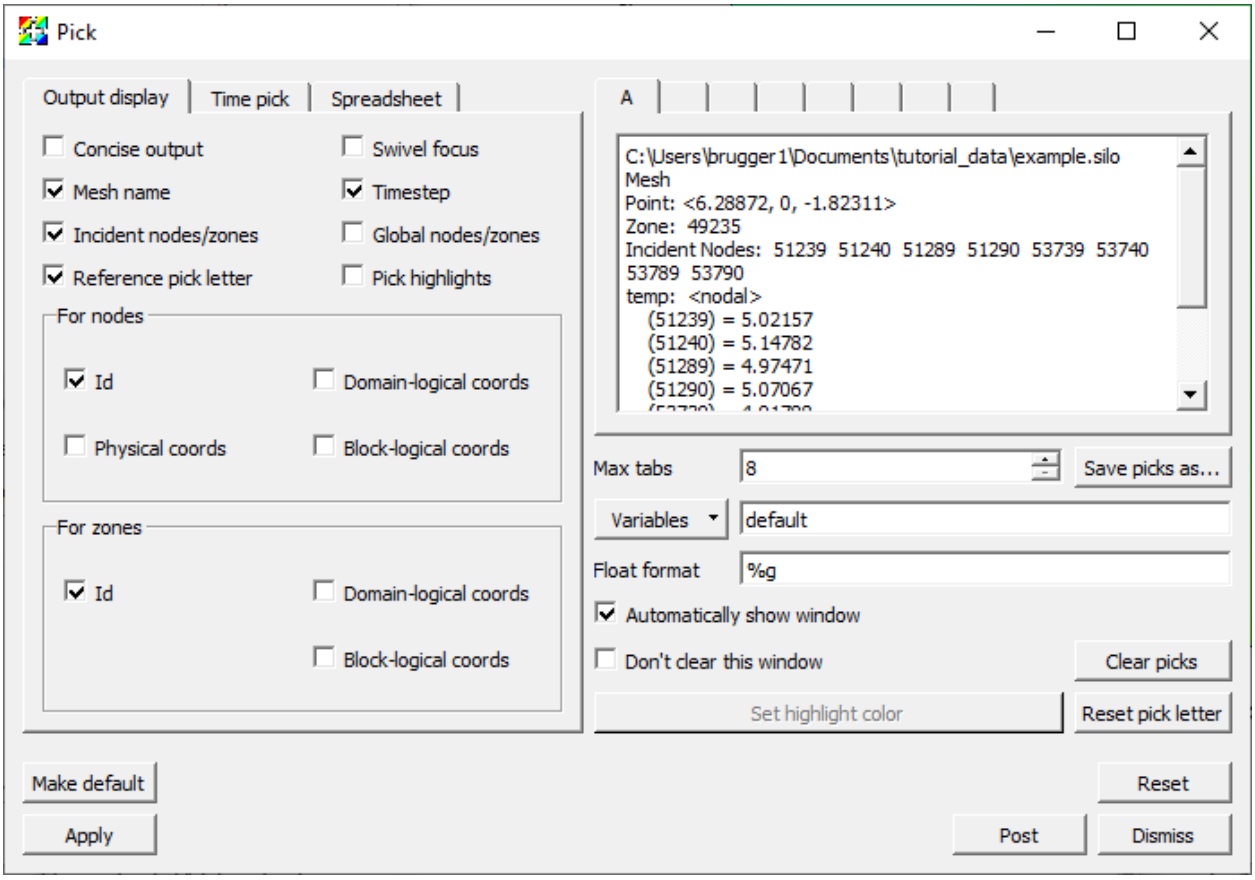


Fig. 6.9: The Pick output window

7. Click *Apply*.
8. Make another pick.

- You get information about pressure, the coordinates of each node, and the logical coordinates for the zone.

6.1.8 Other plots

1. We will experiment with the Contour, Filled Boundary, Label, Vector and Volume plots.

6.1.9 Other operators

1. We will experiment with the Clip and Threshold operators.

6.1.10 Saving an image

1. With a current plot, go to *File->Save window*.

- This saves an image to the filesystem.

On Windows, the default location for saved images is in *Documents/VisIt/My images*.

6.1.11 Saving a database

VisIt can be part of a larger tool chain.

1. If you do not already have one, make a Pseudocolor plot of temp from the "example.silo" database.
2. Apply the Threshold operator and change the range to be 3->max.
3. Click *Draw*.
4. Go to *File->Export database*.
5. This brings up the Export Database window.
6. Change *Export to* to *VTK*.
7. Be sure to set the output directory or the exported file will be written to the working directory *(on Windows that would be the directory where VisIt_ is installed)*.
8. Click *Export*.

- The Export options for VTK writer window will pop up at this point. It allows you to specify the options for the VTK writer. We will use the default options.

9. Click *Ok*.

- A file named "visit_ex_db.vtk" has been saved to the file system.

6.1.12 Subsetting

1. Delete any plots in your visualization window.
2. Open the file "multi_ucd3d.silo".
3. Make a Subset plot of "domains(mesh1)".

- The plot is colored by "domains", which normally correspond to a simulation's processors.

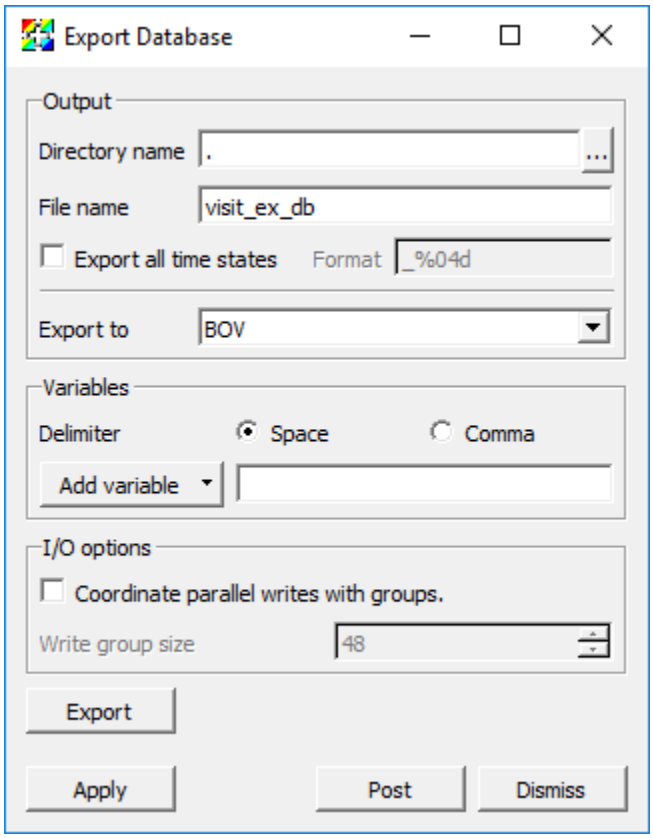


Fig. 6.10: The Export Database window

4. In the Plot list, find the overlapping transparent black and white ovals (like a Venn diagram) and click on it.
5. This brings up the Subset window.

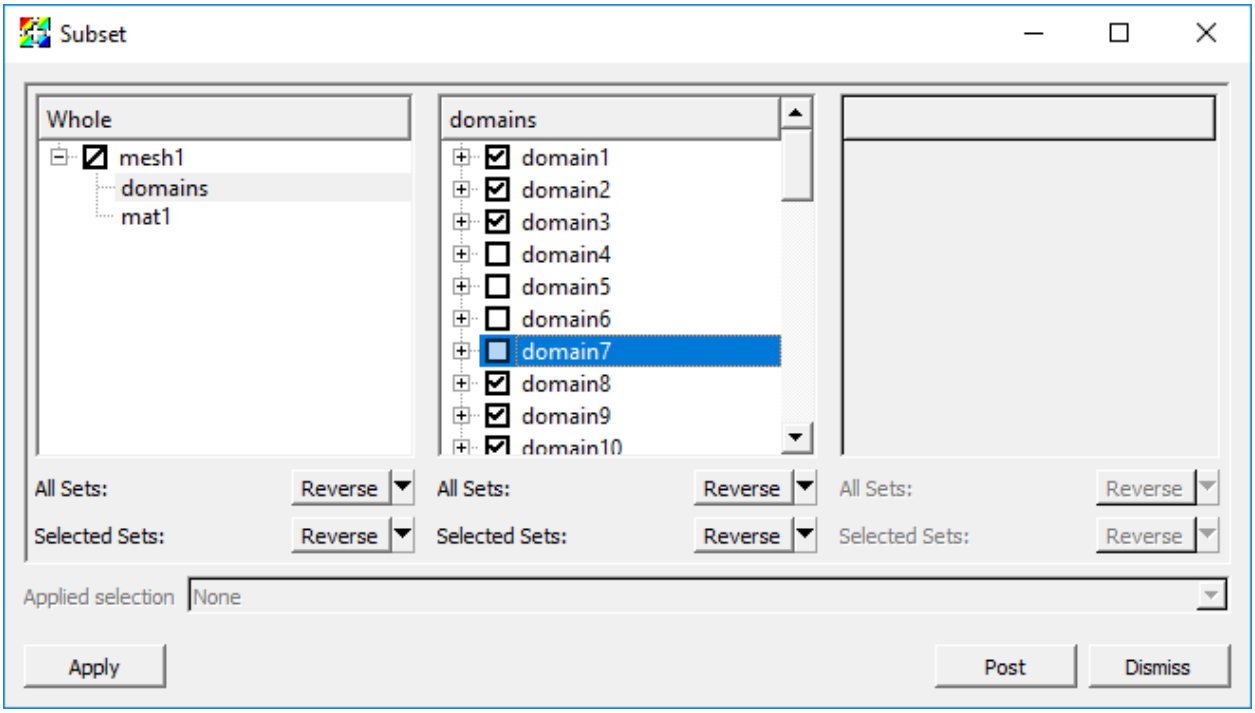


Fig. 6.11: The Subset window

6. Click on *domains* in the left most panel.
   - This will expand the list of domains in the center panel.
7. Turn off some domains and click *Apply*.
   - You will see some of the domains disappear.
   - Subsetting works with any plot type.
8. Turn all the domains back on.
9. Click on *mat1* in the left most panel.
   - This will expand the list of materials in the center panel.
10. Turn off materials 1 and 3.
    - You will see material 2 only, colored by domain.

This mechanism is used to expose subsetting for materials, domains, AMR levels, and other custom subsettable parts.

6.2 Data Analysis

This section describes two important abstractions in VisIt: Queries and Expressions.

6.2.1 Queries

What are queries

Queries are a mechanism for performing data analysis. Example use cases include querying for a number or curve that helps to describe the data set.

Experiment with queries

1. Go to *Controls->Query*.
2. This brings up the Query window.

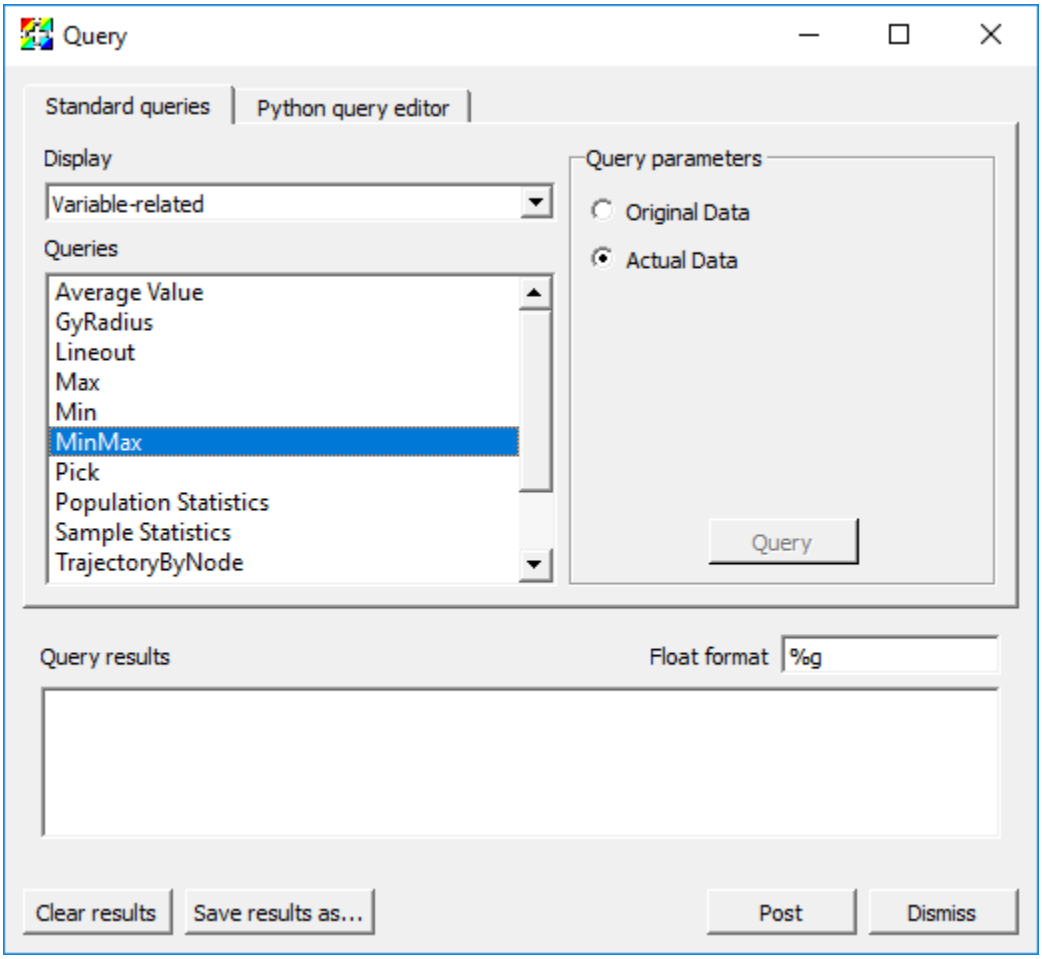


Fig. 6.12: The MinMax query

Variable-related

Variable related queries provide information about variables.

1. Change the *Display* in the Query window to be *Variable-related*.
2. Go back to the *GUI*, delete any plots, open up "example.silo", create a Pseudocolor plot of *temp*, and click *Draw*.

3. Highlight *MinMax* and click *Query*.

- The result will be displayed in the *Query results*. You will see the minimum and maximum values for the variable used by the highlighted plot in the Plot list and their logical and physical coordinates.

4. Apply the Slice operator to your plot.
5. Do another *MinMax* query.

- It gives you the different results. This is because the Query parameter *Actual Data* is selected. This means the answer will be the minimum and maximum constrained to the slice.

6. Change the Query parameter to be *Original Data*.
7. Do another *MinMax* query.

- This time the answer will match the result of the first query. It will display the minimum and maximum for what is in the file, not what is on the screen.

1. Now highlight *Variable Sum* and click *Query*.

- This will sum up the variable values for all cells using the plot highlighted in the Plot list.

2. Now highlight *Weighted Variable Sum* and click *Query*.

- This will sum up all of the values, but it will weight by area (since you have a slice).
- For 3D, it will weight by volume.
- For axi-symmetric 2D calculations, it will weight by revolved volume.

3. Note that both queries have options for doing queries over time (grayed out because we don't have a time varying data set).

- This is for time varying data and will produce a curve in a separate window.

1. Now highlight *Lineout*.

- Note that you must have left *Project to 2D* enabled in the Slice operator for this next one to work correctly.

2. Change the start point to "-5 -5 0" and the end point to "5 5 0".
3. Click *Query*.

- The Lineout query samples data values along a line, producing a 1D dataset from datasets of greater dimension. It creates a new instance of the highlighted plot in the Plot list and copies the plot to another vis window.

4. This is a way to get exact lineouts.
5. You can also take 3D lineouts this way.

1. Now highlight *Pick*.
2. Click *Query*.

- This will provide information about the zone containing the coordinate "0 0 0".

3. Change the mode to *Pick using coordinate to determine node*.
4. Click *Query*.

- This will provide information about the node nearest the coordinate "0 0 0".

5. Change the mode to *Pick using domain and element Id*.
6. Click *Query*.

- This will provide information about the node or zone in the specied domain.

You can also perform a query using the global element id by selecting *Pick using global element Id*. This only works if the file contains global element id information, which this file does not.

Mesh-related

1. Change the *Display* in the Query window to be *Mesh-related*.
2. Experiment with the *2D area*, *SpatialExtents*, *NumZones*, and *Zone Center* queries.

- For the *Zone Center* query, you will set the *Domain* to "0".
- The domain is used for when you have a parallel file, where the data has been "domain decomposed" for parallel processing.

ConnectedComponents related

1. If you haven't already removed the slice operator, do that now, so you have just a Pseudocolor plot of *temp*.
2. Apply the Isovolume operator. Change the *Lower bound* of the Isovolume operator attributes to be "4".
3. You will now see a bunch of blobs in space.
4. Change the *Display* in the Query window to be *ConnectedComponents-related*.
5. Perform the *Number of Connected Components* query.

- It should tell you that there are 15 components.

6. Apply the Clip operator with the default settings.
7. Perform the *Number of Connected Components* query again.

- It should now say there are 14 components.
- Operators affect queries.

6.2.2 Queries over Time

What are queries over time

Queries over time perform analysis through time and generate a time-curve.

Experiment with queries over time

Weighted Variable Sum

1. Go to *Controls->Query*.
2. This brings up the Query window.
3. Go back to the *GUI*, delete any existing plots, open up "wave.visit", and make a Pseudocolor plot of *pressure*.
4. Find and Highlight *Weighted Variable Sum* and click *Do Time Query*.
5. Options for changing the *Starting timestep*, *Ending timestep* and *Stride* will be available.

- Note that these are 0-origin timestate indices and not cycles or times.

6. Click *Query*.

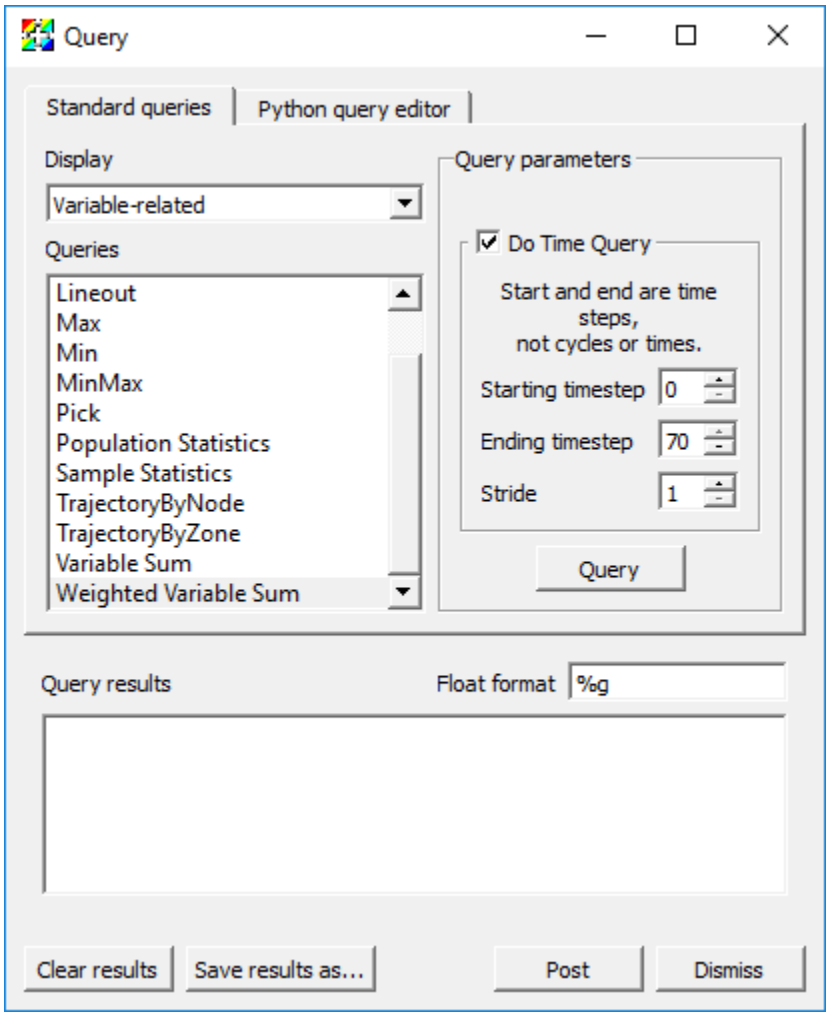


Fig. 6.13: The Weighted Variable Sum query

- The result will be displayed in a new Window. By default the x-axis will be cycle and the y-axis will be the weighted summation of the *pressure*.

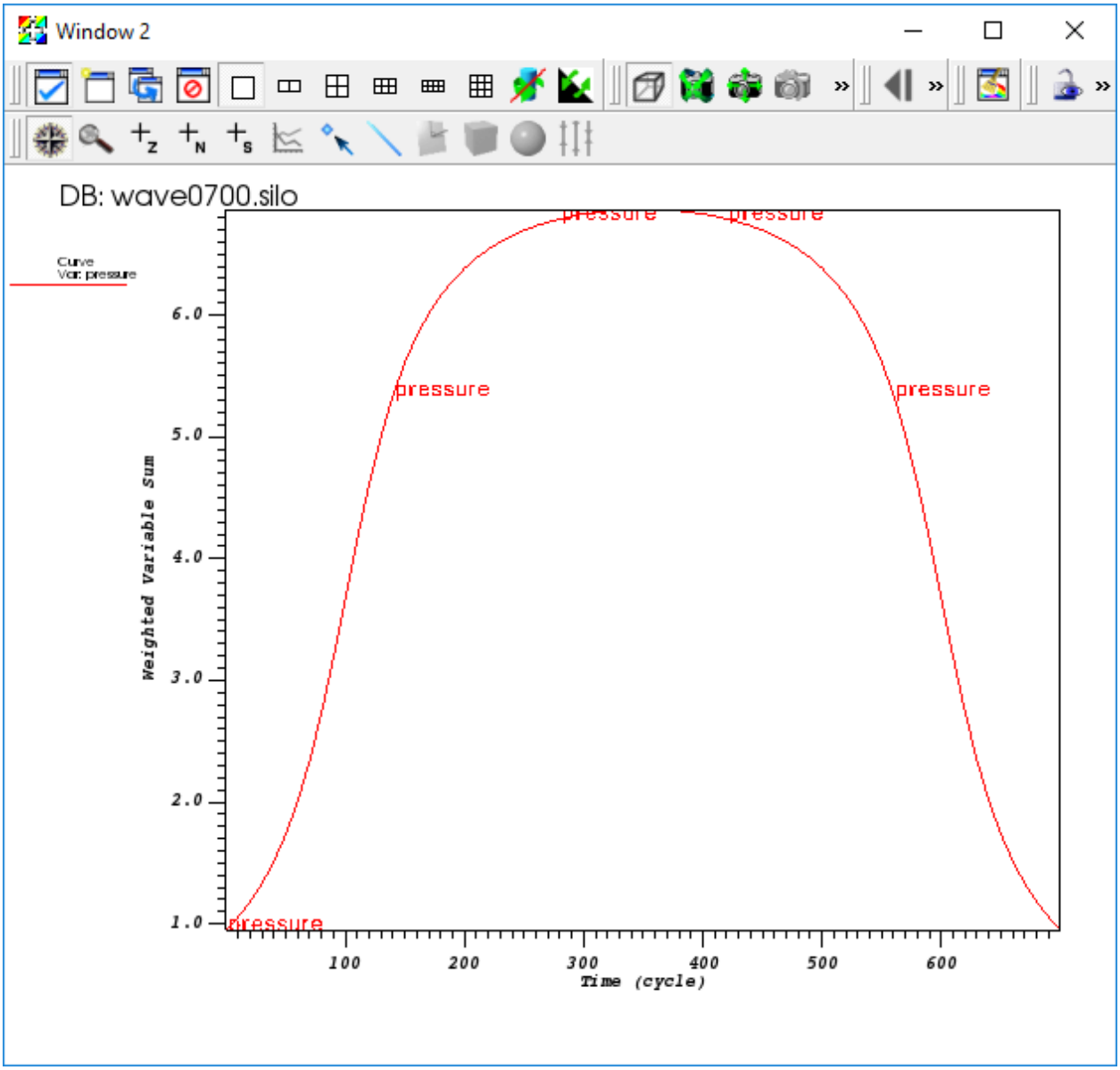


Fig. 6.14: The output of the Weighted Variable Sum query over time

Pick

1. Pick can do multiple-variable time curves.
2. Make *Window 2* active, delete the plot, and make *Window 1* active again.
3. Find and Highlight *Pick* in the Query window and click *Do Time Query* to enable time-curve options.
4. Change the *Variables* option to add *v* using the *Variables->Scalars* dropdown menu.
5. Select *Pick using domain and element Id*. Leave the defaults for *Node Id* and *Domain Id* as "0".

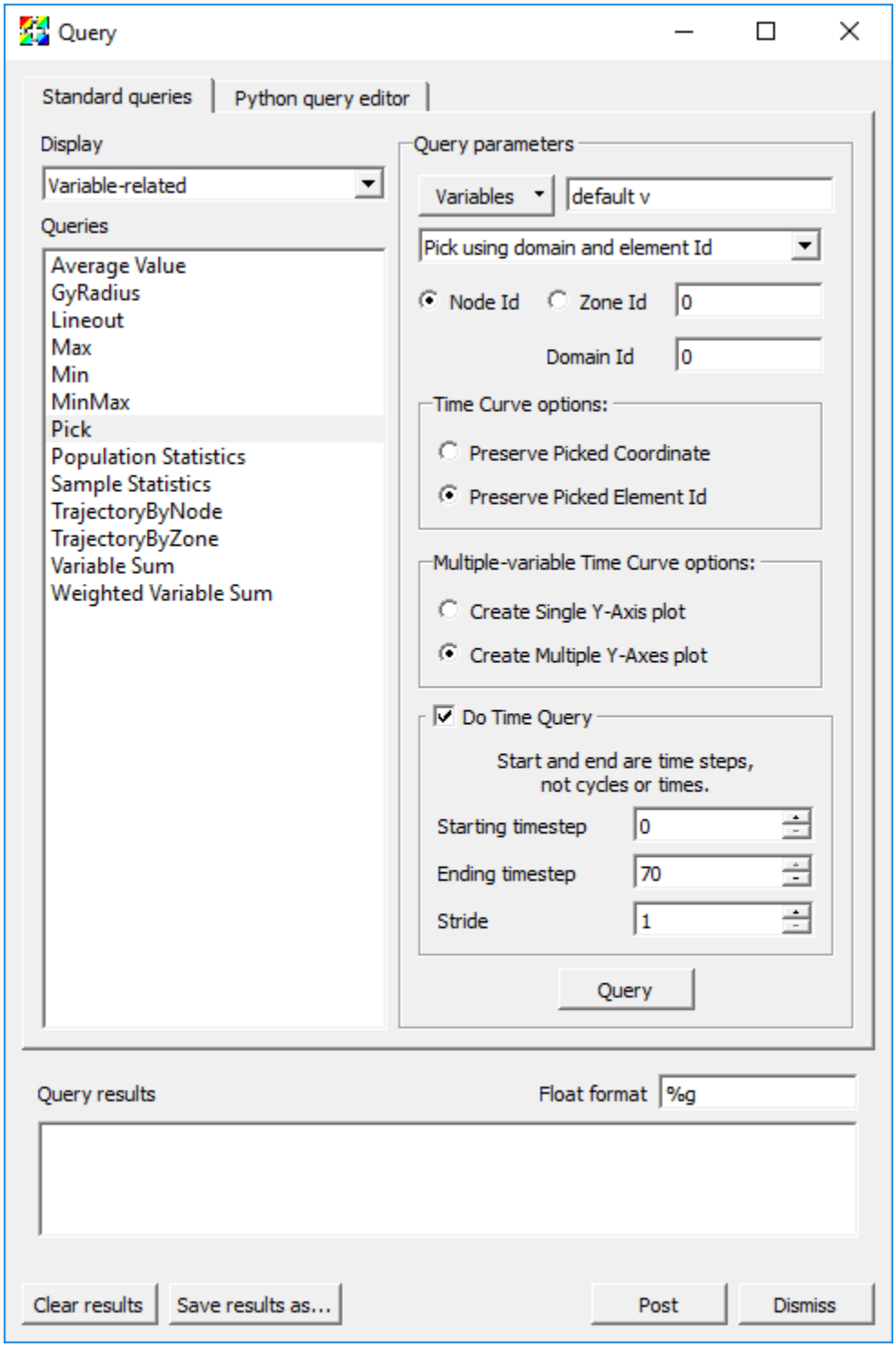


Fig. 6.15: The Pick query

6. Select *Preserve Picked Element Id*.
7. Click *Query*.

- The result will be two curves in a single xy plot.

8. Make *Window 2* active, delete the plot, and make *Window 1* active again.
9. Change the *Multiple-variable Time Curve options* to *Create Multiple Y-Axes plot*.
10. Click *Query*.

- The result will be a Multi-curve plot (multiple axes) in *Window 2*.

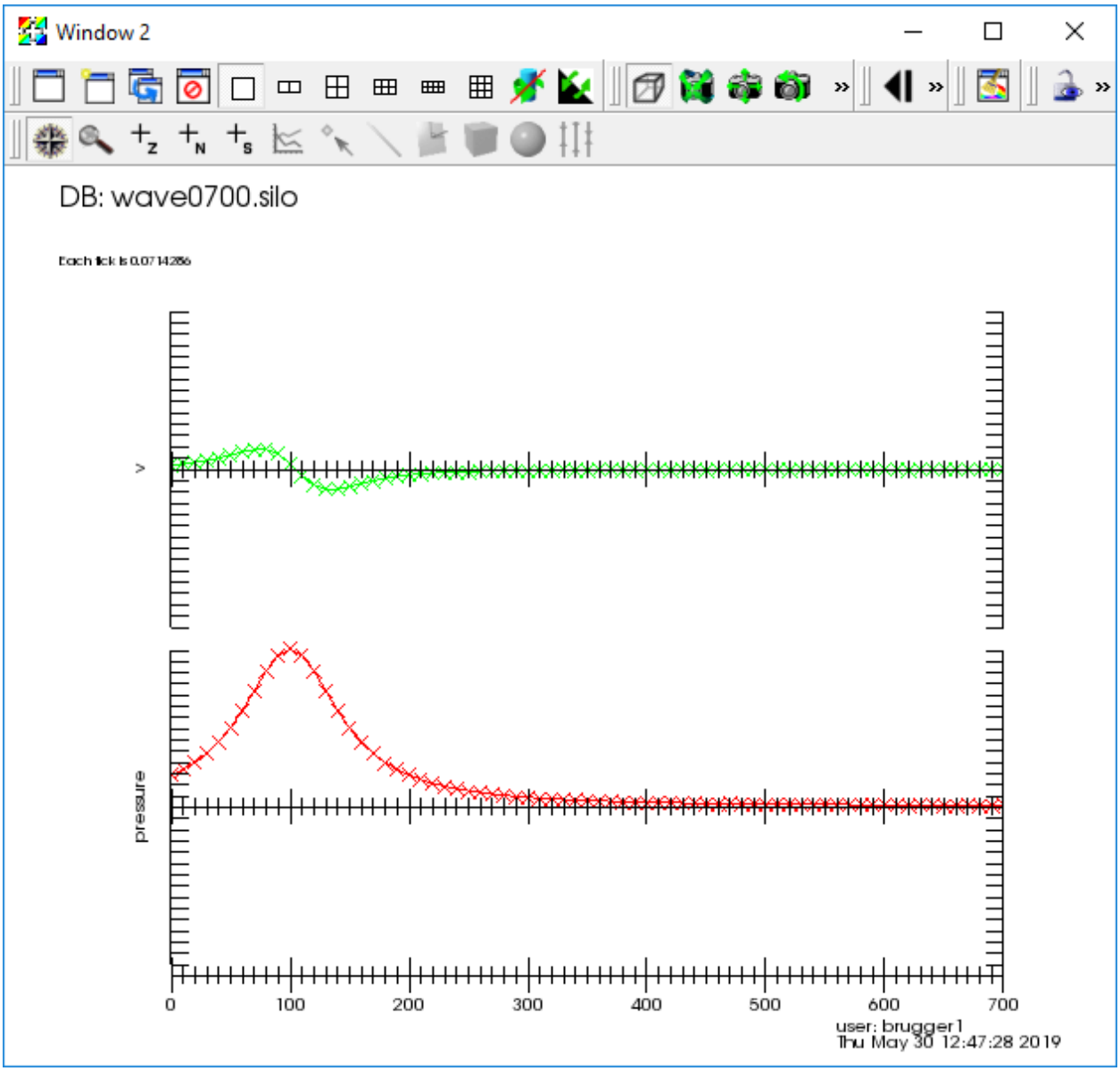


Fig. 6.16: The Pick query output

11. **NOTE:** Time Pick can also be performed via the mouse by first setting things up on the *Time Pick* tab in the Pick window (*Controls->Pick*).

Changing global options

1. Go to *Controls->Query over time options*.
2. This brings up the QueryOverTime window.

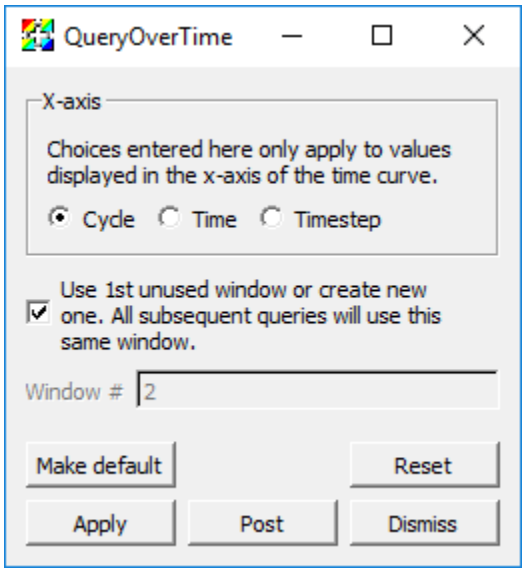


Fig. 6.17: The QueryOverTime window

3. Here you can change the values displayed in the x-axis for all subsequent queries over time.
4. You can also change the window used to display time-curves. By default, the first un-used window becomes the time-curve window, and all subsequent time-curves are generated in the same window.

6.2.3 Built-in queries

VisIt provides a wide variety of built-in queries. See explanations of these here: *Built-in queries*

6.2.4 Expressions

Expressions in VisIt create new mesh variables from existing ones. These are also known as *derived quantities*. VisIt's expression system supports only derived quantities that create a new mesh variable defined over the *entire* mesh. Given a mesh on which a variable named *pressure* is defined, an example of a very simple expression is "2*pressure". On the other hand, suppose one wanted to sum (or integrate) "pressure" over the entire mesh (maybe the mesh represents some surface area over which a force calculation is desired). Such an operation is not an expression in VisIt because it does not result in a new variable defined over the entire mesh. In this example, summing pressure over the entire mesh results in a single, scalar, number, like "25.6". Such an operation is supported instead by VisIt's Variable Sum Query. This tends to be true in general; Expressions define whole mesh variables while Queries define single numerical values (there are, however, some Queries for which this is not strictly true).

A simple algebraic expression, "2*radial"

1. Open up "noise2d.silo".
2. Create a Pseudocolor plot of the variable *radial*.

- Take note of the legend range, “0. . . 28.28”

3. Go to *Controls->Expressions*.
4. Click on *New* in the bottom left.

- This will create an expression and give it a default name, “unnamed1”.

5. Rename this expression by typing “radial2” into the *Name* field

- Take note of the *Type* of the variable. By default, VisIt assumes the type of the new variable you are creating is a s scalar mesh variable (e.g. a single numerical value for each node or zone/cell in the mesh). Here, we are indeed creating a scalar variable and so there is no need to adjust the *Type*. However, in some of the examples that follow, we’ll be creating vector mesh variables and if we don’t specify the correct type, we’ll get an error message.

6. Place the cursor in the *Definition* pane of the *Expressions* dialog.
7. Type the number “2” followed by the C/C++ language symbol for multiplication, “*”.
8. Now, you can either type the name “radial” or you can go to the *Insert Variable. . .* pulldown menu and find and select the *radial* variable there (see picture at right).

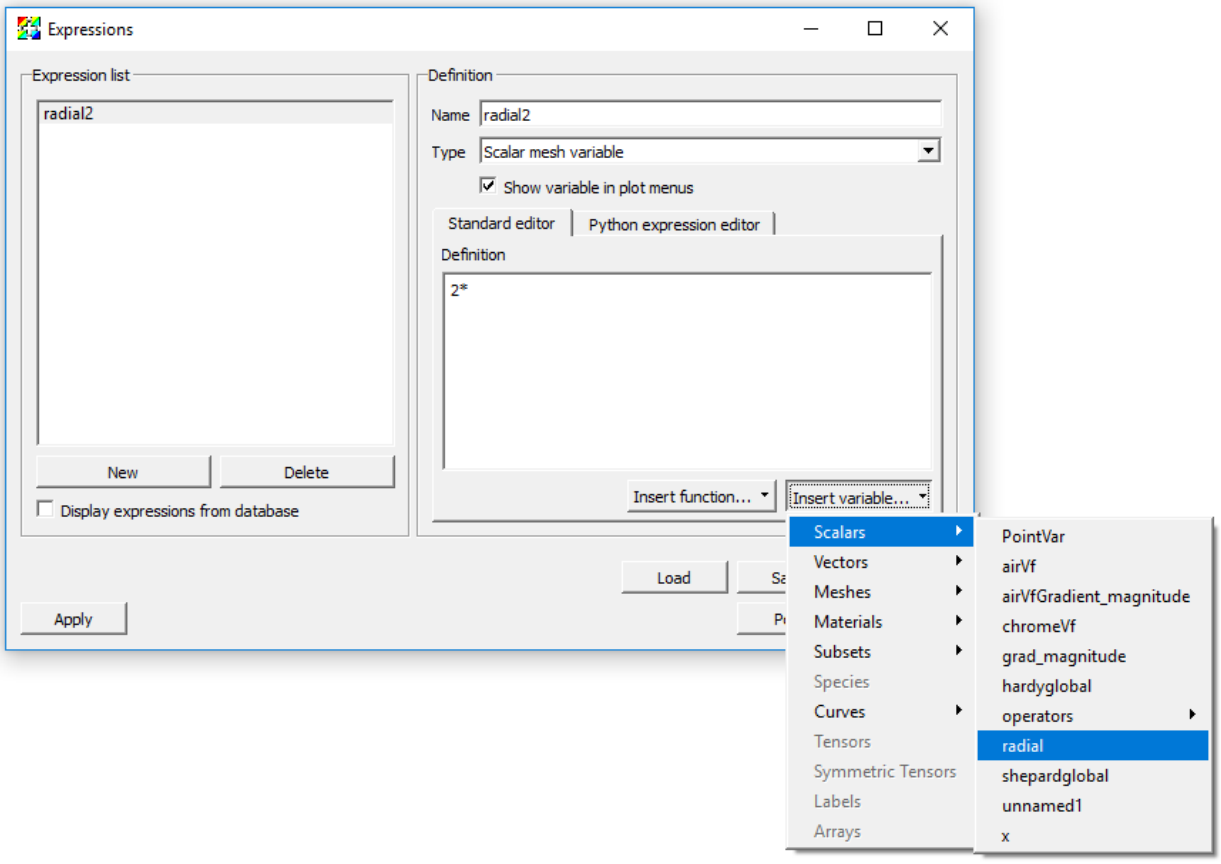


Fig. 6.18: Using the Expressions window Insert variable

9. Click *Apply*.
10. Now, go to the main VisIt *GUI* Panel to the *Variables* pulldown.

- Note that *radial2* now appears in the list of variables there.

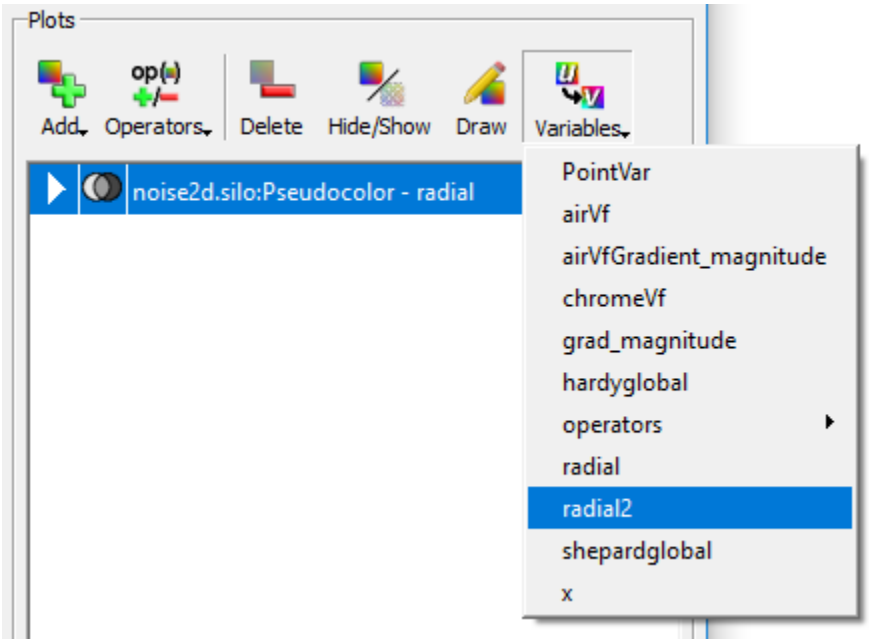


Fig. 6.19: Expression variable appears in the plot menus

11. Select *radial2* from the pull down and click *Draw*.

- Visually, the image will not look any different. But, if you take a close look at the legend you will see it is now showing "0. . . 56.57".

Visit supports several unary and binary algebraic expressions including `+`, `-`, `/`, `\*`, `bitwise-^`, `bitwise-&`, `sqrt()`, `abs()`, `ciel()`, `floor()`, `ln()`, `log10()`, `exp()` and more.

Accessing coordinates (of a mesh) in expressions

Here, we'll use the category of *Mesh* expressions to access the coordinates of a mesh, again, working with "noise2d.silo".

1. Go to *Controls->Expressions*.
2. Click the *New* button and name this expression "Coords".
3. Set the *Type* to *Vector mesh variable* (because coordinates, at least in this 2D example, are a vector quantity).
4. Put the cursor in the *Definition* pane.
5. Go to *Insert Function. . .* and find the *Mesh* category of expressions and then, within it, find the *coord* function expression.

- This should result in the insertion of "coord()" in the *Definition* pane and place the cursor between the two parenthesis characters.
- Note that in almost all cases, the category of *Mesh* expressions expect one or more mesh variables as operands.

6. Now, go to *Insert Variable. . .* pull down and then to the *Meshes* submenu and select *Mesh*.

- This should result in *Mesh* being inserted between the parentheses in the definition.

7. Click *Apply*.

8. Now, we'll define two scalar expressions for the "X" and "Y" coordinates of the mesh. While still in the Expressions window,
    1. Click *New*.
    2. Name the new expression "X".
    - Note that VisIt's expression system is case sensitive so "x" and "X" can be different variable names.
    3. Leave the type as *Scalar mesh variable*
    4. Type into the definition pane, "Coords[0]"
    - This expression uses the array bracket dereference operator "[]" to specify a particular component of an array. In this case, the *array* being derefrenced is the vector variable defined by "Coords".
    - Note that VisIt's expression system always numbers its array indices starting from zero.
    5. Click *Apply*.
    6. Now, repeat these steps to define a "Y" expression for the "Y" coordinates as "Coords[1]".
9. Finally, we'll define the "distance" expression

1. Click the *New* button.
2. Give the new variable the name "Dist" (Type should be *Scalar mesh variable*).
3. Type in the definition "sqrt(X*X+Y*Y)".
4. Click *Apply*.

Now, we'll use the new "Dist" variable we've just defined to display some data.

1. Delete any existing plots from the plot list.
2. Add a Pseudocolor plot of *shepardglobal*.
3. Add an Isovolume operator.
- Although this example is a 2D example and so *volume* doesn't seem to apply, VisIt's Isovolume operator performs the equivalent operation for 2D data.
4. Bring up the Isovolume operator attributes (either expand the plot by clicking on the triangle to the left of its name in the plot list and double clicking on the Isovolume operator there or go to the *OpAtts* menu and bring up Isovolume operator attributes that way).
5. Set the variable to *Dist*.
6. Set the *Lower bound* to "5" and the *Upper bound* to "7".
7. Click *Apply*.
8. Click *Draw*.

You should get the picture below. In this picture, we are displaying a Pseudocolor plot of *shepardglobal*, but Isovolumed by our *Dist* expression in the range "[5...7]".

This example also demonstrates the use of an expression *function*, *coord()* to operate on a mesh and return its coordinates as a vector variable on the mesh.

VisIt has a variety of expression functions that operate on a Mesh including *area* (for 2D meshes), *volume* (for 3D meshes, *revolved_volume* (for 2D cylindrically symmetric meshes), *zonetype*, and more. In addition, VisIt includes the entire suite of *Mesh quality* expressions from the Verdict Library.

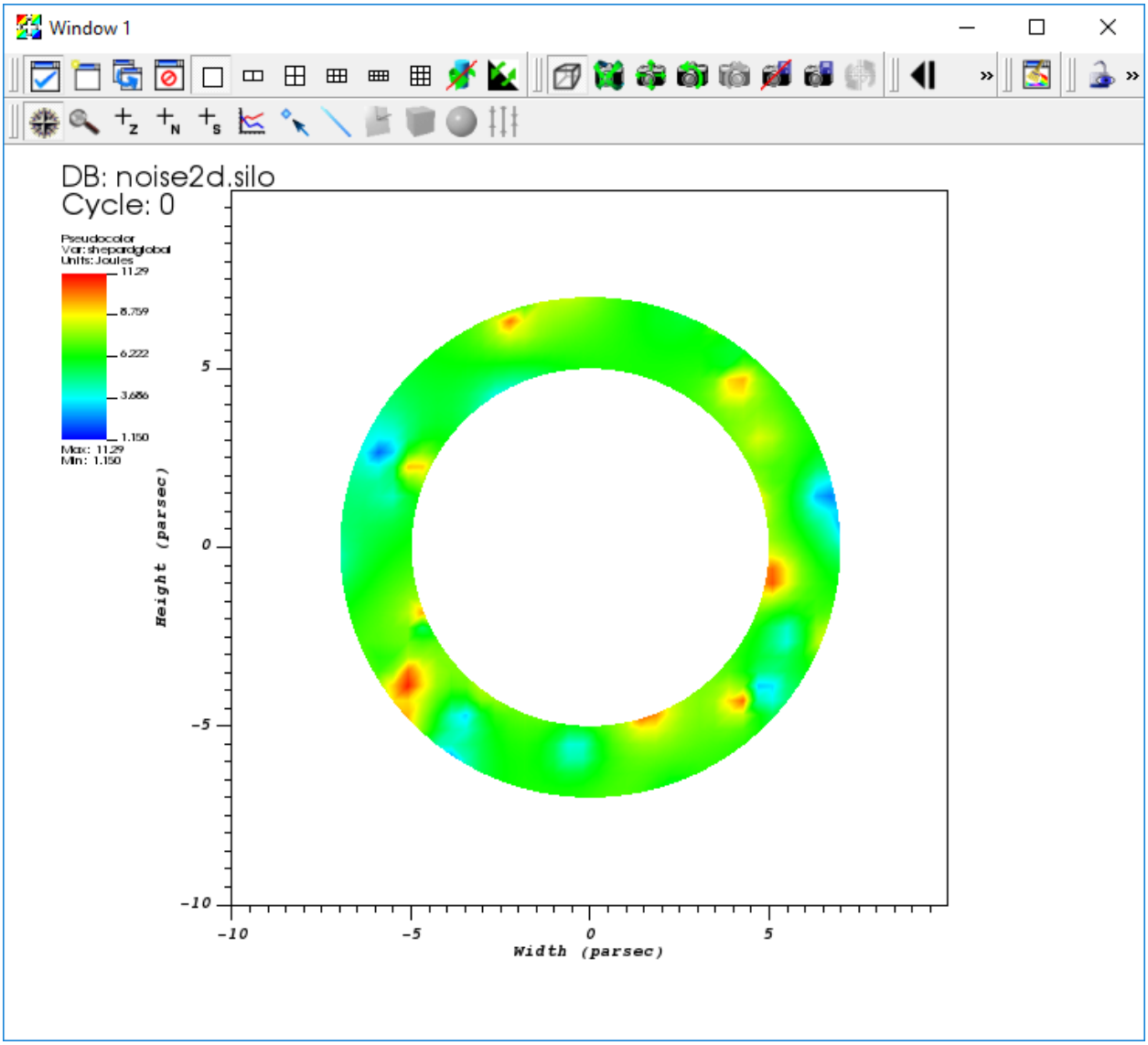


Fig. 6.20: Example of using the radial expression

Creating vector and tensor valued variables from scalars

If the database contains scalar variables representing the individual components of a vector or tensor, VisIt's Expression system allows you to construct the associated vector (or tensor). You create vectors in VisIt's Expression system using the curly bracket *vector compose* "{}" operator. For example, using "noise2d.silo" again as an example, suppose we want to compose a *Vector* valued expression that has "shepardglobal" and "hardyglobal" as components. Here are the steps.

1. Go to *Controls->Expressions*.
2. Click the *New* button and set *Name* to "randvec".
3. Be sure to also set the *Type* to *Vector mesh variable*.
4. Place the cursor in *Definition* pane and type "{shepardglobal, hardyglobal}".
5. Click *Apply*.
6. Go to *Plots->Vector*.
   - You should now see *randvec* appear there as a variable name to plot.
7. Add the Vector plot of *randvec*.

In the example above, we used the *vector compose* operator, "{}" to create a vector variable from multiple scalar variables. We can do the same to create a tensor variable. Recall from calculus that a rank 0 tensor is a scalar, a rank 1 tensor is a vector and a rank 2 tensor is a matrix. So, to create a tensor variable, we use multiple *vector compose* operators nesting within another *vector compose* operator. Here, solely for the purposes of illustration (e.g. this isn't a physically meaningful tensor) we'll use the "X" and "Y" coordinate component scalars we defined earlier together with the *shepardglobal* and *hardyglobal*.

1. Go to *Controls->Expressions*.
2. Click *New* and set the *Name* to "tensor".
3. Be sure to also set the *Type* to *Tensor mesh variable*.
4. Place the cursor in *Definition* pane and type "{ {shepardglobal, hardyglobal}, {X,Y} }".
   - Note the two levels of curly braces. The outer level is the whole rank 2 tensor matrix and the inner curly braces are each row of the matrix.
   - Note that you could also have defined the same tensor expression using two vector expressions like so, "{randvec, Coords}".
5. Click *Apply*.
6. Add a Tensor plot of *tensor* variable.

Variable compatibility gotchas (tensor rank, centering, mesh)

VisIt will allow you to define expressions that it winds up determining to be invalid later when it attempts to execute those expressions. Some common issues are the mixing of incompatible mesh variables in the same expression *without* the necessary additional functions to make them compatible.

Tensor rank compatibility

For example, what happens if you mix scalar and vector mesh variables (e.g. variables of different *Tensor rank*) in the same expression? Again, using "noise2d.silo".

1. Define the expression, "foo" as "grad+shepardglobal" with the *Type Vector mesh variable*.

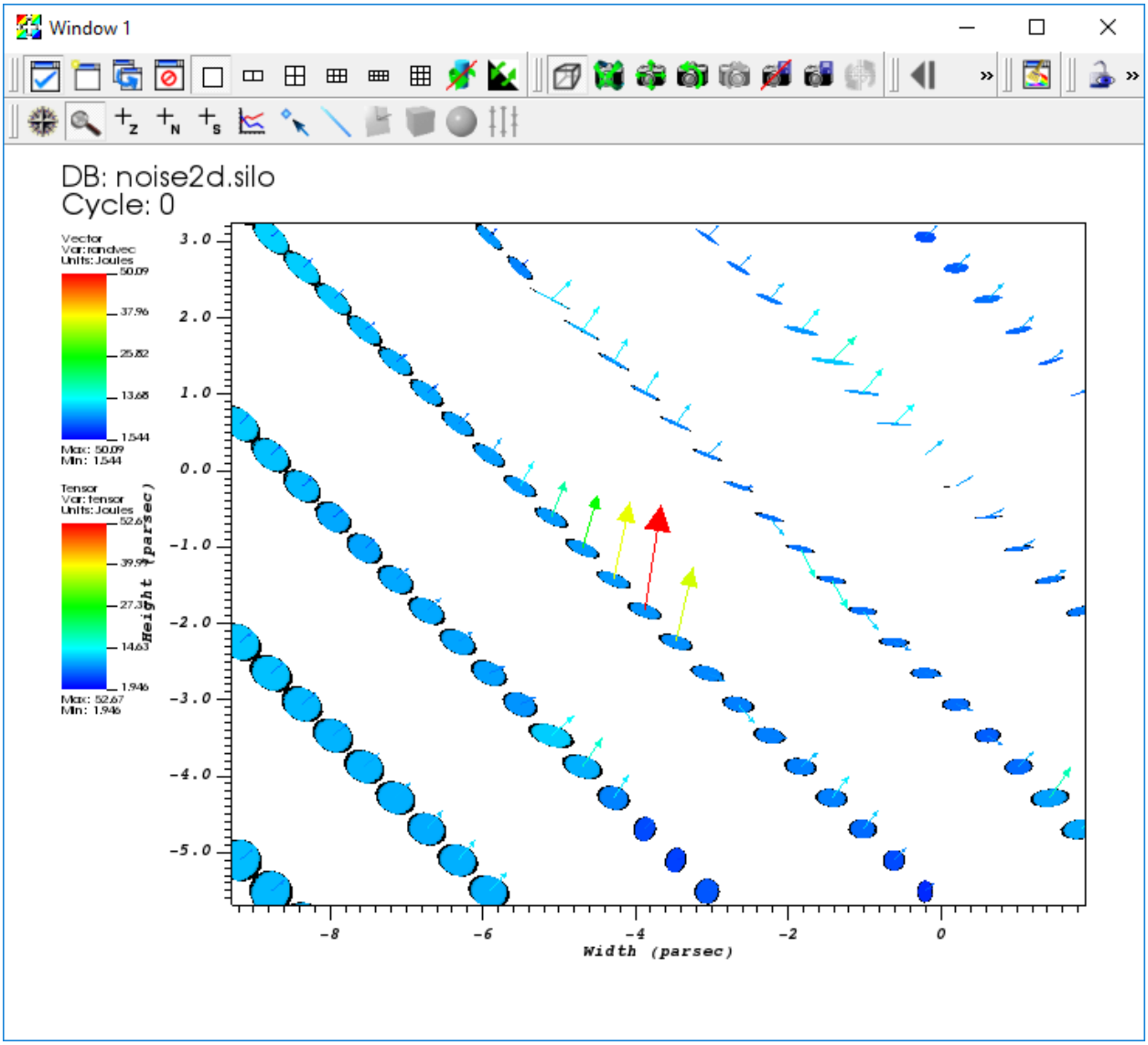


Fig. 6.21: Example of using vector and tensor expressions

- Note that *grad* is a *Vector mesh variable* and *shepardglobal* is a *Scalar mesh variable*.

2. Now, attempt to do a Vector plot of *foo*. This works because VisIt will add the scalar to each component of the vector resulting a new vector mesh variable
3. But, suppose you instead defined *foo* to be of *Type Scalar mesh variable*.

- VisIt will allow you to define this expression. But, when you go to plot it, the plot will fail.

As an aside, as you go back and forth between the Expressions window creating and/or adjusting expression definitions, VisIt makes no attempt to keep track of all the changes you've made in expressions and automatically update plots as expressions change. You will have to manually clear or delete plots to force VisIt to re-draw plots in which you've changed expressions.

In the above example, if on the other hand, you had set type of "foo" to Scalar Mesh Variable, then VisIt would have failed to plot it because it is adding a scalar and a vector variable and the result of such an operation is *always* a *Vector mesh variable*. If what you really intended was a scalar mesh variable, then use one of the expression functions that converts a vector to a scalar (e.g. *magnitude()* function or array dereference operator *[]*) to convert the *Vector mesh variable* in your expression to a scalar mesh variable. So, "grad[i]+shephardglobal" where "i" is "0" or "1" would work to define a scalar mesh variable. Or, "magnitude(grad)+shepardglobal" would also have worked.

Centering compatibility

In "noise2d.silo", some variables are zone centered and some are node centered. What happens if you combine these in an expression? VisIt will default to zone centering for the result. If this is not the desired result, use the "recenter()" expression function, where appropriate, to adjust centering of some of the terms in your expression. For example, again using "noise2d.silo".

1. Define the *Scalar mesh variable* expression "bar" as "shepardglobal+airVf".

- For reference, in "noise2d.silo", "shepardglobal" is node centered while "airVf" is zone centered.

2. Do a Pseudocolor plot of "bar".

- Note that "bar" displays as a zone centered quantity.

3. Now, go back to the expression and recenter "airVf" by adjusting the definition to "shepardglobal+recenter(airVf)".

- The *recenter()* expression function is a *toggle* in that it will take whatever the variable's centering is and swap it (node->zone and zone->node).
- The *recenter()* expression function also takes a second argument, a string of one of the values *toggle*, *zonal*, *nodal* to force a particular behavior.
- Note that when you click *Apply*, the current plot of "bar" does not change. You need to manually delete and re-create the plot (or clear and re-draw the plots).

Finally, note that these two expressions...

- "shepardglobal+recenter(airVf)"
- "recenter(shepardglobal+airVf)"

both achieve a node-centered result. But, each expression is subtly (and numerically) different. The first recenter's "airVf" to the nodes and then performs the summation operator at each node. In the second, there is an implied recentering of "shepardglobal" to the zones first. Then, the summation operator is applied at each zone center and finally the results are recentered back to the nodes. In all likelihood this results in a numerically lower quality result. The moral is that in a complex series of expressions be sure to take care where you want recentering to occur.

Mesh compatibility

In many cases, especially in Silo databases, all the available variables in a database are not always defined on the same mesh. This can complicate matters involving expressions in variables from different meshes.

Just as in the previous two examples of incompatible variables where the solution was to apply some functions to make the variables compatible, we have to do the same thing when variables from different meshes are combined in an expression. The key expression functions which enable this are called *Cross Mesh Field Evaluation* or *CMFE* functions. We will only briefly touch on these here. CMFEs will be discussed in much greater detail in a tutorial devoted to that topic.

Again, using "noise2d.silo"

1. Define the expression "gorf" with definition "PointVar + shepardglobal".

- Note that *PointVar* is defined on a mesh named *PointMesh* while *shepardglobal* is defined on a mesh named *Mesh*.

2. Try to do a Pseudocolor plot of "gorfo". You will get a plot of points and a warning message like this one...

   The compute engine running on host somehost.com issued the following warning: In domain 0, your nodal variable "shepardglobal" has 2500 values, but it should have 100. Some values were removed to ensure VisIt runs smoothly.

So, whats happening here? VisIt is deciding to perform the summation operation on the *PointVar*'s mesh. That mesh consists of 100 points. So, when it encounters the *shepardglobal* variable (defined on *Mesh* with 50x50 nodes), it simply ignores any values in "shepardgloabl" after the first 100. Most likely, this is not the desired outcome.

We have two options each of which involves *mapping* one of the variables onto the other variable's mesh using one of the CMFE expression functions. We can map *shepardglobal* onto *PointMesh* or we can map *PointVar* onto *Mesh*. We'll do both here

Mapping *shepardglobal* onto *PointMesh*

1. Define a new expression named "shepardglobal_mapped".
2. Go to *Insert Function...*, then to the *Comparisons* submenu and select *pos_cmfe*.

- This defines a *position based* cross-mesh field evaluation function. The other option is a *conn_cmfe* or *connectivity-based* which is faster but requires both meshes to be topologically congruent and is not appropriate here.

3. A template for the arguments to the *pos_cmfe* will appear in the *Definition* pane.
4. Replace "<filename:var>" with "<./noise2d.silo:shepardglobal>".

- This assumes the "noise2d.silo" file is in the same directory from which VisIt was started.
- This defines the *source* or *donor* variable to be mapped onto a new mesh.

5. Replace "<meshname>" with "PointMesh".

- This defines the *destination* or *target* mesh the variable is to be mapped onto.

6. Replace "<fill-var-for-uncovered-regions>" with "-1".

- This is needed for position-based CMFE's because the donor variable's mesh and target mesh may not always volumetrically overlap 100%. In places where this winds up being the case, VisIt will use this value to fill in.

7. Now with "shepardglobal_mapped" defined, you can define the desired expression, "PointVar + shepardglobal_mapped" and this will achieve the desired result and is shown below.

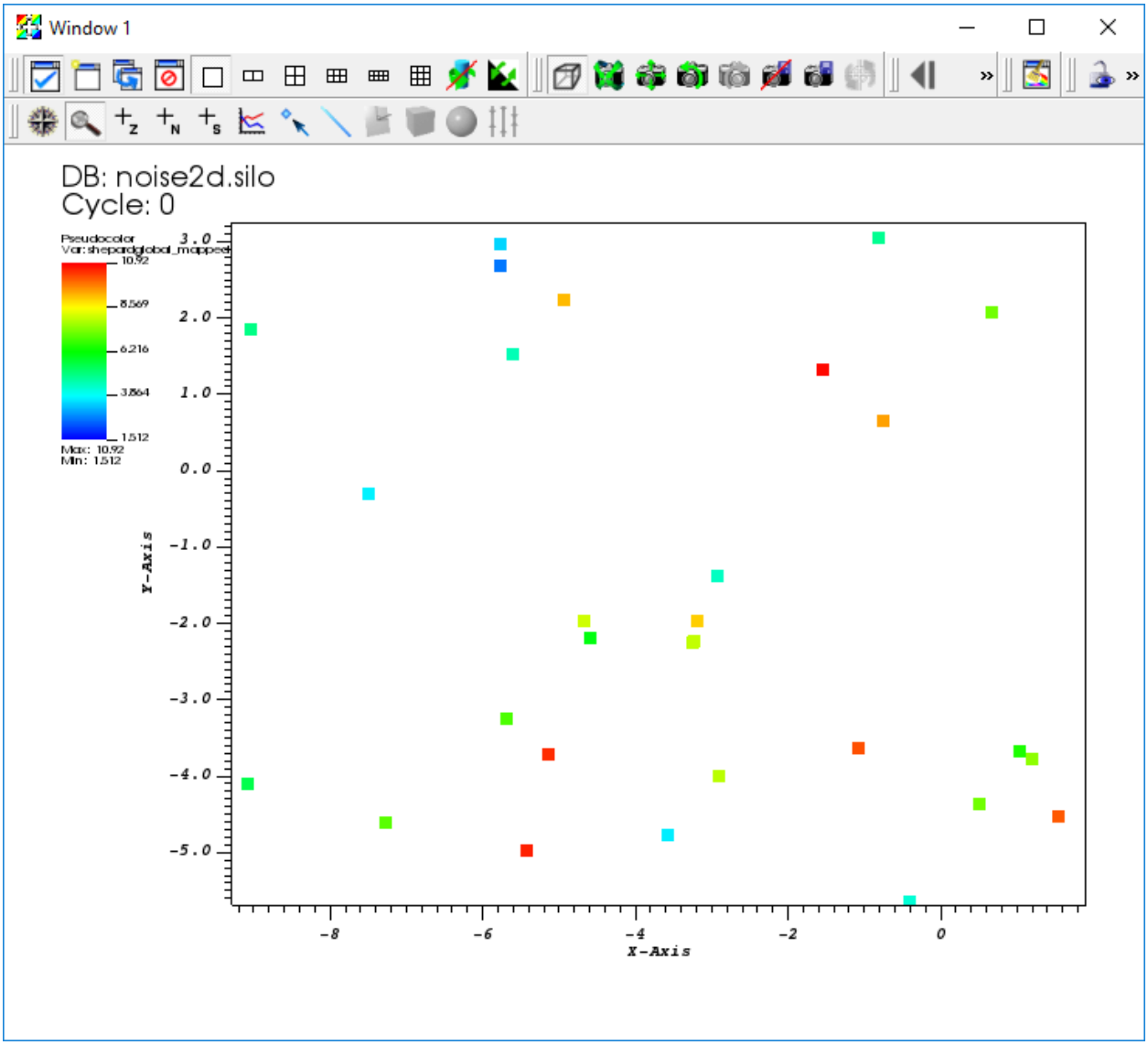


Fig. 6.22: The variable Shepardglobal mapped onto a point mesh

Mapping *PointVar* onto *Mesh*

To be completed. But, cannot map point mesh onto a volumetric mesh. VisIt always returns zero overlap.

Combining expressions and queries is powerful

Suppose you have a database generated by some application code simulating some object being blown apart. Maybe its a 2D, cylindrically symmetric calculation. Next, suppose the code produced a "density" and "velocity" variable. However, what you want to compute is the total mass of some (portion of) of the object that has velocity (magnitude) greater than some threshold, say 5 meters/second. You can use a combination of Expressions, Queries and the Threshold operator to achieve this.

Mass is "density * volume". You have a 2D mesh, so how do you get volume from something that has only 2 dimensions? You know the mesh represents a calculation that is cylindrically symmetric (revolved around the y-axis). You can use the *revolved_volume()* Expression function to obtain the volume of each zone in the mesh. Then, you can multiply the result of *revolved_volume()* by *density* to get mass of each zone in the mesh. Once you have that, you can use threshold operator to display only those zones with velocity (magnitude) greater than 5 and then a variable sum query to add up all the mass moving at that velocity.

Here, we demonstrate the steps involved using the "noise2d.silo" database. Because that database does not quite match the problem assumption described in the preceding paragraphs, we simply re-purpose a few of the variables in the database to serve as our *density* and *velocity* variables in this example. Namely, we define the expression *density* as an alias for *shephardglobal* and *velocity* as an alias for *grad*.

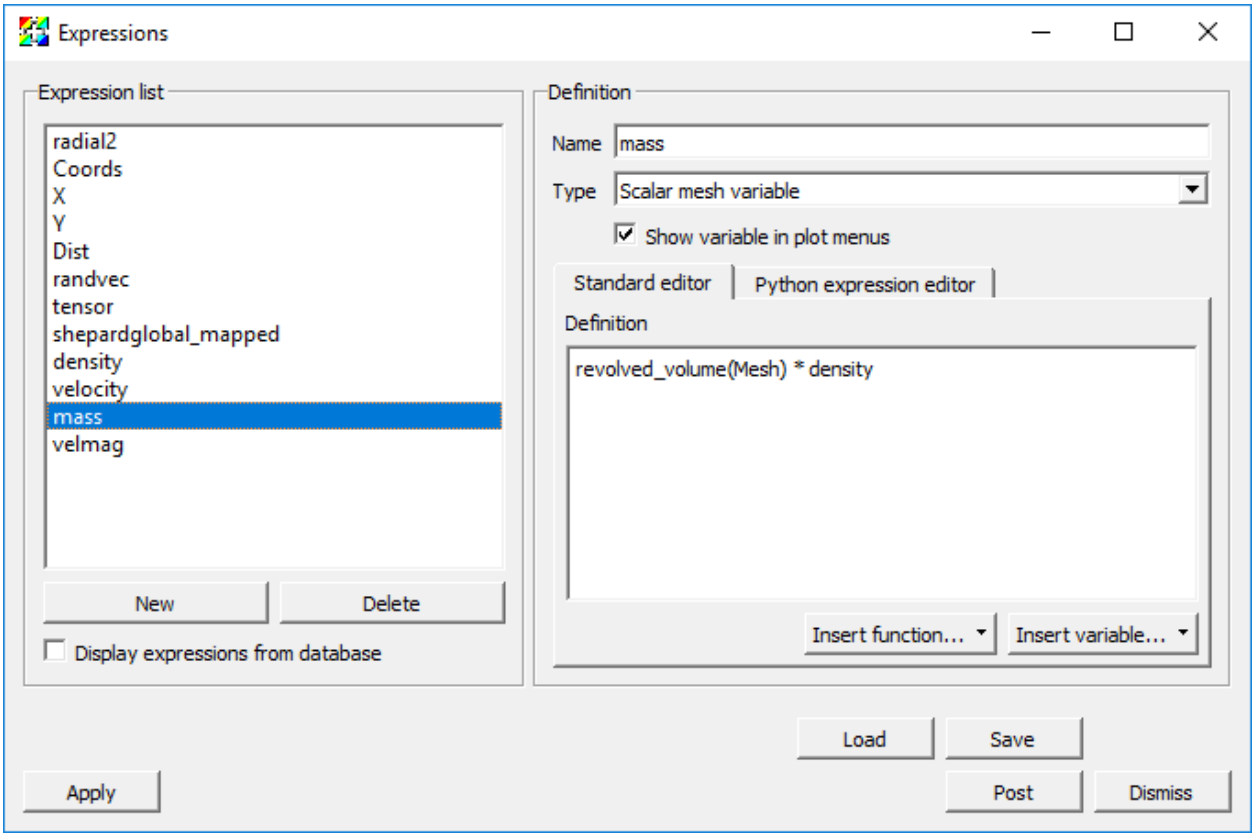


Fig. 6.23: Mass Expression Definition

Here are the steps involved...

1. Go to *Controls->Expressions*.
2. Click *New*.
3. Set the *Name* to “density”.
4. Make sure the *Type* is set to *Scalar mesh variable*.
5. Set the *Definition* to “shepardglobal”.
6. Click *Apply*.
7. Click *New*.
8. Set the *Name* to “velocity”.
9. Make sure the *Type* is set to *Vector mesh variable*.
10. Set the *Definition* to “grad”.
11. Click *Apply*.
12. Click *New*.
13. Set the *Name* to “mass”.
14. Make sure the *Type* is set to *Scalar mesh variable*.
15. Set the *Definition* to “revolved_volume(Mesh) * density”.
16. Click *Apply*.
17. Click the *New* button again (for a new expression).
18. Set the *Name* to “velmag” (for velocity magnitude).
19. Set the *Definition* to “magnitude(velocity)”.
20. Go to *Plot->Pseudocolor->mass*.
21. Click *Draw*.
22. Add *Operator->Threshold*.
23. Open the Threshold operator attributes window.
24. Select the *default* variable and then click *Delete selected variable*.
25. Go to *Add Variable* and select *velmag* from the list of *Scalars*.
26. Set *Lower Bound* to “5”.
27. Click *Apply*.
    - Now the displayed plot changes to show only those parts of the mesh that are moving with velocity greater than 5.
28. Go to *Controls->Query*.
29. Find the *Variable sum query* from the list of queries.
30. Click the *Query* button. The computed result will be a sum of all the individual zones’ masses in the mesh for those zones that are moving with velocity greater than 5.

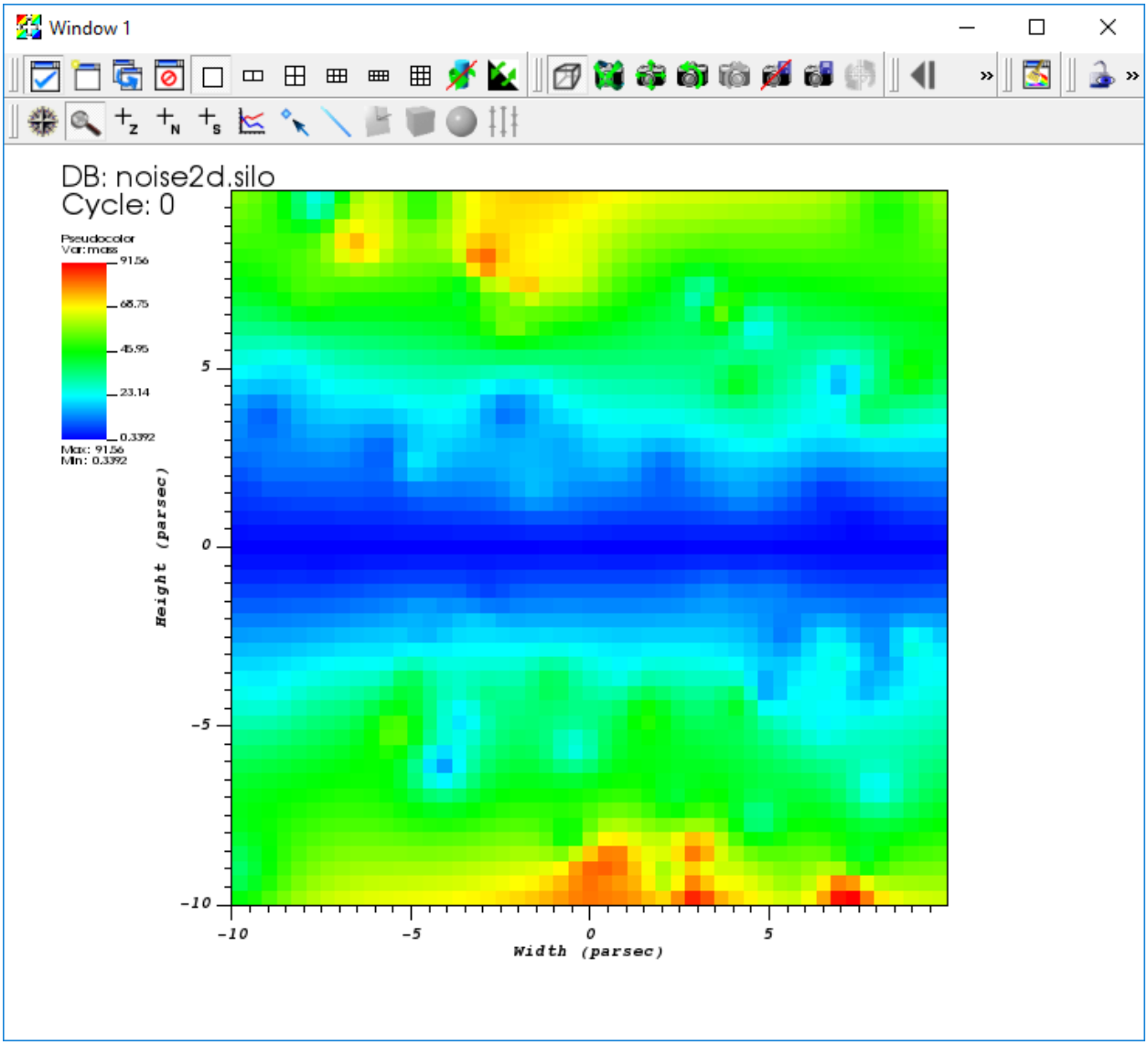


Fig. 6.24: Mass plot

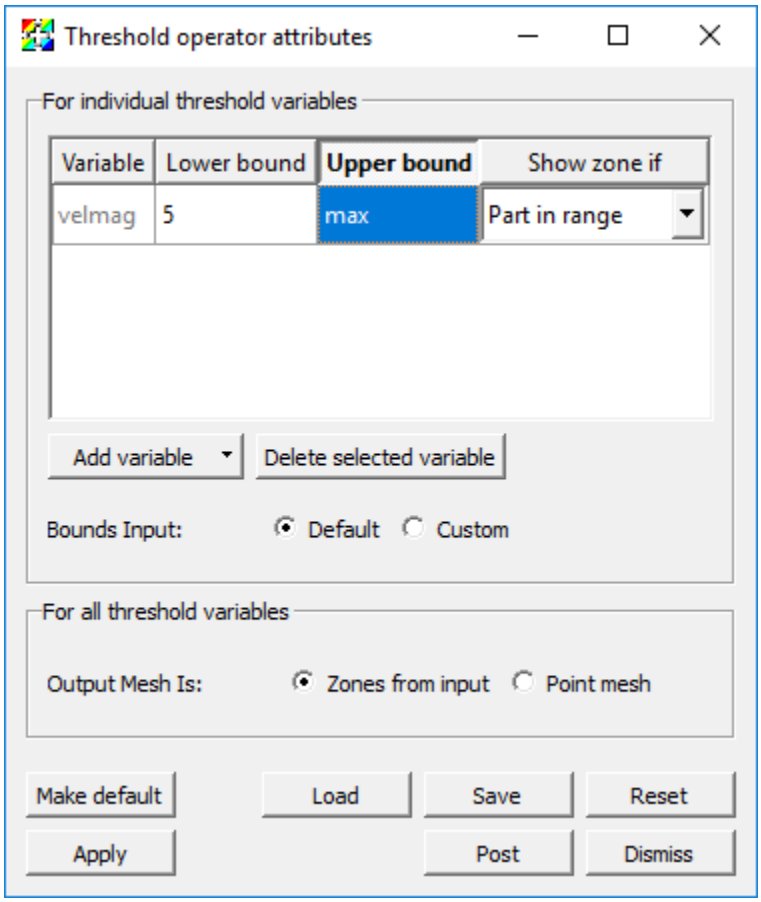


Fig. 6.25: Threshold attributes

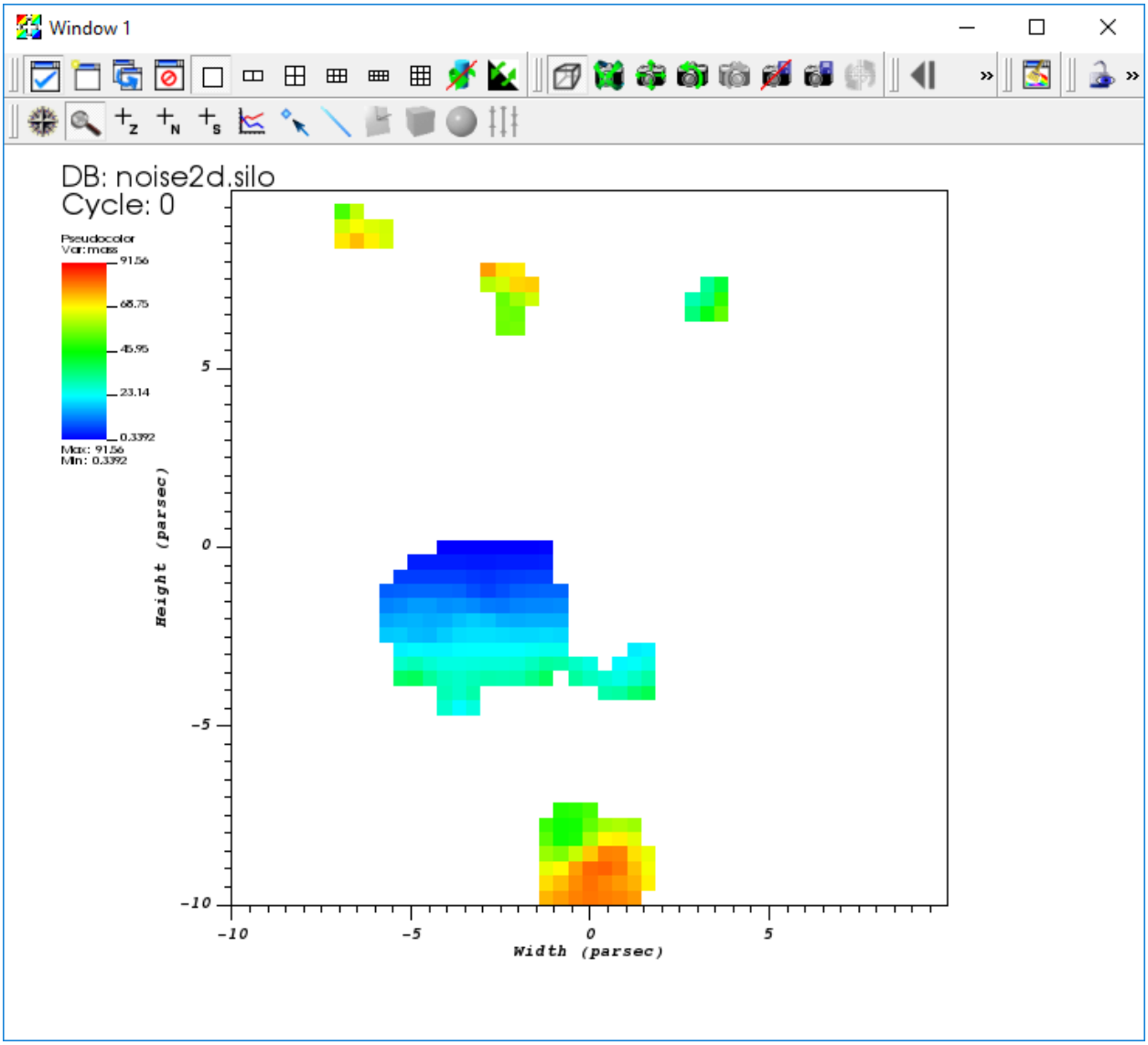


Fig. 6.26: Mass plot after threshold

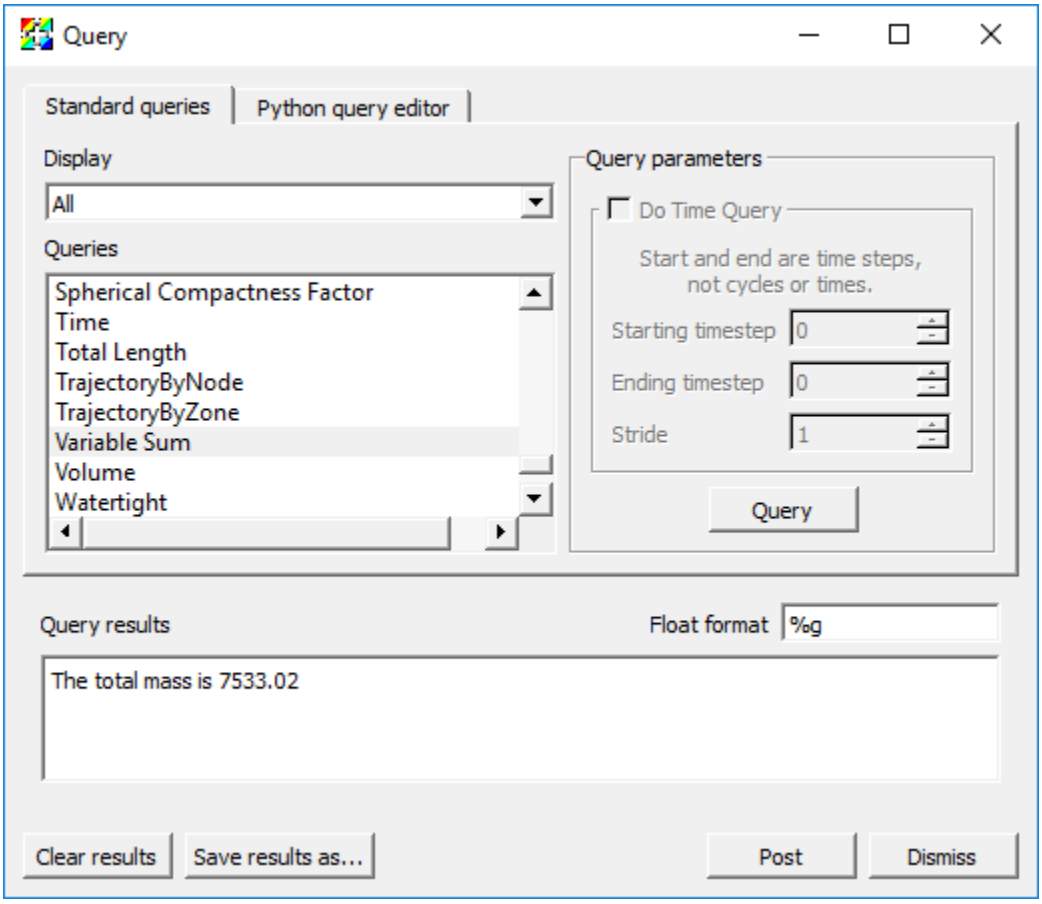


Fig. 6.27: The variable sum query result

Automatic, saved and database expressions

VisIt defines several types of expressions automatically. For all vector variables from a database, VisIt will automatically define the associated magnitude expressions. For unstructured meshes, VisIt will automatically define *mesh quality* expressions. For any databases consisting of multiple time states, VisIt will define *time derivative* expressions. This behavior can be controlled by going to VisIt's *Preferences* dialog and enabling or disabling various kinds of *automatic* expressions.

If you save settings, any expressions you have defined are also saved with the settings. And, they will appear (and sometimes pollute) your menus whether or not they are valid expressions for the currently active database.

Finally, databases are also free to define expressions. In fact, many databases define a large number of expressions for the convenience of their users who often use the expressions in their post-processing workflows. Ordinarily, you never see VisIt's automatic expressions or a database's expressions in the Expression window because they are not editable. However, you can check the *display expressions from database* check box in the Expressions window and VisIt will also show these expressions.

6.3 Scripting

This section describes the VisIt Command Line Interface (CLI).

6.3.1 Command line interface overview

VisIt includes a rich a command line interface that is based on Python 2.7.

There are several ways to use the *CLI*:

1) Launch VisIt in a batch mode and run scripts.
   - Linux: `/path/to/visit/bin/visit -nowin -cli -s <script.py>`
   - macOS: `/path/to/VisIt.app/Contents/Resources/bin/visit -nowin -cli -s <script.py>`
2) Launch VisIt so that a visualization window is visible and interactively issue *CLI* commands.
3) Use both the standard *GUI* and *CLI* simultaneously.

6.3.2 Launching the *CLI*

We will focus on the use case where we have the graphical user interface and *CLI* running simultaneously.

To launch the *CLI* from the graphical user interface:

1) Go to *Controls->Command*.

This will bring up the Commands window. The Command window provides a text editor with Python syntax highlighting and an *Execute* button that tells VisIt to execute the script. Finally, the Command window lets you record your *GUI* actions into Python code that you can use in your scripts.

6.3.3 A first action in the *CLI*

1) Open "example.silo" in the *GUI* if it not already open.
2) Cut-and-paste the following Python commands into the first tab of the Commands window.

```
AddPlot("Pseudocolor", "temp")
# You will see the active plots list in the GUI update, since the CLI and GUI␣
↪communicate.
DrawPlots()
# You will see your plot.
```

3) Click *Execute*.

6.3.4 Tips about Python

1) Python is whitespace sensitive! This is a pain, especially when you are cut-n-pasting things.
2) Python has great constructs for control and iteration, here are some examples:

```
for i in range(100):
   # use i

# strided range
for i in range(0,100,10):
   # use i

if (cond):
 # stmt

import sys
...
sys.exit()
```

6.3.5 Example scripts

We will be using Python scripts in each of the following sections: You can get execute them by:

1) Cut-n-paste-ing them into a tab in the Commands window and executing it.

For all of these scripts, make sure "example.silo" is currently open unless otherwise noted.

Setting attributes

Each of VisIt's Plots and Operators expose a set of *attributes* that control their behavior. In VisIt's *GUI*, these attributes are modified via options windows. VisIt's *CLI* provides a set of simple Python objects that control these attributes. Here is an example setting the minimum and maximum for the Pseudocolor plot

```
DeleteAllPlots()
AddPlot("Pseudocolor", "temp")
DrawPlots()
p = PseudocolorAttributes()
p.minFlag = 1
p.maxFlag = 1
p.min = 3.5
p.max = 7.5
SetPlotOptions(p)
```

Animating an isosurface

This example demonstrates sweeping an isosurface operator to animate the display of a range of isovalues from "example.silo".

```
DeleteAllPlots()
AddPlot("Pseudocolor", "temp")
iso_atts = IsosurfaceAttributes()
iso_atts.contourMethod = iso_atts.Value
iso_atts.variable = "temp"
AddOperator("Isosurface")
DrawPlots()
for i in range(30):
   iso_atts.contourValue = (2 + 0.1*i)
   SetOperatorOptions(iso_atts)
   # For moviemaking, you'll need to save off the image
   # SaveWindow()
```

Using all of VisIt's building blocks

This example uses a Pseudocolor plot with a ThreeSlice operator applied to display *temp* on the exterior of the grid along with streamlines of the gradient of *temp*.

Note that the script below may not work the first time you execute it. In that case delete all the plots and execute the script again.

```
# Clear any previous plots
DeleteAllPlots()
# Create a plot of the scalar field 'temp'
AddPlot("Pseudocolor","temp")
# Slice the volume to show only three
# external faces.
AddOperator("ThreeSlice")
tatts = ThreeSliceAttributes()
tatts.x = -10
tatts.y = -10
tatts.z = -10
SetOperatorOptions(tatts)
DrawPlots()
# Find the maximum value of the field 'temp'
Query("Max")
val = GetQueryOutputValue()
print("Max value of 'temp' = ", val)

# Create a streamline plot that follows
# the gradient of 'temp'
DefineVectorExpression("g","gradient(temp)")
AddPlot("Pseudocolor", "operators/IntegralCurve/g")
iatts = IntegralCurveAttributes()
iatts.sourceType = iatts.SpecifiedBox
iatts.sampleDensity0 = 7
iatts.sampleDensity1 = 7
iatts.sampleDensity2 = 7
iatts.dataValue = iatts.SeedPointID
iatts.integrationType = iatts.DormandPrince
iatts.issueStiffnessWarnings = 0
```

(continues on next page)

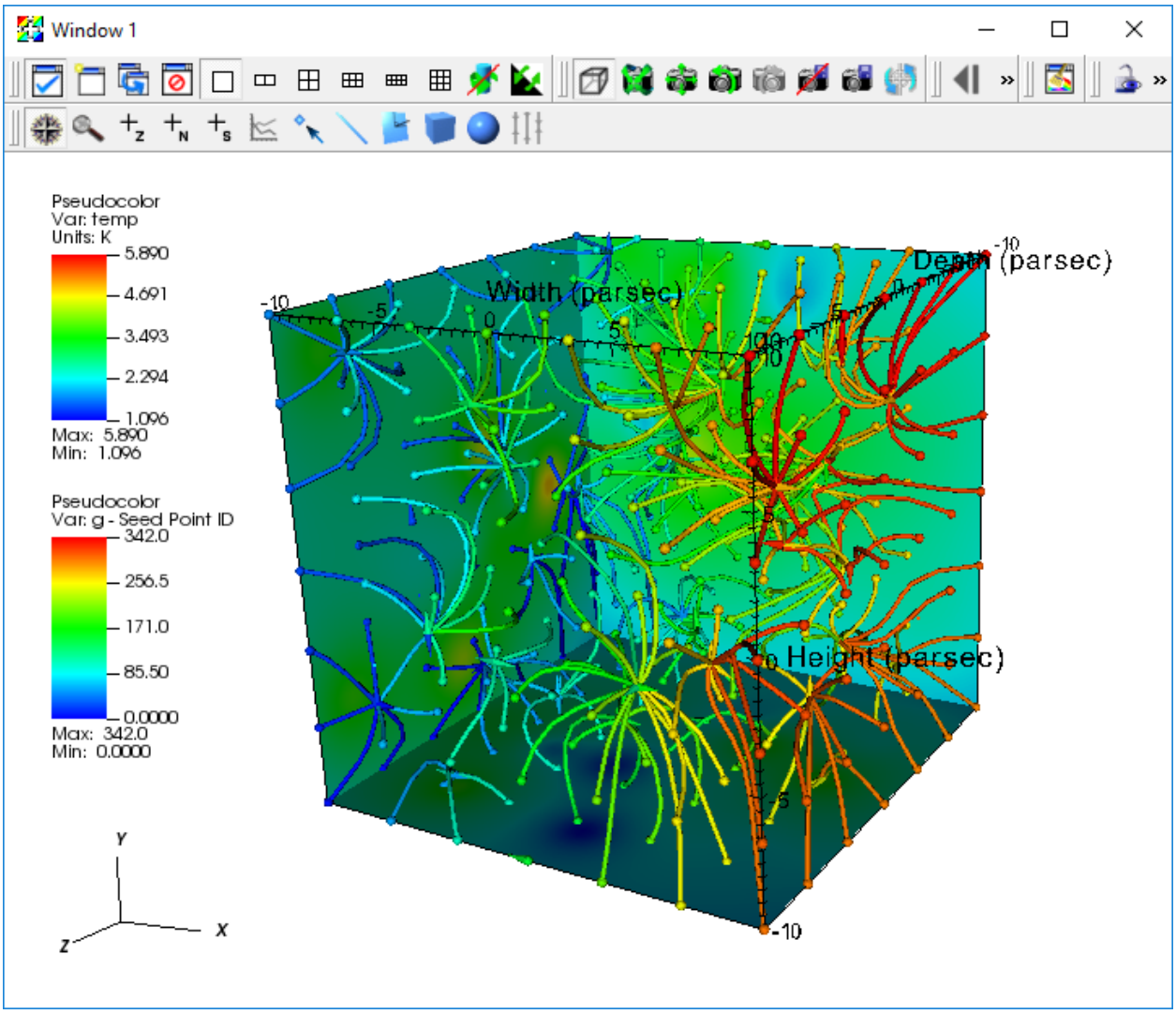


Fig. 6.28: Streamlines

(continued from previous page)

```
iatts.issueCriticalPointsWarnings = 0
SetOperatorOptions(iatts)

# set style of streamlines
patts = PseudocolorAttributes()
patts.lineType = patts.Tube
patts.tailStyle = patts.Spheres
patts.headStyle = patts.Cones
patts.endPointRadiusBBox = 0.01
SetPlotOptions(patts)

DrawPlots()
```

Creating a movie of animated streamline paths

This example extends the "Using all of VisIt's Building Blocks" example by

- animating the paths of the streamlines
- saving images of the animation
- finally, encoding those images into a movie

(Note: Encoding requires ffmpeg is installed and available in your PATH)

```
# import visit_utils, we will use it to help encode our movie
from visit_utils import *

# Set a better view
ResetView()
v = GetView3D()
v.RotateAxis(0,44)
v.RotateAxis(1,-23)
SetView3D(v)

# Disable annotations
aatts = AnnotationAttributes()
aatts.axes3D.visible = 0
aatts.axes3D.triadFlag = 0
aatts.axes3D.bboxFlag = 0
aatts.userInfoFlag = 0
aatts.databaseInfoFlag = 0
aatts.legendInfoFlag = 0
SetAnnotationAttributes(aatts)

# Set basic save options
swatts = SaveWindowAttributes()
#
# The 'family' option controls if visit automatically adds a frame number to
# the rendered files. For this example we will explicitly manage the output name.
#
swatts.family = 0
#
# select PNG as the output file format
#
swatts.format = swatts.PNG
#
```

(continues on next page)

(continued from previous page)

```
# set the width of the output image
#
swatts.width = 1024
#
# set the height of the output image
#
swatts.height = 1024



####
# Crop streamlines to render them at increasing time values over 50 steps
####
iatts.cropValue = iatts.Time
iatts.cropEndFlag = 1
iatts.cropBeginFlag = 1
iatts.cropBegin = 0
for ts in range(0,50):
    # set the integral curve attributes to change the where we crop the streamlines
    iatts.cropEnd = (ts + 1) * .5

    # update streamline attributes and draw the plot
    SetOperatorOptions(iatts)
    DrawPlots()
    #before we render the result, explicitly set the filename for this render
    swatts.fileName = "streamline_crop_example_%04d.png" % ts
    SetSaveWindowAttributes(swatts)
    # render the image to a PNG file
    SaveWindow()

#################
# use visit_utils.encoding to encode these images into a "mp4" movie
#
# The encoder looks for a printf style pattern in the input path to identify the␣
↪frames of the movie.
# The frame numbers need to start at 0.
#
# The encoder selects a set of decent encoding settings based on the extension of the
# the output movie file (second argument). In this case we will create a "mp4" file.
#
# Other supported options include ".mpg", ".mov".
#   "mp4" is usually the best choice and plays on all most all platforms (Linux ,OSX,␣
↪Windows).
#   "mpg" is lower quality, but should play on any platform.
#
# 'fdup' controls the number of times each frame is duplicated.
#  Duplicating the frames allows you to slow the pace of the movie to something␣
↪reasonable.
#
################

input_pattern = "streamline_crop_example_%04d.png"
output_movie = "streamline_crop_example.mp4"
encoding.encode(input_pattern,output_movie,fdup=4)
```

Rendering each time step of a dataset to a movie

This example assumes the "aneurysm.visit" is already opened.

- Create a plot, render all timesteps and encode a movie.

(Note: Encoding requires that ffmpeg is installed and available in your PATH)

```
# import visit_utils, we will use it to help encode our movie
from visit_utils import *
DeleteAllPlots()

AddPlot("Pseudocolor","pressure")
DrawPlots()

# Set a better view
ResetView()
v = GetView3D()
v.RotateAxis(1,90)
SetView3D(v)

# get the number of timesteps
nts = TimeSliderGetNStates()

# set basic save options
swatts = SaveWindowAttributes()
#
# The 'family' option controls if visit automatically adds a frame number to
# the rendered files. For this example we will explicitly manage the output name.
#
swatts.family = 0
#
# select PNG as the output file format
#
swatts.format = swatts.PNG
#
# set the width of the output image
#
swatts.width = 1024
#
# set the height of the output image
#
swatts.height = 1024

#the encoder expects file names with an integer sequence
# 0,1,2,3 .... N-1

file_idx = 0

for ts in range(0,nts,10): # look at every 10th frame
    # Change to the next timestep
    TimeSliderSetState(ts)
    #before we render the result, explicitly set the filename for this render
    swatts.fileName = "blood_flow_example_%04d.png" % file_idx
    SetSaveWindowAttributes(swatts)
    # render the image to a PNG file
    SaveWindow()
    file_idx +=1
```

(continues on next page)

(continued from previous page)

```
################
# use visit_utils.encoding to encode these images into a "mp4" movie
#
# The encoder looks for a printf style pattern in the input path to identify the␣
↪frames of the movie.
# The frame numbers need to start at 0.
#
# The encoder selects a set of decent encoding settings based on the extension of the
# the output movie file (second argument). In this case we will create a "mp4" file.
#
# Other supported options include ".mpg", ".mov".
#   "mp4" is usually the best choice and plays on all most all platforms (Linux ,OSX,␣
↪Windows).
#   "mpg" is lower quality, but should play on any platform.
#
# 'fdup' controls the number of times each frame is duplicated.
#  Duplicating the frames allows you to slow the pace of the movie to something␣
↪reasonable.
#
################

input_pattern = "blood_flow_example_%04d.png"
output_movie = "blood_flow_example.mp4"
encoding.encode(input_pattern,output_movie,fdup=4)
```

Animating the camera

```
def fly():
    # Do a pseudocolor plot of u.
    DeleteAllPlots()
    AddPlot('Pseudocolor', 'temp')
    AddOperator("Clip")
    c = ClipAttributes()
    c.funcType = c.Sphere  # Plane, Sphere
    c.center = (0, 0, 0)
    c.radius = 10
    c.sphereInverse = 1
    SetOperatorOptions(c)
    DrawPlots()

    # Create the control points for the views.
    c0 = View3DAttributes()
    c0.viewNormal = (0, 0, 1)
    c0.focus = (0, 0, 0)
    c0.viewUp = (0, 1, 0)
    c0.viewAngle = 30
    c0.parallelScale = 17.3205
    c0.nearPlane = 17.3205
    c0.farPlane = 81.9615
    c0.perspective = 1

    c1 = View3DAttributes()
    c1.viewNormal = (-0.499159, 0.475135, 0.724629)
    c1.focus = (0, 0, 0)
    c1.viewUp = (0.196284, 0.876524, -0.439521)
```

(continues on next page)

(continued from previous page)

```
    c1.viewAngle = 30
    c1.parallelScale = 14.0932
    c1.nearPlane = 15.276
    c1.farPlane = 69.917
    c1.perspective = 1

    c2 = View3DAttributes()
    c2.viewNormal = (-0.522881, 0.831168, -0.189092)
    c2.focus = (0, 0, 0)
    c2.viewUp = (0.783763, 0.556011, 0.27671)
    c2.viewAngle = 30
    c2.parallelScale = 11.3107
    c2.nearPlane = 14.8914
    c2.farPlane = 59.5324
    c2.perspective = 1

    c3 = View3DAttributes()
    c3.viewNormal = (-0.438771, 0.523661, -0.730246)
    c3.focus = (0, 0, 0)
    c3.viewUp = (-0.0199911, 0.80676, 0.590541)
    c3.viewAngle = 30
    c3.parallelScale = 8.28257
    c3.nearPlane = 3.5905
    c3.farPlane = 48.2315
    c3.perspective = 1

    c4 = View3DAttributes()
    c4.viewNormal = (0.286142, -0.342802, -0.894768)
    c4.focus = (0, 0, 0)
    c4.viewUp = (-0.0382056, 0.928989, -0.36813)
    c4.viewAngle = 30
    c4.parallelScale = 10.4152
    c4.nearPlane = 1.5495
    c4.farPlane = 56.1905
    c4.perspective = 1

    c5 = View3DAttributes()
    c5.viewNormal = (0.974296, -0.223599, -0.0274086)
    c5.focus = (0, 0, 0)
    c5.viewUp = (0.222245, 0.97394, -0.0452541)
    c5.viewAngle = 30
    c5.parallelScale = 1.1052
    c5.nearPlane = 24.1248
    c5.farPlane = 58.7658
    c5.perspective = 1

    c6 = c0

    # Create a tuple of camera values and x values. The x values are weights
    # that help to determine where in [0,1] the control points occur.
    cpts = (c0, c1, c2, c3, c4, c5, c6)
    x=[]
    for i in range(7):
        x = x + [float(i) / float(6.)]

    # Animate the camera. Note that we use the new built-in EvalCubicSpline
    # function which takes a t value from [0,1] a tuple of t values and a tuple
```

(continues on next page)

(continued from previous page)

```
    # of control points. In this case, the control points are View3DAttributes
    # objects that we are using to animate the camera but they can be any object
    # that supports +, * operators.
    nsteps = 100
    for i in range(nsteps):
        t = float(i) / float(nsteps - 1)
        c = EvalCubicSpline(t, x, cpts)
        c.nearPlane = -34.461
        c.farPlane = 34.461
        SetView3D(c)
        # For moviemaking...
        # SaveWindow()

fly()
```

Automating data analysis

```
def TakeMassPerSlice():
   DeleteAllPlots()
   AddPlot("Pseudocolor", "chromeVf")
   AddOperator("Slice")
   DrawPlots()
   f = open("mass_per_slice.ultra", "w")
   f.write("# mass_per_slice\n")
   for i in range(50):
     intercept = -10 + 20*(i/49.)
     s = SliceAttributes()
     s.axisType = s.XAxis
     s.originType = s.Intercept
     s.originIntercept = intercept
     SetOperatorOptions(s)
     Query("Weighted Variable Sum")
     t2 = GetQueryOutputValue()
     str = "%25.15e %25.15e\n" %(intercept, t2)
     f.write(str)
   f.close()

TakeMassPerSlice()
DeleteAllPlots()
OpenDatabase("mass_per_slice.ultra")
AddPlot("Curve", "mass_per_slice")
DrawPlots()
```

Extracting a per-material aggregate value at each timestep

```
####################
# Example that demonstrates looping over a dataset
# to extract an aggregate value at each timestep.
#
#  visit -nowin -cli -s listing_8_extracting_aggregate_values.py wave.visit pressure␣
↪wave_pressure_out
####################
```

(continues on next page)

(continued from previous page)

```
import sys
from visit_utils import *

def setup_plot(dbname,varname, materials = None):
    """
    Create a plot to query.
    """
    OpenDatabase(dbname)
    AddPlot("Pseudocolor",varname)
    if not materials is None:
        TurnMaterialsOff()
        # select the materials (by id )
        # example
        #  materials = [ "1", "2", "4"]
        for material in materials:
            TurnMaterialsOn(materials)
    DrawPlots()

def extract_curve(varname,obase,stride=1):
    """
    Loop over all time steps and extract a value at each one.
    """
    f = open(obase + ".ult","w")
    f.write("# %s vs time\n" % varname)
    nts = TimeSliderGetNStates()
    for ts in range(0,nts,stride):
        print("processing timestate: %d" % ts)
        TimeSliderSetState(ts)
        tval = query("Time")
        # sums plotted variable scaled by
        #   area (2D mesh),
        #   revolved_volume (2D RZ mesh, or
        #   volume (3D mesh)
        rval = query("Weighted Variable Sum")
        # or you can use other queries, such as max:
        # mval = query("Maximum")
        res = "%s %s\n" % (str(tval),str(rval))
        print(res)
        f.write(res)
    f.close()


def open_engine():
    # to open an parallel engine
    #  inside of an mxterm or batch script use:
    engine.open(method="slurm")
    #  outside of an mxterm or batch script
    # engine.open(nprocs=21)

def main():
    nargs = len(sys.argv)
    if nargs < 4:
        usage_msg = "usage: visit -nowin -cli -s visit_extract_curve.py "
        usage_msg += "{databsase_name} {variable_name} {output_file_base}"
        print(usage_msg)
        sys.exit(-1)
    # get our args
```

(continues on next page)

(continued from previous page)

```
    dbname  = sys.argv[1]
    varname = sys.argv[2]
    obase   = sys.argv[3]
    # if you need a parallel engine:
    # open_engine()
    setup_plot(dbname,varname)
    extract_curve(varname,obase)


if __visit_script_file__ == __visit_source_file__:
    main()
    sys.exit(0)
```

6.3.6 Recording *GUI* actions to Python scripts

VisIt's Commands window provides a mechanism to translate *GUI* actions into their equivalent Python commands.

1) Open the Commands Window by selecting ''Controls Menu->Command"

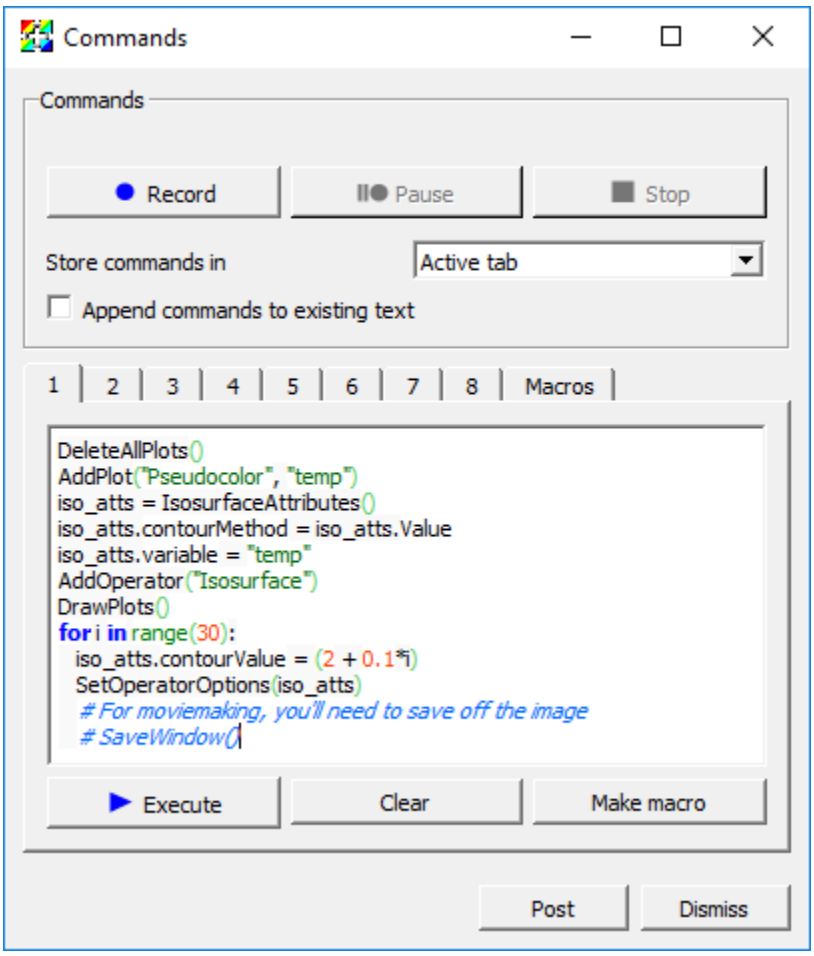


Fig. 6.29: The Commands window

2) Click the *Record* button.

3) Perform *GUI* actions.

4) Return to the Commands Window.

5) Select a tab to hold the python script of your recorded actions.

6) Click the *Stop* button.

7) The equivalent Python script will be placed in the tab in the Commands window.

- Note that the scripts are very verbose and contain some unnecessary commands, which can be edited out.

6.3.7 Learning the *CLI*

Here are some tips to help you quickly learn how to use VisIt's *CLI*:

1) From within VisIt's python *CLI*, you can type "dir()" to see the list of all commands.

- Sometimes, the output from "dir()" within VisIt's python *CLI* is a little hard to look through. So, a useful thing on Linux to get a nicer list of methods is the following shell command (typed from *outside* VisIt's python *CLI*)...

```
echo "dir()" | visit -cli -nowin -forceinteractivecli | tr ',' '\n' | tr -d " '"␣
↪| sort
```

- Or, if you are looking for *CLI* functions having to do with a specific thing...

```
echo "dir()" | visit -cli -nowin -forceinteractivecli | tr ',' '\n' | tr -d " '"␣
↪| grep -i material
```

2) You can learn the syntax of a given method by typing "help(MethodName)"

- Type "help(AddPlot)" in the Python interpreter.

3) Use the *GUI* to Python recording featured outlined in *Recording GUI actions to Python scripts*.

4) Use ''WriteScript()" function, which will create a python script that describes all of your current plots.

- For more details, see WriteScript.

5) When you have a Python object, you can see all of its attributes by printing it.

```
s = SliceAttributes()
print s
# Output:
originType = Intercept  # Point, Intercept, Percent, Zone, Node
originPoint = (0, 0, 0)
originIntercept = 0
originPercent = 0
originZone = 0
originNode = 0
normal = (0, -1, 0)
axisType = YAxis  # XAxis, YAxis, ZAxis, Arbitrary, ThetaPhi
upAxis = (0, 0, 1)
project2d = 1
interactive = 1
flip = 0
originZoneDomain = 0
originNodeDomain = 0
meshName = "default"
```

(continues on next page)

(continued from previous page)

```
theta = 0
phi = 0
```

See *the section on apropos*, `help` and `lsearch` for more information on finding stuff in VisIt's *CLI*.

6.3.8 Advanced features

1) You can set up your own buttons in the VisIt gui using the *CLI*. See *VisIt Run Commands (RC) File*.
2) You can set up callbacks in the *CLI* that get called whenever events happen in VisIt. See Python callbacks.
3) You can create your own custom Qt *GUI* that uses VisIt for plotting. See PySide recipes.

6.4 Aneurysm

This tutorial provides a short introduction to VisIt's features while exploring a finite element blood flow simulation of an aneurysm. The simulation was run using the LifeV finite element solver and made available for this tutorial thanks to Gilles Fourestey and Jean Favre, Swiss National Supercomputing Centre.

6.4.1 Open the dataset

This tutorial uses the aneurysm dataset.

1. Download the aneurysm dataset.
2. Click on the *Open* icon to bring up the File open window.
3. Navigate your file system to the folder containing "aneurysm.visit".
4. Highlight the file "aneurysm.visit" and then click *OK*.

6.4.2 Plotting the mesh topology

First we will examine the finite element mesh used in the blood flow simulation.

Create a Mesh plot

1. Go to *Add->Mesh->Mesh*.
2. Click *Draw*.

After this, the mesh plot is rendered in VisIt's Viewer window. Modify the view by rotating and zooming in the viewer window.

Modify the Mesh plot settings

1. Double click on the Mesh plot to open the Mesh plot attributes window.
2. Experiment with settings for:

- *Mesh color*
- *Opaque color*

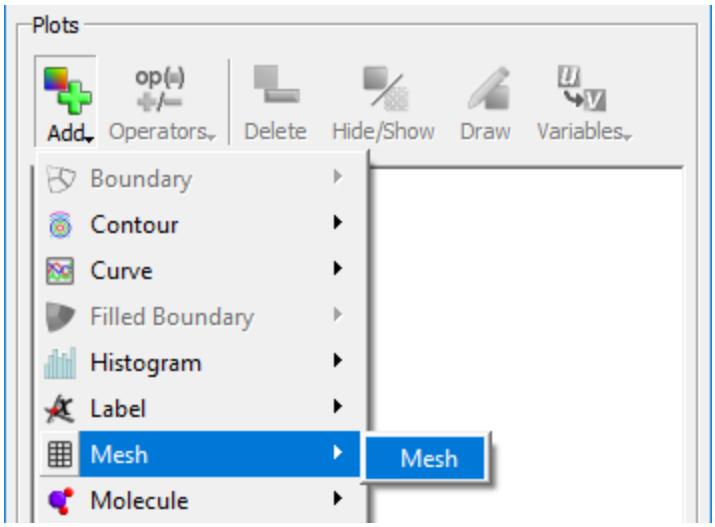


Fig. 6.30: Adding a mesh plot.

- *Opaque mode* - When the Mesh plot's opaque mode is set to automatic, the Mesh plot will be drawn in opaque mode unless it is forced to share the visualization window with other plots, at which point the Mesh plot is drawn in wireframe mode. When the Mesh plot is drawn in wireframe mode, only the edges of each externally visible cell face are drawn, which prevents the Mesh plot from interfering with the appearance of other plots. In addition to having an automatic opaque mode, the Mesh plot can be forced to be drawn in opaque mode or wireframe mode by selecting the *On* or *Off*. This is best demonstrated with the Pseudocolor plot of pressure present.
- *Show internal zones*

You will need to click *Apply* to commit the settings to your plot.

Query the mesh properties

VisIt's Query interface provides several quantitative data summarization operations. We will use the query interface to learn some basic information about the simulation mesh.

1. Go to *Controls->Query* to bring up the Query window.
2. Select *NumZones* and click *Query*.
   - This returns the number of elements in the mesh.
3. Select *NumNodes* and click *Query*.
   - This returns the number of vertices in the mesh

Note: The terms "zones", "elements", and "cells" are overloaded in scientific visualization, as are the terms "nodes", "points", and "vertices".

Additional exercises

- What type of finite element was used to construct the mesh?
- How many elements are used to construct the mesh?
- How many vertices are used to construct the mesh?

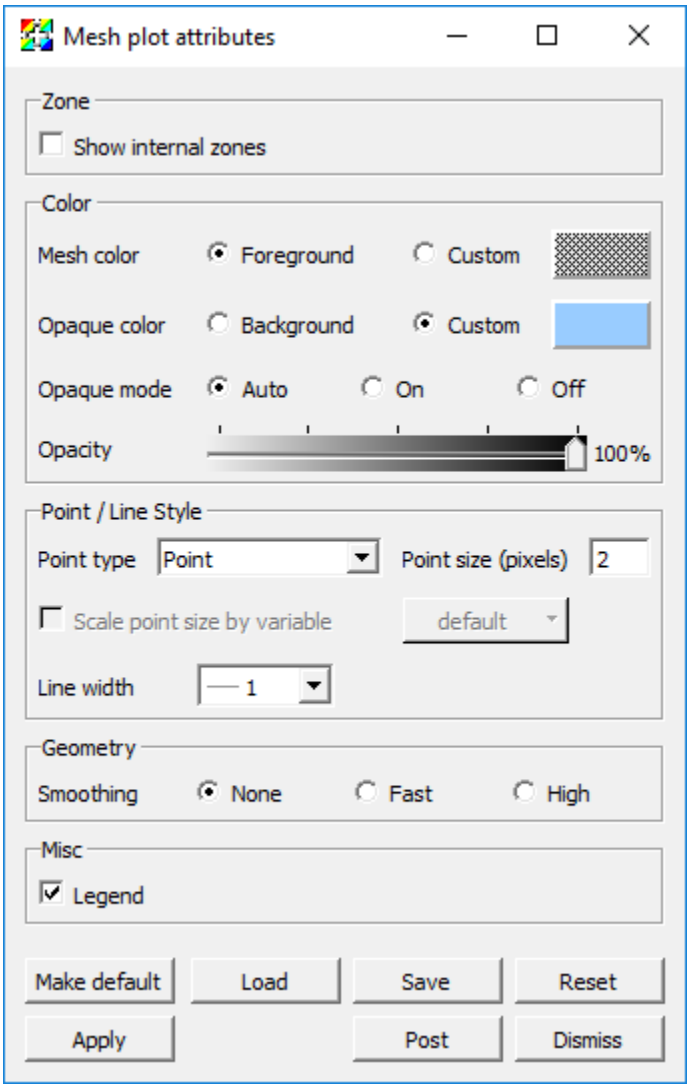


Fig. 6.31: The mesh plot attributes window.

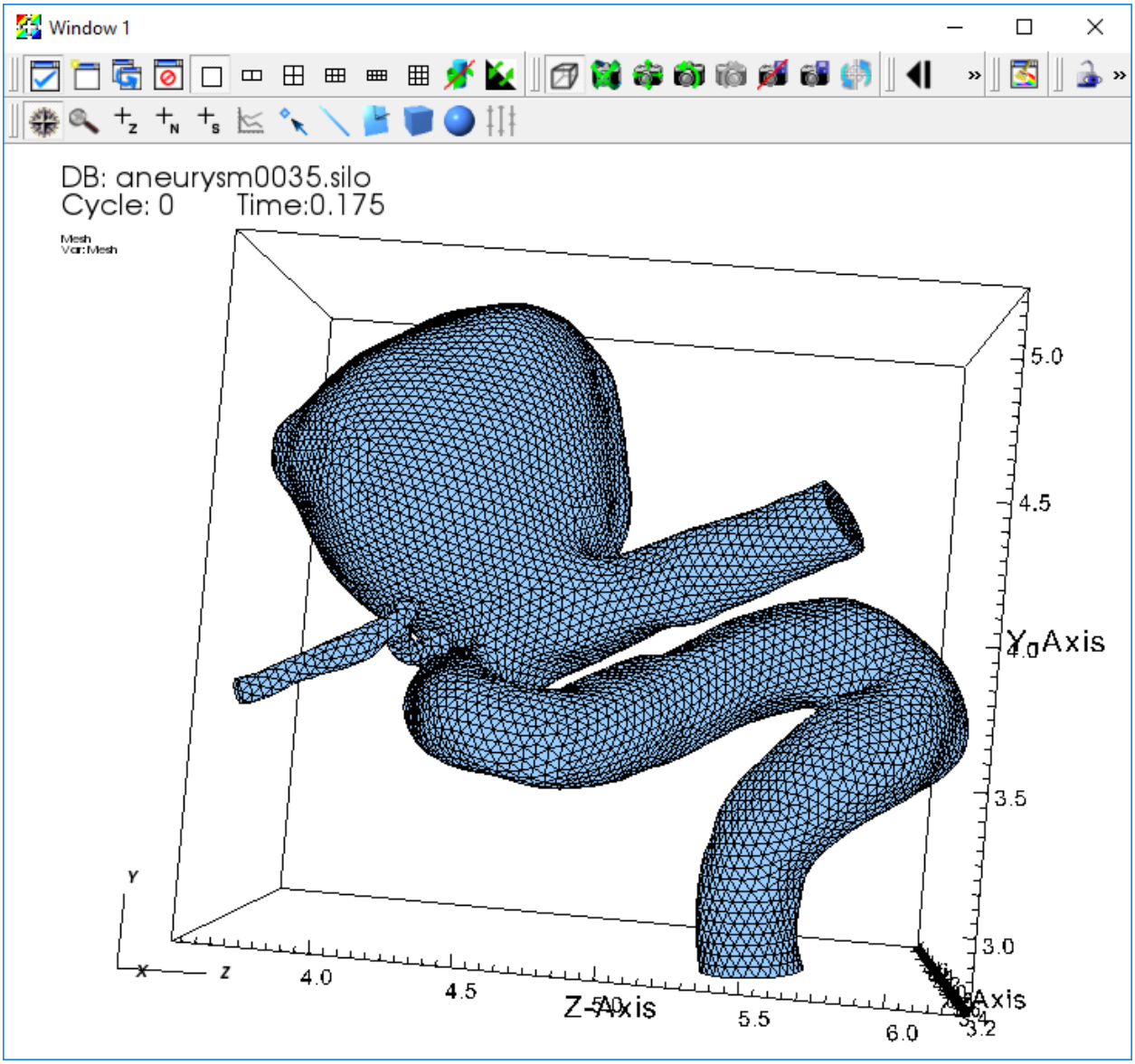


Fig. 6.32: The mesh plot of the aneurysm.

- On average, how many vertices are shared per element?

6.4.3 Examining scalar fields

In addition to the mesh topology, this dataset provides two mesh fields:

- A scalar field "pressure", associated with the mesh vertices.
- A vector field "velocity", associated with the mesh vertices.

VisIt automatically defines an expression that allows us to use the magnitude of the "velocity" vector field as a scalar field on the mesh. The result of the expression is a new field named "velocity_magnitude".

We will use Pseudocolor plots to examine the "pressure" and "velocity_magnitude" fields.

1. Go to *Add->Pseudocolor->Pressure*.
2. Click *Draw*.
3. Double click on the Pseudocolor plot to bring up the Pseudocolor plot attributes window.
4. Change the color table to *Spectral* and check the *Invert* button.
5. Click *Apply*.
6. Click *Draw*.
7. Click *Play* in the *Time* animation controls above the plot list on the main *GUI* window.

You will see the pressure field animate on the exterior of the mesh as the simulation evolves.

Experiment with:

- Setting the Pseudocolor plot limits.
- Hiding and showing the Mesh plot.

When you are done experimenting, stop animating over time steps using the *Stop* button.

Query the maximum pressure over time

We can use the "pressure" field to extract the heart beat signal. We want to find the maximum pressure value across the mesh elements at each time step of our dataset. VisIt provides a *Query over time* mechanism that allows us to extract this data.

First, we need to set our query options to use timestep as the independent variable for our query.

1. Go to *Controls->Query over time options*.
2. Select *Timestep*.
3. Click *Apply* and *Dismiss*.

Now we can execute the *Max* query on all of our time steps and collect the results into a curve.

1. Click on the Pseudocolor plot to make sure it is active.
2. Go to *Controls->Query* to bring up the Query window.
3. Select *Max*.
4. Check *Do Time Query*.
5. Click *Query*.

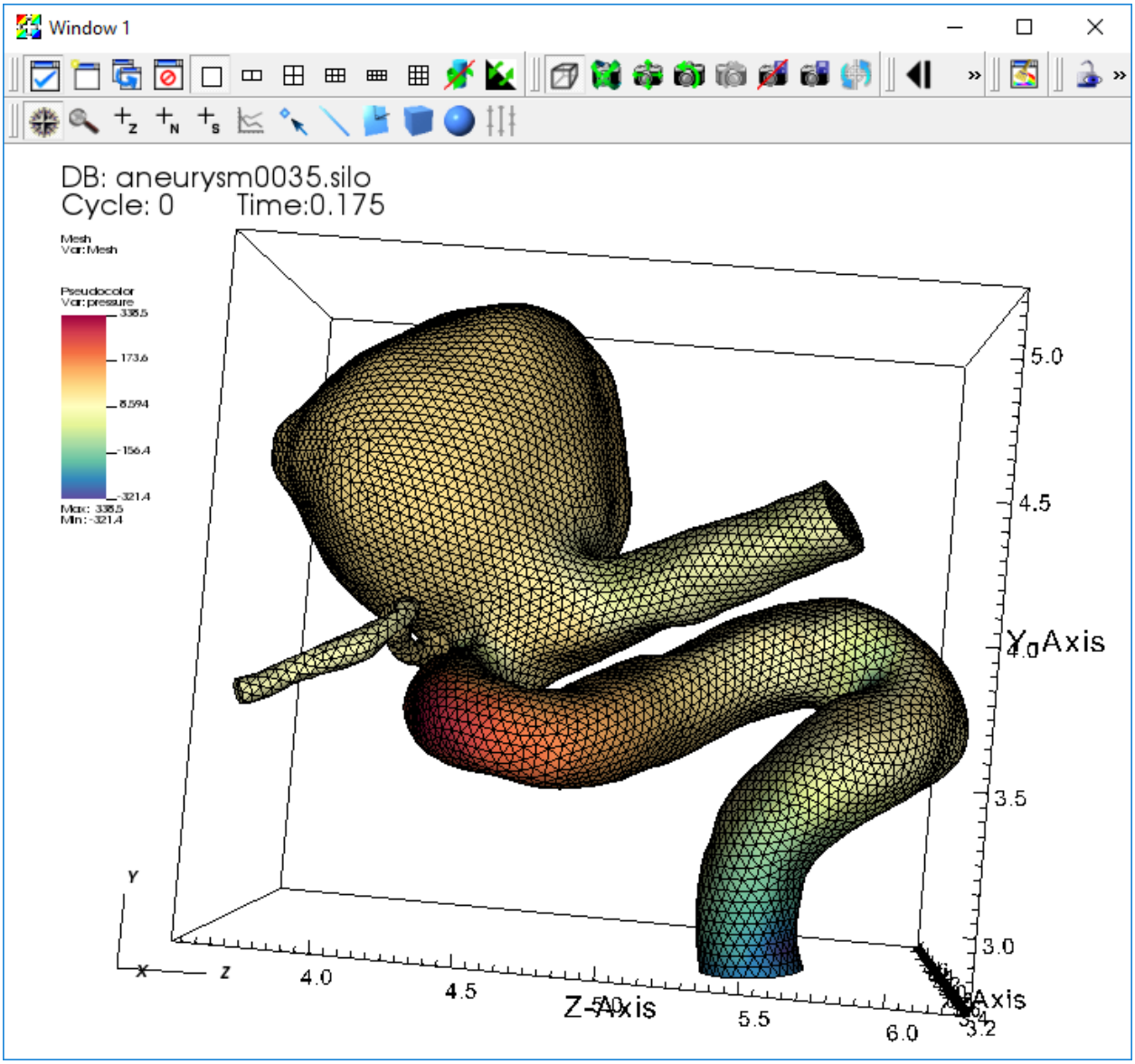


Fig. 6.33: The pseudocolor plot of the pressure.

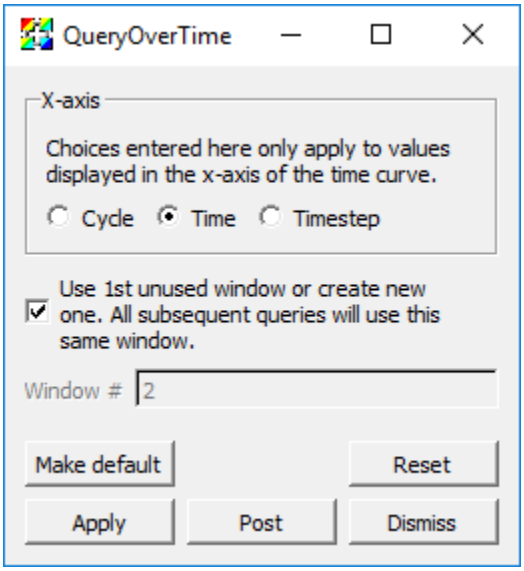


Fig. 6.34: The QueryOverTime attributes window.

This will process the simulation output files and create a new window with a curve that contains the maximum pressure value at each time.

Additional exercises

- How many heart beats does this dataset cover?
- Estimate the number of beats per minute of the simulated heart (each cycle is 0.005 seconds).

6.4.4 Contours and sub-volumes of high velocity

Examining the velocity magnitude

Next we create a Pseudocolor plot to look at the magnitude of the "velocity" vector field.

1. Delete all your existing plots by selecting them all and clicking *Delete*.
2. Go to *Add->Pseudocolor->velocity_magnitude*.
3. Open the Pseudocolor plot attributes window and set the color table options as before.
4. Click *Draw*.

Notice that the velocity at the surface of the mesh is zero. To get a better understanding of the flow inside the mesh, we will use operators to extract regions of high blood flow.

Creating a semi-transparent exterior mesh plot

When looking at features inside the mesh, it helps to have a partially transparent view of the whole mesh boundary for reference. We will add a Subset plot to create this view of the mesh boundary.

1. Uncheck *Apply operators to all plots*.
2. Go to *Add->Subset->Mesh*.

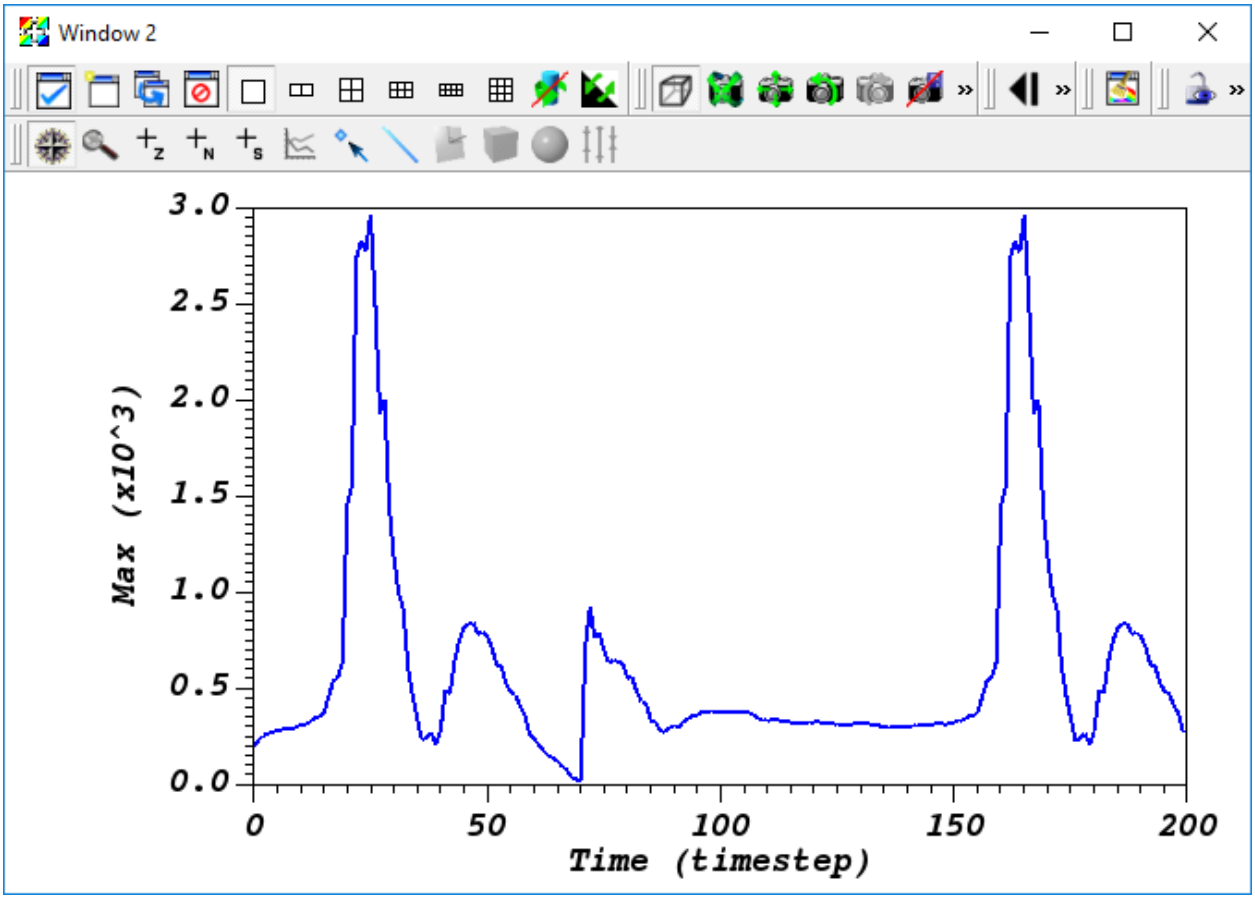


Fig. 6.35: The query over time plot.

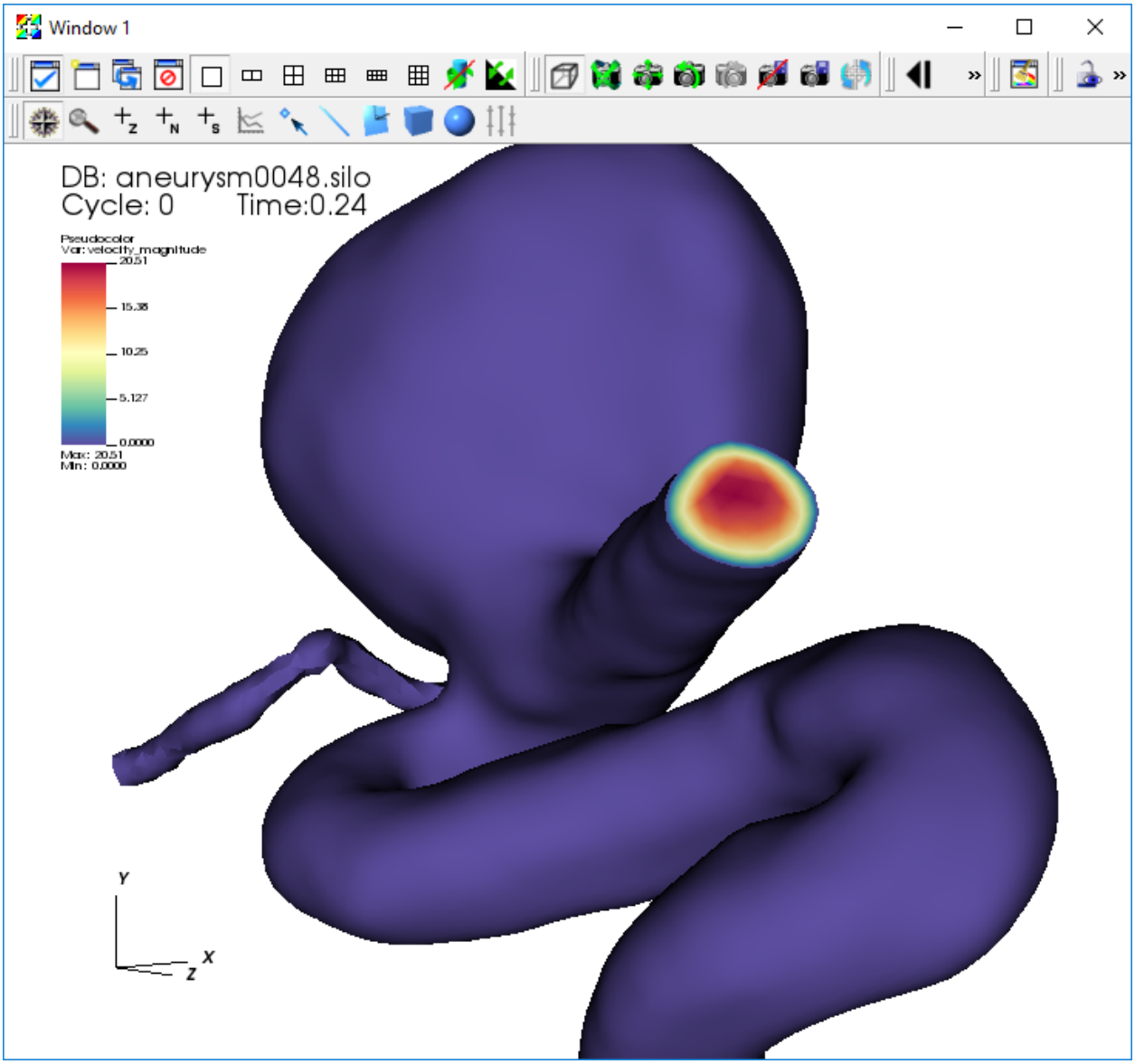


Fig. 6.36: The pseudocolor plot of the velocity magnitude.

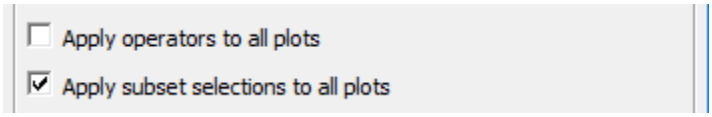


Fig. 6.37: The apply operators to all plots setting.

3. Open the Subset plot attributes window.
4. Change the color to *Light Blue*.
5. Set the *Opacity* slider to *25%*.
6. Click *Apply*.
7. Click *Draw*.

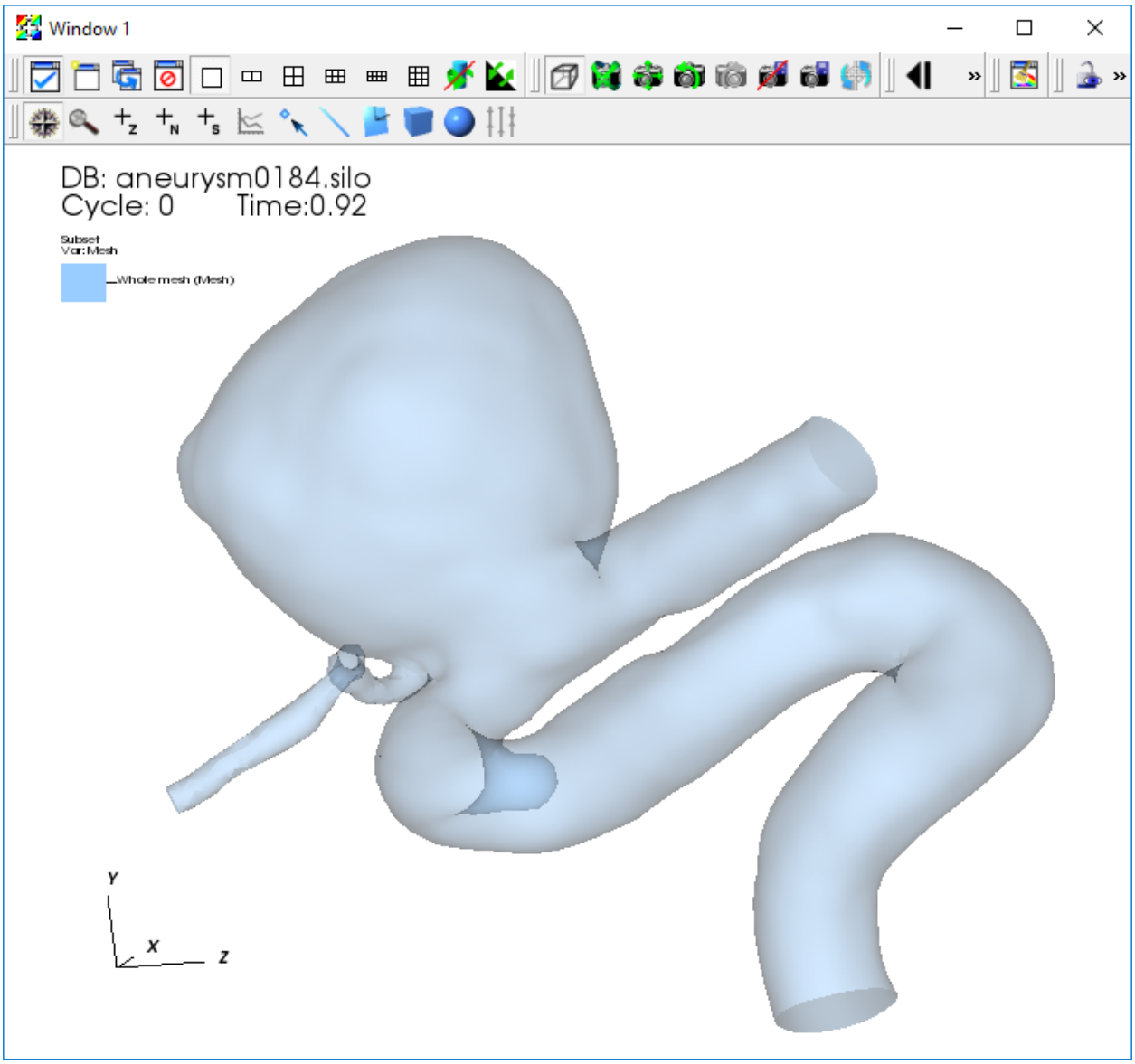


Fig. 6.38: The transparent subset plot.

Contours of high velocity

Now we will extract contour surfaces at high velocity values using the Isosurface operator.

1. Select the Pseudocolor plot in the plot list.

2. Go to *Operators->Slicing->Isosurface*.
3. Open the Isosurface operator attributes window.
4. Set *Select by* to *Value*, and use "10 15 20".
5. Click *Apply* and *Dismiss*.
6. Click *Draw* and press the *Play* button to animate the plot over time.

You will see the contour surfaces extracted from the "velocity_magnitude" field animate as the simulation evolves.

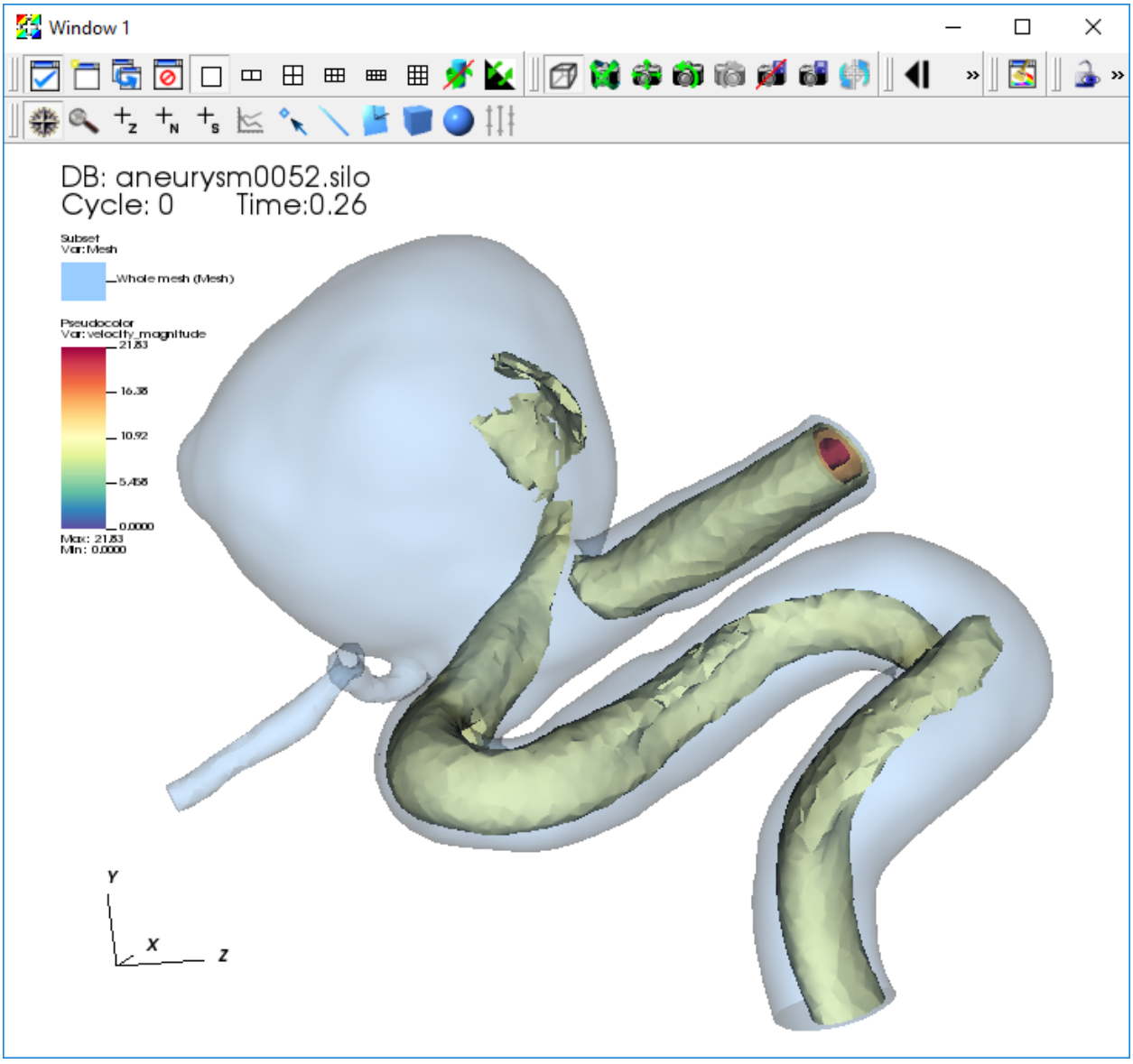


Fig. 6.39: The transparent subset plot with iso surfaces of velocity magnitude.

Sub-volumes of high velocity

As an alternative to contours, we can also extract the sub-volume between two scalar values using the Isovolume operator.

1. Click *Stop* to stop the animation.
2. Remove the Isosurface operator.
3. Go to *Operators->Selection->Isovolume*.
4. Open the Isovolume operator attributes window.
5. Set the *Lower bound* to “10” and the *Upper bound* to “20”.
6. Click *Apply* and *Dismiss*.
7. Click *Draw* and press the *Play* button to animate the plot over time.

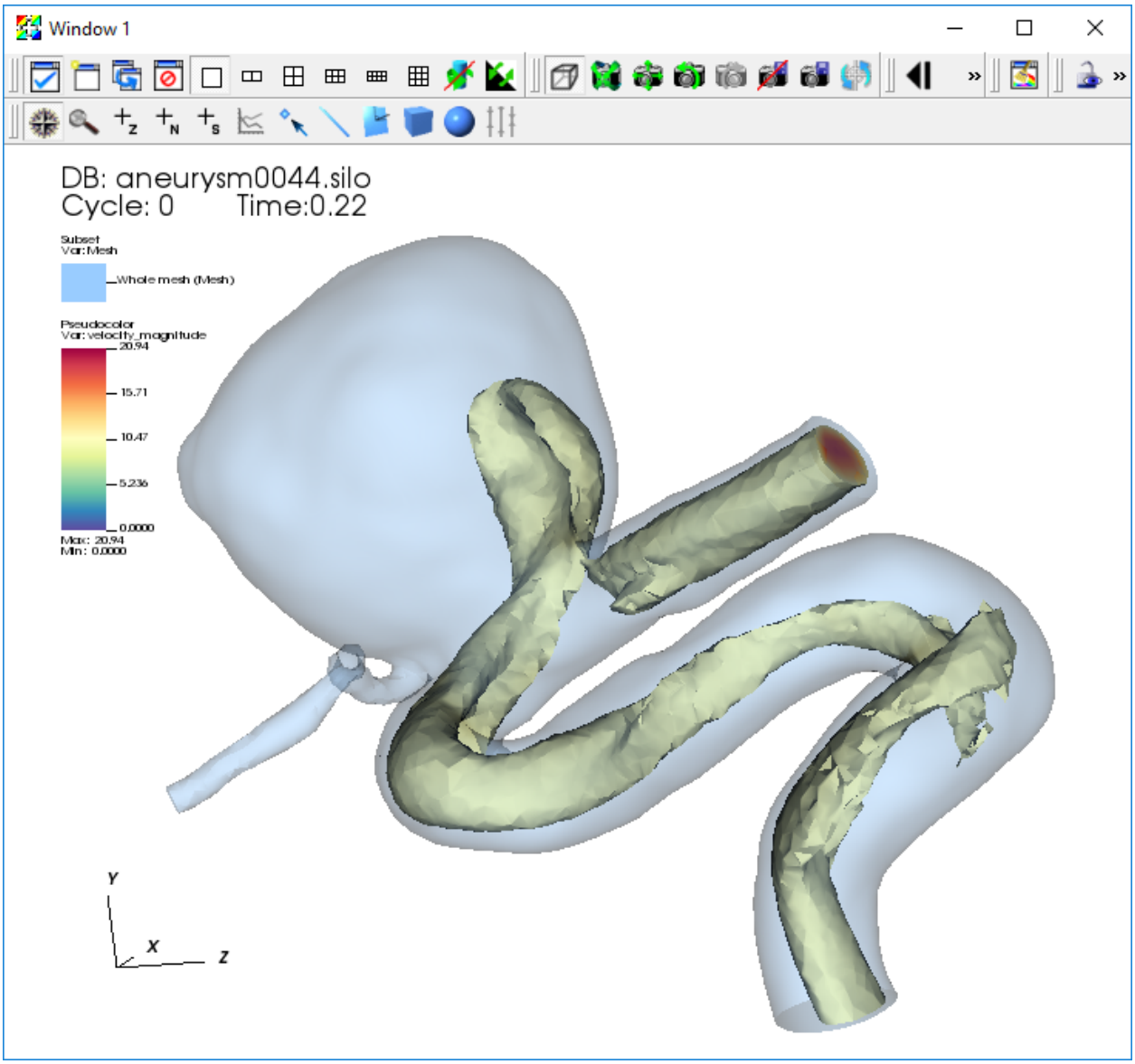


Fig. 6.40: The transparent subset plot with an iso volume of velocity magnitude.

6.4.5 Visualizing the velocity vector field

This section of the tutorial outlines using glyphs, streamlines, and pathlines to visualize the velocity vector field from the simulation.

Plotting the vector field directly with glyphs

VisIt's Vector plot renders a vector field at each time step as a collection of arrow glyphs. This allows us to see the direction of the vectors as well as their magnitude. We will create a vector plot to directly view the simulated "velocity" vector field.

1. Go to *Add->Vector->velocity*.
2. Open the Vector plot attributes window.
3. Go to the *Sampling* tab.
4. Set *Stride* to "5".
5. Go to the *Color* section on the *Data* tab.
6. Change the *Magnitude* to *Spectral*, and check the *Invert* option.
7. Go to the *Geometry* tab.
8. In the *Scale* section, set the *Scale* to "0.5".
9. In the *Style* section, set *Arrow body* to *Cylinder*.
10. In the *Rendering* section, set *Geometry Quality* to *High*.
11. Click *Apply* and *Dismiss*.
12. Click *Draw*.
13. Click *Play*.

Examining features of the flow field with streamlines

To explore the flow field further we will seed and advect a set of streamlines near the inflow of the artery. Streamlines show the path massless tracer particles would take if advected by a static vector field. To construct Streamlines, the first step is selecting a set of spatial locations that can serve as the initial seed points.

We want to center our seed points around the peak velocity value on a slice near the inflow of the artery. To find this location, we query a sliced pseudocolor plot of the "velocity_magnitude".

1. Go to *Add->Pseudocolor->velocity_magnitude*.
2. Open the Pseudocolor plot attributes window and set the color table options as before.
3. Go to *Operators->Slicing->Slice*.
4. Open the Slice operator attributes window.
5. In the *Normal* section set *Orthogonal* to *Y Axis*.
6. In the *Origin* section select *Point* and set the value to "3 3 3".
7. In the *Up Axis* section uncheck *Project to 2D*.
8. Click *Apply* and *Dismiss*.
9. Click *Draw*.

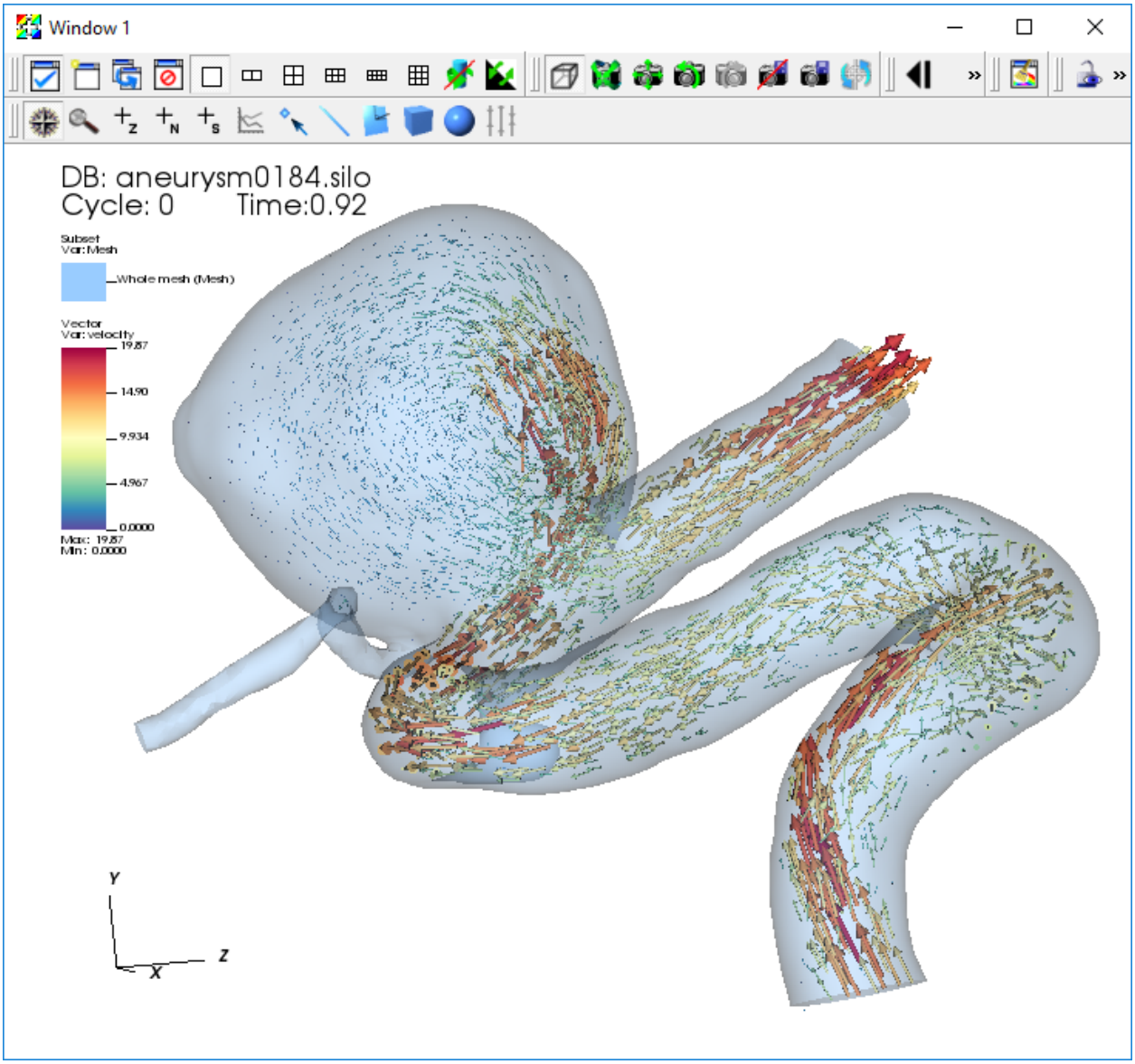


Fig. 6.41: The vector plot of velocity.

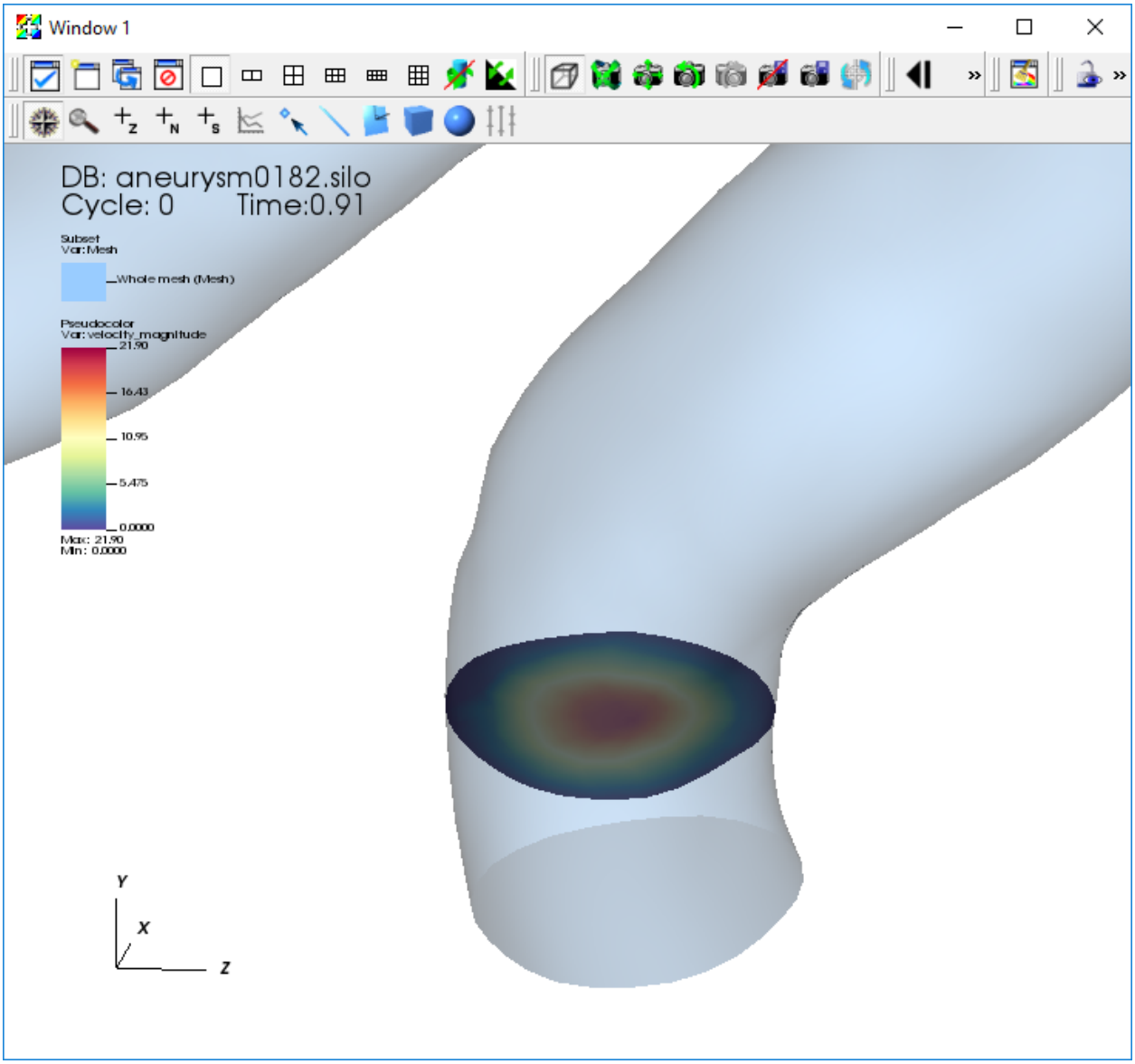


Fig. 6.42: The velocity magnitude on a slice.

Query to find the maximum velocity on the slice

1. Click to make sure the Pseudocolor plot of your "velocity_magnitude" slice is active.
2. Go to *Controls->Query*.
3. Select *Max*.
4. Select *Actual Data*.
5. Click *Query*.

This will give you the maximum scalar value on the slice and the x,y,z coordinates of the node associated with this value. We will use the x,y,z coordinates of this node to seed a set of streamlines.

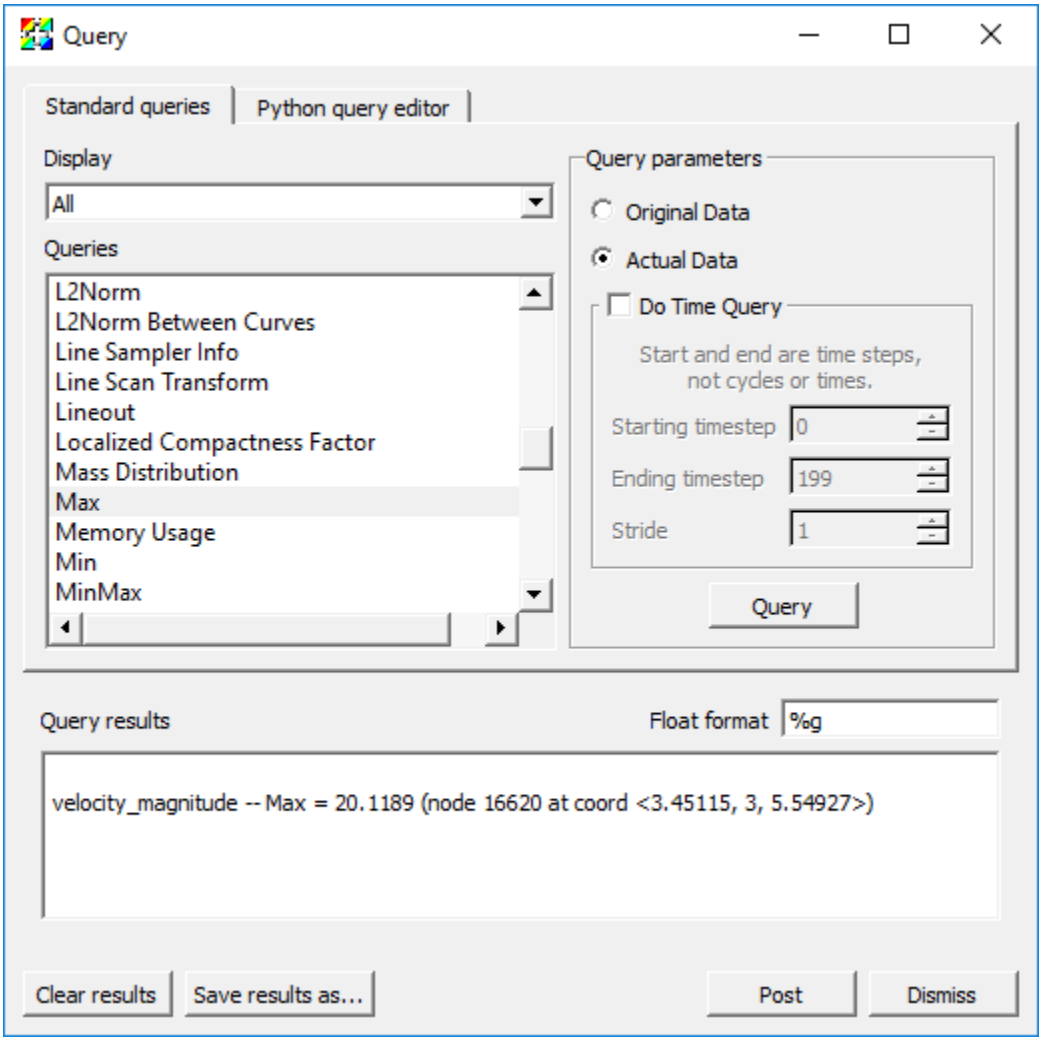


Fig. 6.43: The result of the velocity magnitude query.

Plotting streamlines of velocity

1. Go to *Add->Pseudocolor->operators->IntegralCurve->velocity*.
2. Open the IntegralCurve operator attributes window.

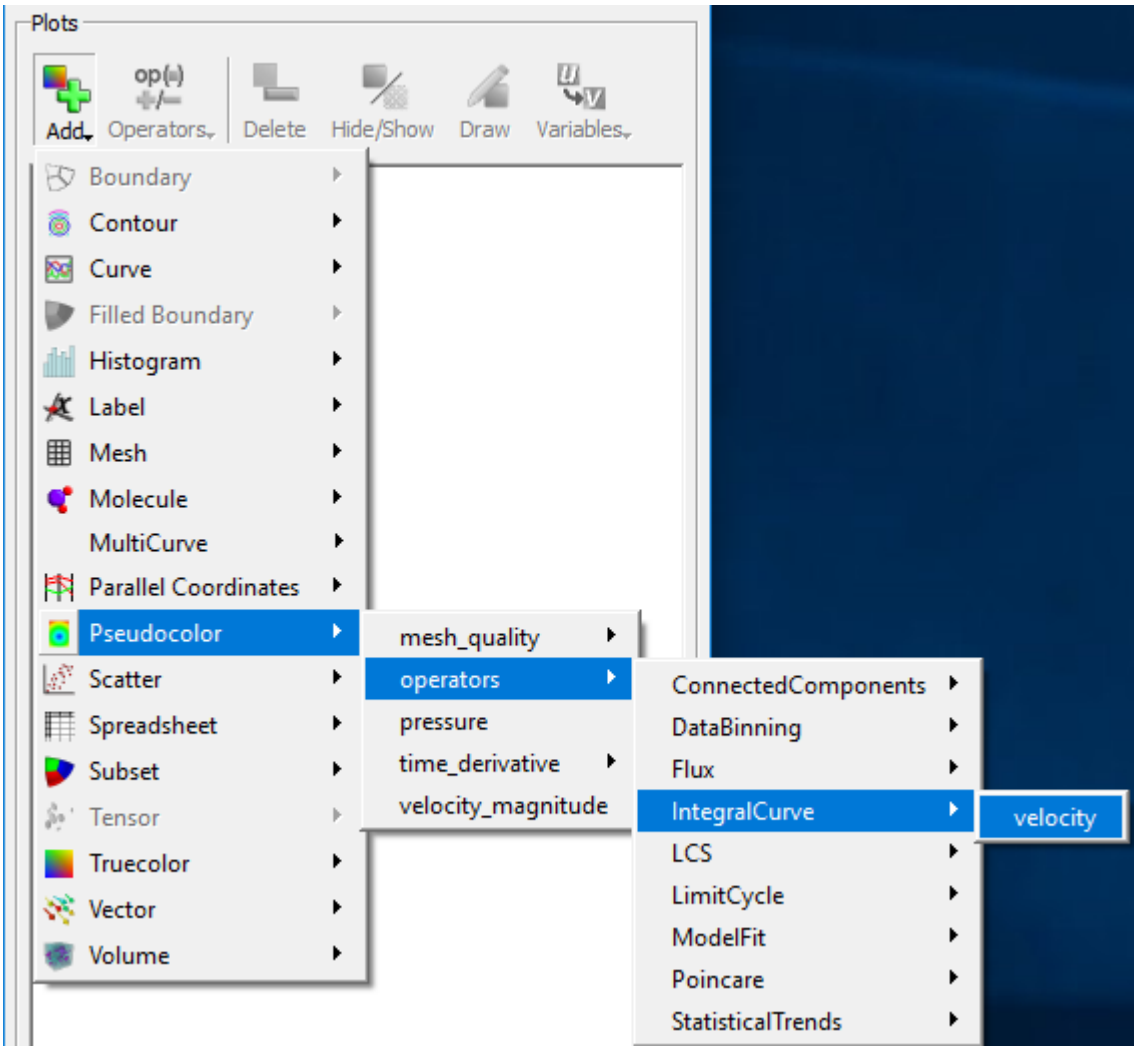


Fig. 6.44: Creating a streamline plot with the IntegralCurve operator.

3. Go to the *Source* section on the *Integration* tab.
4. Set the *Source type* to *Circle*.
5. Set the *Origin* to the value returned from the max query: "3.45115 3 5.54927", excluding any commas in the input text box.
6. Set the *Normal* to the y-axis: "0 1 0".
7. Set the *Up axis* to the z-axis: "0 0 1".
8. Set the *Radius* to "0.12".
9. Go to the *Sampling* section.
10. Set *Sampling along:* to *Boundary*.
11. Set *Samples in Theta:* to "12".
12. Go to the *Advanced* tab.
13. In the *Warnings* section, uncheck all of the warning checkboxes.
14. Click *Apply* and *Dismiss*.
15. Open the Pseudocolor plot attributes window.
16. Go to the *Data* tab.
17. In the *Color* section set the *Color table* to *Reds*.
18. Go to the *Line* section on the *Geometry* tab.
19. Set *Line type* to *Tubes*.
20. Set *Tail* to *Sphere*.
21. Set *Head* to *Cone*.
22. Set the head and tail *Radius* to "0.03".
23. Click *Apply* and *Dismiss*.
24. Click *Draw*.
25. Use the time slider controls to view a few time steps.

Examining features of the flow field with pathlines

Finally, to explore the time varying behavior of the flow field we will use pathlines. Pathlines show the path massless tracer particles would take if advected by the vector field at each timestep of the simulation.

We will modify our previous IntergralCurve options to create pathlines.

1. Set the time slider controls to the first timestep.
2. Open the IntegralCurve attributes window.
3. Go to the *Appearance* tab.
4. In the *Streamlines vs Pathlines* section select *Pathline*.
5. In the *Pathlines Options* section set *How to perform interpolation over time* to *Mesh is static over time*.
6. Click *Apply* and *Dismiss*.

This will process all 200 files in the dataset and construct the pathlines that originate at our seed points.

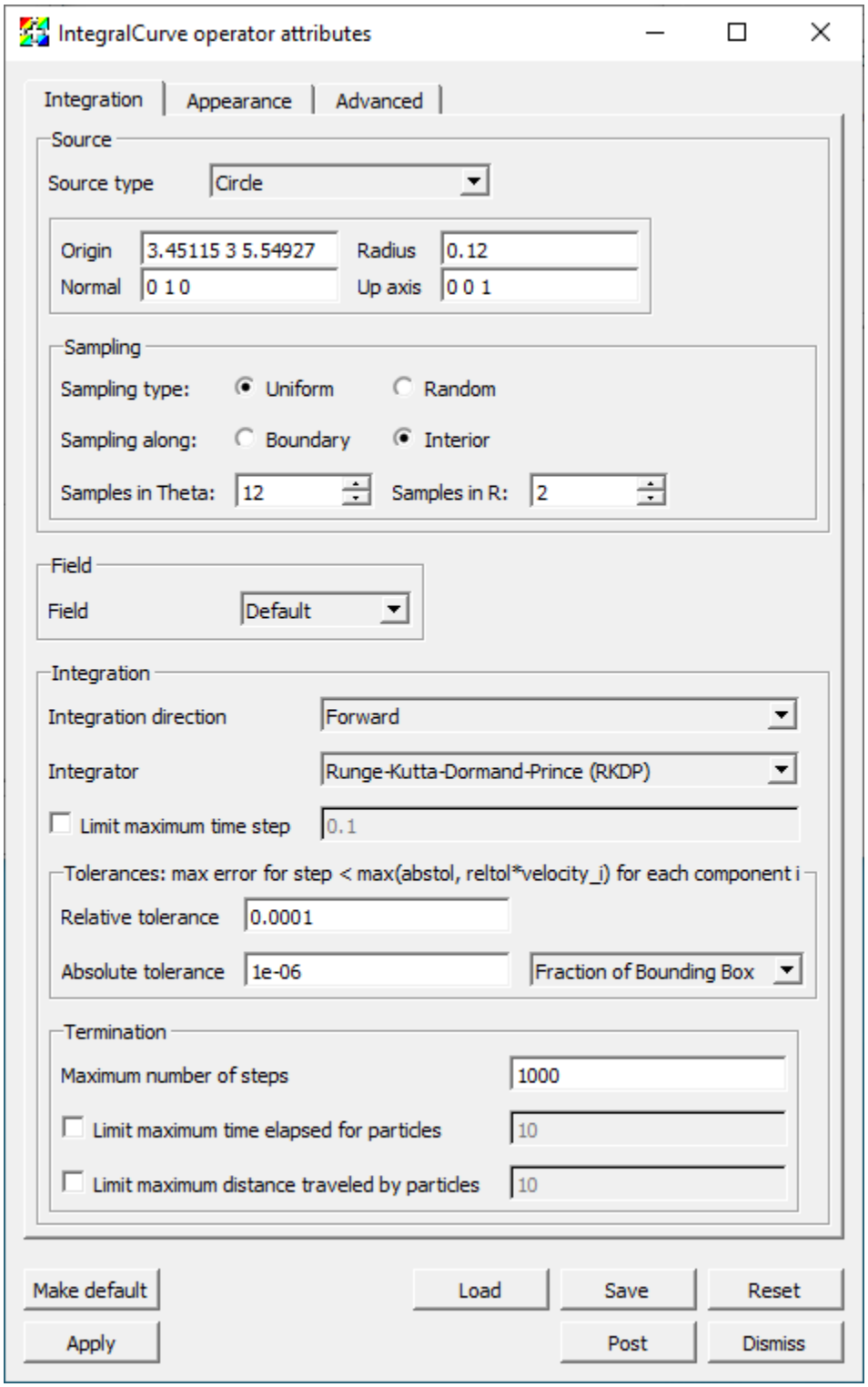


Fig. 6.45: The IntegralCurve operator attributes.

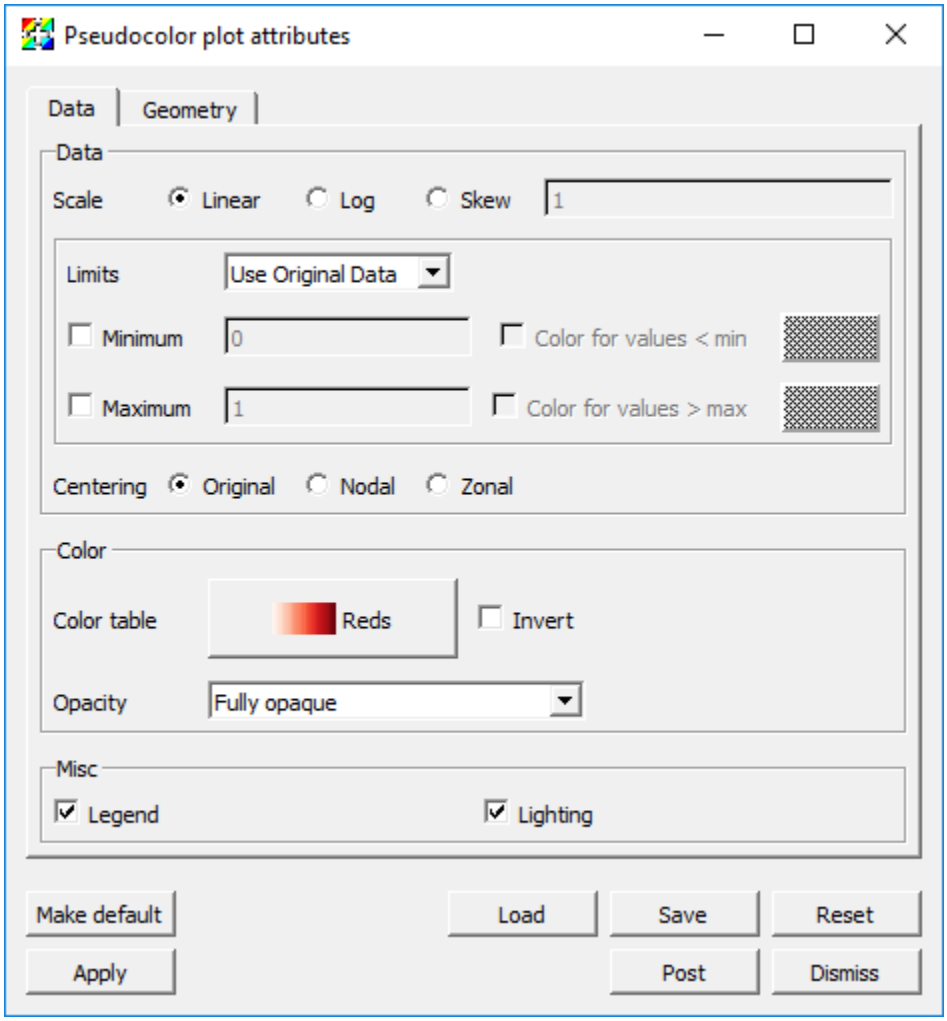


Fig. 6.46: The Pseudocolor attributes for the streamline data.

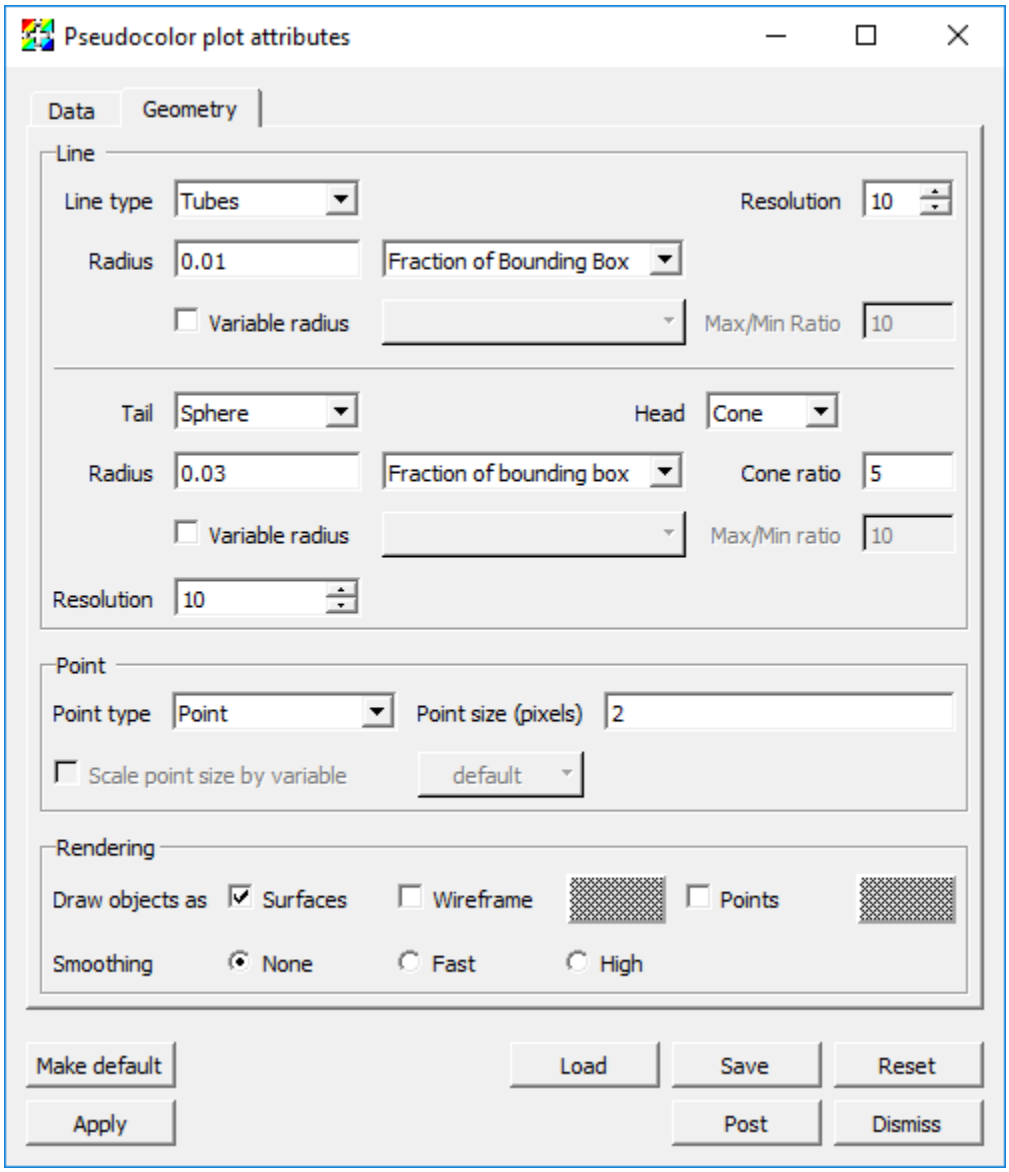


Fig. 6.47: The Pseudocolor attributes for the streamline geometry.

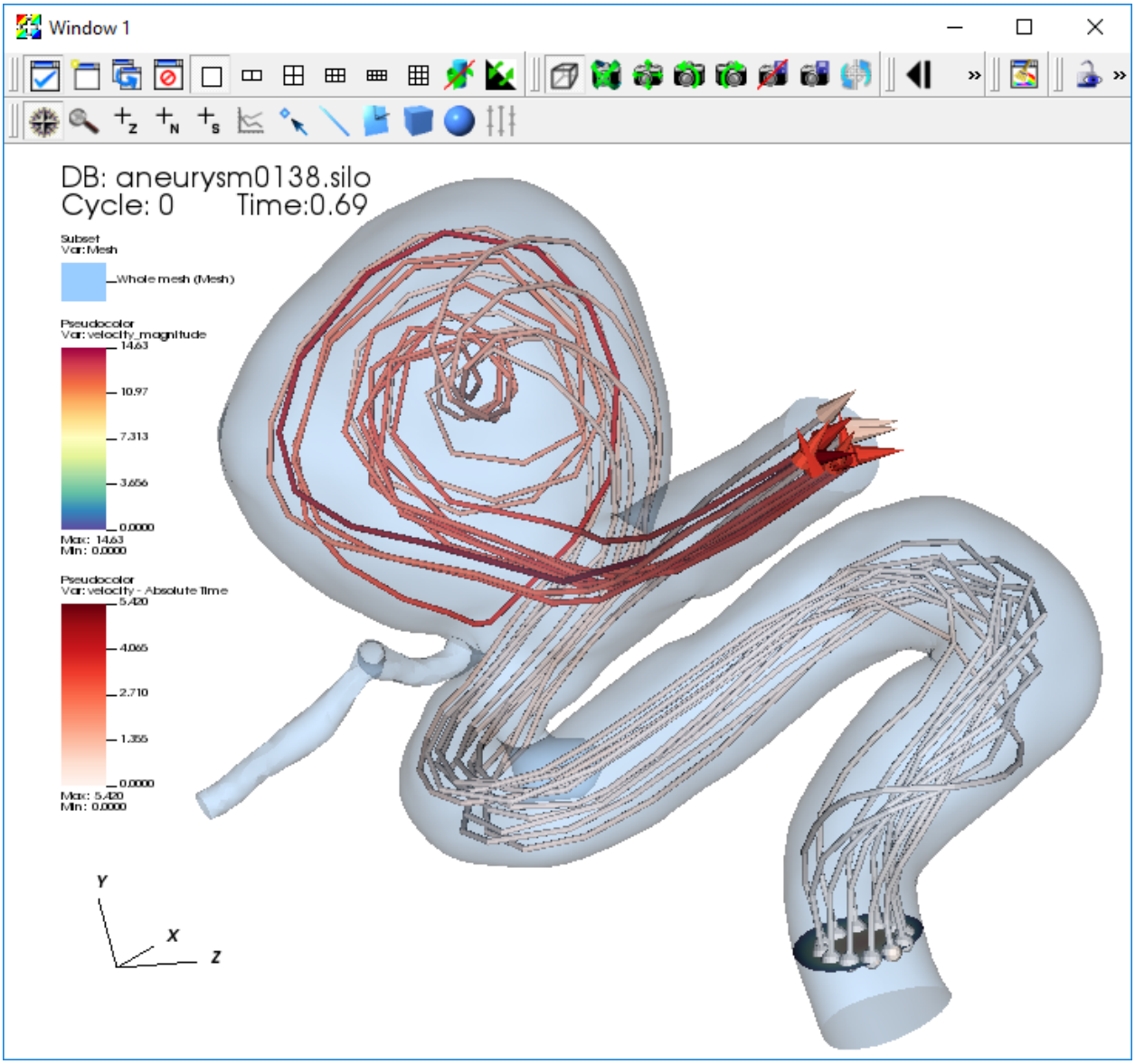


Fig. 6.48: The streamlines of velocity.

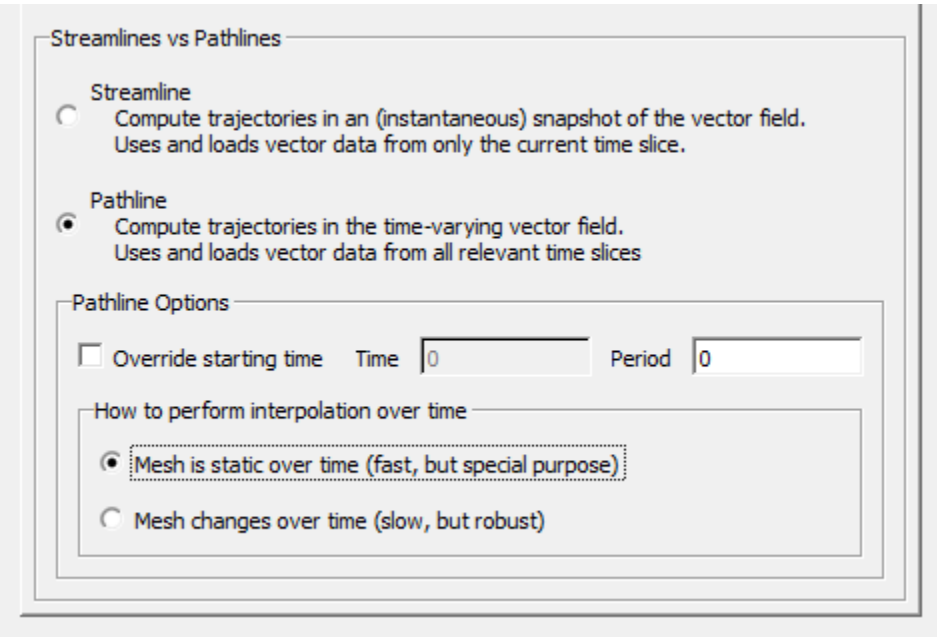


Fig. 6.49: The IntegralCurve operator pathline attributes.

6.4.6 Calculating the flux of a velocity field through a surface

To calculate a flux, we will need the original velocity vector, the normal vector of the surface, and VisIt's Flux Operator. We will calculate the flux through a cross-slice located at Y=3, at the beginning of the artery.

Creating the slice and showing velocity glyphs

First we will directly plot the velocity vectors that exist on the slice through the 3D mesh.

1. Delete any existing plots.
2. Go to *Add->Vector->velocity*.
3. Open the Vector plot attributes window.
4. Go to the *Sampling* tab and set the *Fixed number* to "40".
5. Go to the *Geometry* tab.
6. Set *Arrow body* to *Cylinder*.
7. Set *Geometry Quality* to *High*.
8. Click *Apply* and *Dismiss*.
9. Go to *Operators->Slicing->Slice*.
10. Open the Slice operator attributes window.
11. Set *Normal* to *Arbitrary* and to "0 1 0".
12. Set *Origin* to *Intercept* and to "3".
13. Uncheck *Project to 2D*.
14. Click *Make default*, *Apply* and *Dismiss*.

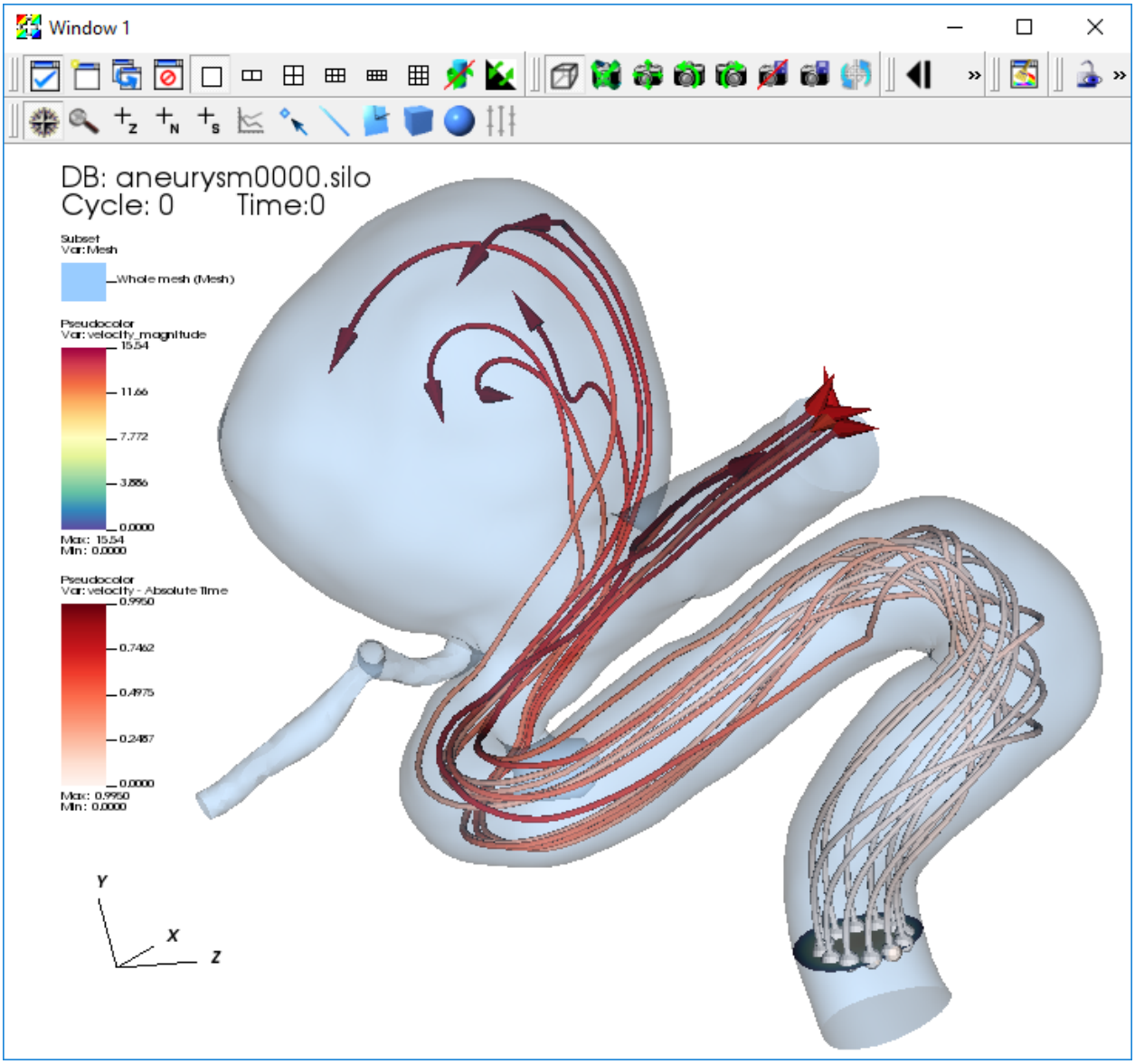


Fig. 6.50: The pathlines of velocity.

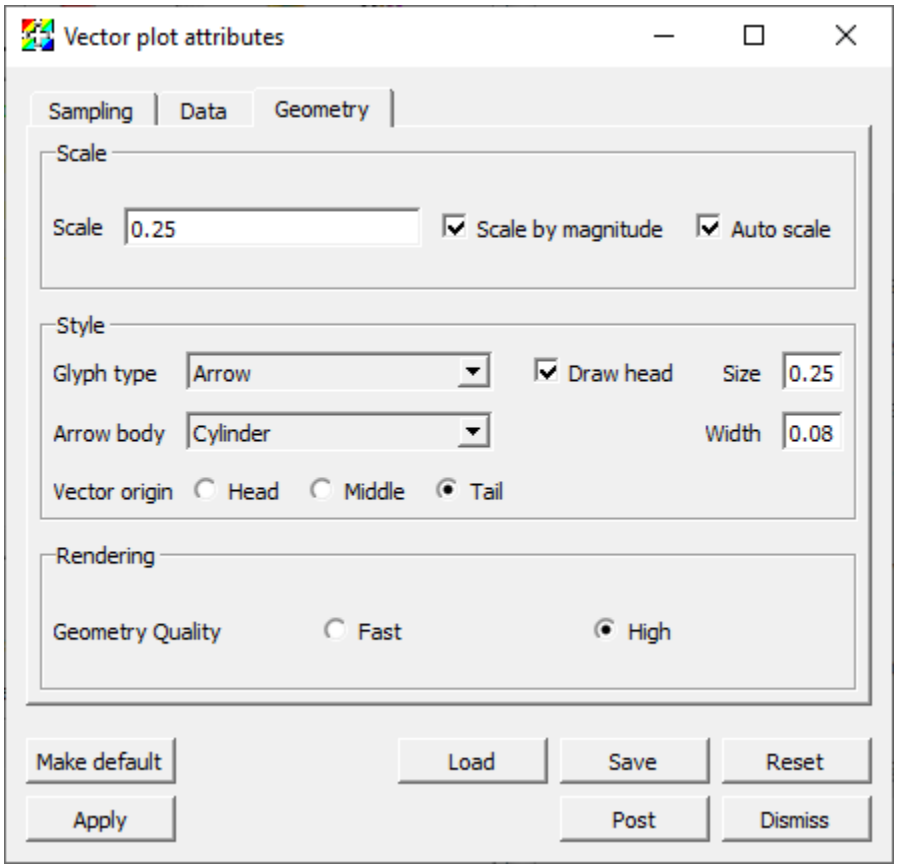


Fig. 6.51: The Vector plot attributes.

15. Click *Draw*.

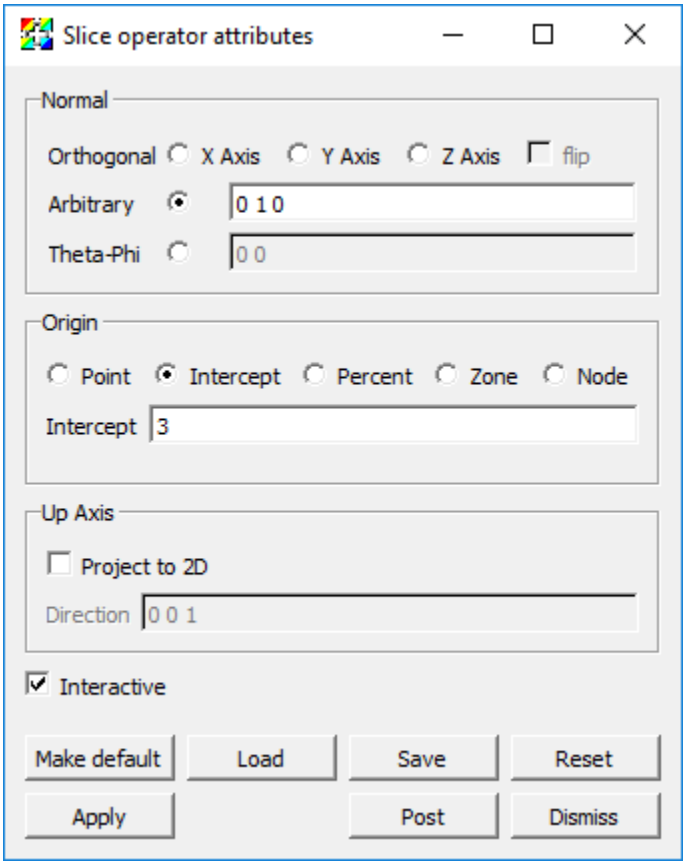


Fig. 6.52: The Slice operator attributes.

In order to give some context to the Vector plot of velocity on the slice let's add a Pseudocolor plot of velocity_magnitude on the same slice and a Mesh plot.

16. Go to *Add->Pseudocolor->velocity_magnitude*.
17. Open the Pseudocolor plot attributes window.
18. Set *Limits* to *Use Current Plot*.
19. Click *Apply* and *Dismiss*.
20. Go to *Operators->Slicing->Slice*.
21. Click *Draw*.
22. Go to *Add->Mesh->Mesh*.
23. Open the Mesh plot attributes window.
24. Set *Mesh color* to *Custom* and select a medium grey color.
25. Click *Apply* and *Dismiss*.
26. Click *Draw*.
27. Zoom in to explore the plot results.

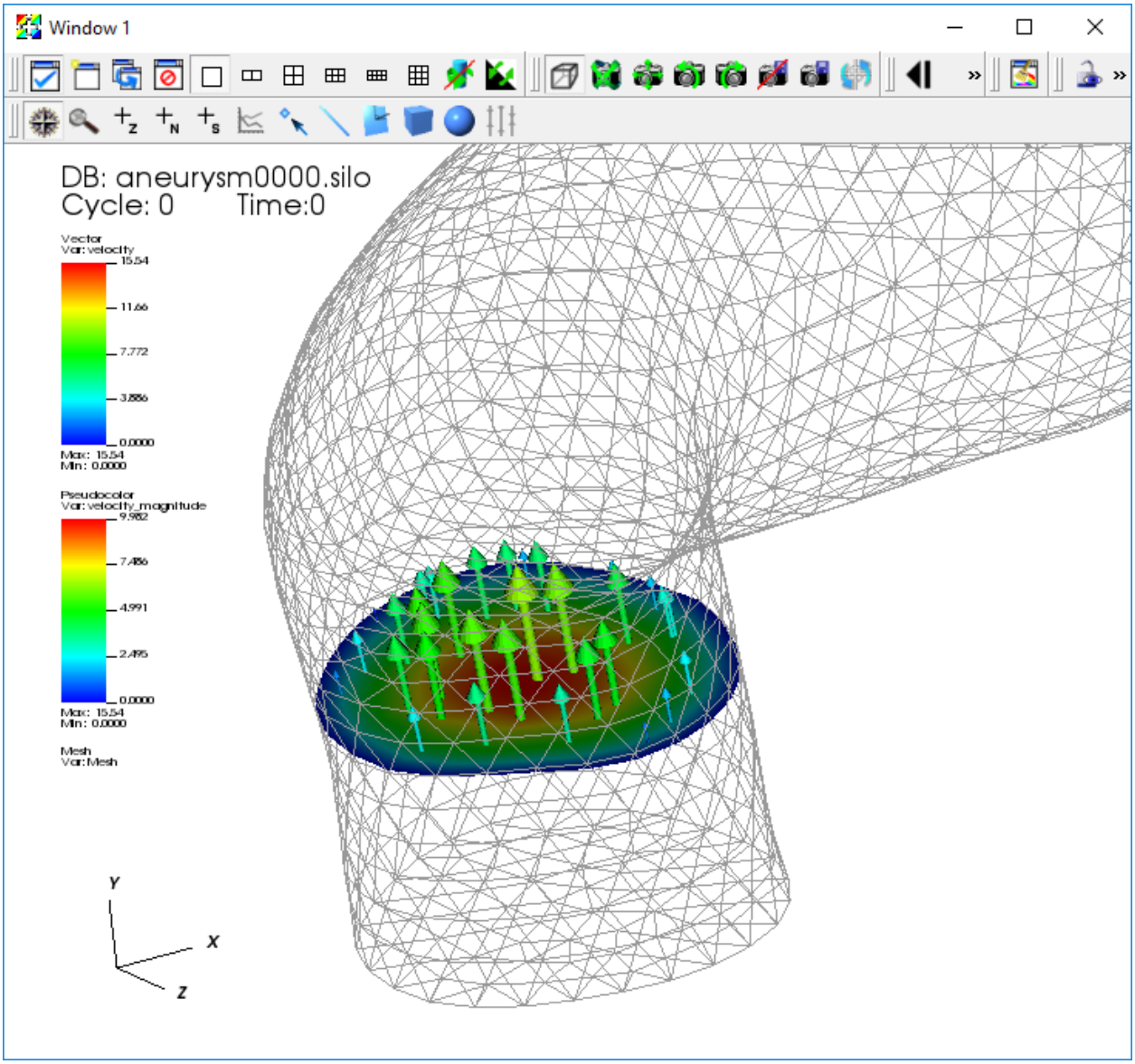


Fig. 6.53: The velocity on the slice.

The Vector plot uses glyphs to draw portions of the instantaneous vector field. The arrows are colored according to the speed at each point (the magnitude of the velocity vector). Next we create an expression to evaluate the vectors normal to the Slice. These normals should all point in the Y direction.

Creating a vector expression and using the DeferExpression operator

We will use VisIt's pre-defined expression to evaluate the normals on a cell-by-cell basis.

1. Go to *Controls->Expressions*.
2. Click *New*.
3. Change the *Name* to "normals" and the *Type* to *Vector mesh variable*.
4. Go to *Insert function->Miscellaneous->cell_surface_normal* in the *Standard editor* tab.
5. Go to *Insert variable->Meshes->Mesh* in the *Standard editor* tab.

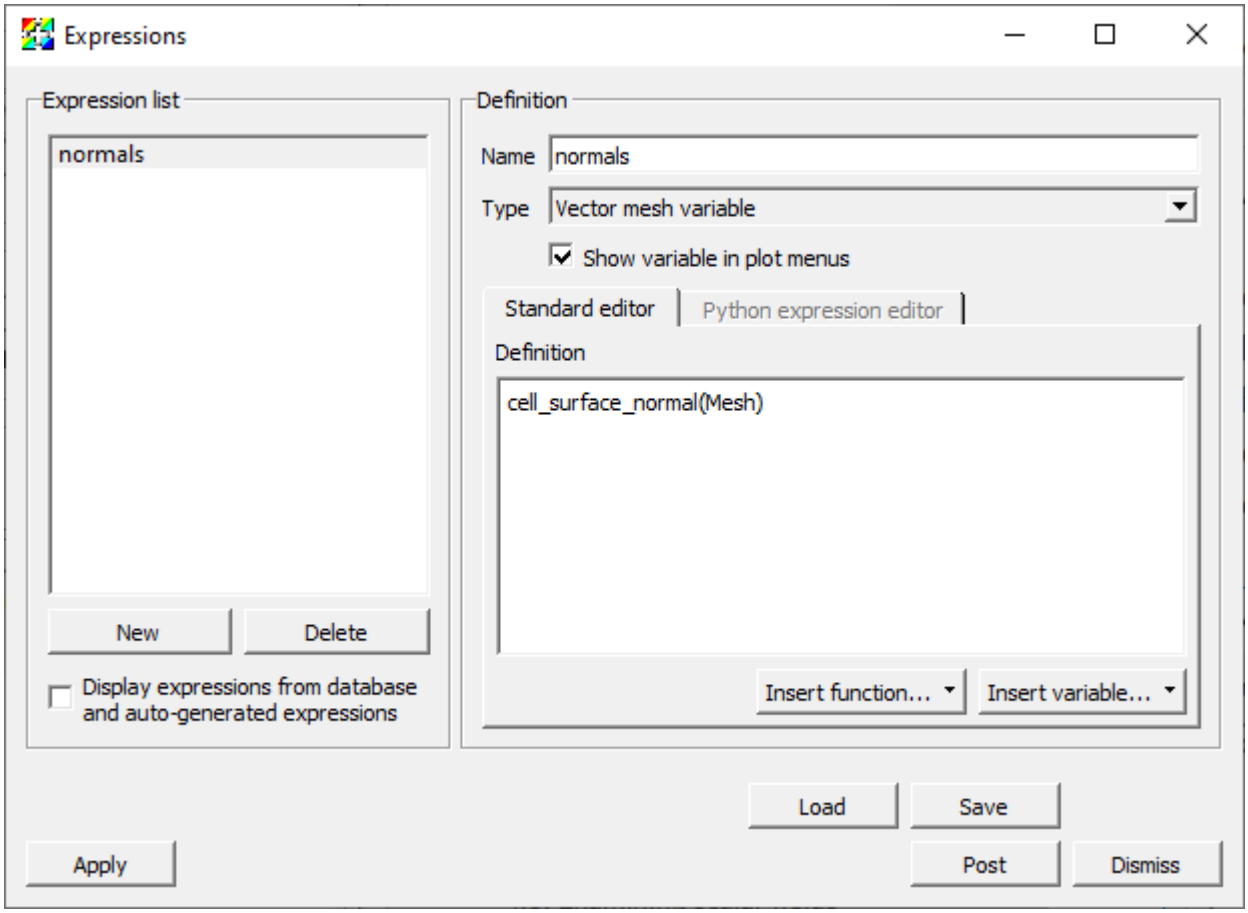


Fig. 6.54: The Expressions window.

6. Click *Apply* and *Dismiss*.
7. Return to the Vector plot and change its variable to "normals".

You will then get the error message saying: *The 'normals' expression failed because The Surface normal expression can only be calculated on surfaces. Use the ExternalSurface operator to generate the external surface of this object. You must also use the DeferExpression operator to defer the evaluation of this expression until after the external*

surface operator. In fact, VisIt cannot use the name *Mesh* which refers to the original 3D mesh. It needs to defer the evaluation until after the Slice operator is applied. Thus, we need to add the Defer Expression operator.

8. Go to *Operators->Analysis->DeferExpression*.
9. Open the DeferExpression operator attributes window.
10. Go to *Variables->Vectors->normals*.

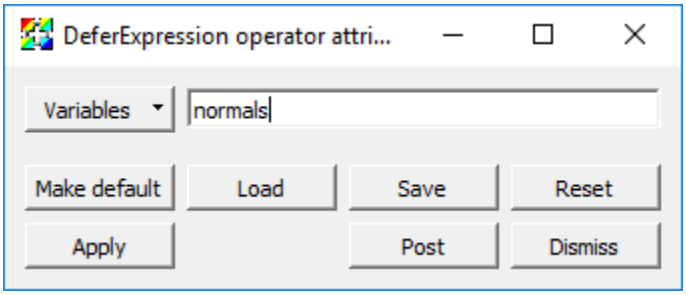


Fig. 6.55: The DeferExpression window.

11. Click *Apply* and *Dismiss*.
12. Click *Draw*.
13. Verify that all your normals point in the up (Y) direction.

Calculating the flux on the slice

We are now ready for the final draw.

1. Go to *Add->Pseudocolor->operators->Flux->Mesh*.
2. Go to *Operators->Slicing->Slice*.
3. Open the Slice operator attributes window.
4. Verify that the default values previously saved are used.
5. Move the Slice operator above the Flux operator.
6. Go to *Operators->Analysis->DeferExpression*.
7. Move the DeferExpression operator above the Flux operator just below the Slice operator.
8. Open the Flux operator attributes window.
9. Set the *Flow field* to "velocity".
10. Click *Apply* and *Dismiss*.
11. Click *Draw*.

Verify that you have a display that is cell-centered, and that will vary with the Time slider

12. Get the numerical value of the flux by query-ing for the *Weighted Variable Sum*.

6.5 Potential Flow

This tutorial demonstrates VisIt's features while exploring results from simple simulations of potential based flow around an obstruction, specifically an airfoil. Potential flow assumes irrotational flow. That is, there is no rotational

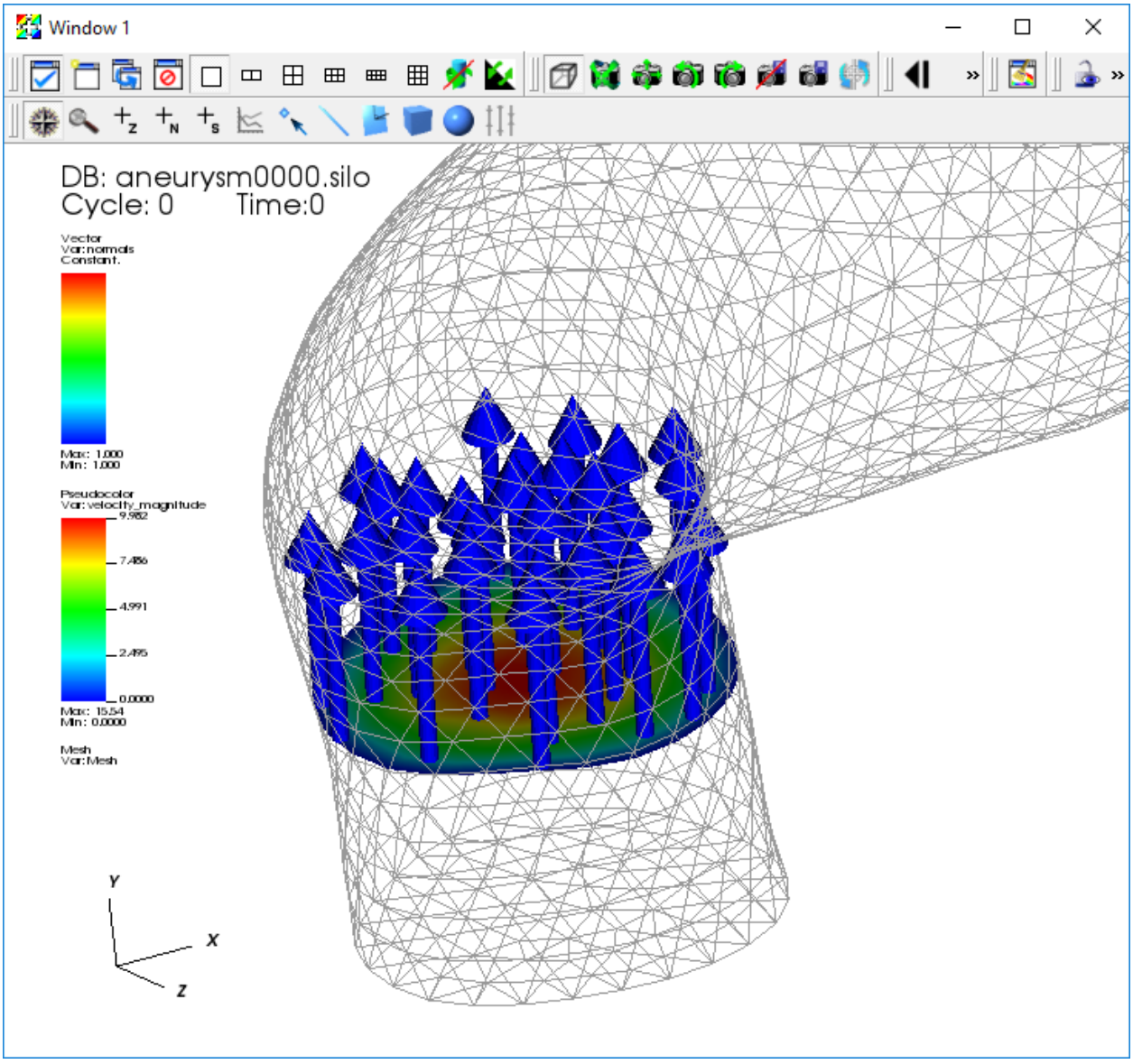


Fig. 6.56: The Vector plot of the normals.

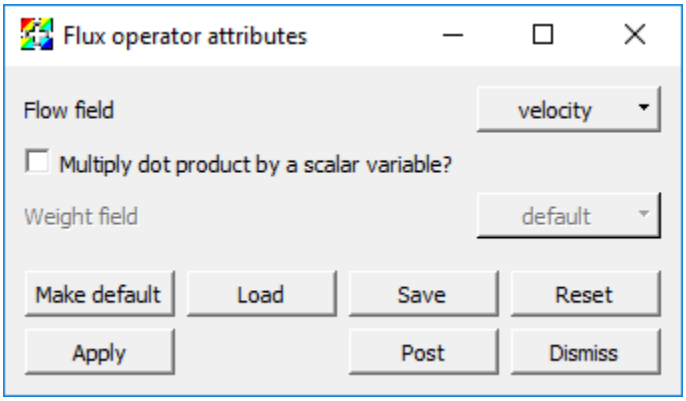


Fig. 6.57: The Flux operator attributes window.

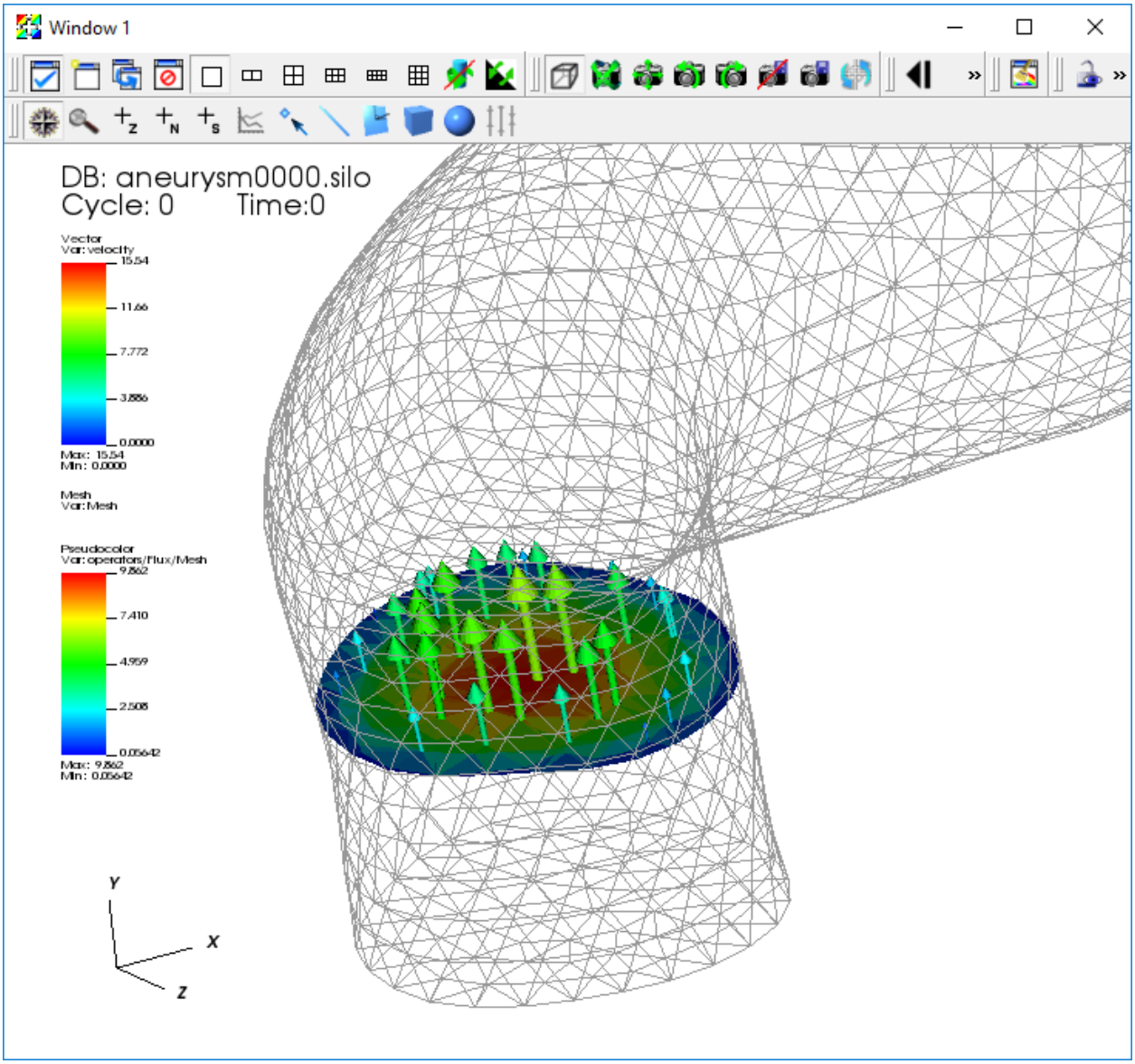


Fig. 6.58: The Vector plot of the flux.

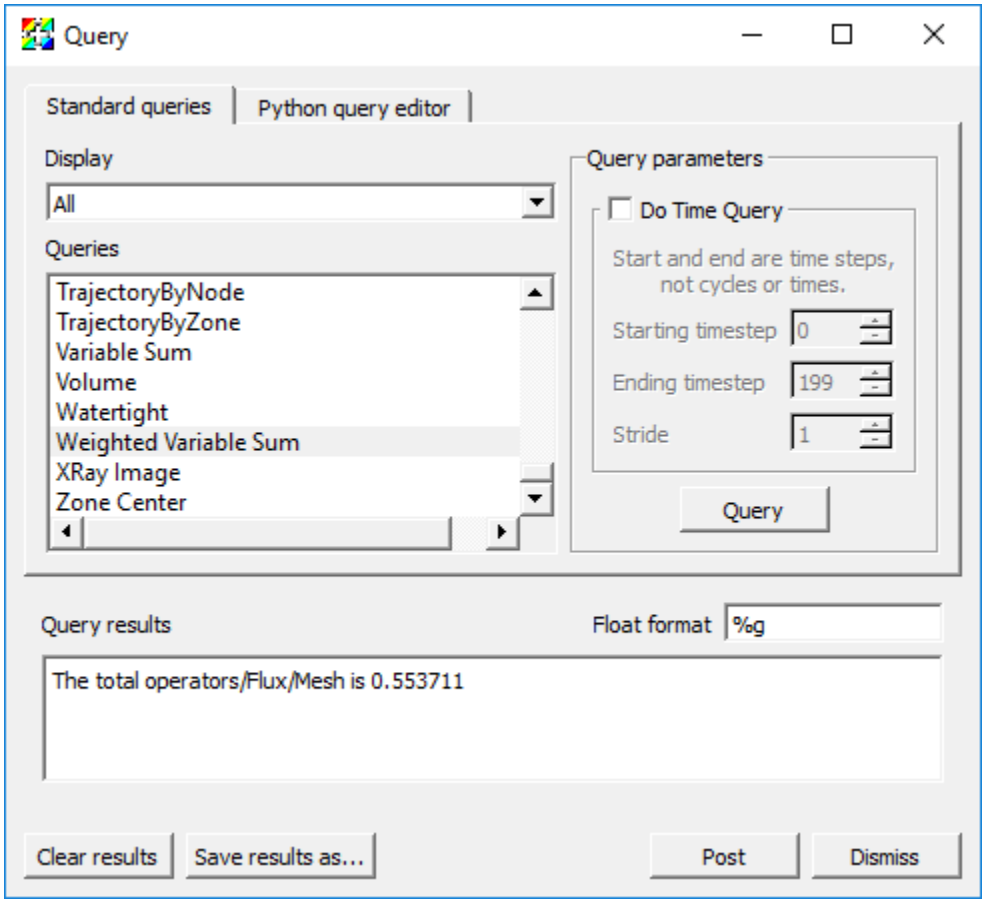


Fig. 6.59: The result of the Weighted Variable Sum query.

motion in the flow, no vortices or eddies. This assumption is valid for low velocities and certain types of gases/fluids and obstructions. When the flow does involve rotation, a more complex solution involving Navier-Stokes equations is required.

The potential flow solver is a mini-app developed using the MFEM finite element library. The example is available for this tutorial thanks to Aaron Fischer and Mark Miller of LLNL. The data set includes VTK output files for a set of solutions where the angle of attack of the velocity varies from -5 degrees to 25 degrees.

6.5.1 Open the dataset

This tutorial uses the potential flow dataset.

1. Download the potential flow dataset.
2. Click on the *Open* icon to bring up the File open window.
3. Navigate your file system to the folder containing "potential_flow_ang_sweep.visit".
4. Highlight the file "potential_flow_ang_sweep.visit" and then click *OK*.

6.5.2 Plotting the mesh topology

First we will examine the mesh used by the solver.

Create a Mesh plot

1. Go to *Add->Mesh->main*.
2. Click *Draw*.

After this, the mesh plot is rendered in VisIt's Viewer window. This is a 2D mesh, modify the view by planning and zooming in the viewer window. Zoom in near the airfoil and look at the mesh structure.

Modify the Mesh plot settings

1. Double click on the Mesh plot to open the Mesh plot attributes window.
2. Experiment with settings for:

- *Mesh color*
- *Opaque color*
- *Opaque mode*

You will need to click *Apply* to commit the settings to your plot.

6.5.3 Examining the velocity magnitude

In addition to the mesh topology, this dataset provides a vector field "v", representing the velocity, associated with the mesh vertices.

VisIt automatically defines an expression that allows us to use the magnitude of the "v" vector field as a scalar field on the mesh. The result of the expression is a new field named "v_magnitude".

We will use Pseudocolor plots to examine the "pressure" and "velocity_magnitude" fields.

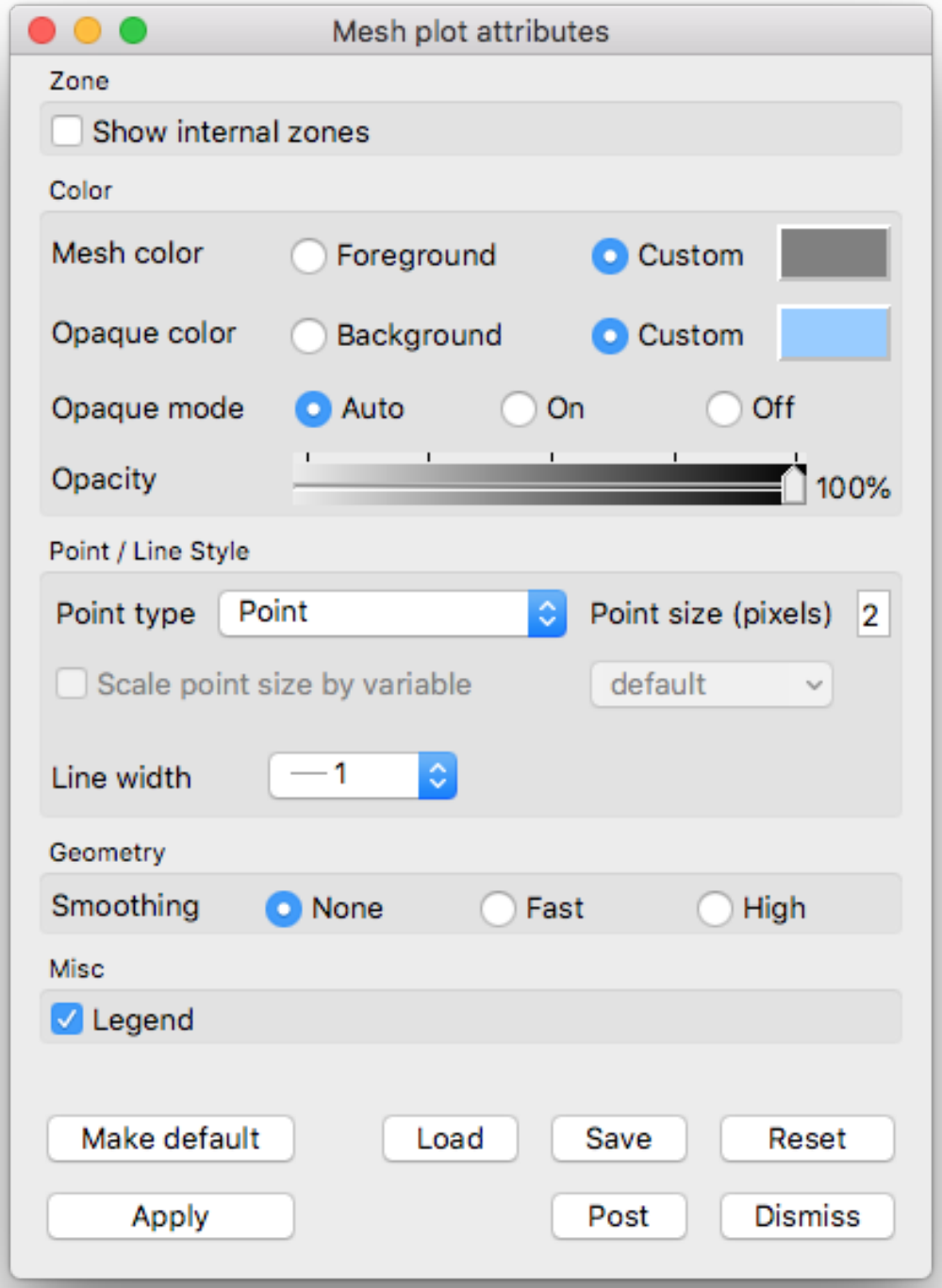


Fig. 6.60: Example mesh plot settings for the Potential Flow data.

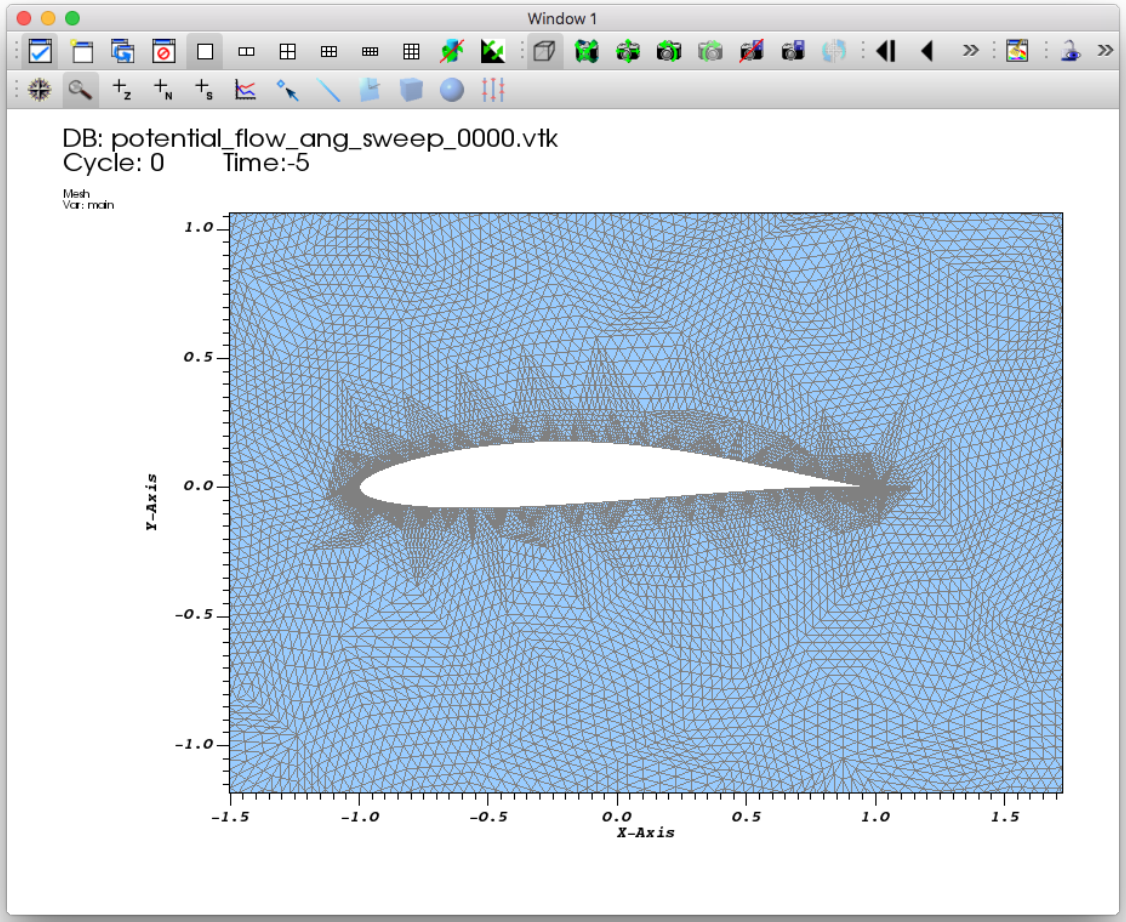


Fig. 6.61: Example mesh plot result the Potential Flow data.

1. Go to *Add->Pseudocolor->v_magnitude*.
2. Click *Draw*.
3. Double click on the Pseudocolor plot to bring up the Pseudocolor plot attributes window.
4. In the *Limits* section, enable the *Maximum* checkbox and set the limit to *1*.
5. In the *Color* section,cChange the color table to *Spectral* and check the *Invert* button.
6. Click *Apply*.
7. Click *Draw*.
8. Drag the *Time* animation controls above the plot list on the main *GUI* window.

You will see the velocity magnitude solutions for the different angles of attack.

Experiment with the *Color for values > max* option to see where the range is being clipped.

Contours of velocity magnitude

Now we will add an additional plot to view velocity magnitude contours

1. Go to *Add->Contour->v_magnitude*.
2. Double click on the Contour plot to bring up the Contour plot attributes window.
3. In the *Contour Levels* section, enable the *Maximum* checkbox and set the limit to *1*.
4. In the *Lines* section, set the *Line width* to *2*.
5. Click *Apply*.
6. Click *Draw*.
7. Drag the *Time* animation controls above the plot list on the main *GUI* window.

You will see the contours of the velocity magnitude solutions for the different angles of attack.

Delete the contour plot when you are finished exploring, but keep the pseudocolor plot.

6.5.4 Visualizing the velocity vector field

This section of the tutorial outlines using glyphs and streamlines to visualize the velocity vector field from the simulation.

Plotting the vector field directly with glyphs

VisIt's Vector plot renders a vector field at each time step as a collection of arrow glyphs. This allows us to see the direction of the vectors as well as their magnitude. We will create a vector plot to directly view the simulated "v" vector field.

1. Go to *Add->Vector->v*.
2. Open the Vector plot attributes window.
3. Go to the *Vectors* tab.
4. Set *Stride* to "17".
5. Go the the *Data* tab.
6. In the *Limits* section, enable the *Maximum* checkbox and set the value to "1".

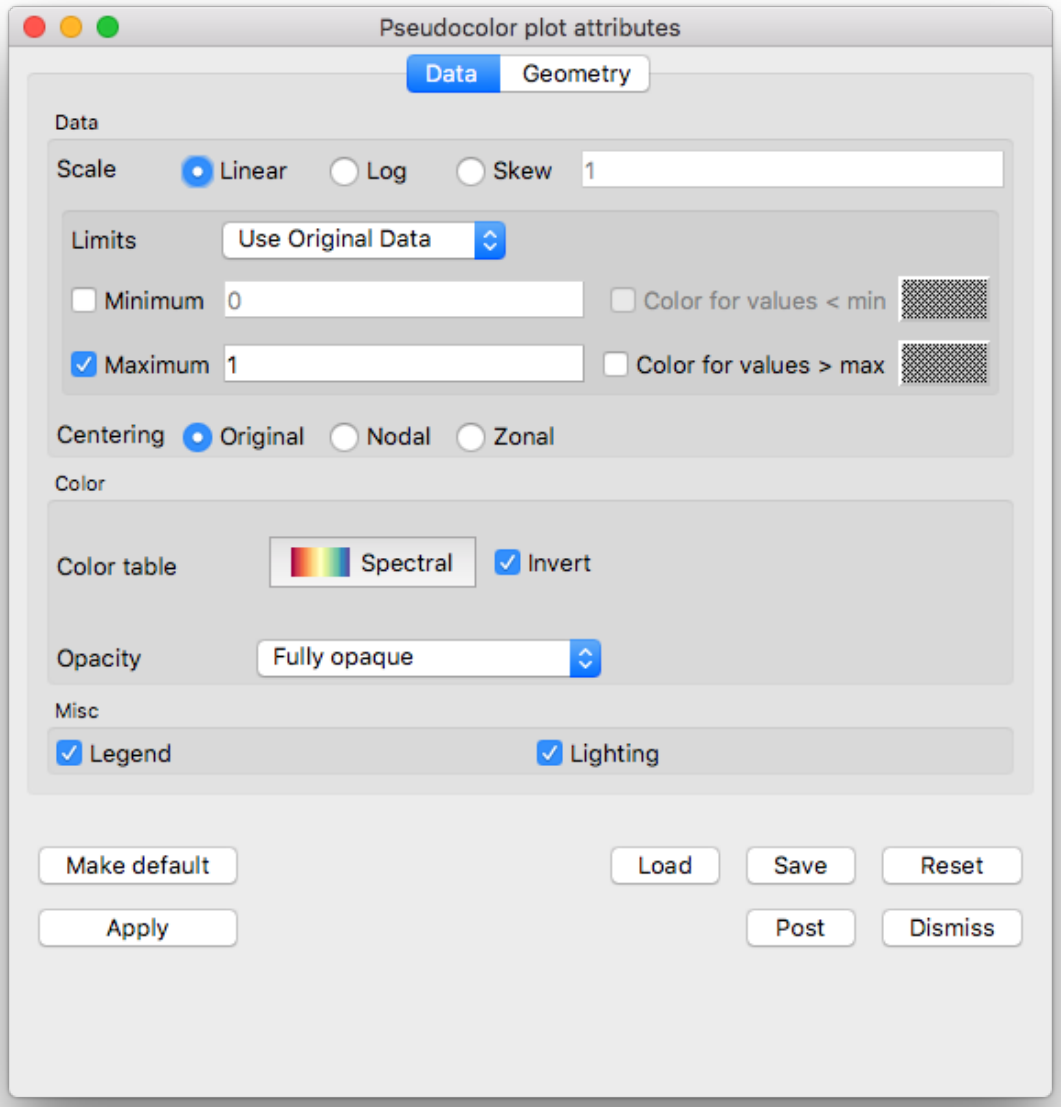


Fig. 6.62: The pseudocolor plot attributes for the velocity magnitude example.

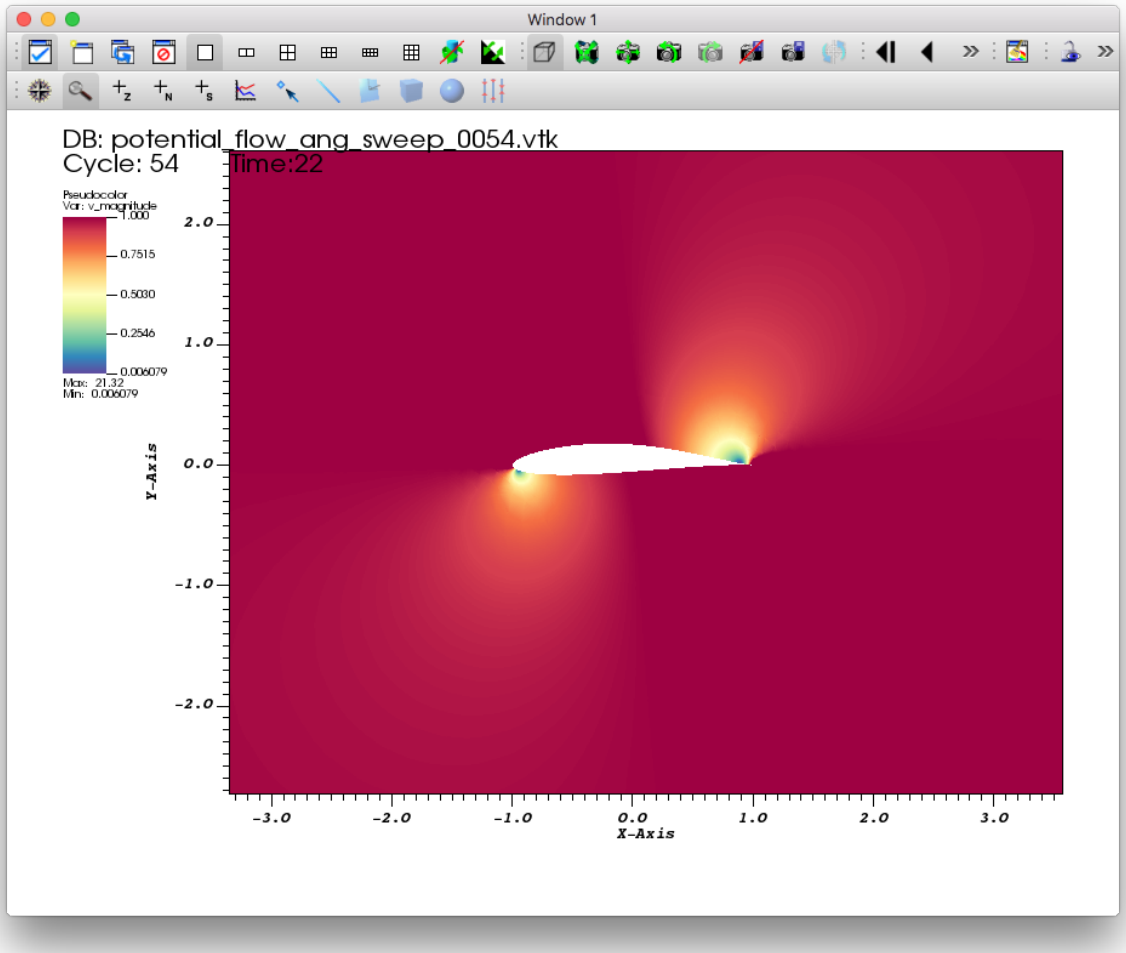


Fig. 6.63: The pseudocolor plot of the velocity magnitude.

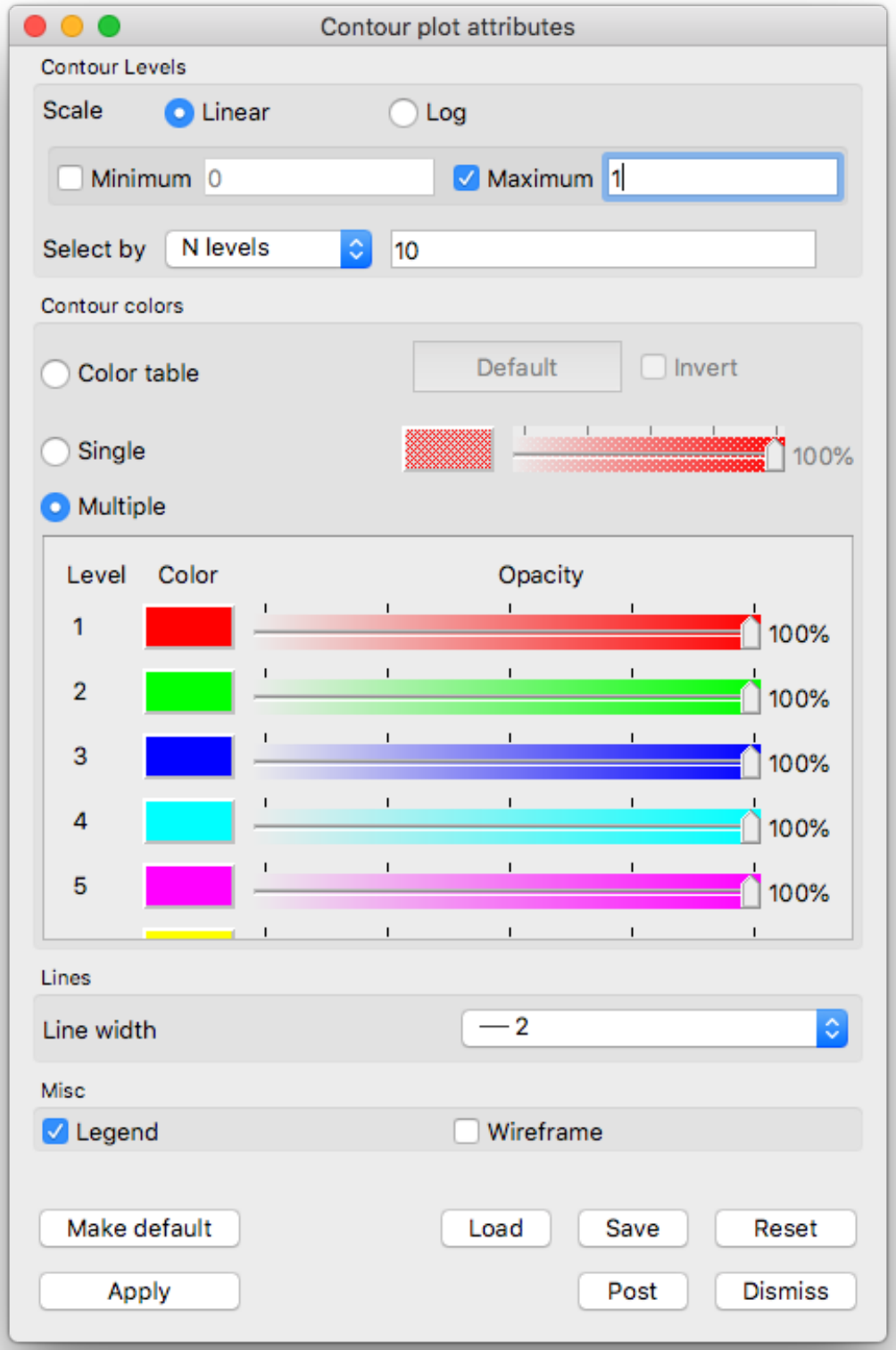


Fig. 6.64: Example contour plot settings for Potential Flow velocity magnitude.

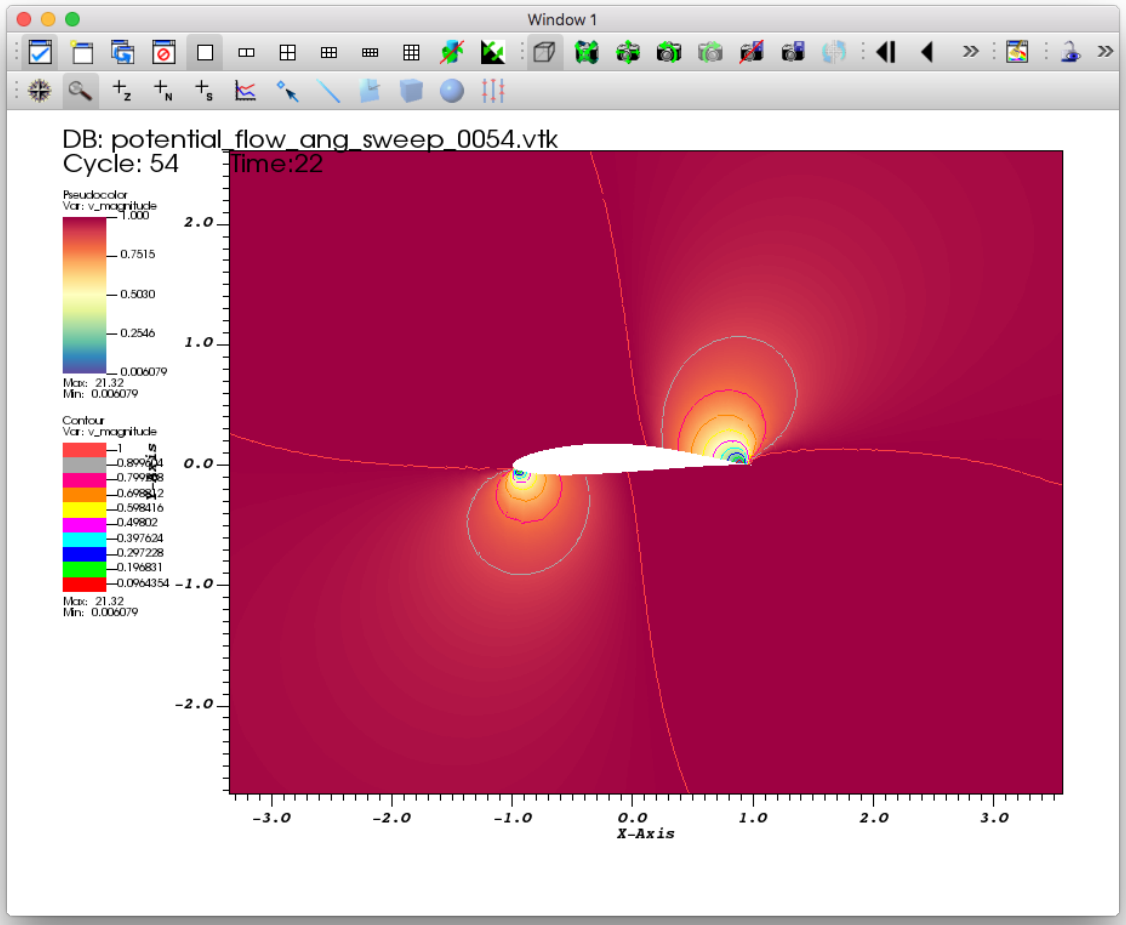


Fig. 6.65: A contour plot of the velocity magnitude.

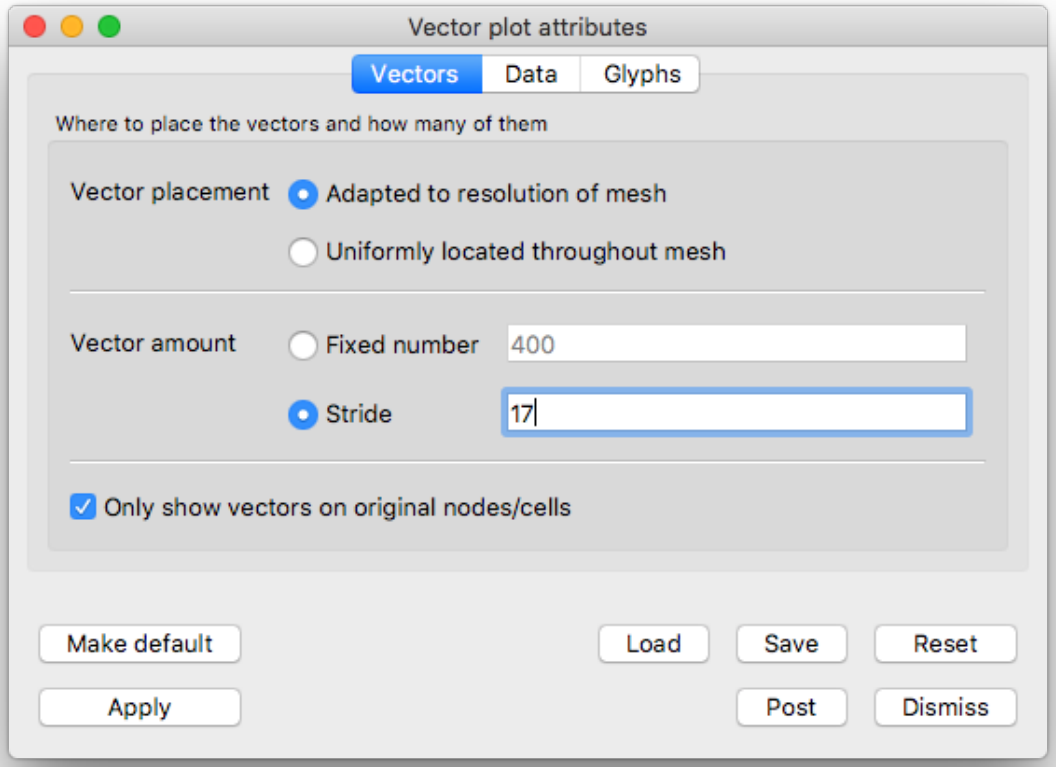


Fig. 6.66: *Vectors* tab settings for example vector plot of velocity

7. In the *Color* section, change the *Magnitude* to *viridis*

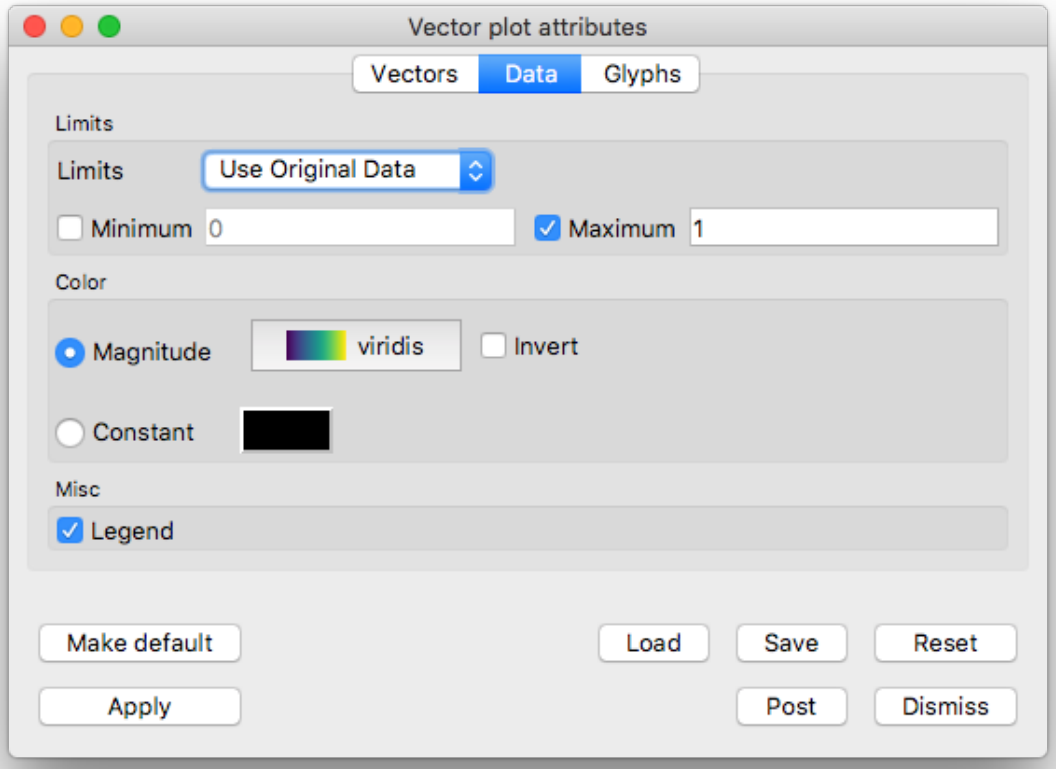


Fig. 6.67: *Data* tab settings for example vector plot of velocity

8. Go to the *Glyphs* tab.
9. In the *Scale* section, uncheck *Scale by magnitude* and *Auto scale*.
10. Click *Apply* and *Dismiss*.
11. Click *Draw*.
12. Zoom in near the airfoil.
13. Drag the *Time* animation controls above the plot list on the main *GUI* window.

You will see glyphs of velocity solutions for the different angles of attack.

Delete the vector plot when you are finished exploring, but keep the pseudocolor plot.

Examining features of the flow field with streamlines

To explore the flow field further we will seed and advect a set of streamlines on the left side of the mesh. Streamlines show the path massless tracer particles would take if advected by a static vector field. To construct Streamlines, the first step is selecting a set of spatial locations that can serve as the initial seed points.

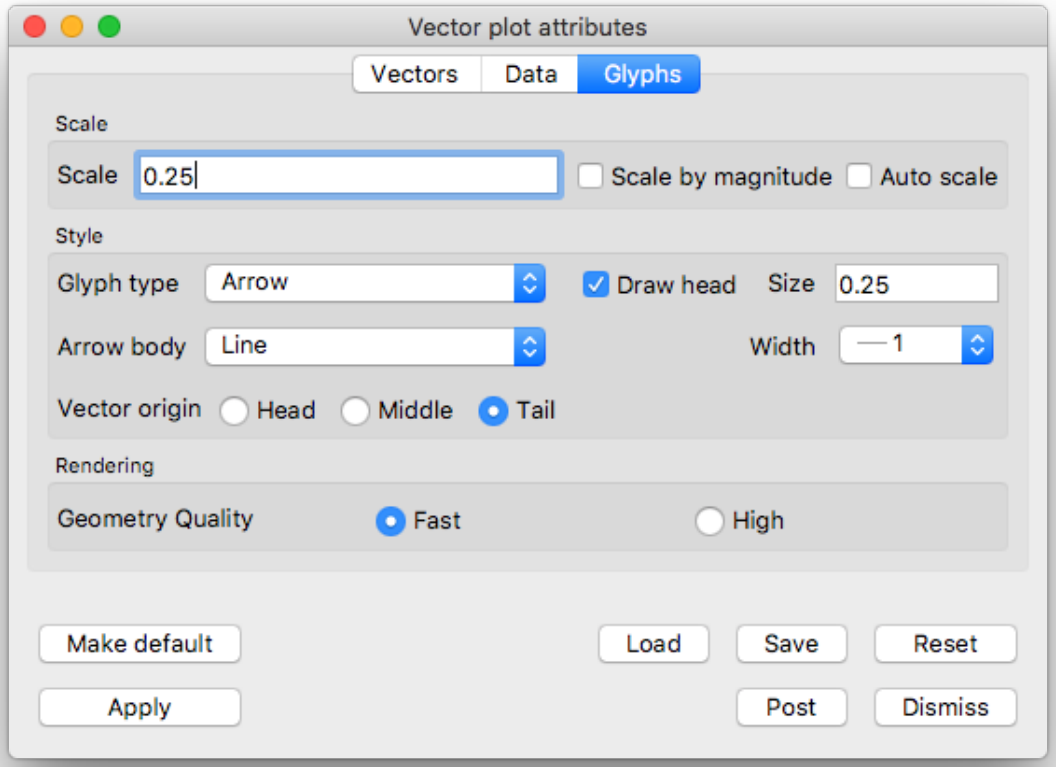


Fig. 6.68: *Glyphs* tab settings for example vector plot of velocity

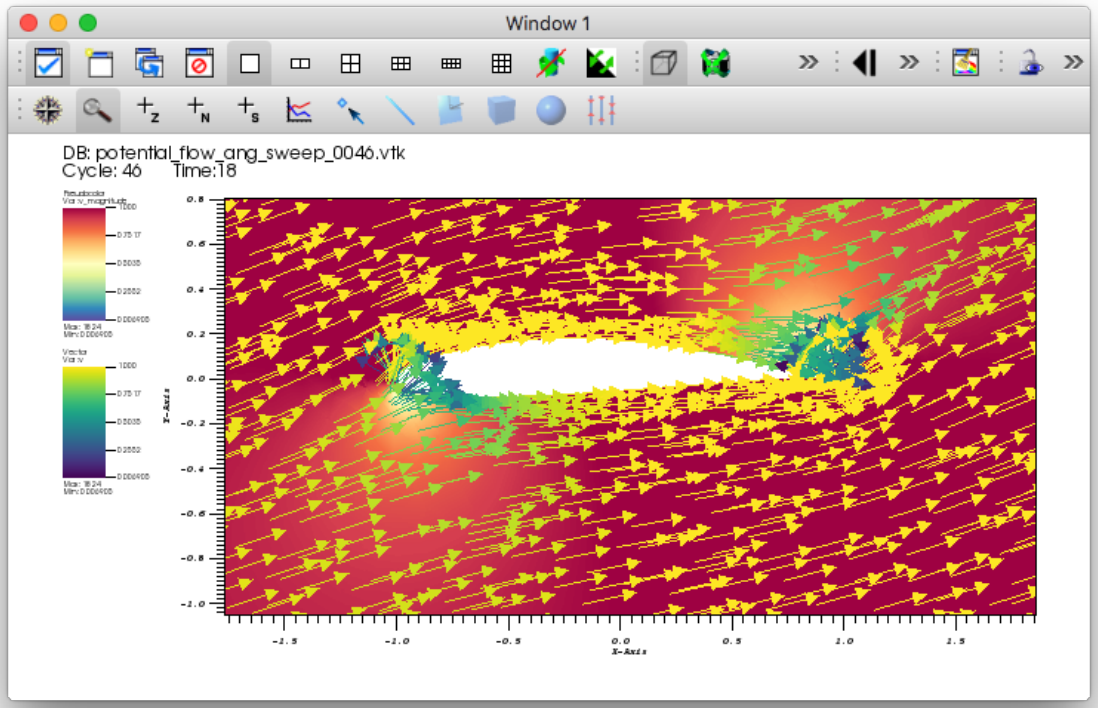


Fig. 6.69: The vector plot of velocity.

The flow moves left to right, we will use a vertical line of seed points on the left side of the mesh.

Plotting streamlines of velocity

1. Go to *Add->Pseudocolor->operators->IntergralCurve->v*.

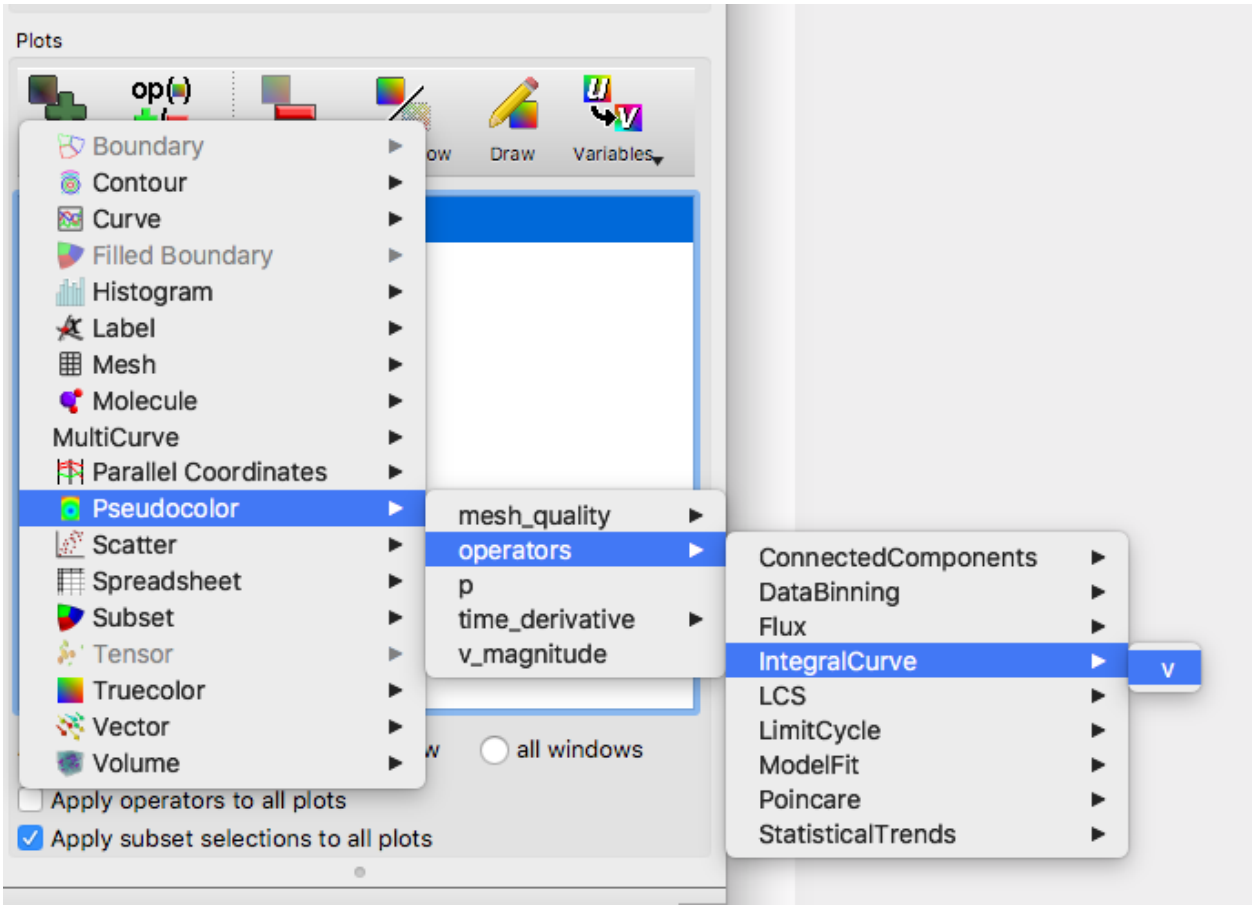


Fig. 6.70: Creating a streamline plot with the IntegralCurve operator.

2. Open the IntegralCurve operator attributes window.
3. Go to the *Source* section on the *Integration* tab.
4. Set the *Source type* to *Line*.
5. Set the *Start* to "-2 -2 0", excluding any commas in the input text box.
6. Set the *Stop* to "-2 2 0".
7. Set *Samples along line* to "10".
8. Click *Apply* and *Dismiss*.
9. Click *Draw* on the Main GUI
11. Open the Pseudocolor plot attributes window.
12. Go to the *Data* tab.
13. In the *Color* section set the *Color table* to *viridis*.

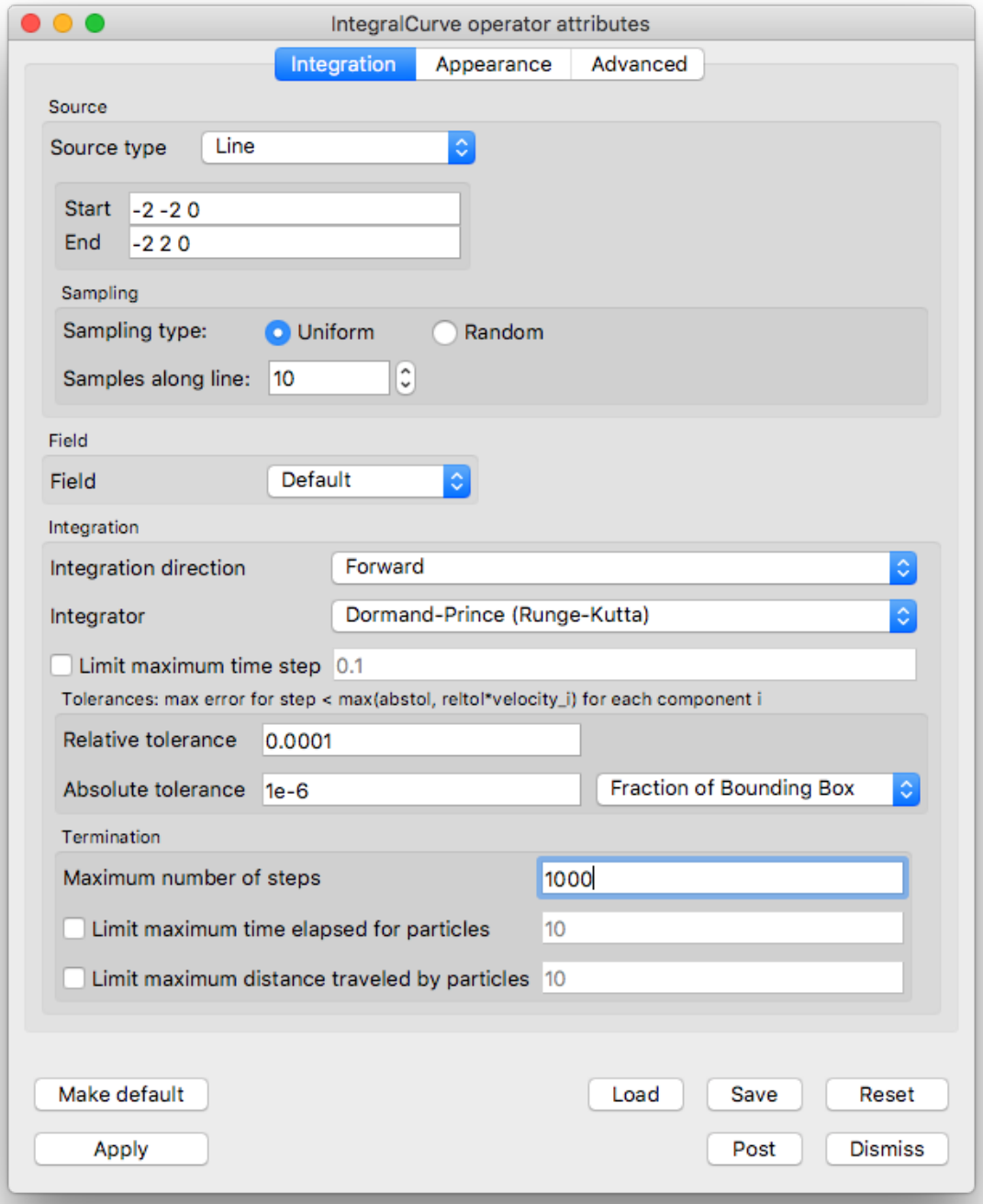


Fig. 6.71: The IntegralCurve operator attributes.

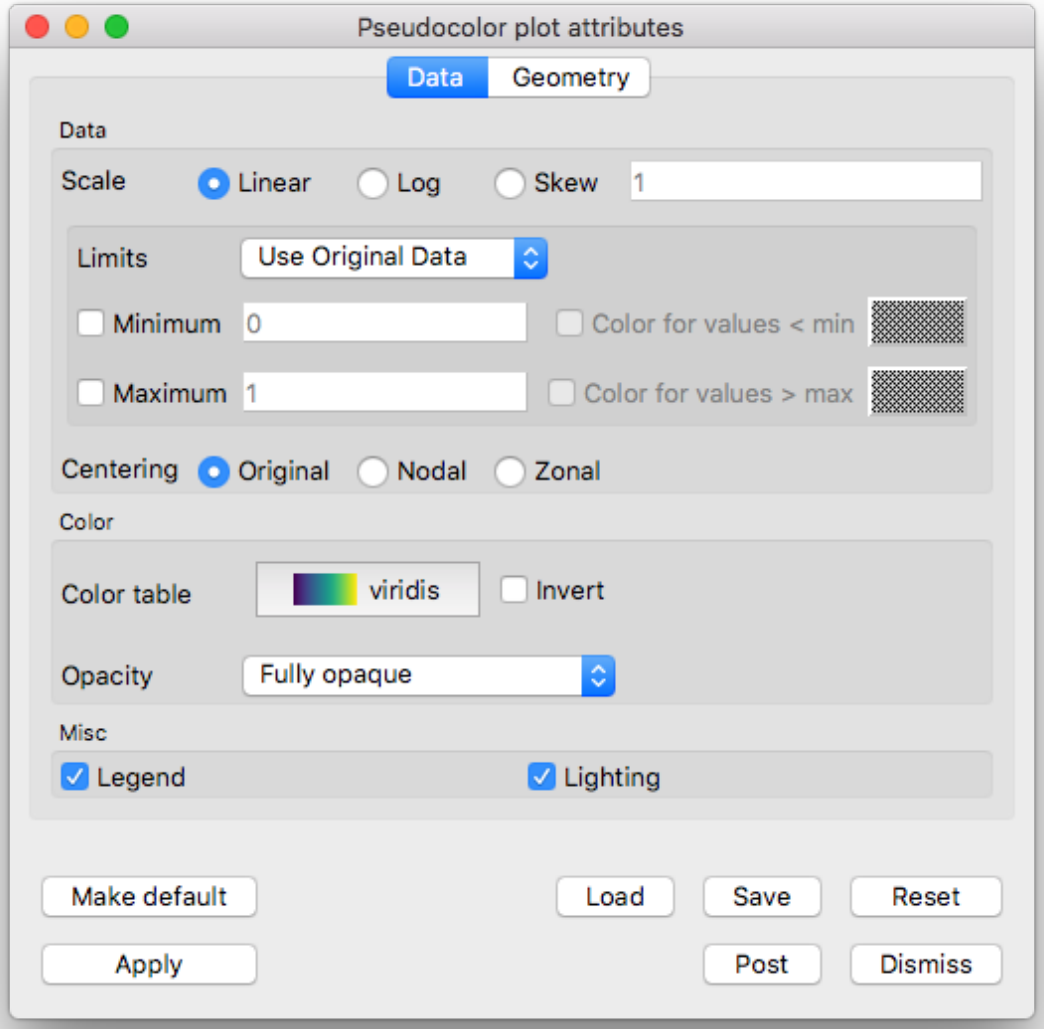


Fig. 6.72: The Pseudocolor attributes for the streamline data.

14. Go to the *Line* section on the *Geometry* tab.
15. Set *Line type* to *Ribbons*.
16. Set *Tail* to *Sphere*.
17. Set the tail *Radius* to "0.025".
18. Click *Apply* and *Dismiss*.
19. Click *Draw*.
20. Use the time slider controls to view a few different angles of attack solutions.

6.6 MRI

This tutorial provides a short introduction to visualizing MRI data using VisIt. We'll be relying on the Analyze data format, which is developed at the Mayo Clinic.

6.6.1 Open the dataset

This tutorial uses the MRI dataset.

1. Download the MRI dataset.
2. Click on the *Open* icon to bring up the File open window.
3. Navigate your file system to the folder containing "s01_anatomy_stripped.img".
4. Highlight the file "s01_anatomy_stripped.img" and then click *OK*.

6.6.2 Plotting areas of interest

First, we'll add a Pseudocolor plot and isoloate the visualization to an area that we're interseted in. In this case, it's a human brain located within the dataset.

Create a Pseudocolor plot

1. Go to *Add->Pseudocolor->Variable*.
2. Click *Draw*.

The Pseudocolor plot should now be rendered in VisIt's Viewer window. Modify the view by rotating and zooming in the viewer window. You'll notice that the visualization doesn't look very interesting at this point. This is because what we're really interested in seeing is hidden within the dataset.

Add an Isovolume operator

Adding an Isolvolume operator will help us remove sections of the dataset that we're uninterested in.

1. Go to *Operators->Selection->Isovolume* to add the Isovolume operator.
2. Click on the triangle to the left of your Pseudocolor plot, and double click Isovolume to open up the Isovolume attributes.
3. Once you've opened the Isovolume attributes, set the Lower bound to 30, and click *Apply*.

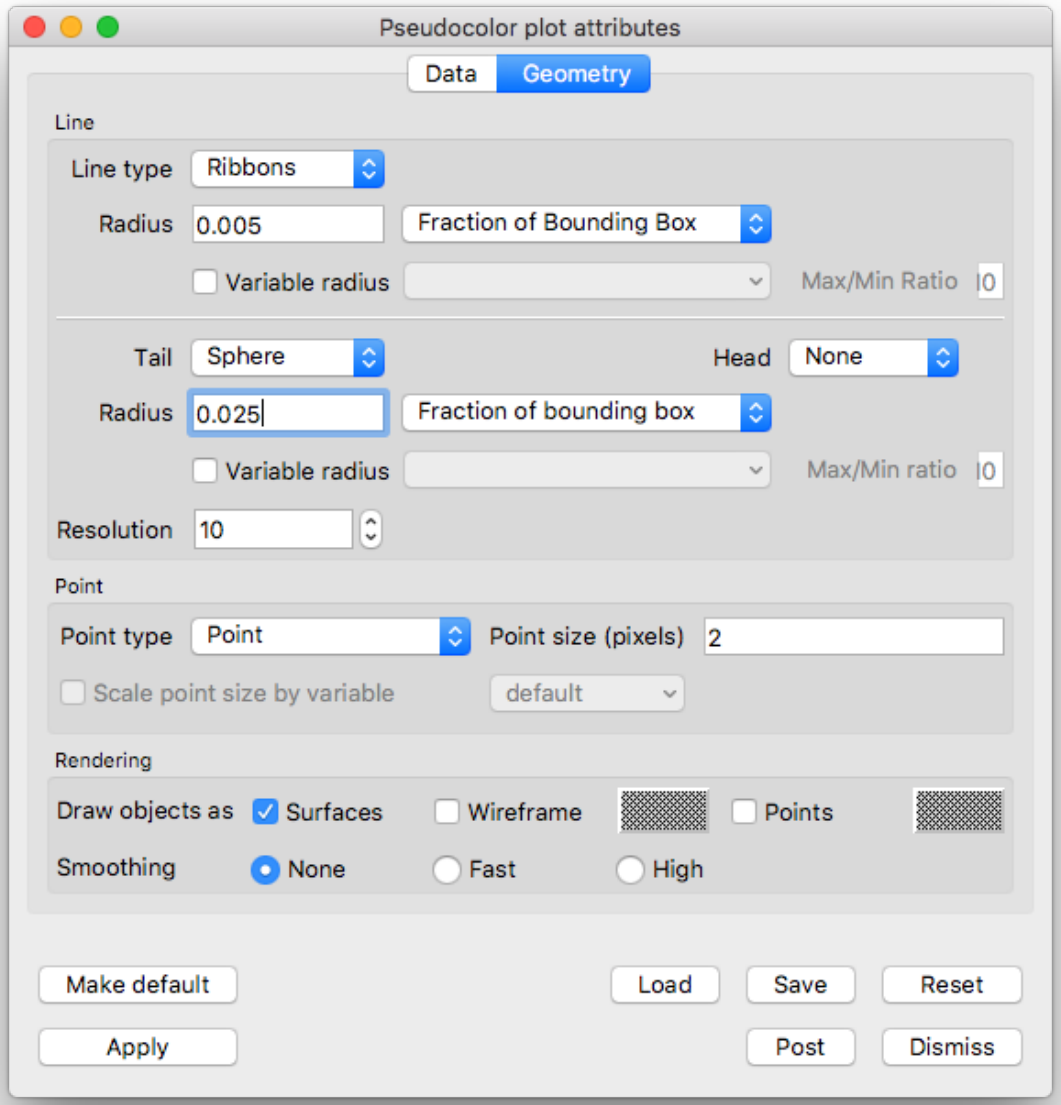


Fig. 6.73: The Pseudocolor attributes for the streamline geometry.

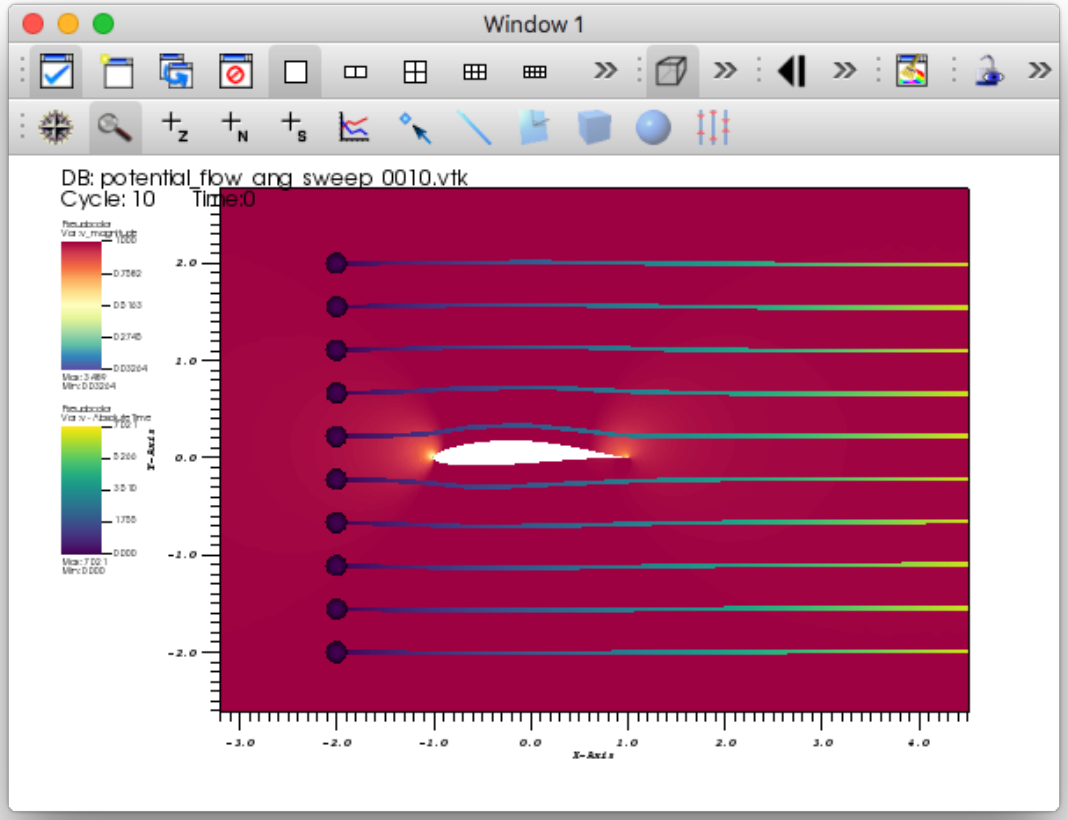


Fig. 6.74: The streamlines of velocity at 0 degree angle of attack.

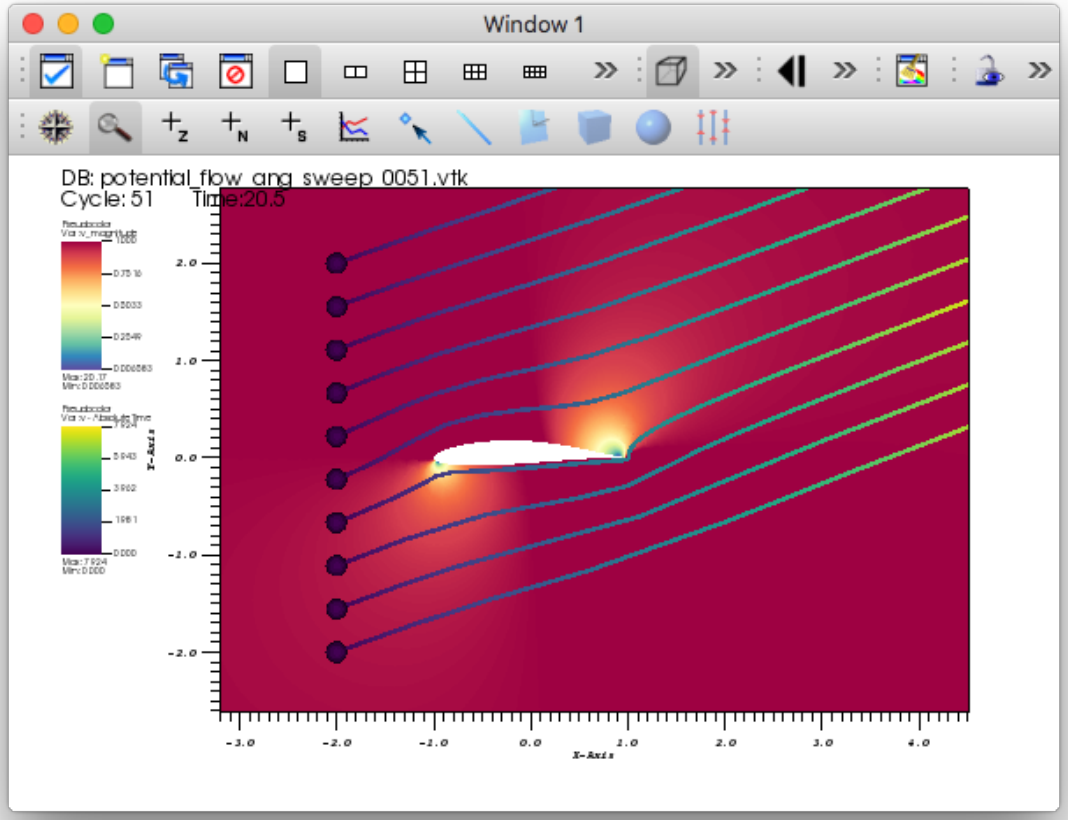


Fig. 6.75: The streamlines of velocity at 20.5 degree angle of attack.

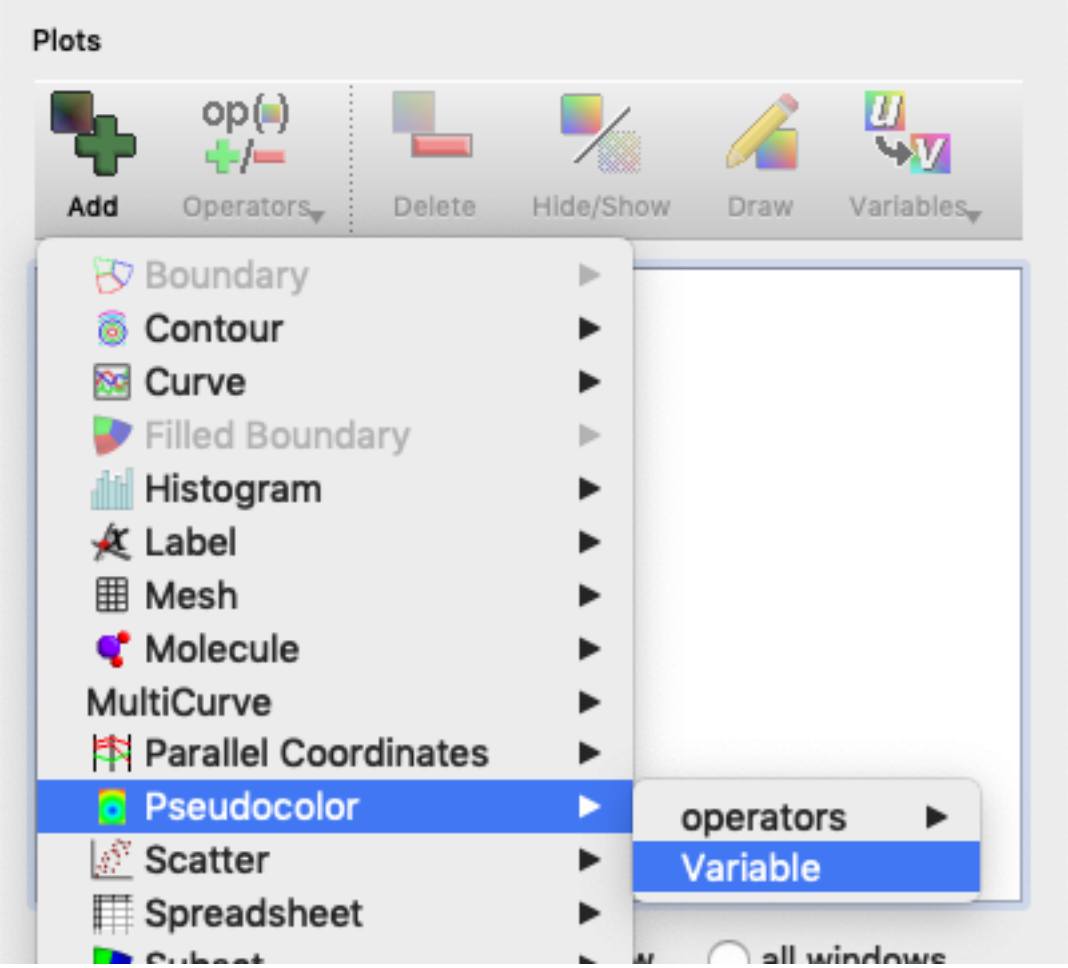


Fig. 6.76: Adding a Pseudocolor plot.

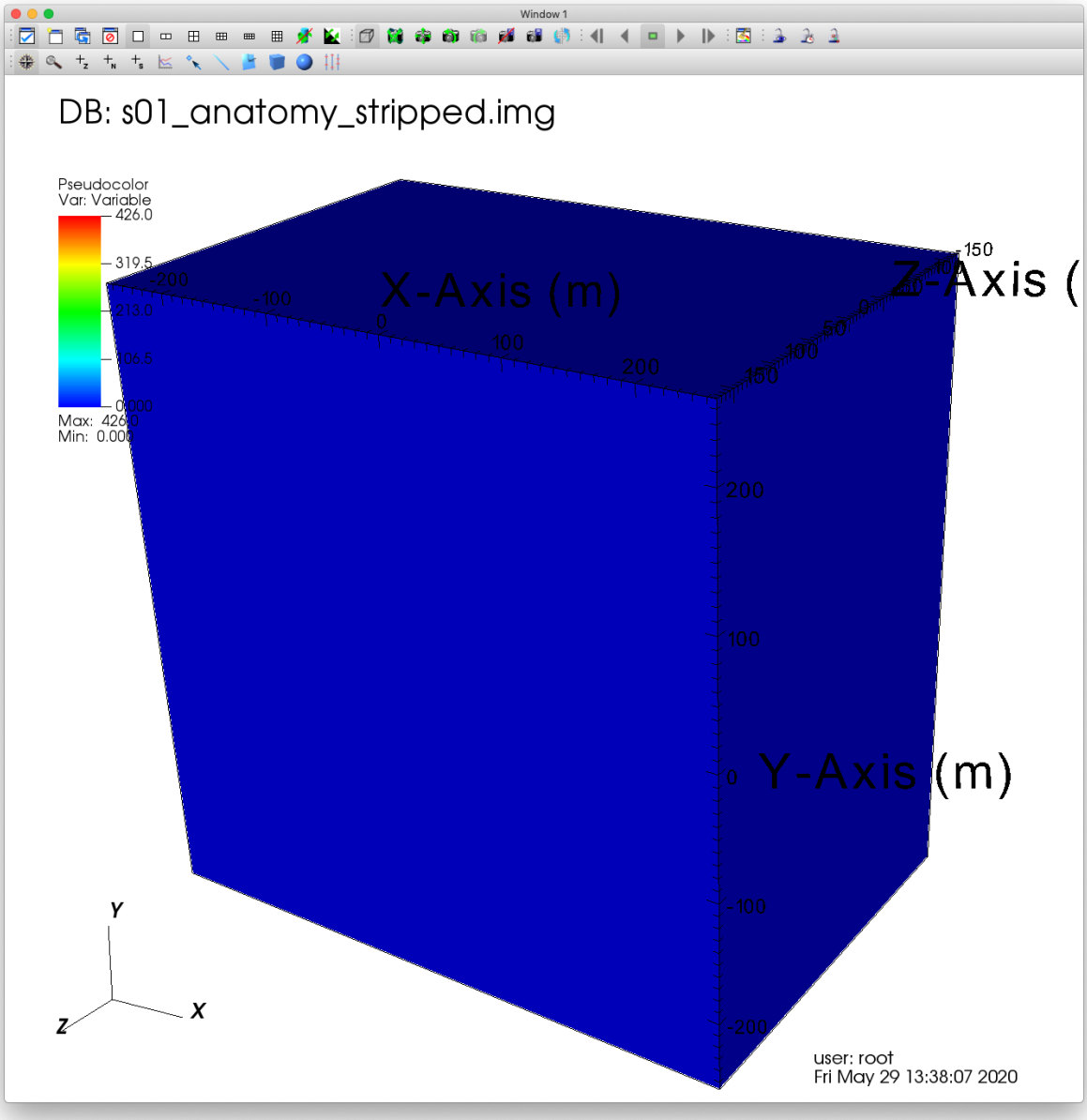


Fig. 6.77: Visualizing our dataset.

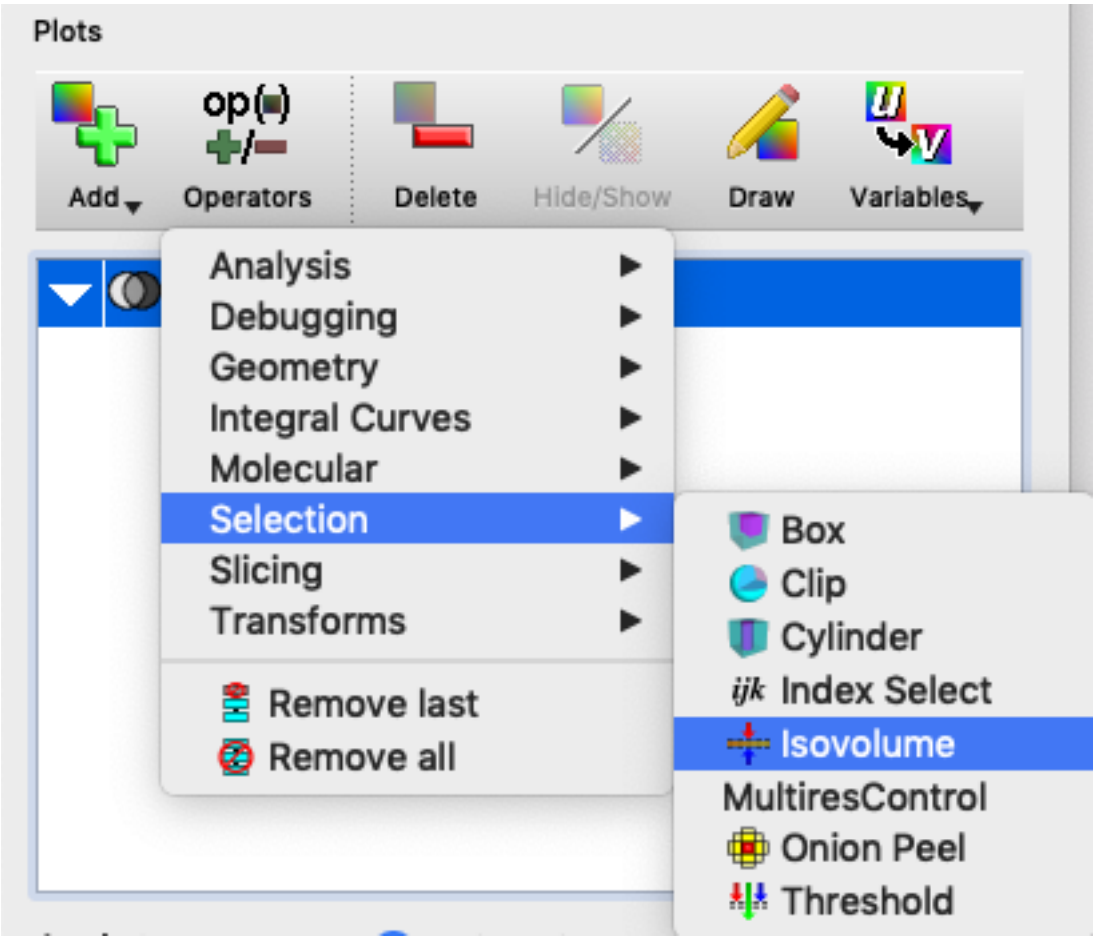


Fig. 6.78: Adding a Isovolume operator.

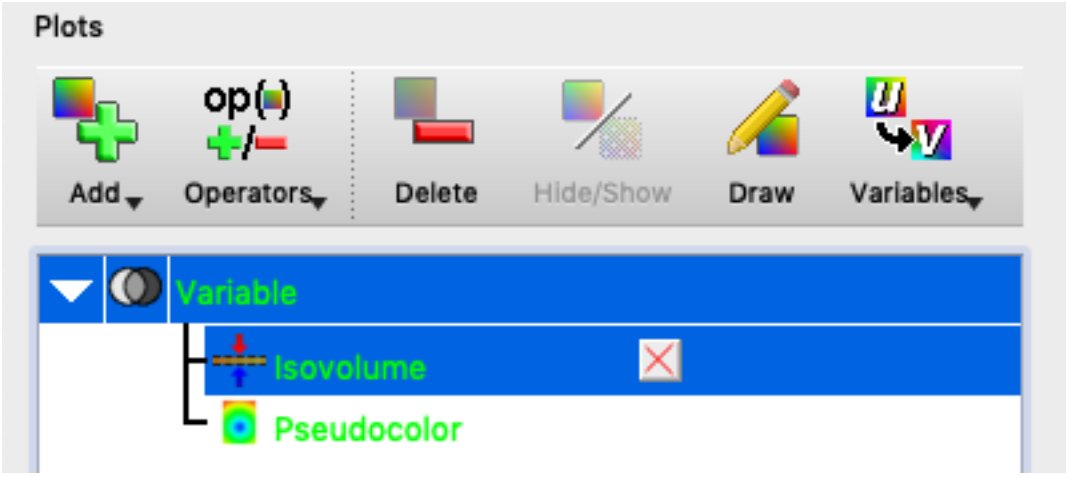


Fig. 6.79: Opening the Isovolume attributes.

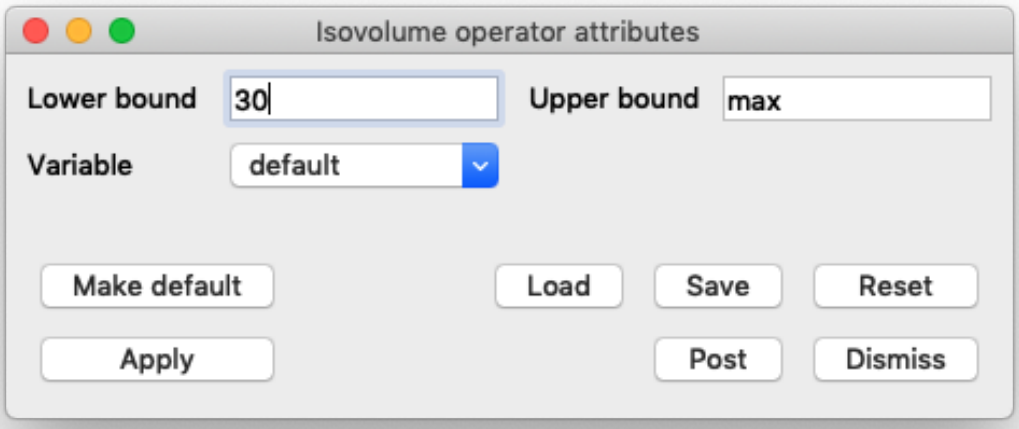


Fig. 6.80: Changing the Isovolume attributes.

4. Click *Draw*. You will now see a visualization of a human brain.
5. You can experiment with changing the lower and upper bounds of the Isolvoume attributes to visualize different sections of the dataset.

Change the color table

The default color table doesn't add much to the visualization, so let's change the color table to better suite our needs. In this case, we'll choose Pastel1.

1. Double click Pseudocolor to open up the Pseudocolor attributes.
2. Once there, you can choose your color table.
3. Click *Apply* to finalize the change.

6.6.3 Exploring our MRI dataset

Now that we've located and visualized the inner section of our dataset, we can further explore characteristics local to this region.

Performing a Slice

First, we're going to slice out a single cross-section for closer examination.

1. Go to *Operators->Slicing->Slice* to add the Slice operator.
2. Double click on the Slice to bring up the Slice attributes window.
3. There are a lot of options to configure here. For now, we'll leave all of the default settings except for Project to 2D. Uncheck this box.

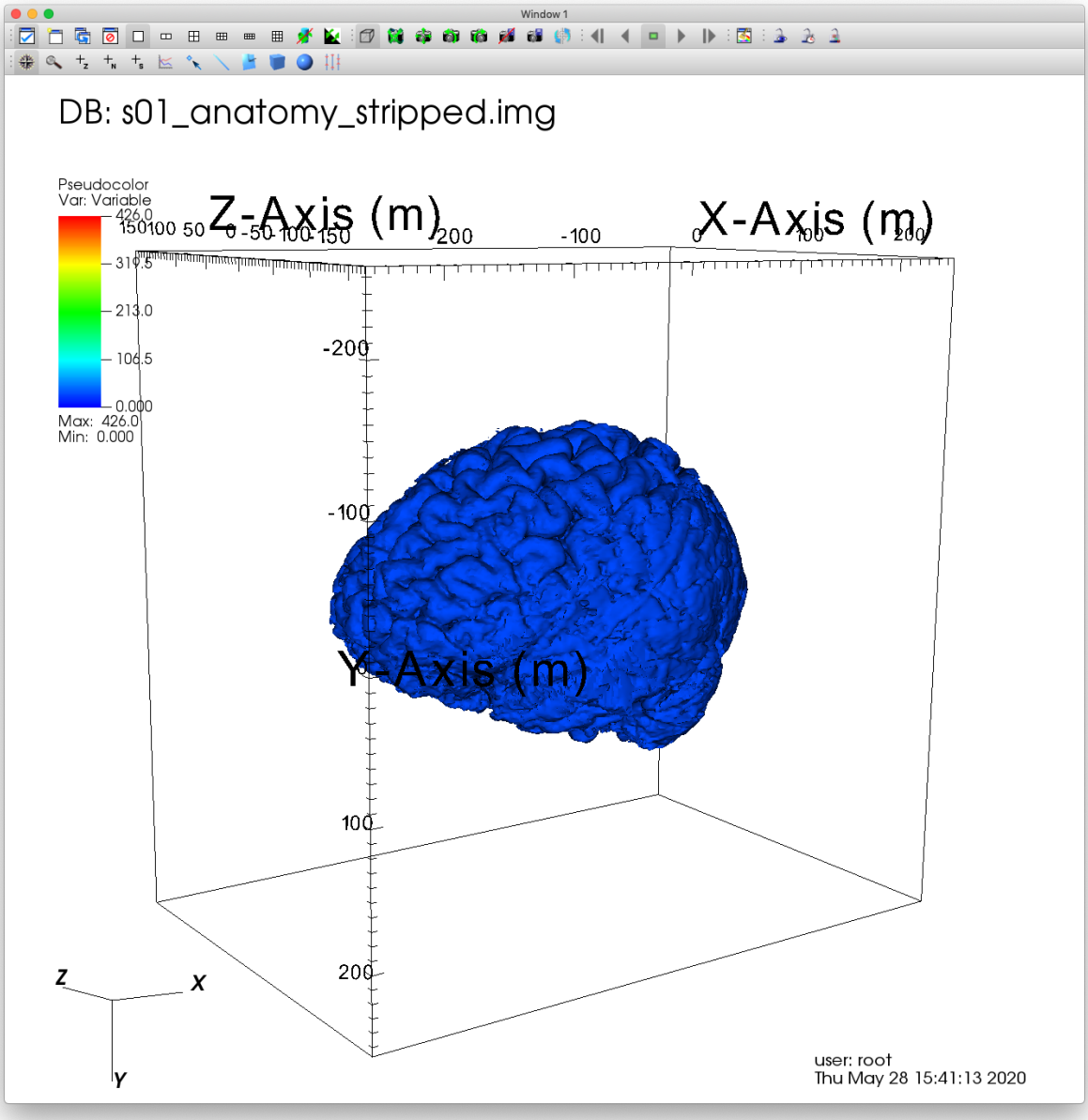


Fig. 6.81: Visualizing the underlying data of our dataset.

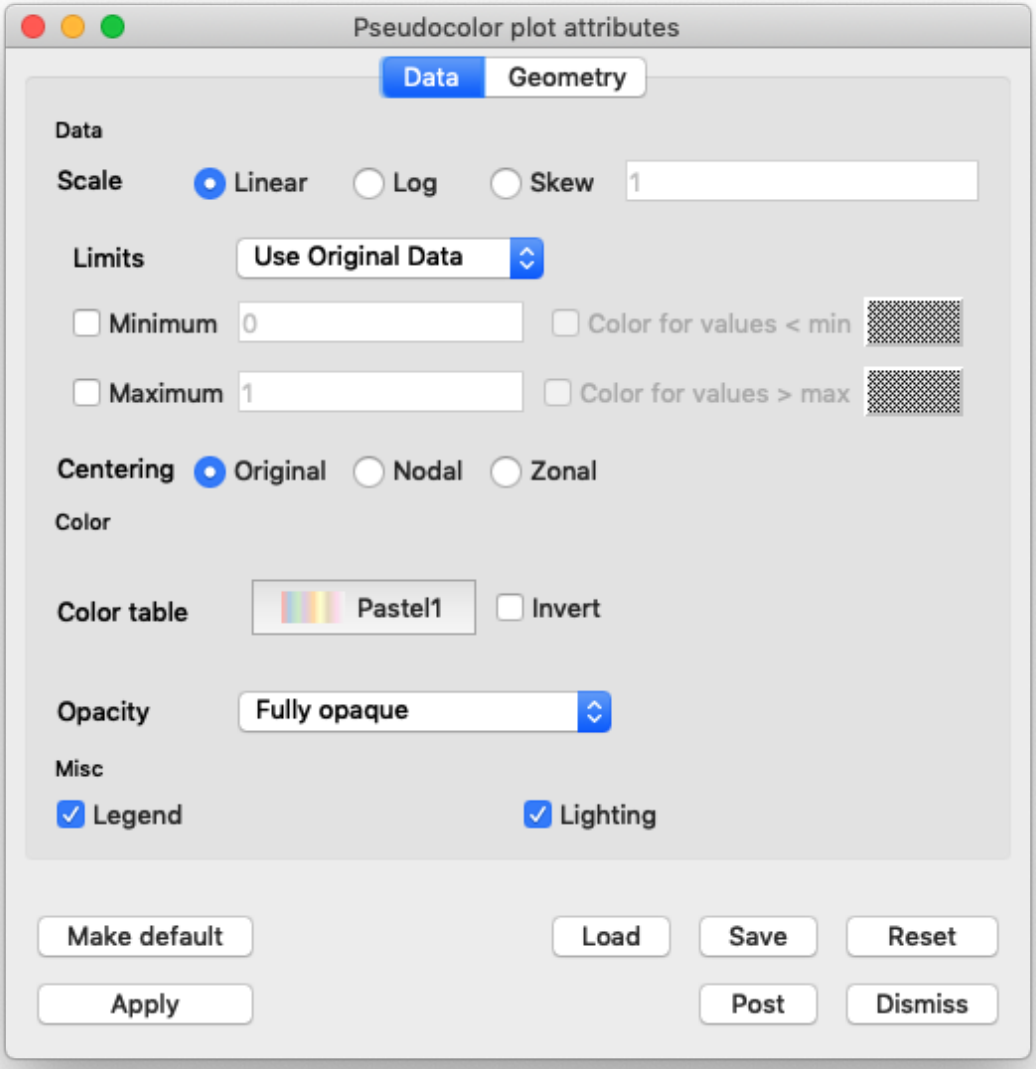


Fig. 6.82: Changing the color table.

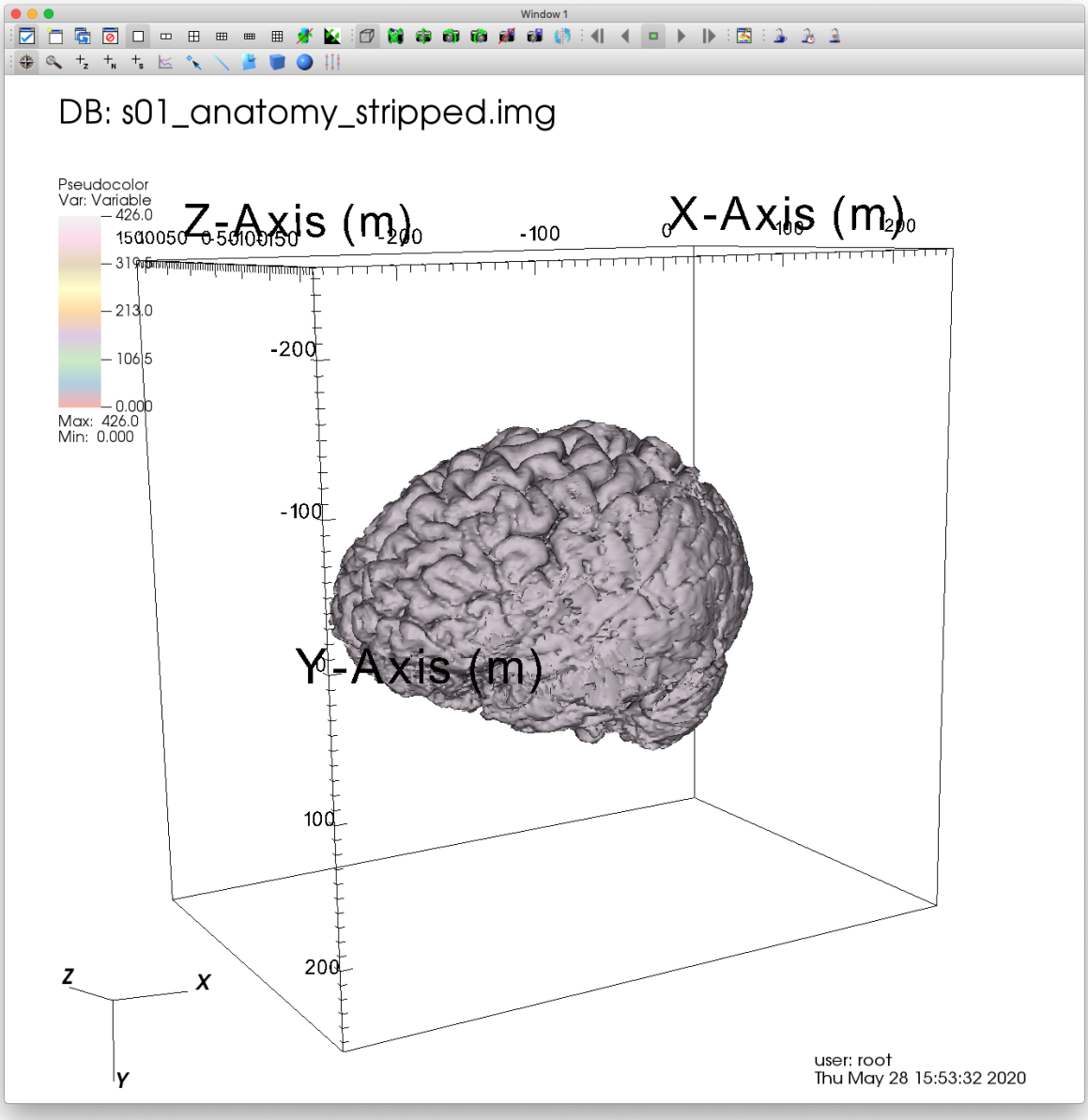


Fig. 6.83: Visualizing our updated color table.

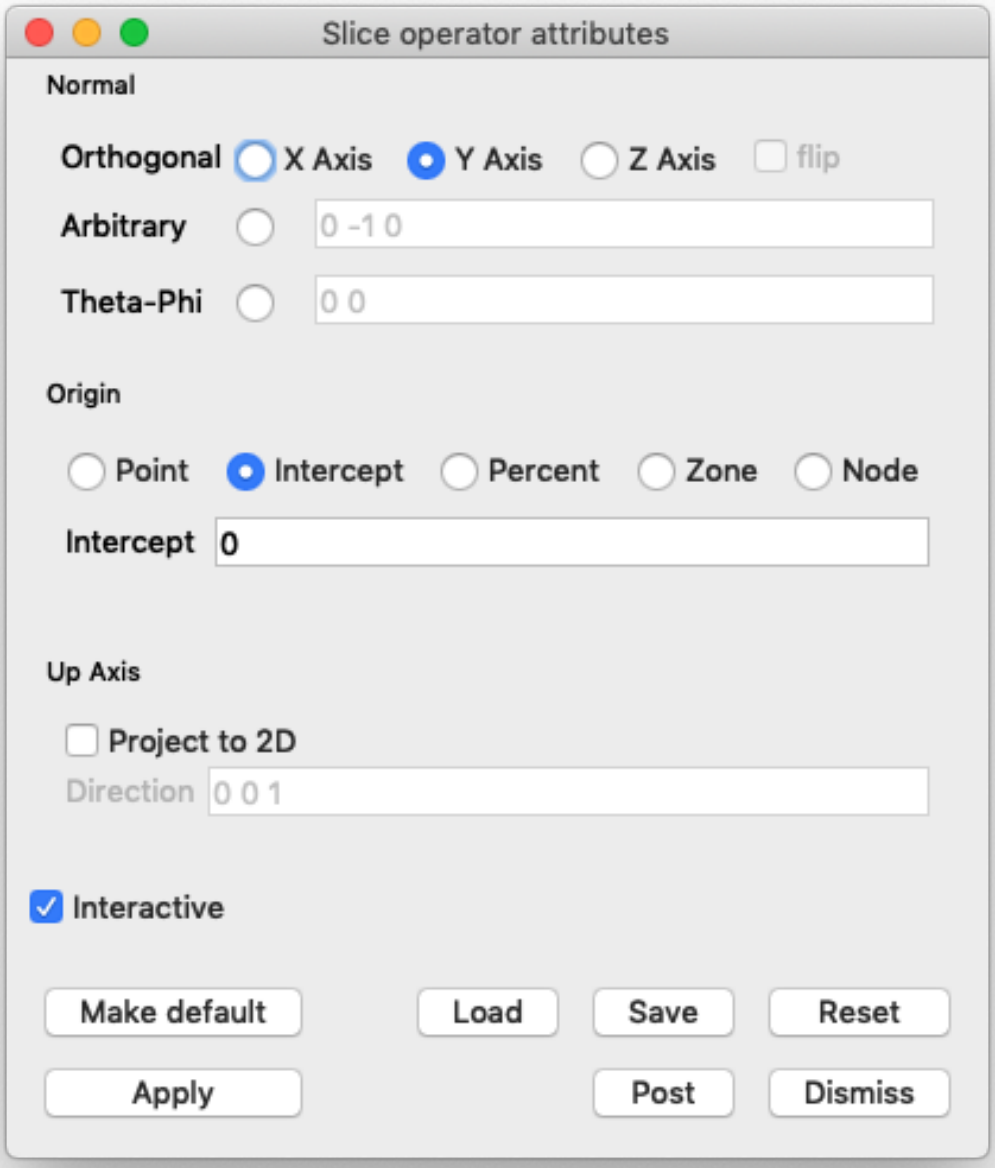


Fig. 6.84: Changing the Slice attributes.

4. Click *Apply*.
5. Click *Draw*.

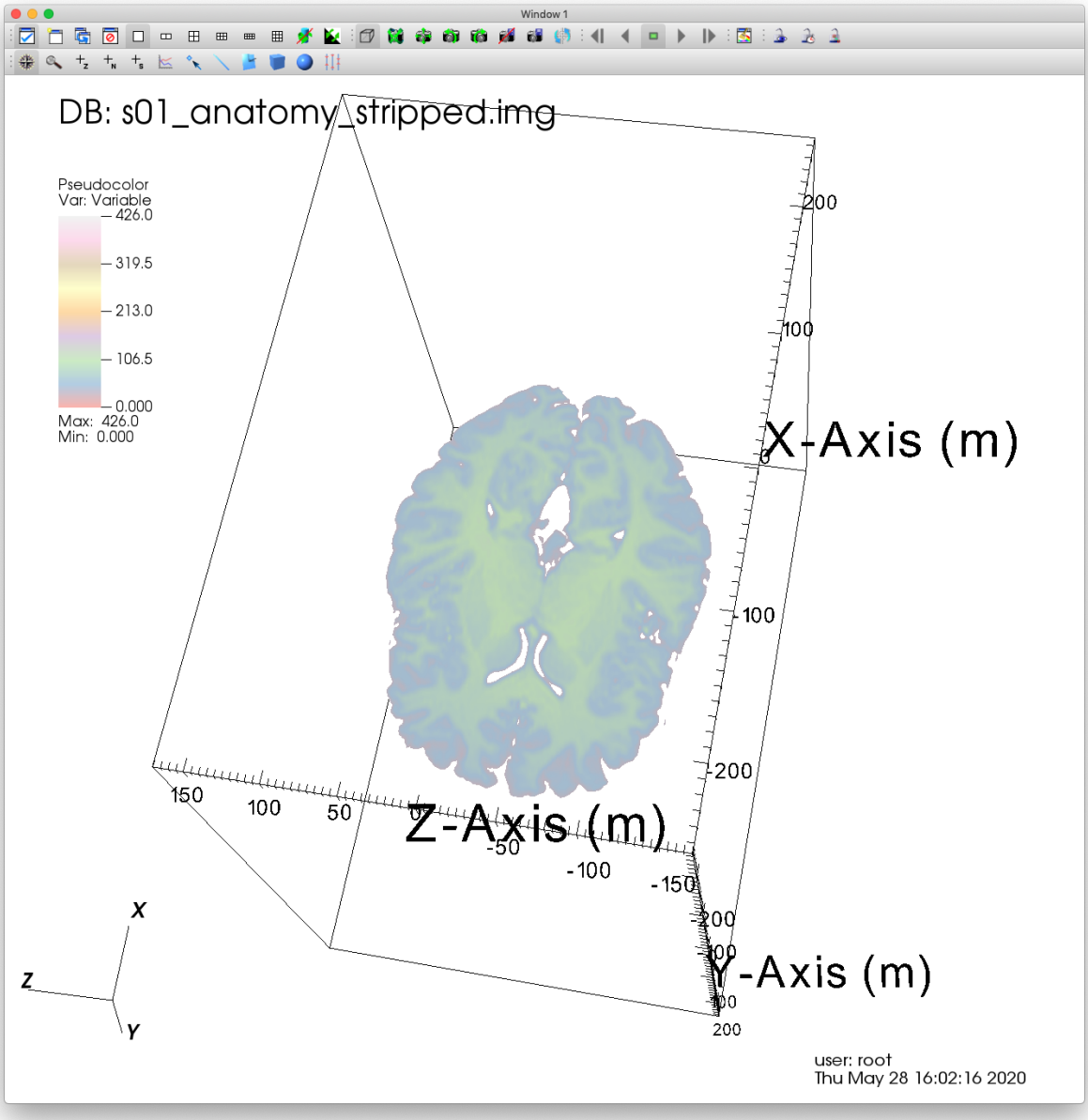


Fig. 6.85: Visualizing a Slice of our MRI dataset.

Performing a ThreeSlice

Another usefull operator that is similar to Slice is ThreeSlice. This operator creates three axis aligned slices of a 3D dataset, one in each dimension.

1. Remove the Slice operator by clicking the X button to the right of the added Slice.
2. Go to *Operators->Slicing->ThreeSlice* to add the ThreeSlice operator.

3. Double click on the ThreeSlice to bring up the ThreeSlice attributes window. You can move the location of each slice by changing the X, Y, and Z values.

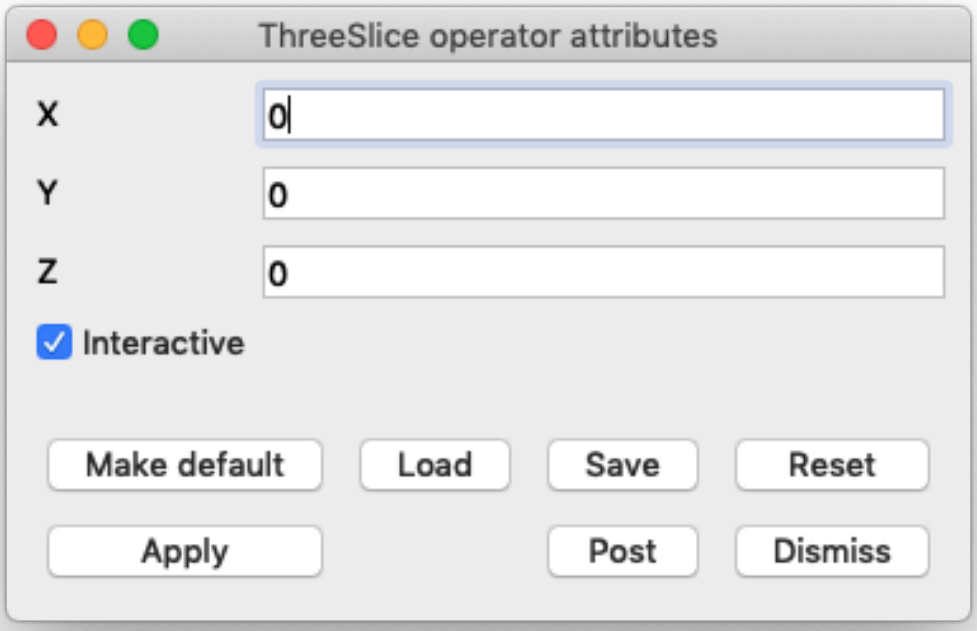


Fig. 6.86: The ThreeSlice attributes.

4. Click *Apply*.
5. Click *Draw*.

Performing a ThreeSlice using the point tool

Along with directly entering the X, Y, Z coordinates for your ThreeSlice in the attributes window, Visit also provides the option of using an interactive Point tool for determing these coordinates.

1. In the top left-hand corner of the visualization window, you'll find a button that activates the Point tool. *Click* this button.
2. Once activated, you will see a point surrounded by a red box within the visualization window.
3. Before changing the orientation of our Point tool, *Click* on the ThreeSlice attributes window so that VisIt understands that we want to associate this Point tool with these attributes.
4. *Click* and drag the red box to change the location of the point defining the X, Y, Z coordinates of the ThreeSlice. VisIt will automatically update the plot.
5. *Click* the Point tool button again to deactivate the tool.

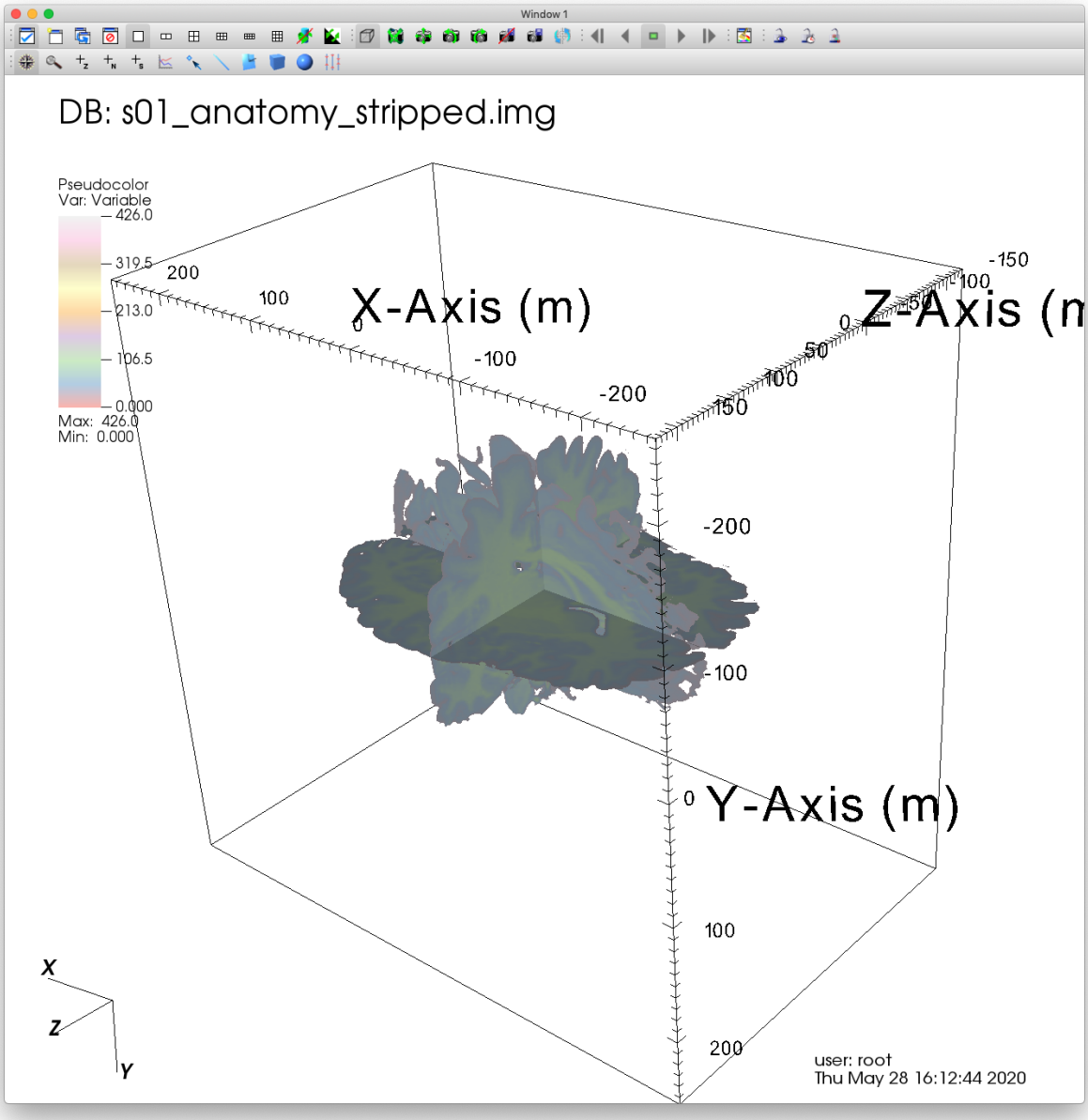


Fig. 6.87: Visualizing a ThreeSlice of our MRI dataset.

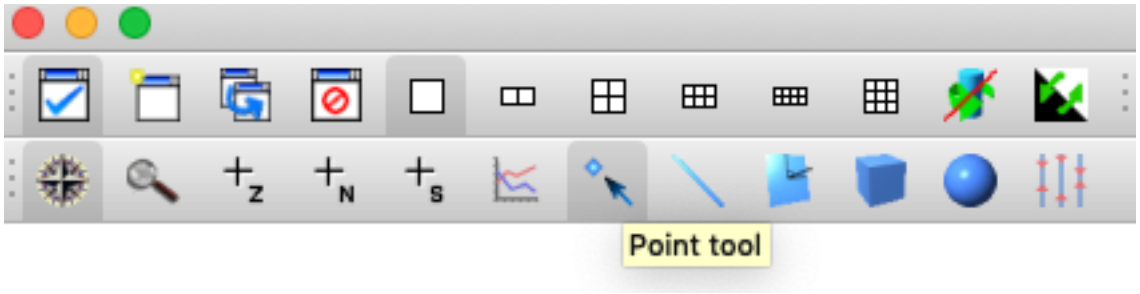


Fig. 6.88: Activating the Point tool.

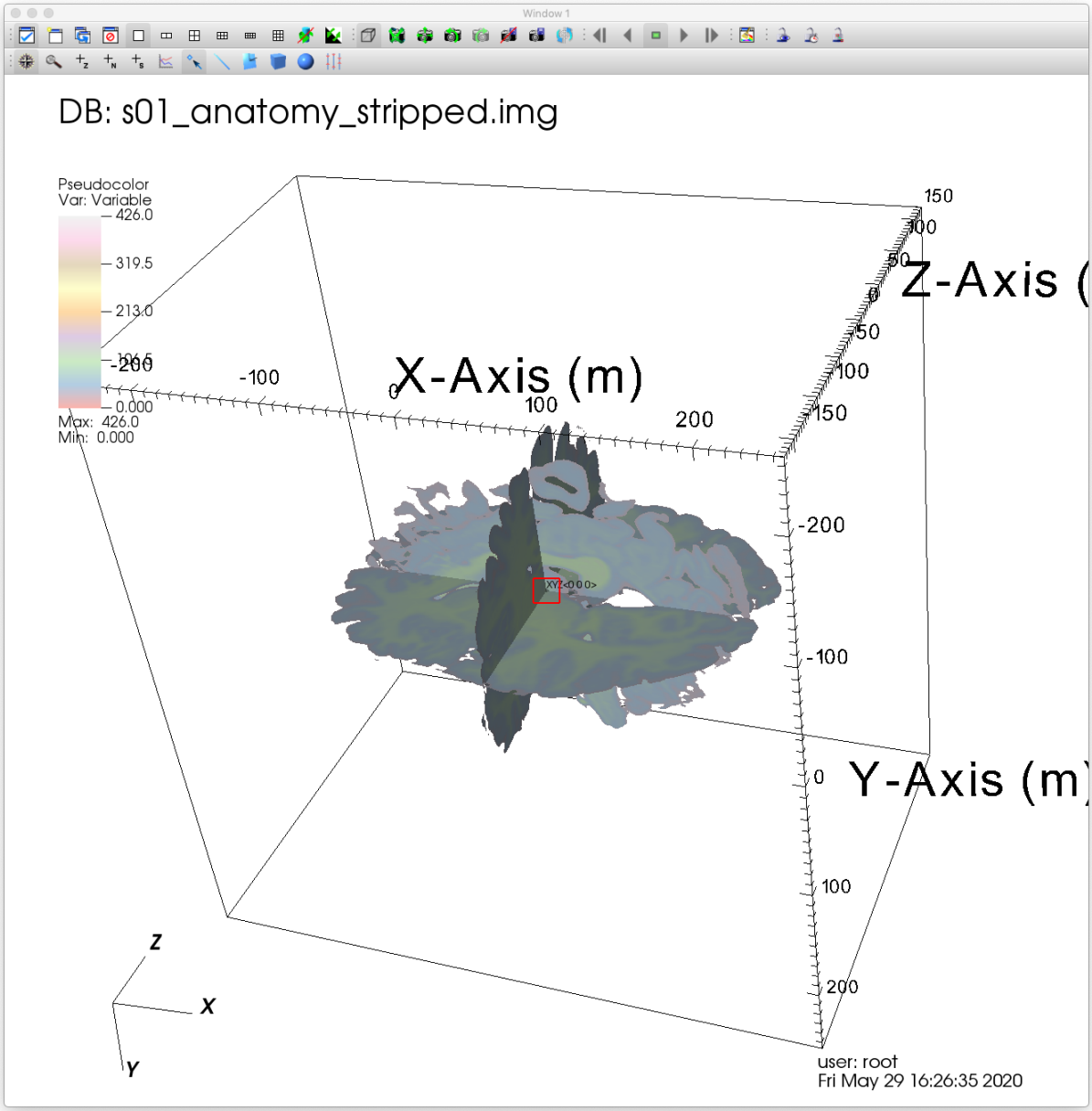


Fig. 6.89: The activated Point tool.

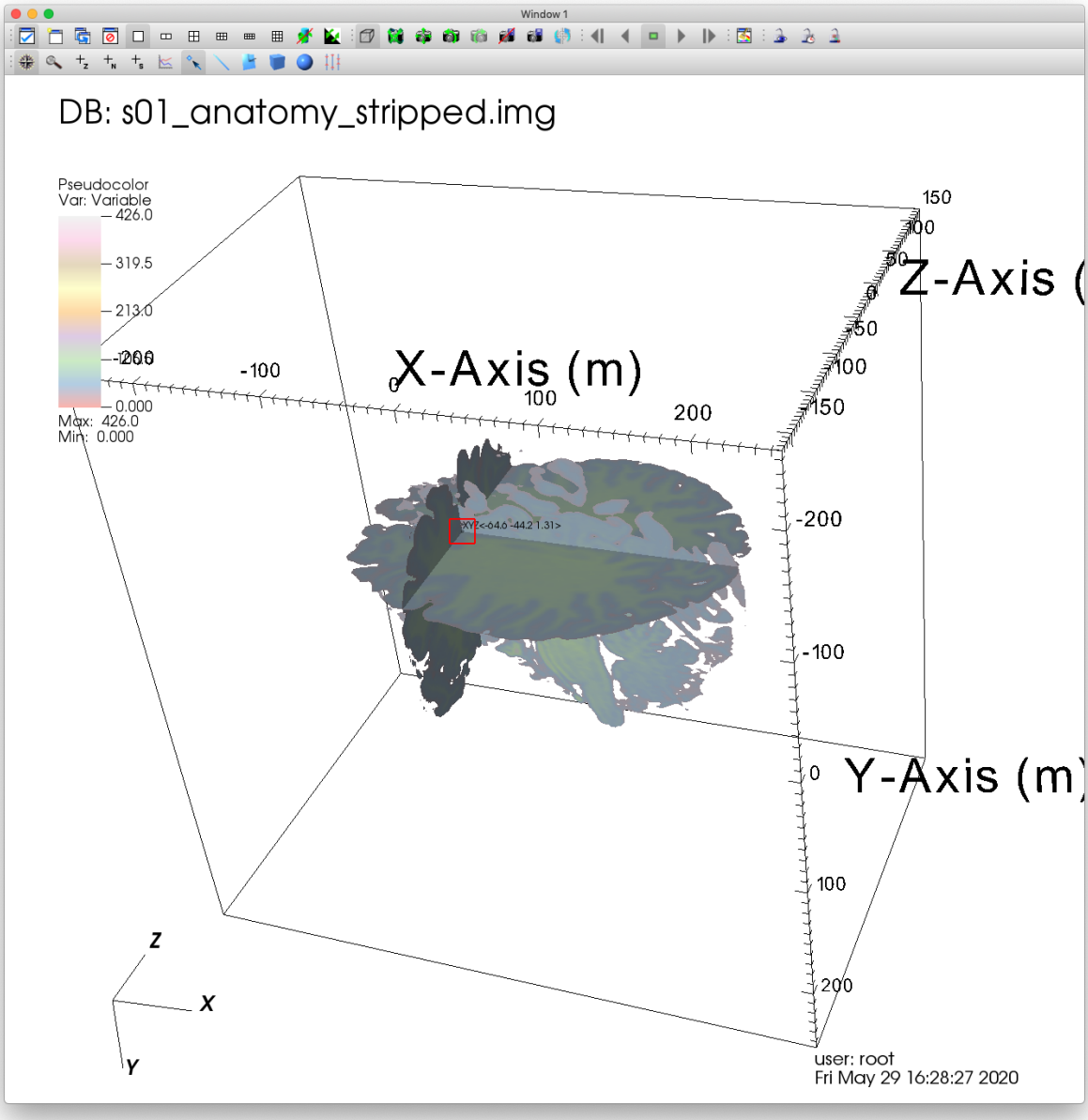


Fig. 6.90: Performing a ThreeSlice with the Point tool.

Performing a Clip

One more way to view the interior of your dataset is to perform a Clip, which clips away entire sections of your data. There are many ways to perform your Clip, each of which has it's own benefits.

Performing a Clip using a single plane

1. Remove the ThreeSlice operator by clicking the X button to the right of the added ThreeSlice.
2. Go to *Operators->Selection->Clip* to add a Clip operator.
3. Double click on the Clip to bring up the Clip attributes window. Again, there are many settings to configure here. The default settings use a single plane for performing the Clip.
4. Click *Apply*.
5. Click *Draw*.

Performing a Clip using two planes

1. Return to the Clip attributes window, check the Plane 2 box, and change the normal of Plane 2 to "0 -1 0".
2. Click *Apply*.

Performing a Clip using three planes

1. Return to the Clip attributes window, and check the Plane 3 box. Next, change the origin of Plane 3 to "0 0 -50".
2. Click *Apply*.

Performing a Clip using a sphere

Let's update the settings of our Clip so that we remove a spherical section of the data.

1. Double click on the Clip to bring up the Clip attributes window again. Change the Slice type to Sphere. The attribute options should change significantly. Set the Center to "0 100 0", and set the radius to 150.
2. Click *Apply*.
3. Click *Draw*.

Performing a Clip using the Plane tool

VisIt also provides an interactive Plane tool that can be used to determine your intersecting plane by orienting a 3D axis within the dataset.

1. First, *Click* the Reset button in the Clip attributes window to reset the Clip attributes to their default state.
2. In the top left-hand corner of the visualization window, you'll find a button that activates the Plane tool. *Click* this button.
3. Once activated, you will see a 3D axis defining a plane within the visualization window.
4. Before changing the orientation of our Plane tool, *Click* on the Clip attributes window so that VisIt understands that we want to associate this Plane tool with these attributes.

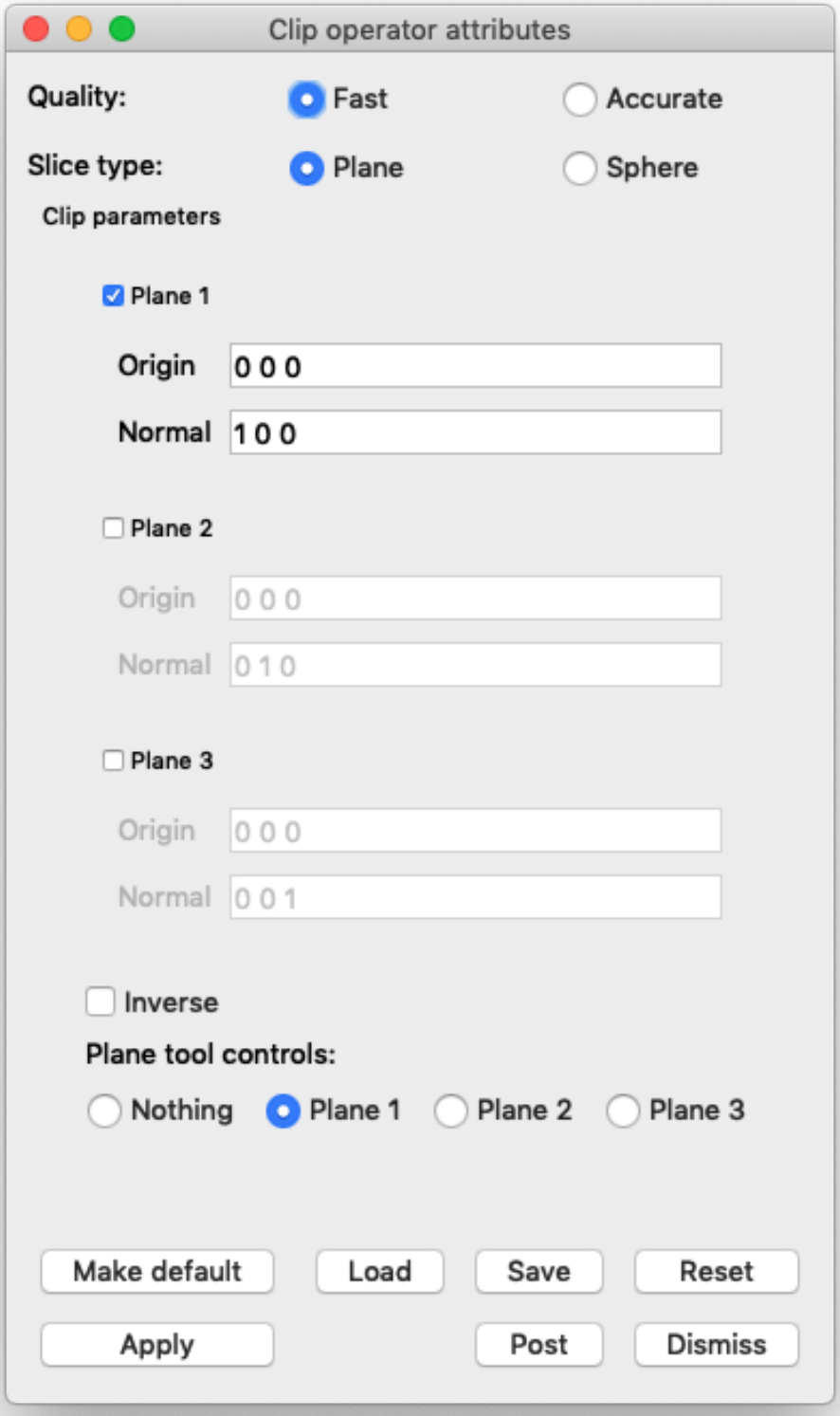


Fig. 6.91: The Clip attributes.

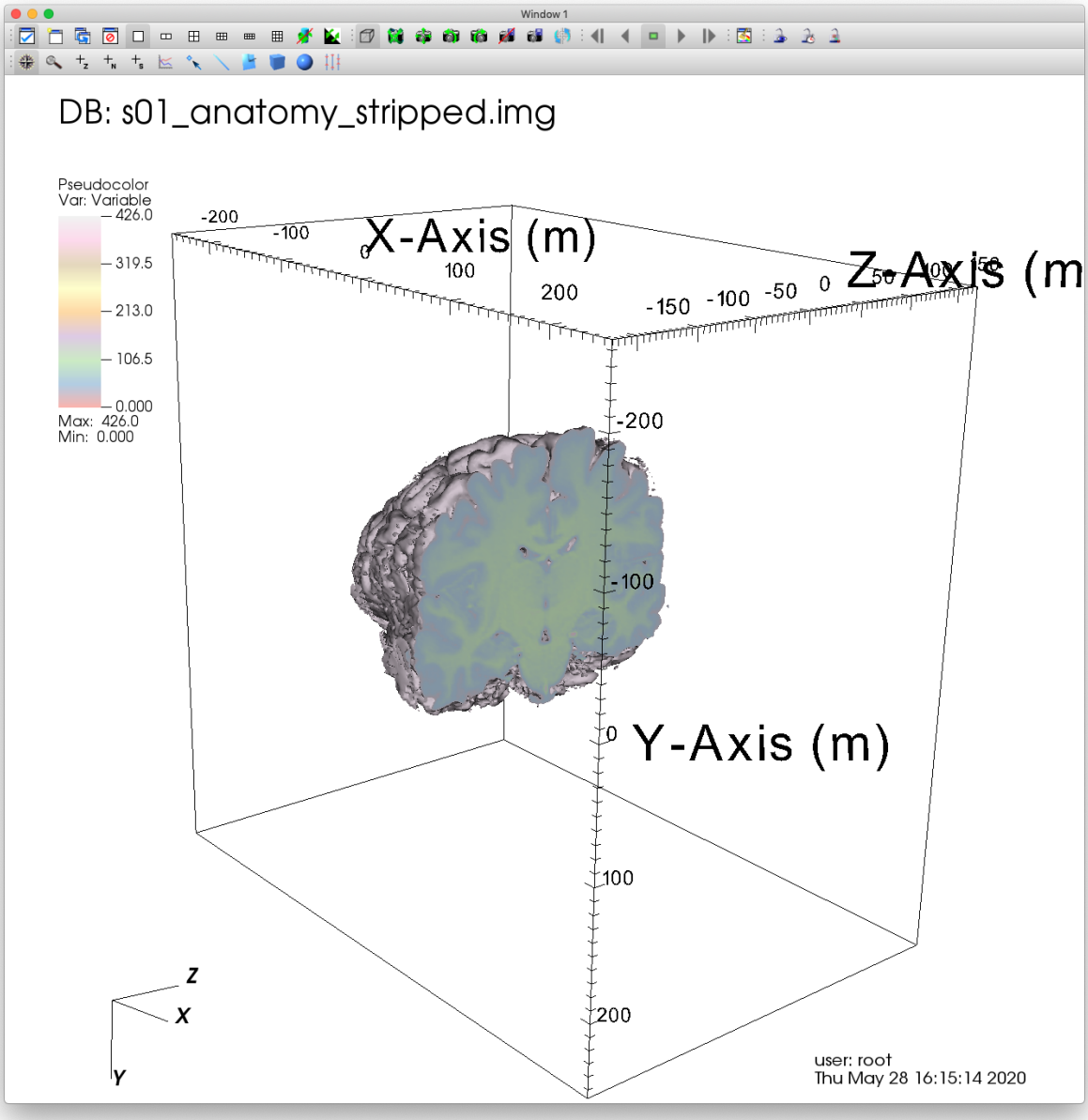


Fig. 6.92: Visualizing a Clip of our MRI dataset.

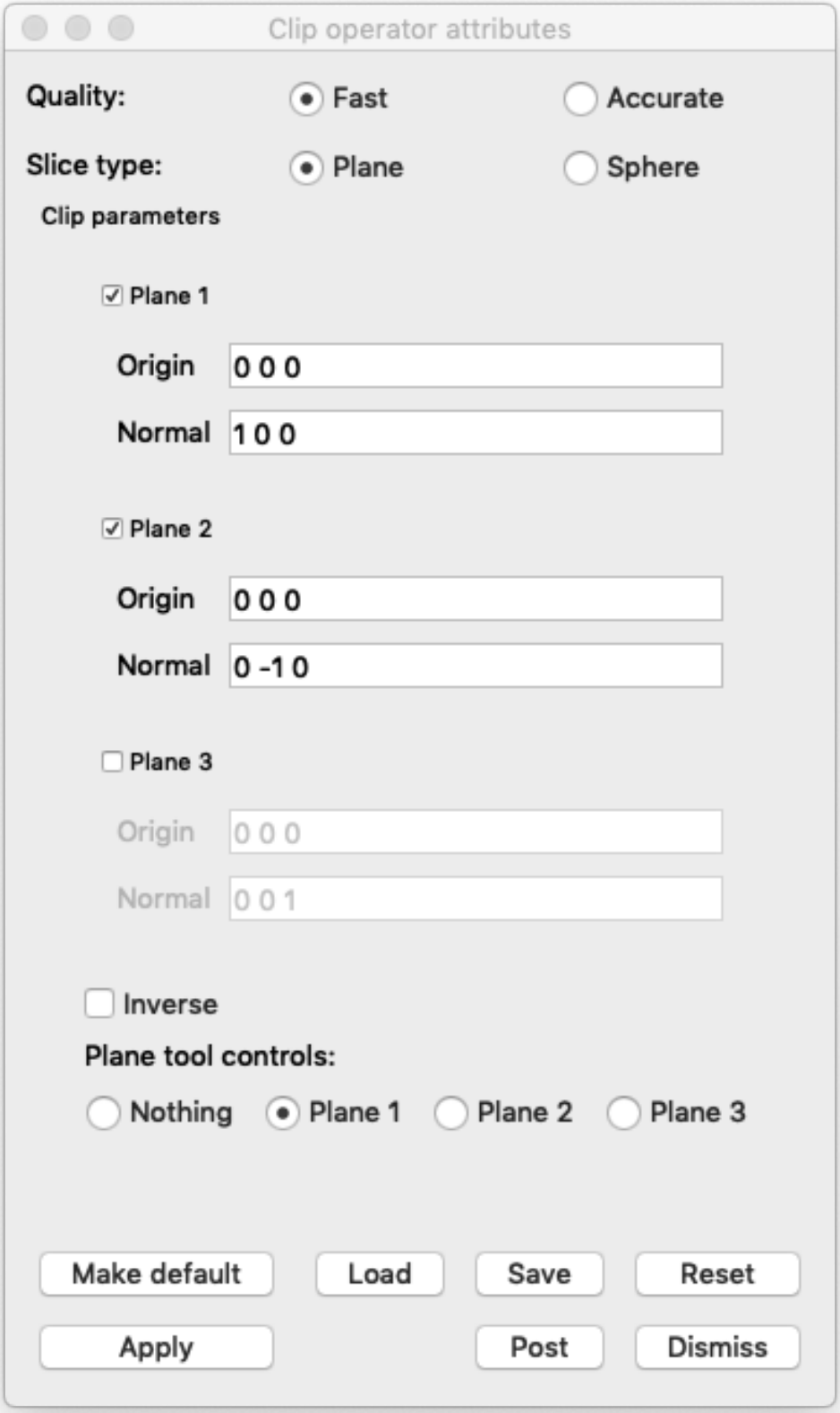


Fig. 6.93: Altering the Clip attributes.

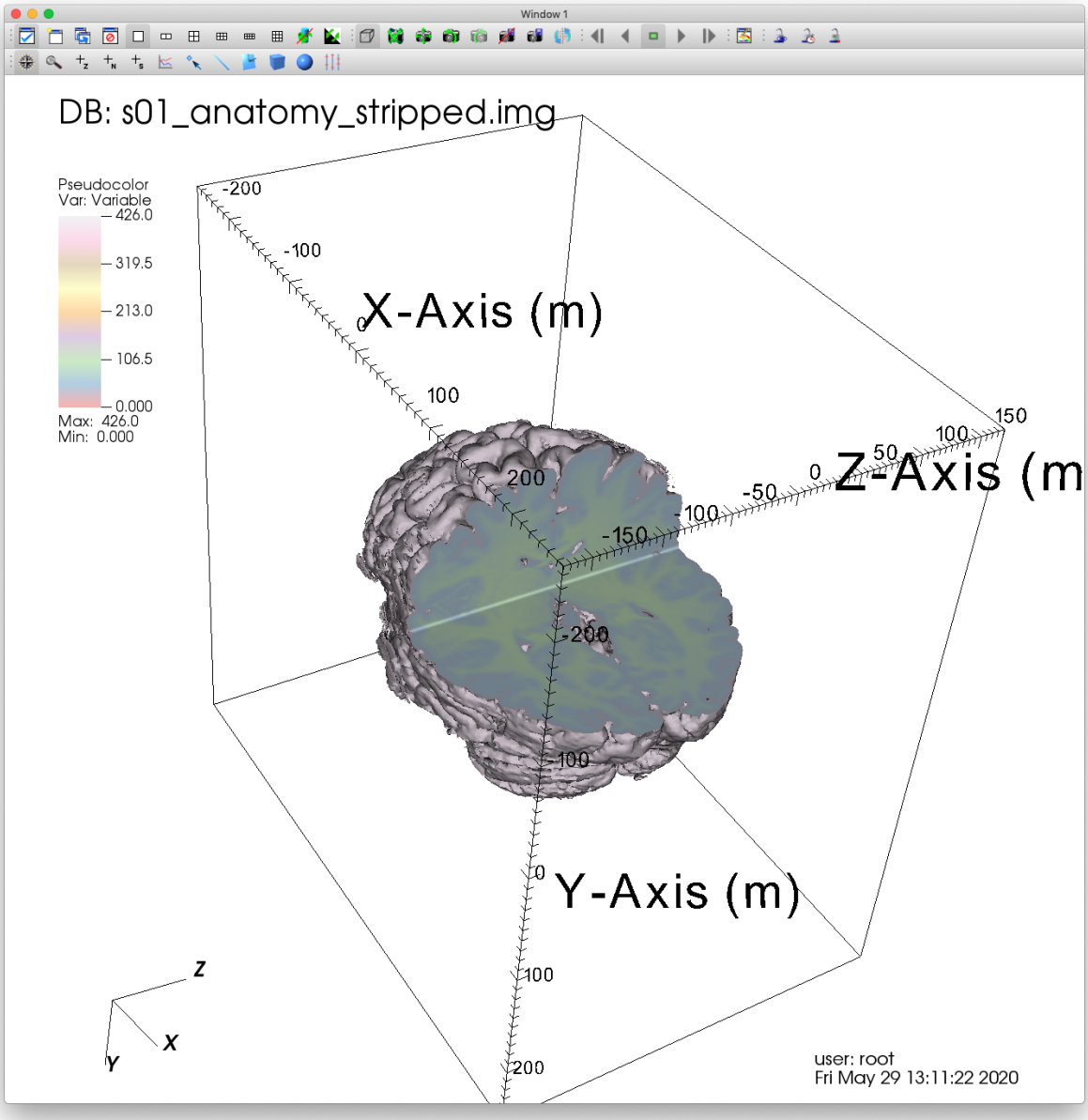


Fig. 6.94: Visualizing a 2 Plane Clip of our MRI dataset.

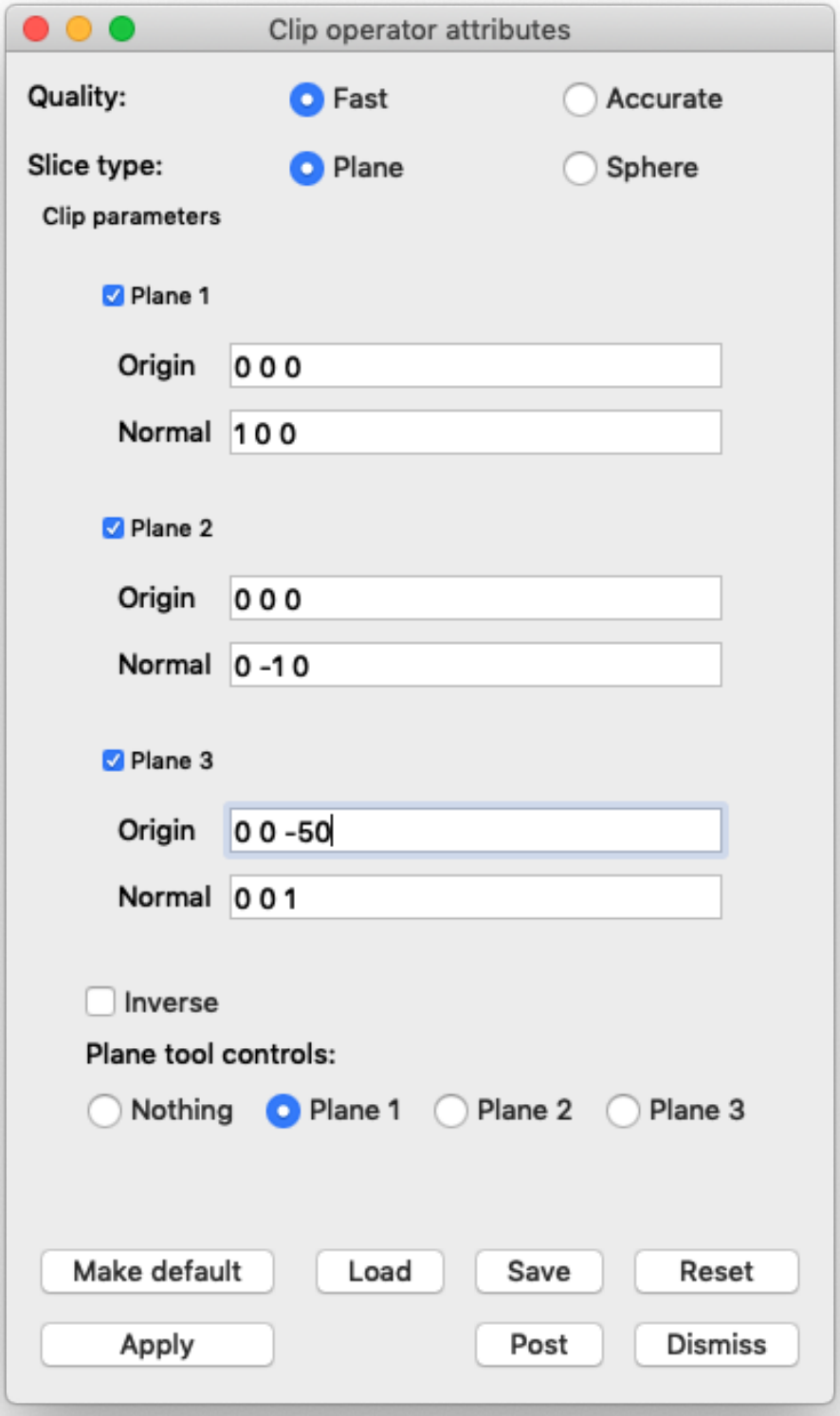


Fig. 6.95: Altering the Clip attributes.

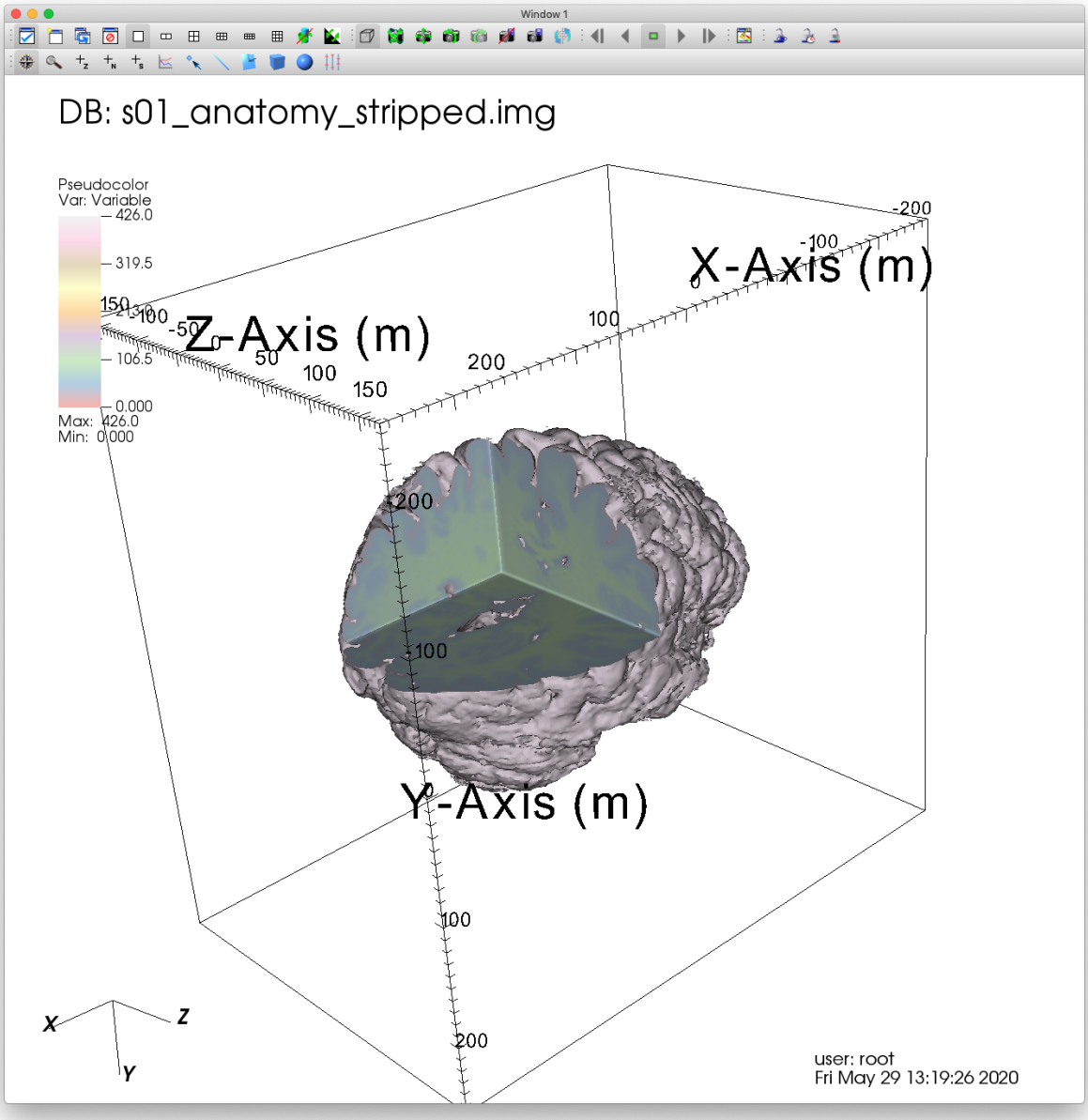


Fig. 6.96: Visualizing a 3 Plane Clip of our MRI dataset.

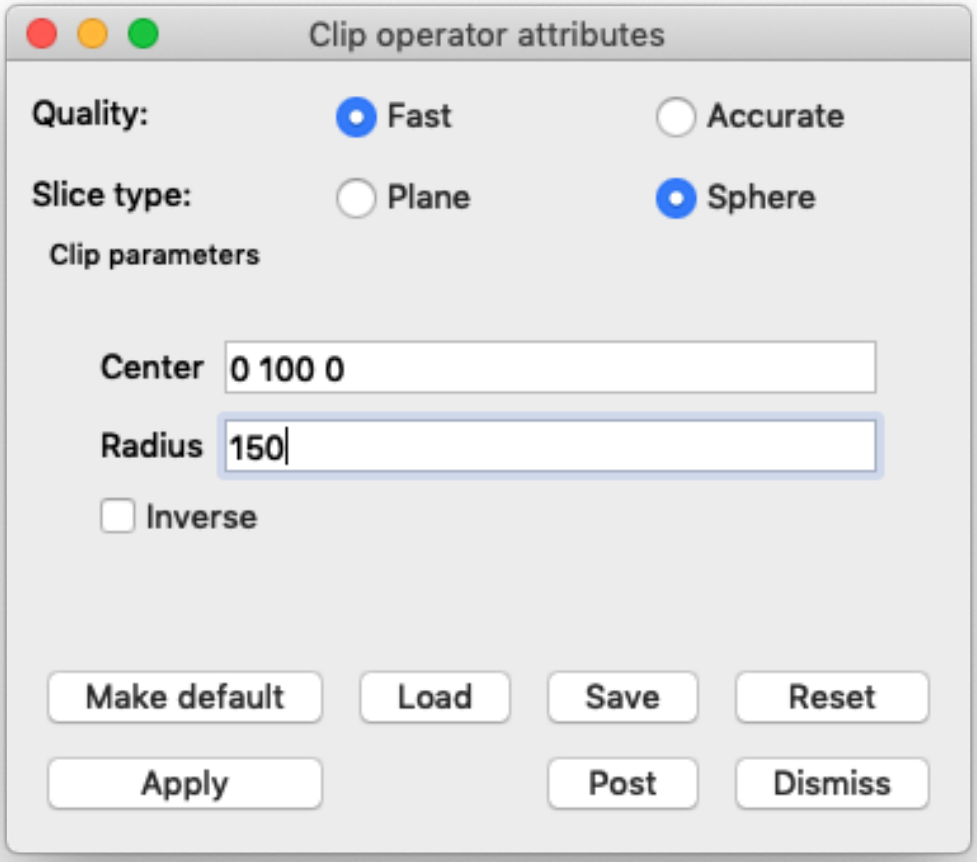


Fig. 6.97: Changing the Clip attributes.

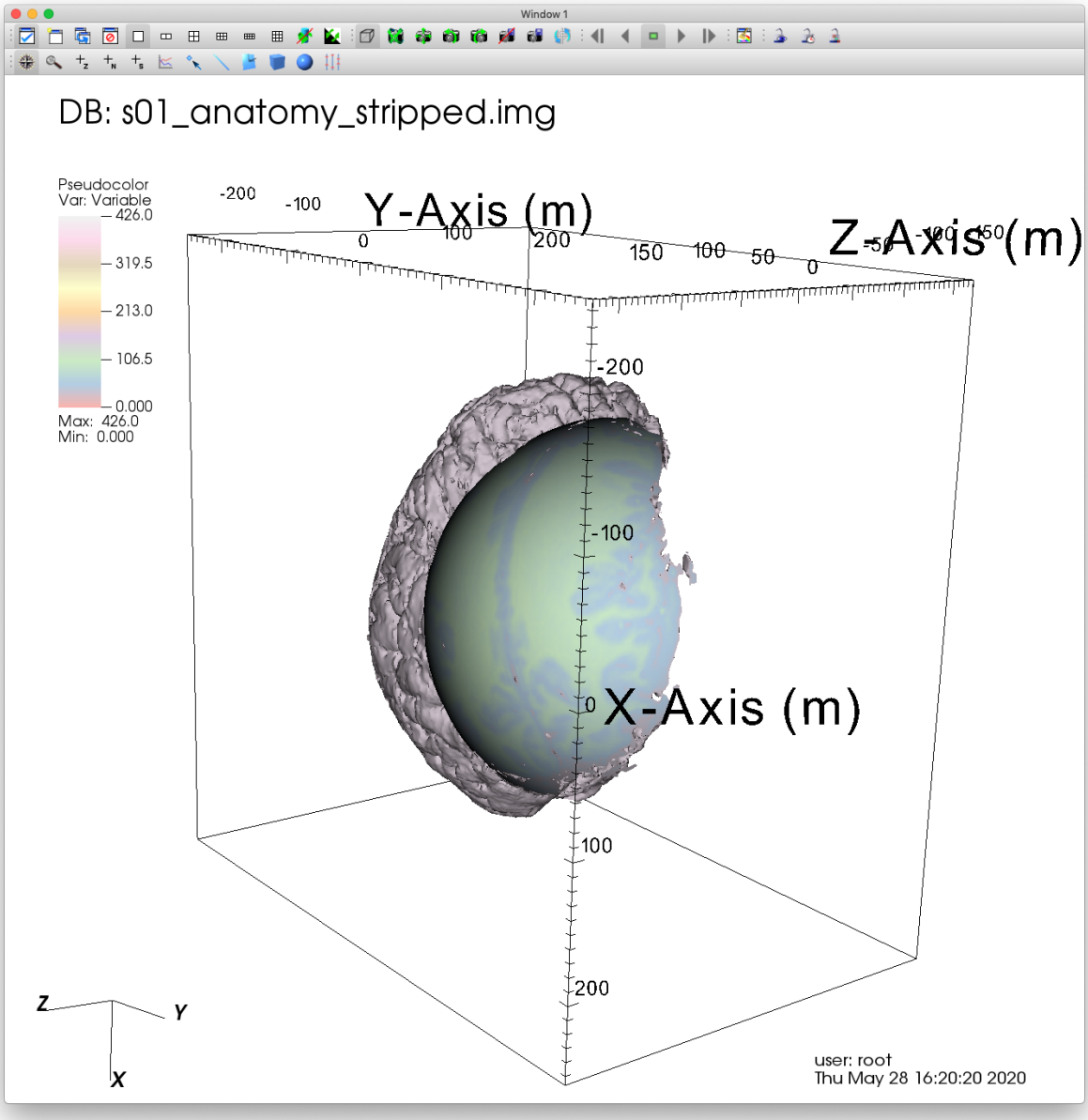


Fig. 6.98: Visualizing a spherical Clip of our MRI dataset.

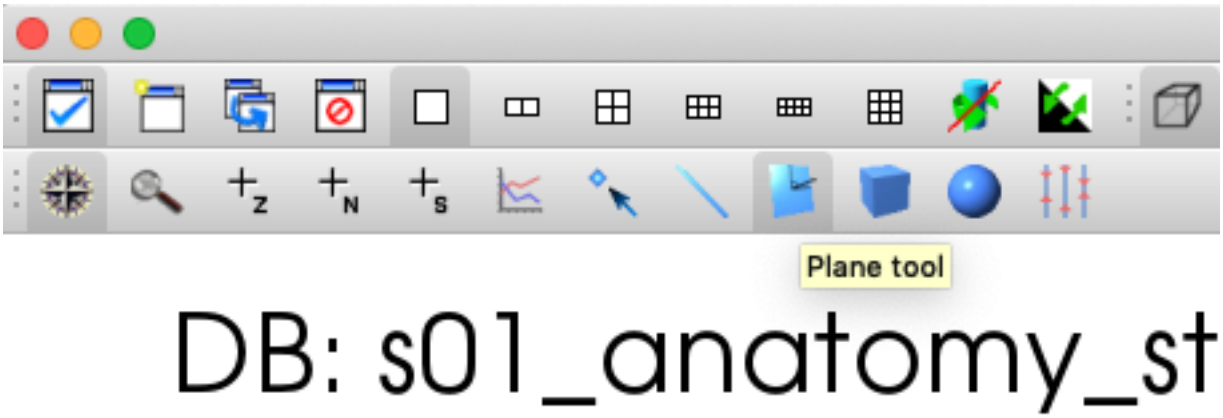


Fig. 6.99: Activating the Plane tool.

5. You will see several red boxes aligned with various points of the Plane tool. Click and drag these red boxes to re-orient the plane you are defining. VisIt will automatically perform a Clip at the newly oriented plane.

Performing a Clip using the Sphere tool

Much like the Plane tool, VisIt also provides a Sphere tool, which allows us to interactively define a sphere that can be used to set the Clip attributes.

1. *Click* the Plane tool button to deactivate the Plane tool.
2. *Click* the Sphere tool button, which is in the same row as the Plane tool.
3. Return to the Clip attributes window and change the Slice type to Sphere. *Click* Apply.
4. You can change the shape and location of the Sphere tool by clicking and dragging the red boxes associated with the Sphere.

6.7 Connected Components

VisIt provides an expression and set of queries to help identify and summarize connected subcomponents of a mesh. These capabilities can help isolate or compute statistics of complex features embedded in your data. The connected components algorithm used is unique in that it can not only process simple meshes, but it can also efficiently handle large meshes partitioned in a distributed-memory setting. This tutorial provides an introduction of how to use VisIt's connected components capabilities. The algorithm is not discussed here, for more details on the algorithm see[1] .

6.7.1 Open the dataset

This tutorial uses the "example.silo" dataset from our tutorial_data.

1. Download the tutorial_data and extract the example files.

[1] C. Harrison, J. Weiler, R. Bleile, K. Gaither, H. Childs. "A Distributed-Memory Algorithm for Connected Components Labeling of Simulation Data" in Topological and Statistical Methods for Complex Data, J. Bennett, F. Vivodtzev, V. Pascucci. Eds. Springer Berlin Heidelberg, pp. 3–21., December 2014

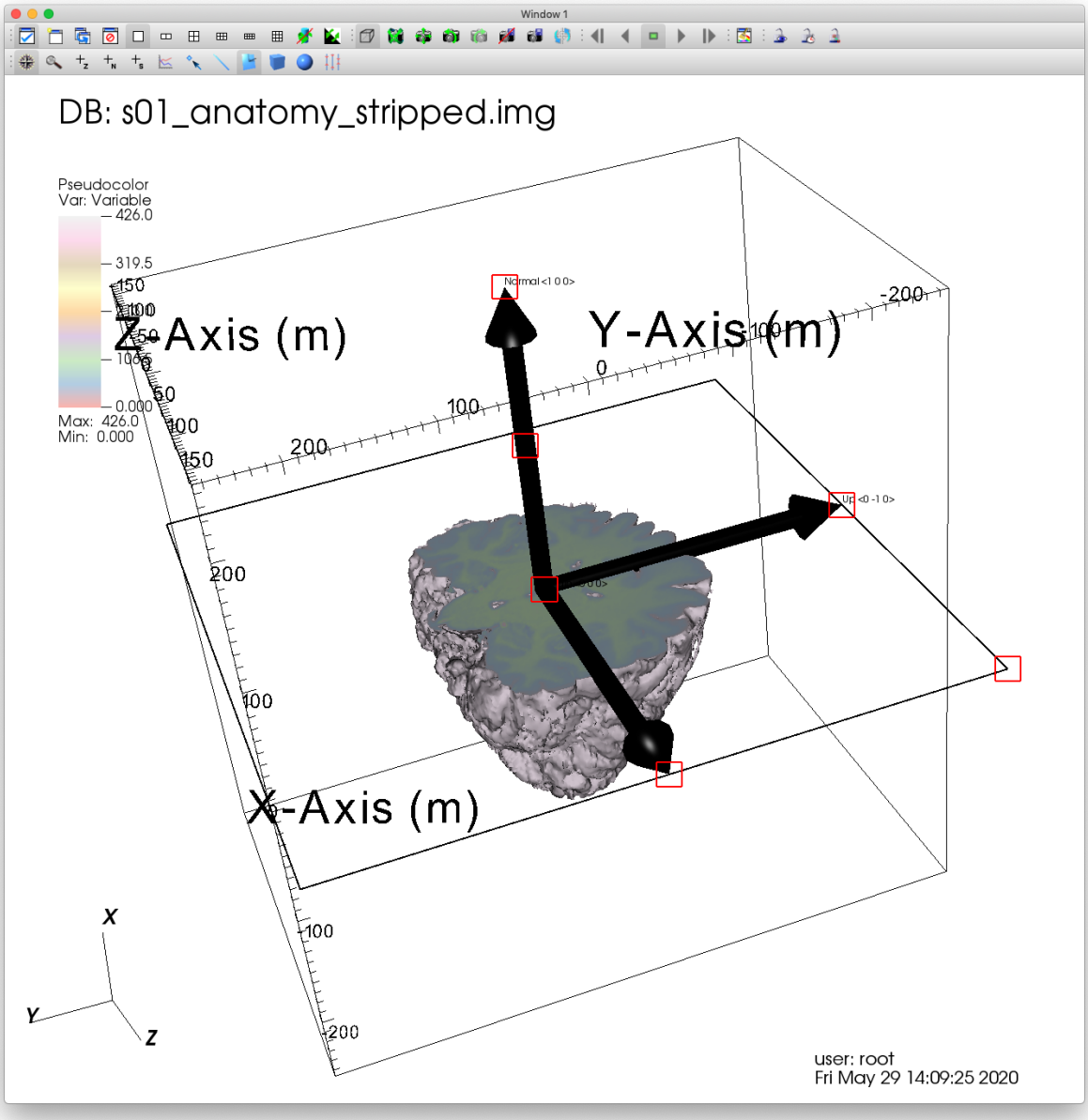


Fig. 6.100: The activated Plane tool.

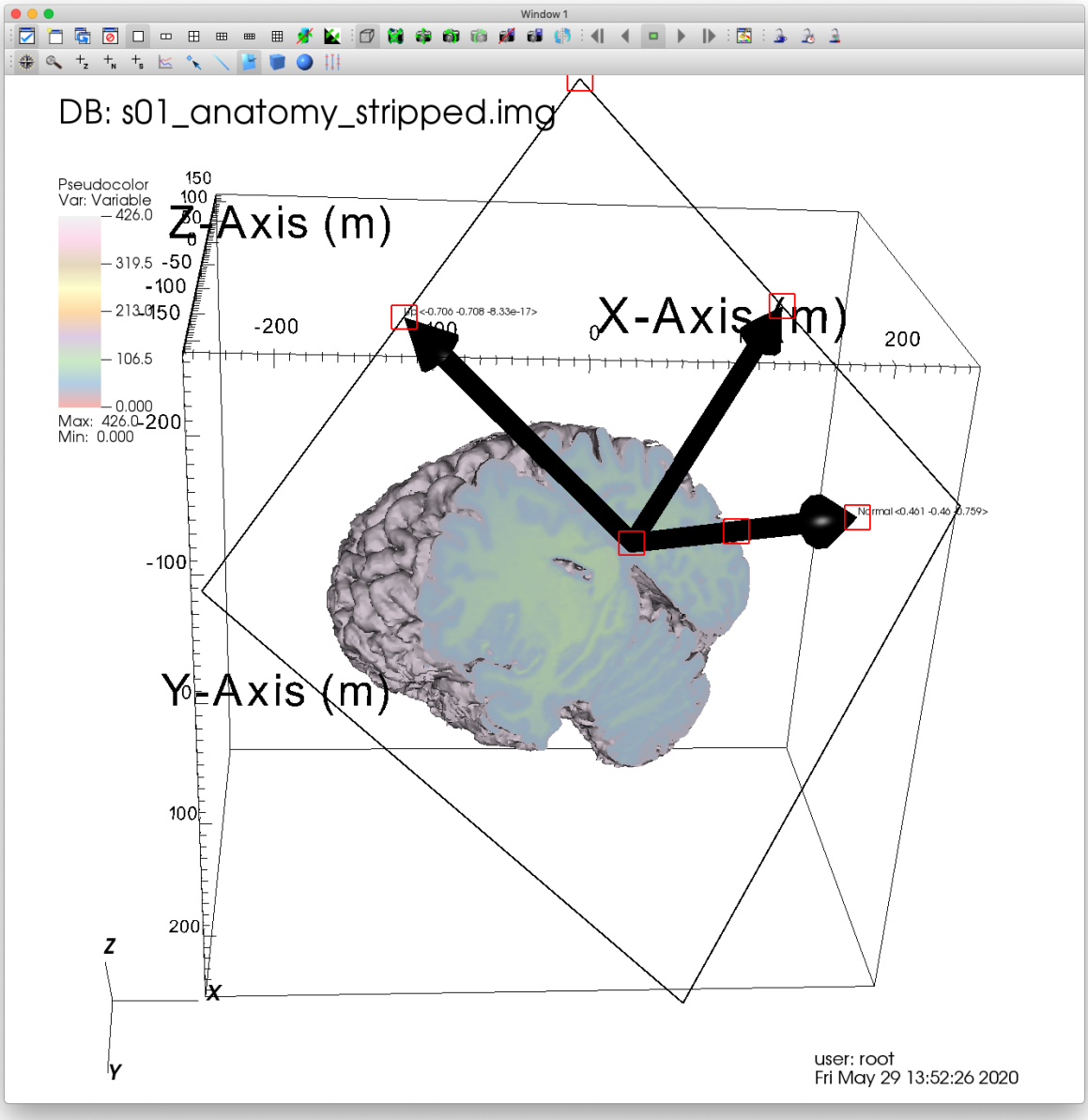


Fig. 6.101: Performing a Clip with the Plane tool.

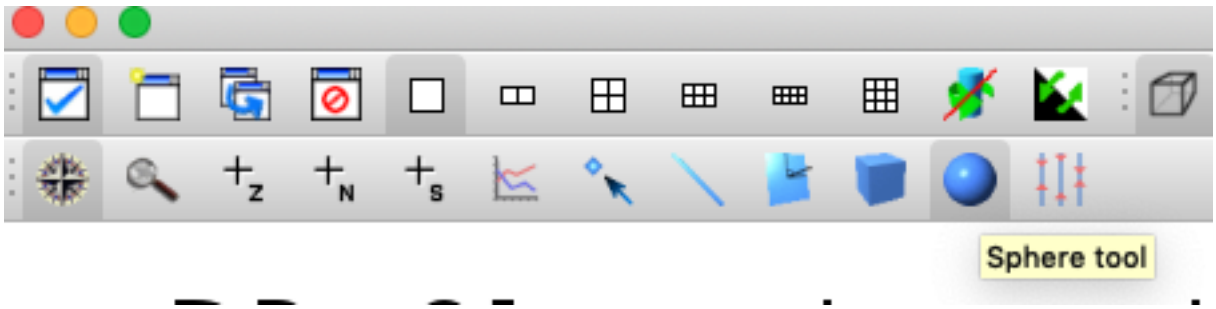


Fig. 6.102: Activating the Sphere tool.

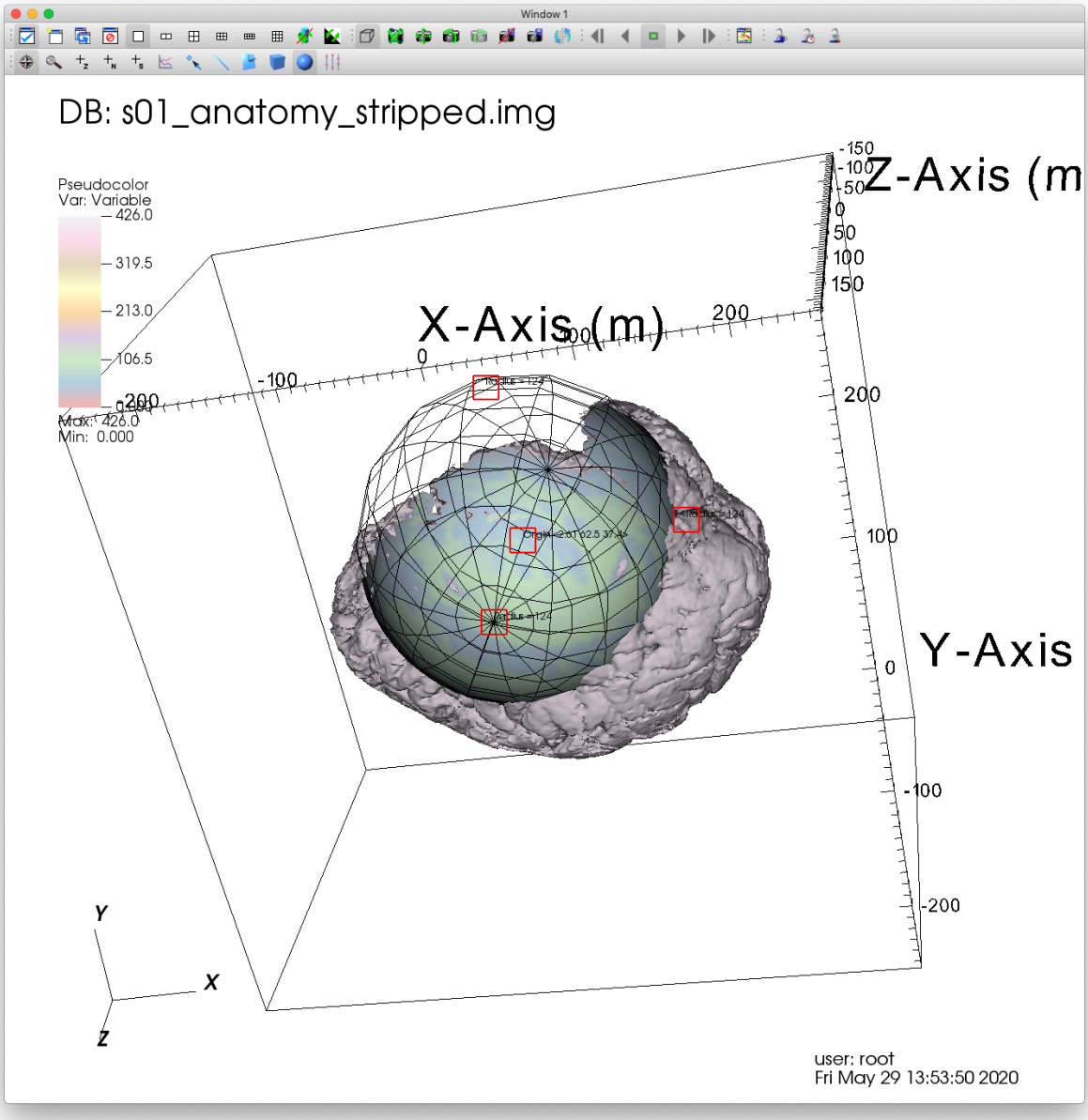


Fig. 6.103: Performing a Clip with the Sphere tool.

2. Start VisIt and click the *Open* icon to bring up the File open window.
3. Navigate your file system to the folder containing "example.silo".
4. Highlight the file "example.silo" and then click *OK*.

6.7.2 Use a scalar field to cut our mesh

This example mesh starts fully connected. First we cut our example mesh to create a mesh with smaller connected regions that we will label. In the "example.silo" dataset regions of the high *pressure* values produce an interesting pattern to explore. In practice, volume fractions or density values are also useful fields to use.

We compare using two operators, Theshold and Isovolume, to cut the mesh according to a scalar field. These methods produce different mesh topologies which influence the Connected Components Labeling result.

Threshold

First, we create a new mesh using the Threshold operator.

1. In the plot list, click *Add->Pseudocolor->pressure*.
2. In the plot list, click *Operators->Selection->Isovolume*
3. Click on the triangle to the left of your Pseudocolor plot and double click Threshold to open up the Threshold attributes.
4. Once you've opened the Threshold attributes, remove the default row. Add a new variable, select *pressure* and set the Lower bound to *3.9*.
5. Click *Apply* and dismiss the Threshold attributes Window.
6. Optionally, open the Pseudocolor attributes Window and change the color table used. These plots use the `Spectral` color table.
7. Click *Draw*. You will now see a visualization of the thresholded mesh.

The Threshold operator simply excludes elements from the mesh based on if the threshold criteria is met.

Isovolume

Next, we create a new mesh using the Isovolume operator. We created this in a new viewer window, so we can easily look at both results side-by-side.

1. Create a new viewer window
2. Delete any plots in the new window.
3. In the plot list, click *Add->Pseudocolor->pressure*.
4. In the plot list, click *Operators->Selection->Isovolume*
5. Click on the triangle to the left of your Pseudocolor plot and double click Isovolume to open up the Isovolume attributes.
6. Once you've opened the Isovolume attributes, set Variable to *pressure* and set the Lower bound to *3.9*.
7. Click *Apply* and dismiss the Threshold attributes Window.
8. Optionally, open the Pseudocolor attributes Window and change the color table used. These plots use the `Spectral` color table.
9. Click *Draw*. You will now see a visualization of the cut mesh.

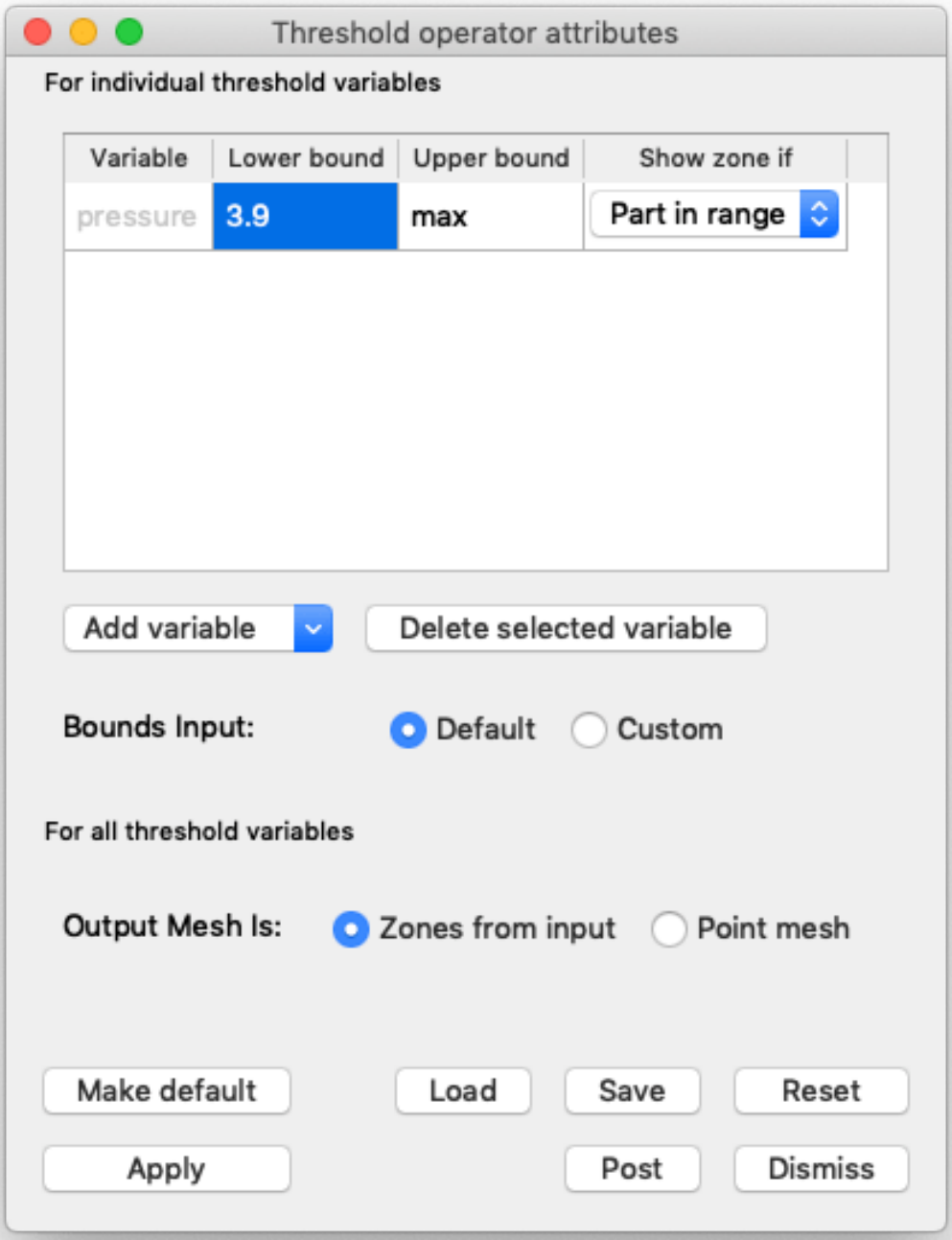


Fig. 6.104: Setting the Isovolume attributes.

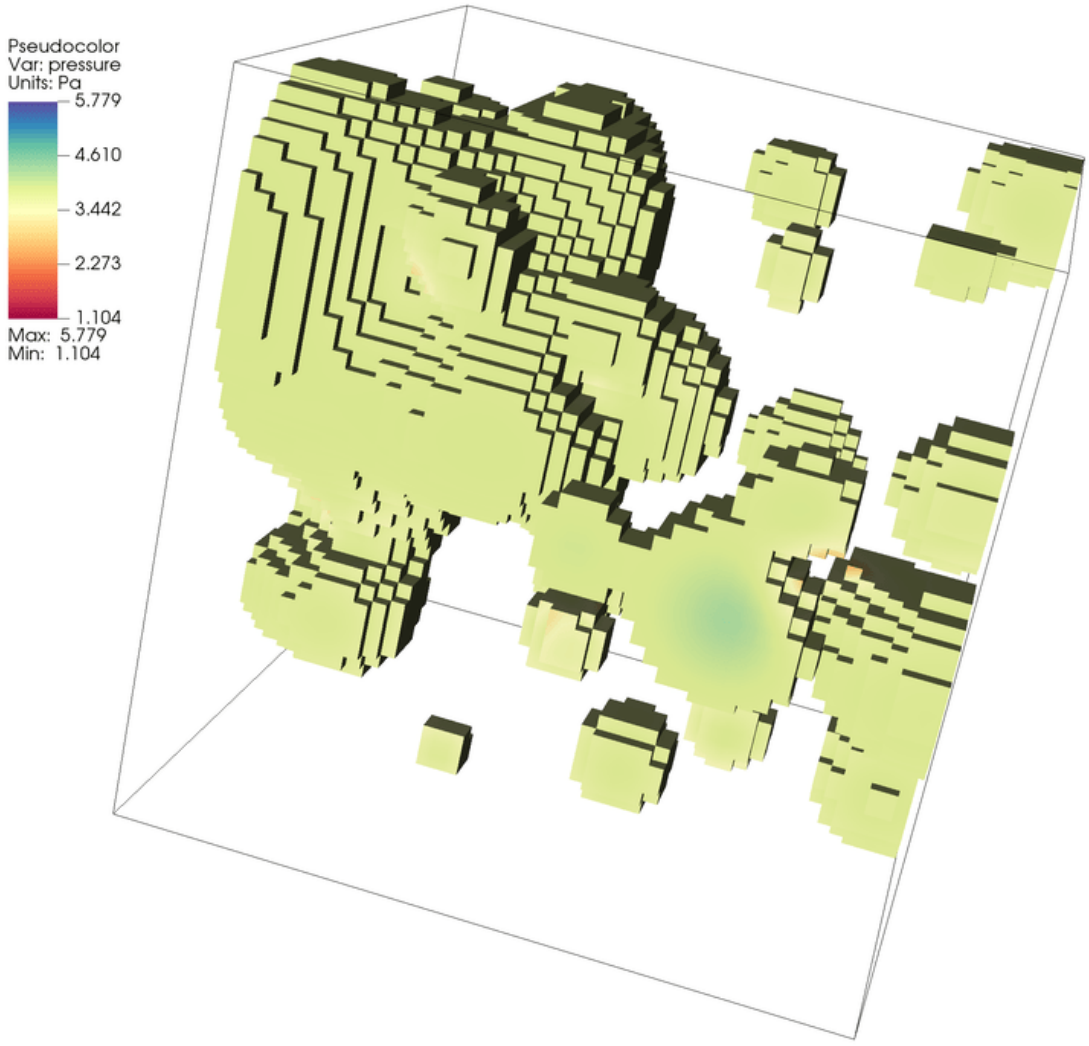


Fig. 6.105: Visualizing our thresholded example of our dataset.

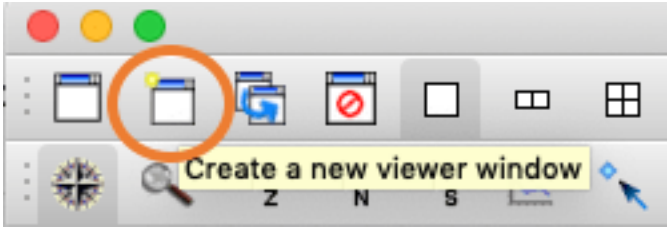


Fig. 6.106: Click to create a new viewer window.

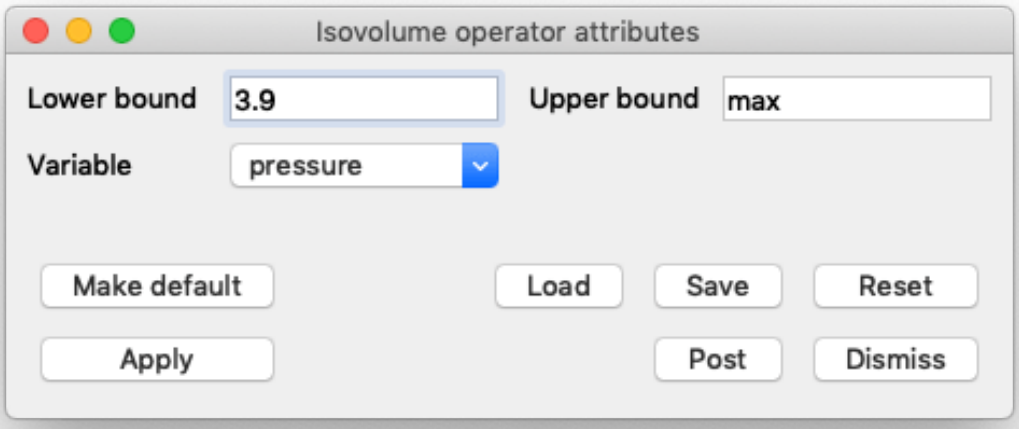


Fig. 6.107: Setting the Isovolume attributes.

The Isovolume cuts the mesh to include volumes between two Isosurfaces.

6.7.3 Labeling Connected Components with an Expression

The thresholded mesh has a blocky structure and submeshes remain connected at the edges of blocks. The isovolumed mesh has interpolated cuts, which create a smoother result and less connected submeshes. Next, we use the *conn_components* expression to view and compare connected submeshes for each of these plots.

1. Open the Expressions Window (*Options Menu->Expressions*)
2. Create a new expression named *ccl* with the definition `conn_components(Mesh)`.
3. Click *Apply* and dismiss the Expressions Window.

Now we use the *ccl* expression with our existing pipelines.

Connected Components of Threshold Result

1. Make Window 1 active (The window with the Threshold operator pipeline)
2. In the plot list, click *Operators->Analysis->DeferExpression*
3. Click on the triangle to the left of your Pseudocolor plot, and double click DeferExpression to open up the DeferExpression attributes.
4. Once you've opened the DeferExpression attributes, add *ccl* to the list of deferred expressions.
5. Click *Apply* and dismiss the DeferExpression attributes Window.
6. Use the Variables menu to change the active variable to *ccl*.
7. Optionally, open the Pseudocolor attributes Window and change the color table used. These plots use the `Spectral` color table.

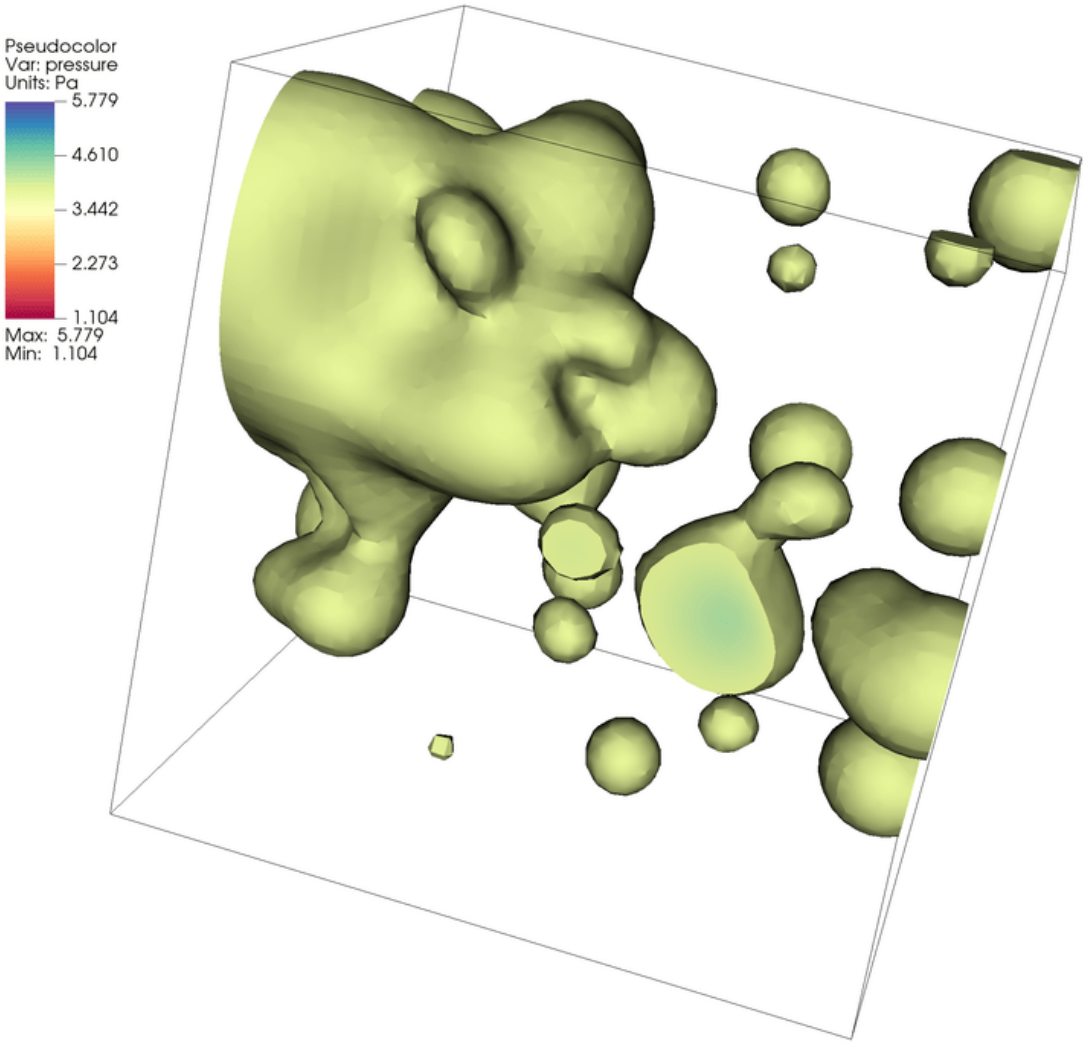


Fig. 6.108: Visualizing an isovolume from our example of our dataset.

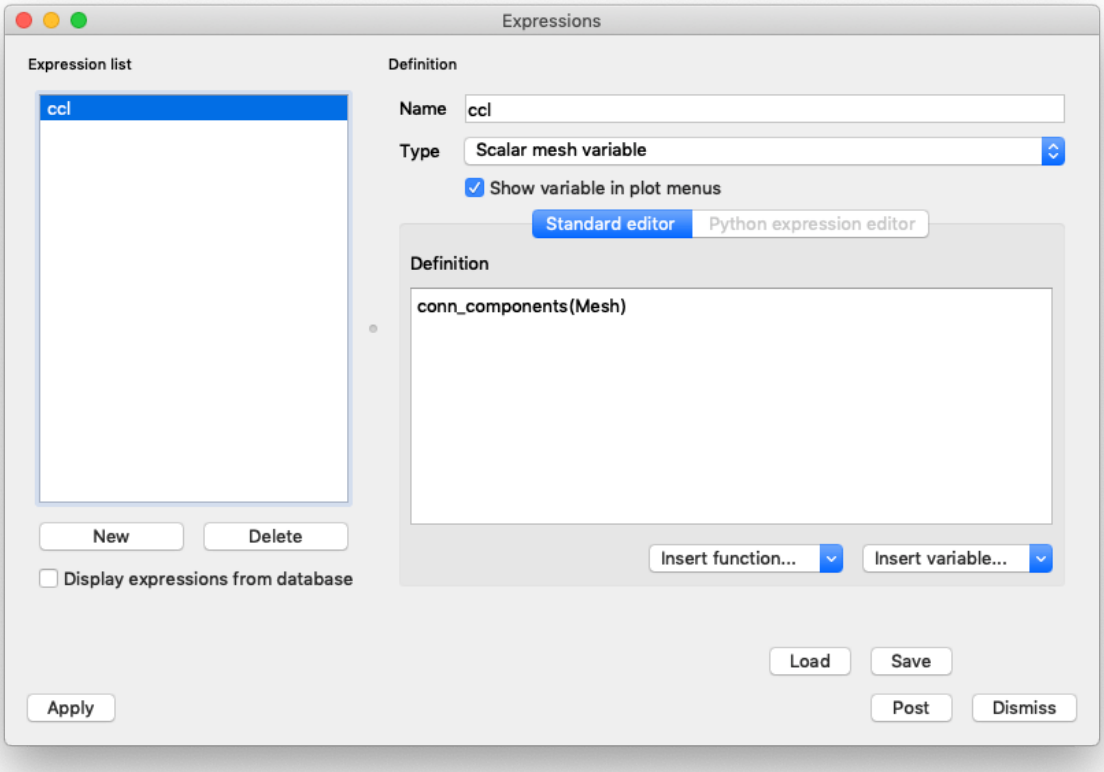


Fig. 6.109: Defining a *conn_components* expression

Fig. 6.110: Changing active window to 1

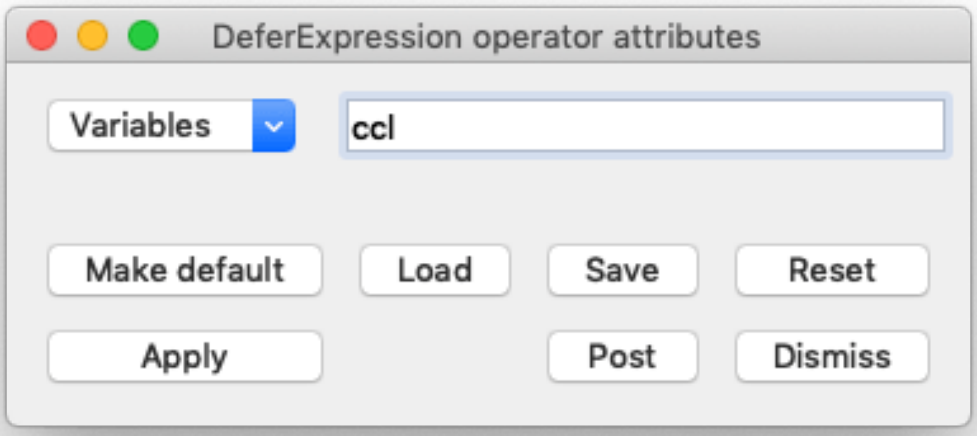


Fig. 6.111: Setting the DeferExpression attributes. This operator instructs VisIt to execute the *ccl* expression on the Threshold result, instead of the original mesh.

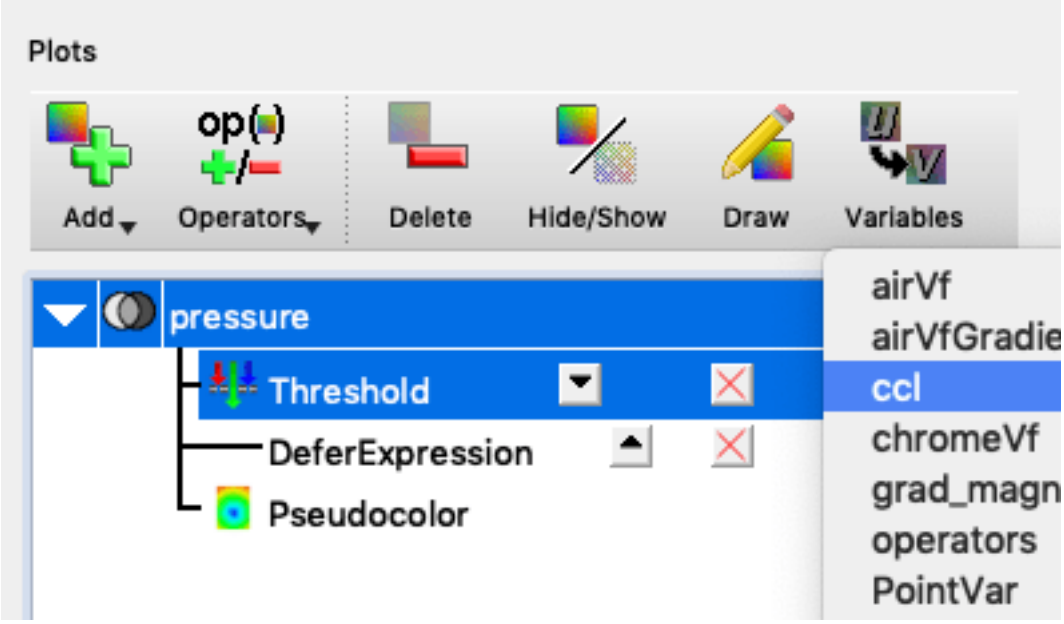


Fig. 6.112: Changing active plot variable to *ccl*.

8. Click *Draw*. You will now see the mesh rendered with Connected Component Labels.

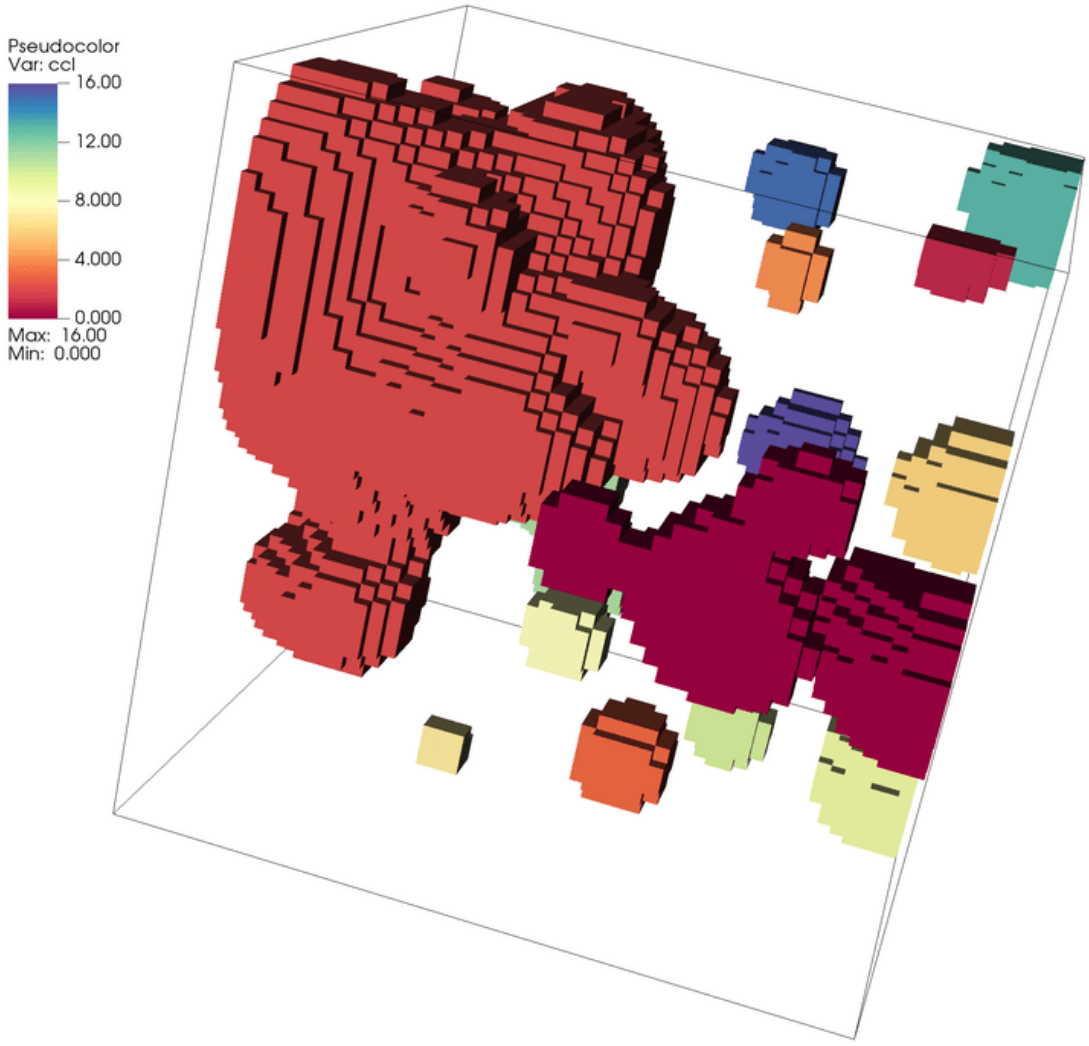


Fig. 6.113: Connected Components of the Threshold Result

We now see a Pseudocolor plot of a new scalar field where each element is associated with its connected component label. In this case we have 17 connected components labeled using ids 0 - 16.

Connected Components of Isovolume Result

1. Make Window 2 active (The window with the Isovolume operator pipeline)
2. Repeat the steps above to add a *DeferExpression* operator and set it up to defer the *ccl* expression.
3. Use the Variables menu to change the active variable to *ccl*.

Fig. 6.114: Changing active window to 2

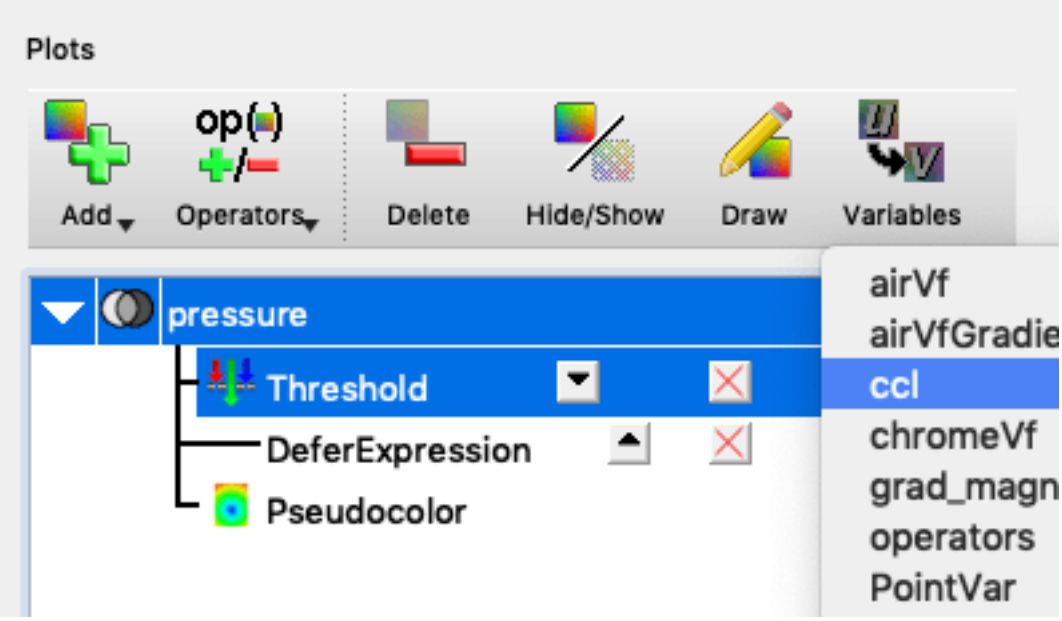


Fig. 6.115: Changing active plot variable to *ccl*.

4. Optionally, open the Pseudocolor attributes Window and change the color table used. These plots use the `Spectral` color table.
5. Click *Draw*. You will now see the mesh rendered with Connected Component Labels.

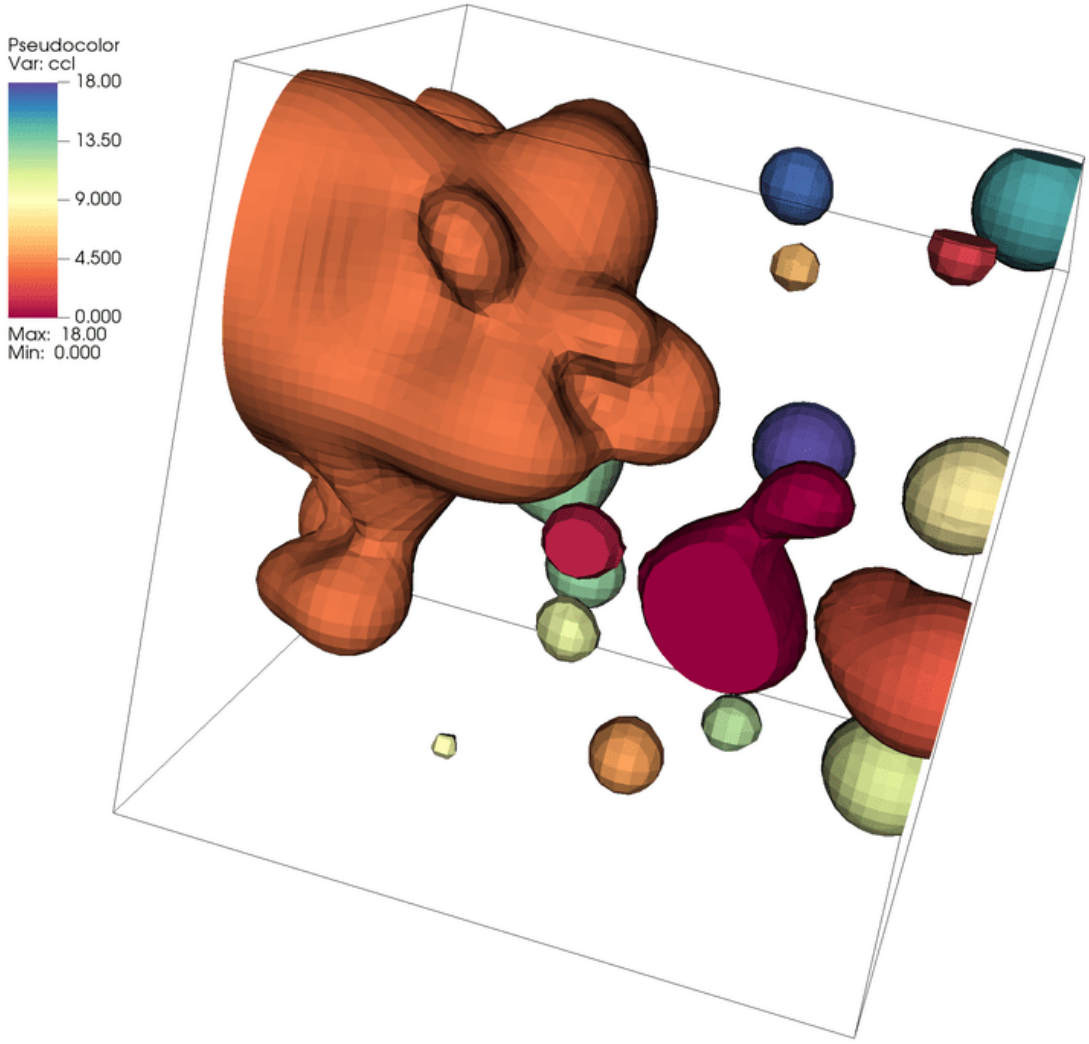


Fig. 6.116: Connected Components of the Isovolume Result

Again, we now have Pseudocolor plot of a new scalar field where each element is associated with its connected component label. In this case we have 19 connected components labeled using ids 0 - 18. You can lock the views between the two windows to compare the differences in the meshes and identify where the connected components differ.

6.7.4 Using the Connected Component Summary query

The Connected Components Summary query computes and aggregates all component info and optionally writes this data to an okc file (an XMDF File format)

The query returns the following details of each component:

Component Data	Fields
Component Id	comp_label
Centroid	comp_x, comp_y, comp_z
Number of Cells (or Zones) per component	comp_num_cells
Area (if the dataset is 2D)	comp_area
Volume (if the dataset is 3D or RZ)	comp_volume
Variable Sum	comp_sum
Weighted Variable Sum	comp_weighted_sum
Spatial Bounding Box	comp_bb_x_min, comp_bb_x_max, comp_bb_y_min, comp_bb_y_max, comp_bb_z_min, comp_bb_z_max
Number of MPI Tasks spanned by the component	comp_num_procs

Next, we use the Connected Components Summary via python on one of our plots to obtain this info.

1. Launch VisIt's Command Line Interface (CLI) (*Controls Menu->Launch CLI*)

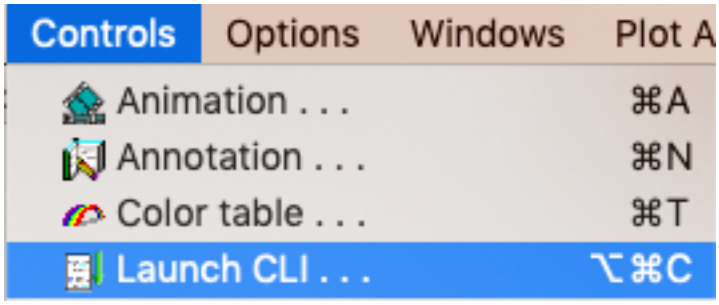


Fig. 6.117: Launch the CLI

2. Run the following code snippets (Example output below is from the Isovolume case)

   Python Snippet

```
# Execute our connected components query and get the result
Query("Connected Components Summary")
res = GetQueryOutputObject()

# Show names in the results dictionary
print(res.keys())

# Print the array of per component volumes
print(res["comp_volume"])
```

   Output

```
"Found 19 connected components.\nComponent summary information saved to cc_
↪summary.okc, which can be imported into VisIt"
```

```
['comp_bb_x_max', 'comp_bb_z_max', 'comp_sum', 'comp_y', 'comp_num_procs', 'comp_
↪bb_x_min', 'comp_weighted_sum', 'comp_bb_y_min', 'comp_z', 'comp_volume', 'comp_
↪x', 'comp_bb_z_min', 'comp_num_cells', 'connected_component_count', 'comp_bb_y_
↪max']
```

```
(37.97730226694259, 3.2019942591930146, 1.610134229606217, 33.371787344299676,␣
↪907.2334157190477, 4.499707094552377, 0.9447130410516479, 7.414511301985026, 15.
↪064008190720848, 0.18155970817315392, 3.1918362871108457, 22.247388229041434, 2.
↪625056508686029, 78.24442360391282, 2.4172440068352756, 23.494122927868506, 18.
↪57216353121875, 6.944255799937935, 8.499496779401833)
```

Python Snippet

```
# Print all of the results in a ~human friendly way
import json
print(json.dumps(res,indent=2))
```

Output

Lots of text, so we omit it here!

6.8 Remote Usage

VisIt can be used remotely in several different manners. Some use capabilities native to VisIt, such as running VisIt in client/server mode, and some use external mechanisms, such as VNC. We will also touch briefly on using batch allocations in an interactive manner.

VisIt can run remotely in the following ways:

- *Using X display forwarding*.
  - Easiest to setup and convenient to use.
  - Lowest interactivity performance.
- *Using a VNC client*.
  - More complex to set up.
  - Convenient to use.
  - Provides high interactivity performance.
- *Using client/server*.
  - More complex to set up.
  - Provides highest interactivity performance.

6.8.1 Using X Display forwarding though ssh

When VisIt is running with X display forwarding through ssh, it is completely running on the remote system and sending all its graphics commands over ssh. In one sense this is the easiest to use since you just launch VisIt on you remote system and you are ready to go. Since you are typically already logged into the remote system and already in the directory of interest there is no additional setup required, such as entering passwords or navigating the remote directory structure. Unfortunately it is also the lowest performing option. Graphical user interfaces typically send lots of small messages back between the remote system and the local display. If there is a high latency between them then simple operations such as clicking on buttons and bringing up new windows may take a long time. Furthermore, the

rendering performance of the visualization windows suffers because VisIt can't leverage the graphics processing unit on the local system.

When using X Display forwarding you need to have an X Server running on the display of your local system. In the case of Linux and MacOS, both will have X Servers running by default. In the case of Windows you will need to install a X Server on your system and enable it. Fortunately, most people will already have an X Server installed on their system if they are using ssh to login to the supercomputing center.

Typically, X display forwarding is enabled by default and all you need to do is launch VisIt on the remote system once you have ssh'ed to the remote system.

When starting ssh from a command line you will need to use the "-Y" option.

```
ssh -Y
```

Some X Servers may need to have their default options set for use with VisIt. This is primarily because VisIt uses OpenGL for rendering and not all X Servers are configured properly to work with OpenGL.

Configuring X-Win32 for use with VisIt

The default setting X-Win32 sometimes are not set to work well with OpenGL. This isn't always the case and will depend on the graphics card installed on your system. If VisIt crashes on your system you will need to do the following.

1. Bring up the X-Win32 control panel.
2. Go to the *Window* tab.
3. Turn off *Use Direct2D*.
4. Turn on *Use Software Renderer for OpenGL*.
5. Click *Apply*.
6. At this point you should exit all the windows associated with X-Win32 and re-establish you connections to the remote system.

6.8.2 Using VNC

When using VNC it looks and behaves just like you were logged into an X Window display running at the supercomputing site that is constrained to a single window and is separate from the windowing system running on your local system. It provides all the conveniences of X display forwarding but at a much higher interactivity level since the networking between the remote computer and the VNC server will provide high bandwidth and low latency. Ideally you would do all your interactions with the supercomputer center through the VNC client. The one draw back is that the VNC server compresses the video stream it sends to the VNC client in order to provide high interactivity. This may result in small compression artifacts in the images you see in the VNC client.

This portion of the tutorial on using VNC will focus on using RealVNC at the Lawrence Livermore National Laboratory (LLNL). Using VNC at other computer centers will be similar, but unique to each site.

Installing VNC

If your system is an LLNL managed system you can install it via the LLNL workstations catalog for MacOS or Windows. Alternatively, you can download the RealVNC client and install it on your desktop. VNC clients not supplied by RealVNC will not work at LLNL.

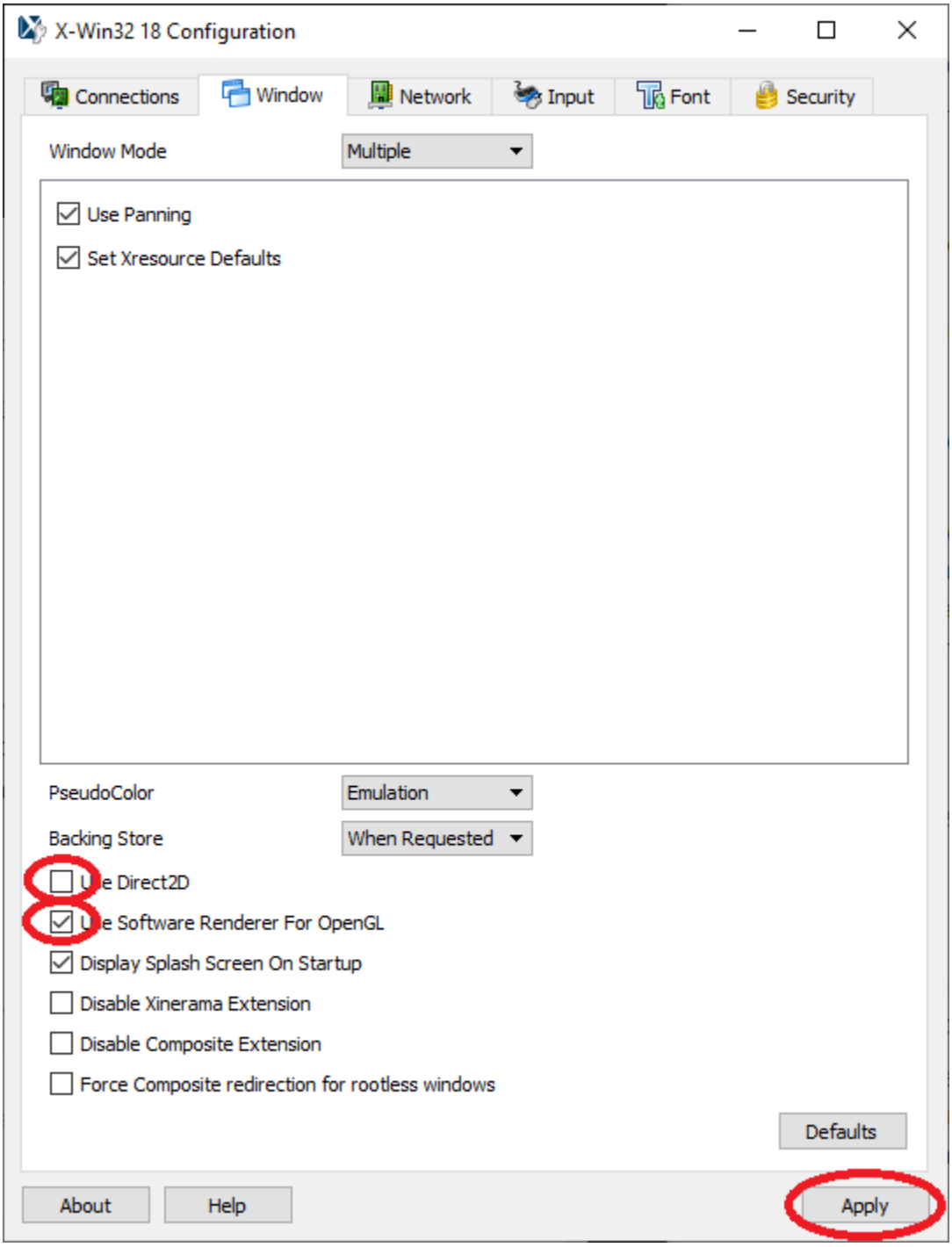


Fig. 6.118: The X-Win32 control panel

Installing RealVNC on an LLNL managed Windows system

1. Select *LANDESK Management->Portal Manager* from the Start menu.
2. Click on *RealVNC Viewer* in the list of software packages.
3. Click *Launch* to install the package.

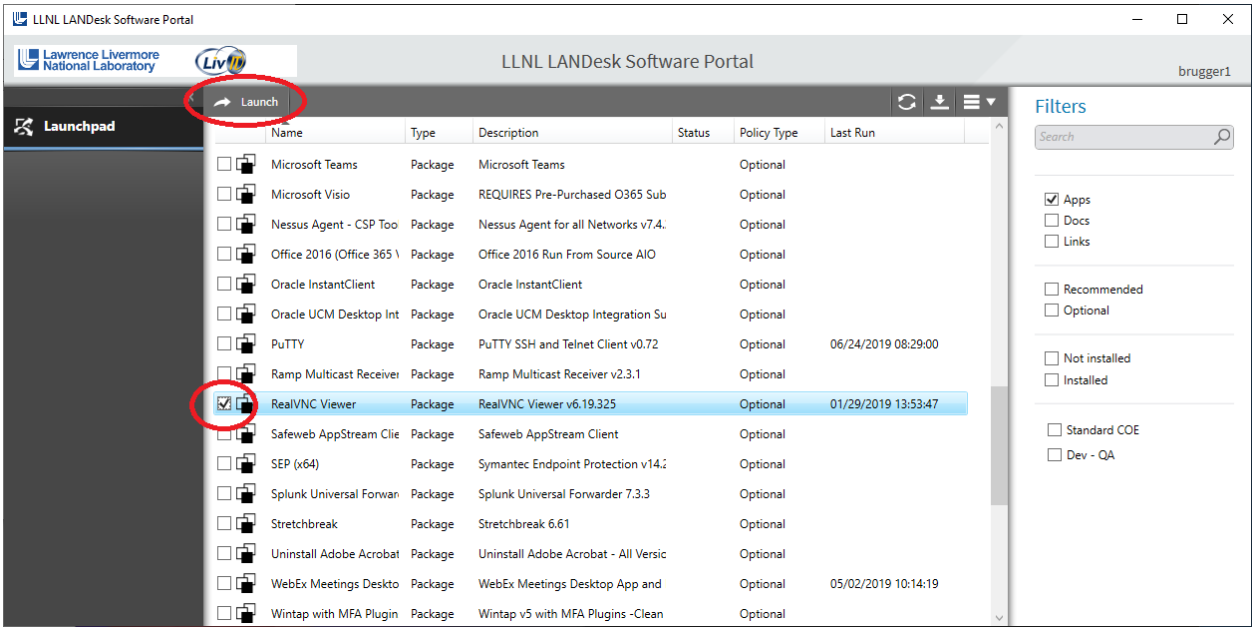


Fig. 6.119: The LLNL LANDesk Software Portal

Installing RealVNC on an LLNL managed Mac system

1. Start MacPatch from *Applications->MacPatch.app*.
2. Select the **Software** tab and scroll down until you find the *RealVNC Viewer*.
3. Click the *Install* button in the right column to install the package.

Starting up the RealVNC client

There is a lot of additional content on using RealVNC at Livermore Computing.

At this point we will focus on running RealVNC on Windows. Other than starting the Viewer, everything should be pretty much the same for Windows, MacOS and Linux.

1. Select *RealVNC->VNC Viewer* from the Start menu.
2. This will bring up the VNC Viewer.

Now we are ready to create the profiles for logging into the CZ and RZ.

1. Select *File->New connection...*.
2. This will bring up the Properties window.
3. Change the *VNC Server* field to "czvnc.llnl.gov:5999".
4. Change the *Name* field to "CZ VNC".

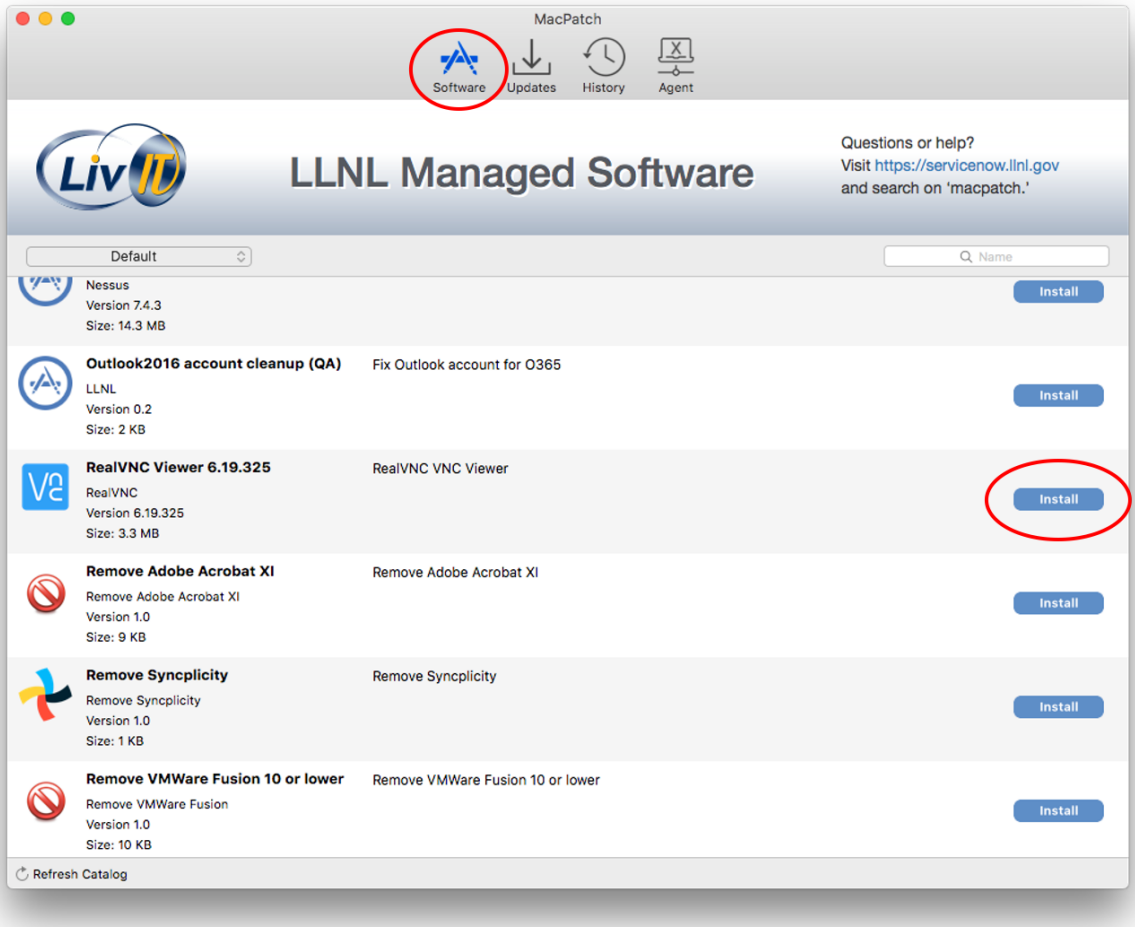


Fig. 6.120: MacPatch: LLNL Managed Software

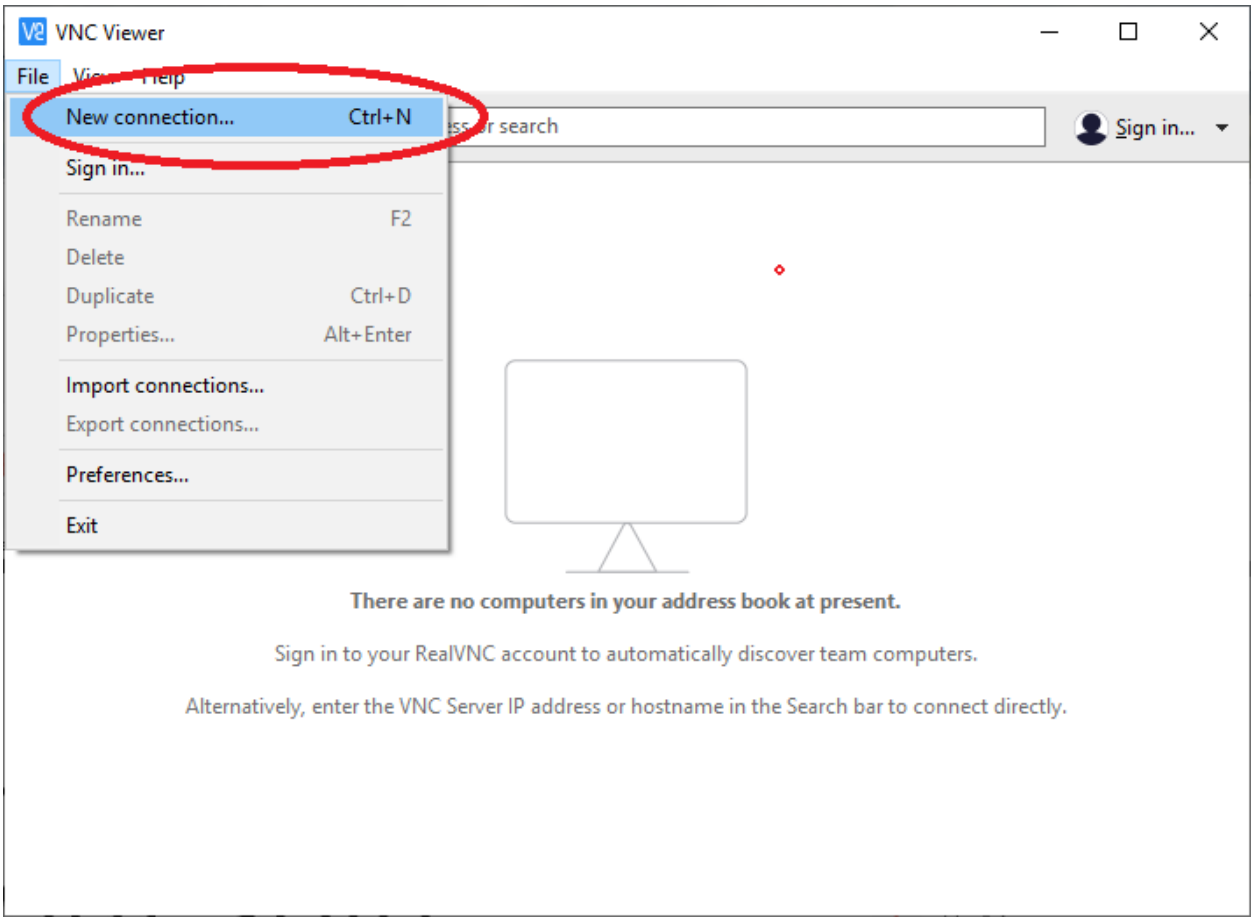


Fig. 6.121: The VNC Viewer

5. Click *Ok*.

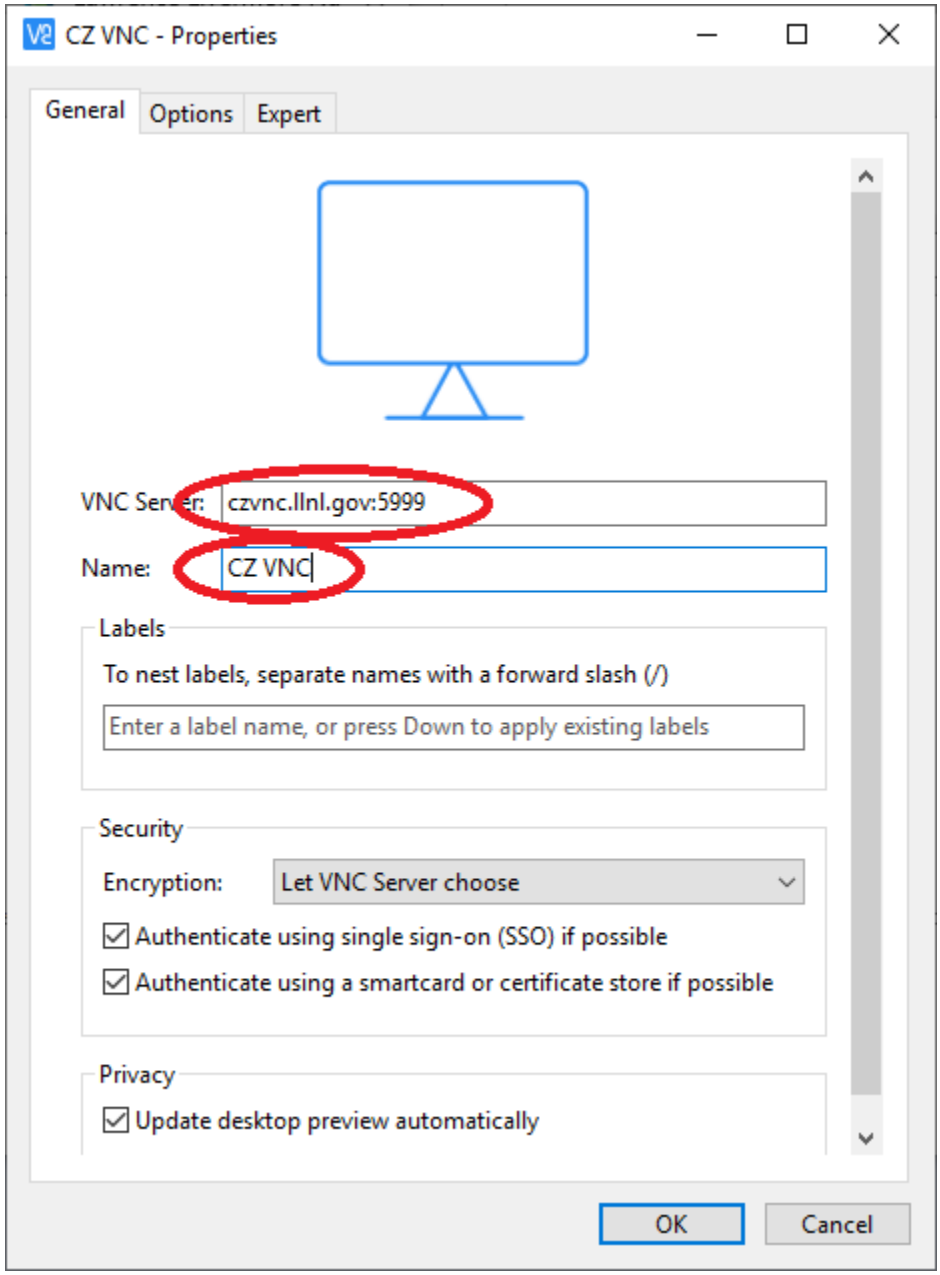


Fig. 6.122: The VNC Viewer Properties window

6. This will create a profile for logging into the CZ VNC.
7. Now do the same for the RZ.
8. Select *File->New connection...*.
9. Change the *VNC Server* field to "rzvnc.llnl.gov:5999".
10. Change the *Name* field to "RZ VNC".
11. Click *Ok*.

12. Your VNC Viewer window should now contain two connection profiles.

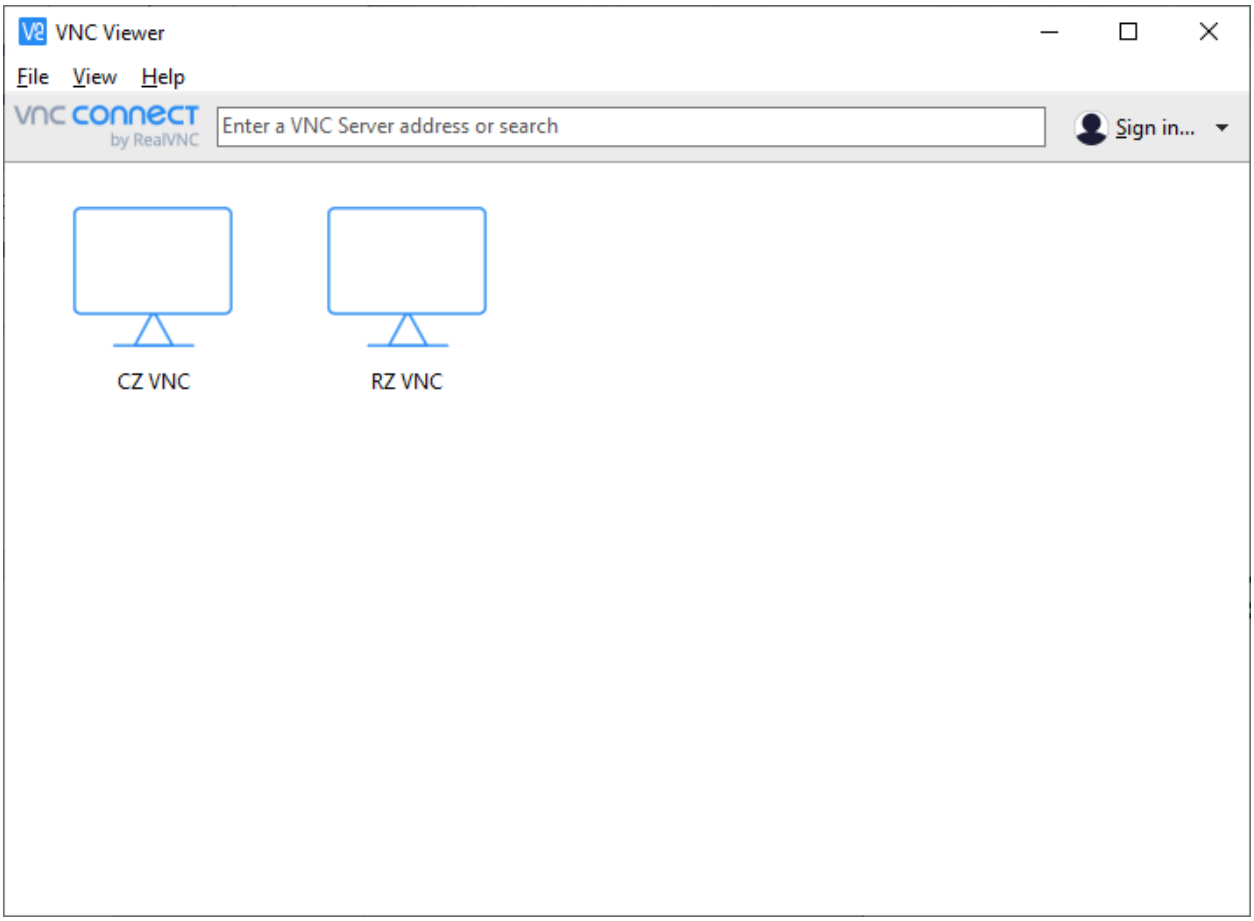


Fig. 6.123: The VNC Viewer with two profiles

Now we are ready to login to one of the systems.

1. Double click on the *CZ VNC* icon
2. This will bring up a login window.
3. Enter your CZ username and password.

This will bring up a Linux desktop. The resolution of the desktop will probably be low if you have never used the VNC server before. This is so that it isn't too large if you are on a laptop.

To change the resolution of the display dynamically, bring up a terminal and use the xrandr command.

1. Select *Applications->Terminal*
2. Enter "xrandr" in the terminal to get a list of supported resolutions.
3. Enter "xrandr -s 1280x720" in the terminal to change the resolution to 1280 by 720.
4. Change the resolution back to something more appropriate to your screen.

Recommended resolutions are:

- Dell laptop running Windows: 1280 x 720
- A high-resolution external monitor: 1920 x 1200

- A Mac laptop: 1680 x 1050 (Retina Display) or 1440 x 900

When using VisIt you should ssh to another CZ machine so that you don't overload the VNC server. You should use version 3.1.1 of VisIt for the best performance on a VNC client. Versions prior to 3.0.0 will not work properly with VNC.

1. Enter "ssh quartz".
2. Enter "visit -v 3.1.1".
3. Run VisIt as normal.

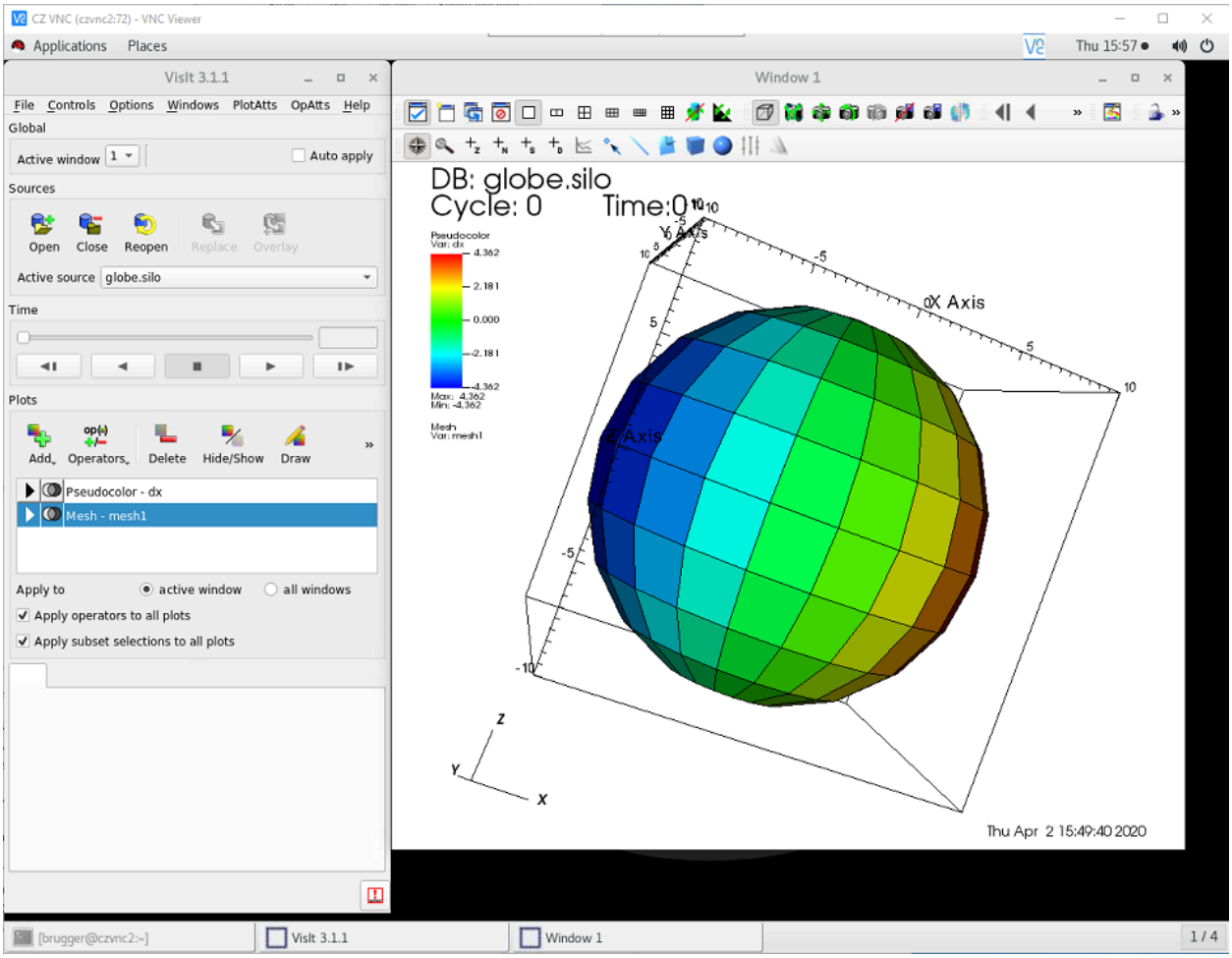


Fig. 6.124: VisIt running on the VNC Viewer

Troubleshooting VNC issues

Sometimes you can't see anything because the default screen is too large. There are two solutions to this issue, one is to reduce the resolution of the desktop and the other is to have the window scale automatically. To reduce the desktop resolution:

1. Use the scroll bars to navigate to upper left hand corner and bring up a terminal.
2. From the terminal use the "xrandr" command to change the resolution as described *here*.

To have the desktop scale automatically:

1. Go to the slide-out menu at the top center and rest your mouse below the title bar.

Fig. 6.125: The slide-out menu

2. Click the *Scale automatically* icon.

Fig. 6.126: Clicking on the Scale automatically icon

3. The window should now resize and you can use the VNC client.

Sometimes the response gets really slow when the VNC server is under heavy load. One solution is to reduce the picture quality.

1. Go to the slide-out menu at the top center and rest your mouse below the title bar.
2. Click the *Properties* icon.

Fig. 6.127: Clicking on the Properties icon

3. Click on the *Options* tab.
4. Set the *Picture quality* to *Low*.
5. Click *Ok*.

6.8.3 Using client/server

When VisIt is running in a client/server mode, a portion of VisIt is running on your local system and a portion is running on a remote compute resource such as a supercomputing center. This will always give better performance

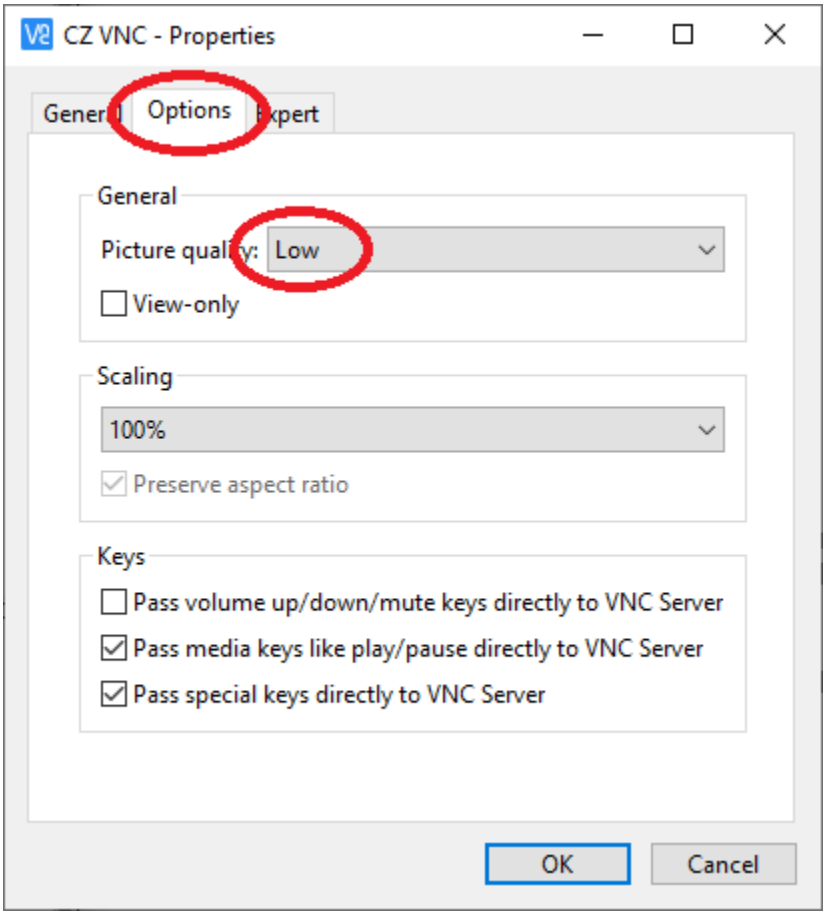


Fig. 6.128: Setting the Picture quality to Low

than running on a remote system using X display forwarding, since interactions with the graphical user interface will be faster and VisIt will be able to leverage the graphics processing unit on your local system. The portion running on your local system is referred to as the client and the portion running on the remote compute resource is referred to as the server. The client is responsible for the graphical user interface and the rendering window, while the server is responsible for accessing the data on the remote system, processing it, and sending back geometry to be rendered or images to be displayed.

When running in client/server mode, VisIt makes use of host profiles that provide information on how to run VisIt on the remote system, such as where VisIt is installed and information about the batch system. VisIt comes with host profiles for many different supercomputing systems. This portion of the tutorial will use the Livermore Computing Center at LLNL.

Installing the host profiles for your computer center

The first thing you will need to do is make sure you have the host profiles installed for the remote system. You can check this by bringing up the *Host profiles* window.

1. Select *Options->Host profiles...* to bring up the *Host profiles* window.
2. If the list of *Hosts* is blank or doesn't contain the host of interest, you will need proceed with steps 4 - 10.
3. Click the *Dismiss* button.
4. Select *Options->Host profiles and configuration setup...* to bring up the *Setup Host Profiles and Configuration* window.

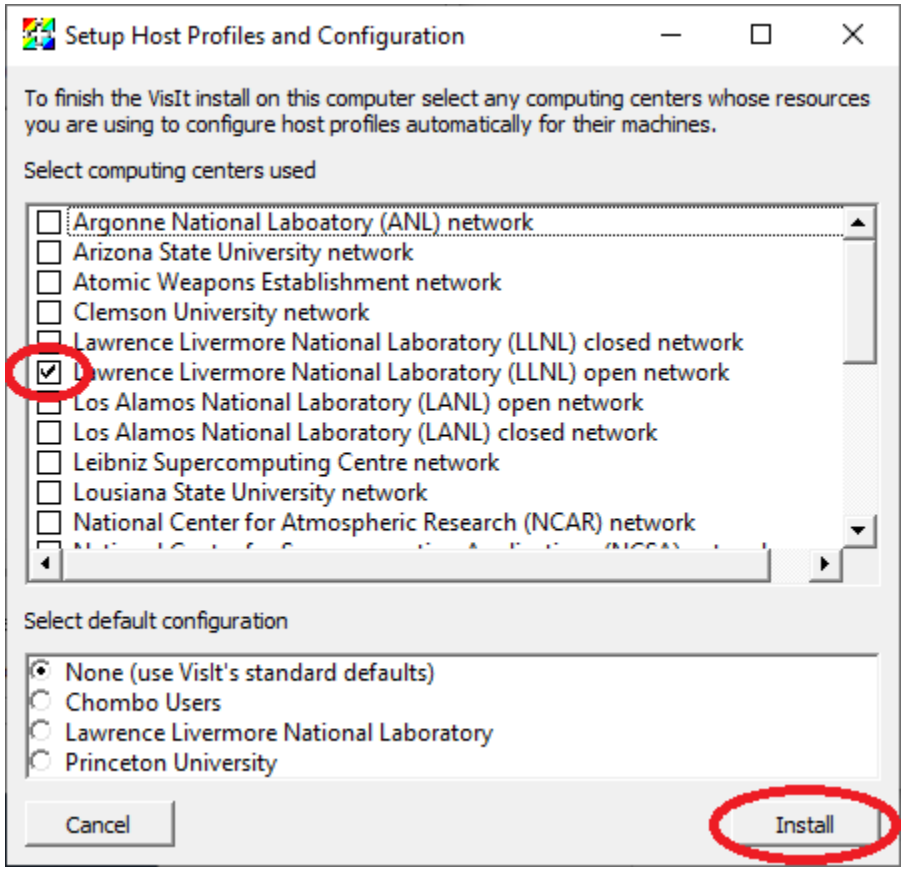


Fig. 6.129: The *Setup Host Profiles and Configuration*

5. Click on the *Lawrence Livermore National Laboratory (LLNL) open network*.
6. Click *Install*.
7. Restart VisIt.
8. Select *Options->Host profiles...* to bring up the *Host profiles* window.
9. You should now see the host profiles for LLNL.
10. Click the *Dismiss* button.

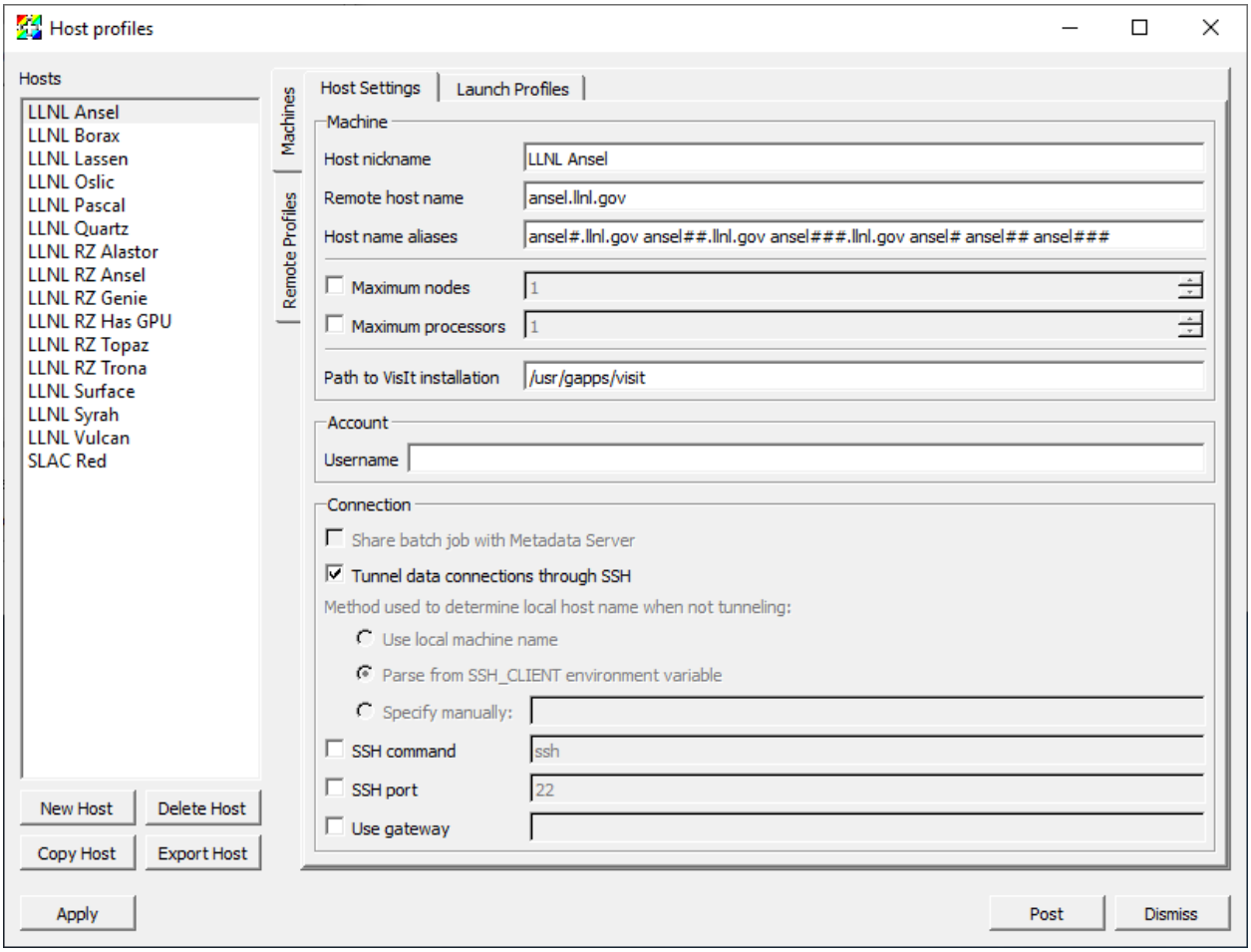


Fig. 6.130: The *Host profiles* window with the host profiles for LLNL

Connecting to a remote system

You are now ready to connect to the remote system.

1. Click on the *Open* icon in the *Sources* section of the main window to bring up the *File open* window.
2. Click on the *Host* pulldown menu and select *LLNL Quartz*.
3. This will bring up a window to enter your password.
4. If your username is different on the remote system from the one on your local system you will need to click on *Change username* and change your username.

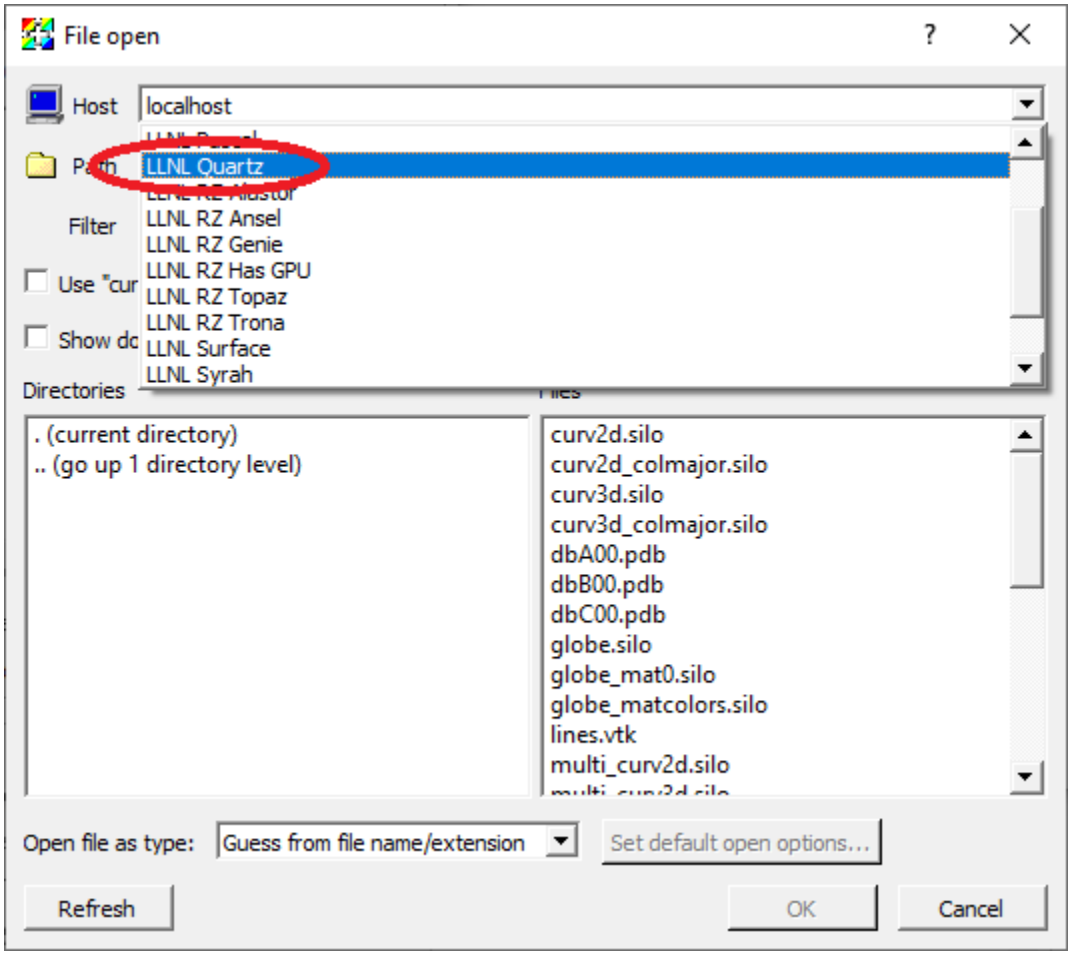


Fig. 6.131: The *File open* window

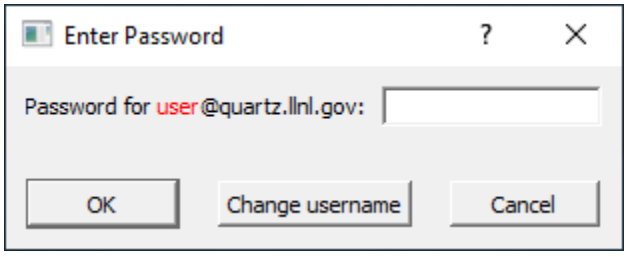


Fig. 6.132: The *Enter Password* window

5. The *File open* will now open to your home directory on the remote system.

You are now ready to open files, create plots and do everything you are used to doing with VisIt.

File locations when running client/server

When running in client/server mode some files are stored on the local system and some are stored on the remote system. Most files are stored or saved on the local system. Some examples include:

- Images
- Movies
- Host profiles
- Settings
- Color tables

The main exception is when exporting data. Those results are saved on the remote system. This is usually what you want since you will most likely want to open it on the remote system for further processing.

The window that exports databases is unable to browse the remote file system, so you will need to carefully type in the path to the directory to save it in.

6.8.4 Using batch systems interactively

When VisIt normally uses the batch system, it submits the parallel compute engine to the batch system and then the compute engine runs until it exits. Sometimes VisIt exits because of a crash. Once that happens you will lose the rest of the batch allocation and you will need to submit a batch job, which may not always happen immediately. One way around this is to get a batch job and then run all of VisIt in batch system using X display forwarding (ideally from within a VNC client).

One such mechanism is mxterm, a utility available at LLNL. It submits a batch job and pops up an xterm. From the xterm, the user can start VisIt as many times as they want until the batch job time limit expires. There may be similar mechanisms available at other supercomputing centers. If not, it would be fairly straightforward to create such a script for the batch system at your supercomputing center.

Using mxterm

The basic mxterm command is:

```
mxterm <nnodes> <ntasks> <nminutes> <-q queue_name>
```

An example that gets 1 node with 36 tasks for 30 minutes in the pdebug queue.

```
mxterm 1 36 30 -q pdebug
```

When the job starts an xterm window will appear on your screen.

When using an mxterm, you will need to use the mxterm profile when starting your compute engine.

6.9 Making Movies

Making movies with VisIt runs the gamut from creating a simple movie that shows the time evolution of a simulation to movies that contain multiple image sequences, where the image sequences may contain:

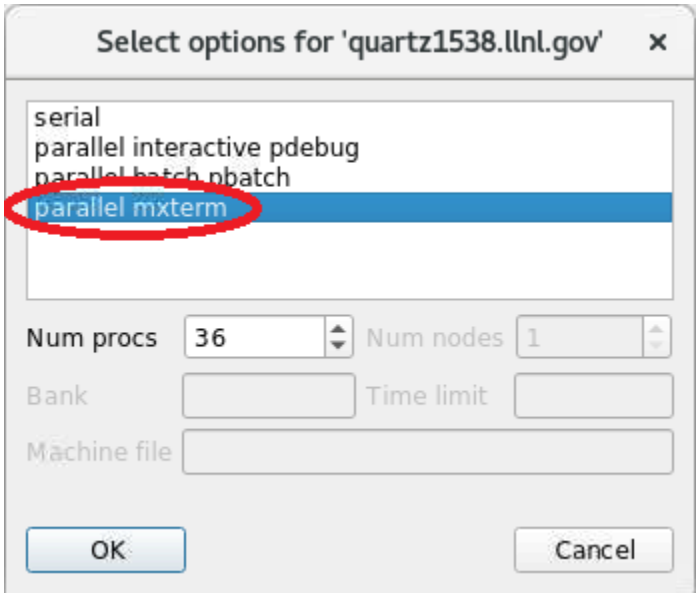


Fig. 6.133: Selecting the mxterm host profile

1. Titles
2. Fade-ins
3. Image sequences that involve moving the camera around or through the data.
4. Image sequences where each image contains multiple components such as a 3d view of the data and a curve showing the time evolution of a value.
5. Image sequences where operator attributes are modified such as animating a slice plane moving through a data set.

Simple movies can be made with the Save movie wizard and more complex movies are made using Python scripts. This tutorial will focus on creating simple movies with the Save movie wizard and using Python scripts.

6.9.1 Creating a movie of a simulation evolving over time

The simplest type of movie to create is a movie of a simulation evolving over time. There are several steps to making such a movie.

1. Create a good image for a single time state. This is typically the first or last time state.
2. Animate the movie to make sure the entire movie looks good and change things if they don't.
3. Create the images and encode the movie.

6.9.2 Creating a good image from a single time state

This tutorial uses the **dbreak3d** dataset – available at http://www.visitusers.org/index.php?title=Tutorial_Data

The dataset simulates the evolution of water and air in a water tank after an interface holding a column of water is instantaneously removed.

Display the tank

1. Open the file *dbreak3d_boundaries.silo*.
2. Create a Subset plot of *domains*.
3. Click *Draw*.
4. The Subset plot shows the different faces that comprise the water tank. We do not want to view all of the boundaries because they will block the fluid data, so next we turn off a few of the boundary faces that are identified as *domains* in the data file.
5. We would like to turn off the magenta and yellow boundaries. From the Subset plot legend we can see that those are *domain5* and *domain6*.
6. Bring up the *Subset* window by clicking on the Ven Diagram next to the Subset plot in the plot list.
7. Click on *domains* to expand the list of domains and deselect *domain5* and *domain6*.
8. Click *Apply*.
9. Now let's make all the faces the same color.
10. Double click on the Subset plot in the plot list to bring up the *Subset plot attributes* window.
11. Select *Single* and choose the light pastel green color.
12. Click *Apply* and *Dismiss*.

Display the water

The water information is stored in the file *dbreak3d_fluid.visit* and contains information about the time evolution of the water. The boundary of the water can be created using the *alpha1* variable. It represents the volume fraction of water in a cell. A value of 0.0 means that the cell doesn't contain any water. A value of 1.0 means that the cell is completely filled with water. The region containing the water can be extracted by using the Isovolume operator, selecting the region where the volume fraction is between 0.5 and 1.0. Let's get started.

1. Open the file *dbreak3d_fluid.visit*.
2. Create a Pseudocolor plot of *alpha1*.
3. Double click on the Pseudocolor plot in the plot list to bring up its attributes.
4. Change the *Color table* to *PuBu*.
5. Change the *Opacity* to *Constant*.
6. Set the *Opacity* slider value to *65%*.
7. Click *Apply* and *Dismiss*.
8. Deselect *Apply operators to all plots* on the main control window below the plot list. This will allow you to apply the Isovolume operator to just the Pseudocolor plot.
9. Go to *Operators->Selection->Isovolume* to add the Isovolume operator to the Pseudocolor plot.
10. Click on the triangle next to the Pseudocolor plot to expand the Pseudocolor plot.
11. Double click on the Isovolume operator to bring up its attributes.
12. Set the *Lower bound* to *0.5*.
13. Select *alpha1* as the *Variable* option.
14. Click *Apply* and *Dismiss*.

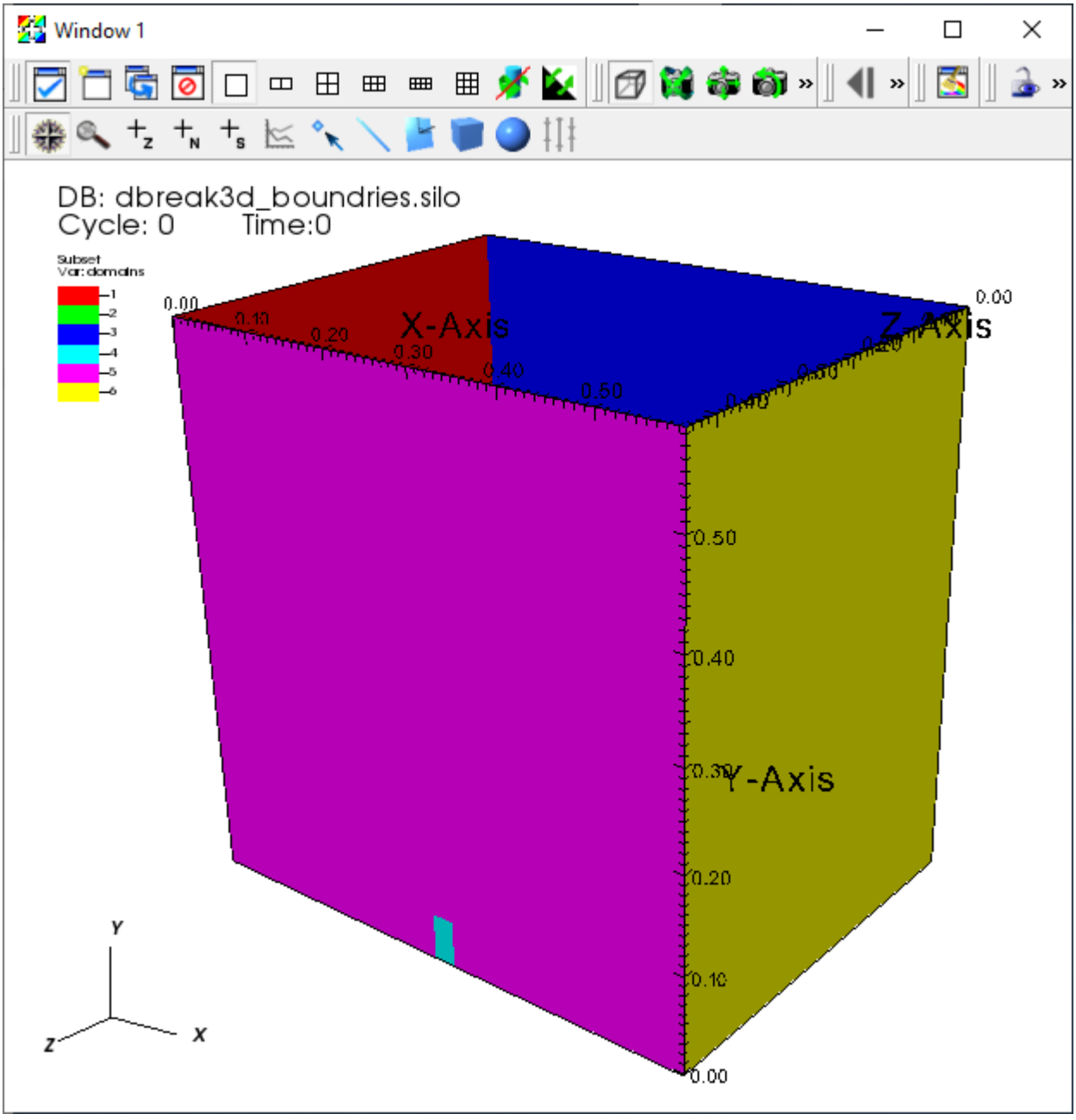


Fig. 6.134: The default *Subset* plot of the boundaries.

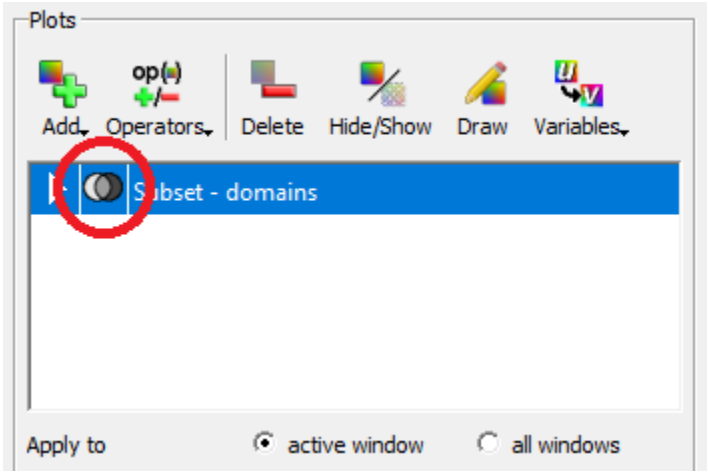


Fig. 6.135: Bringing up the *Subset* window from the plot list.

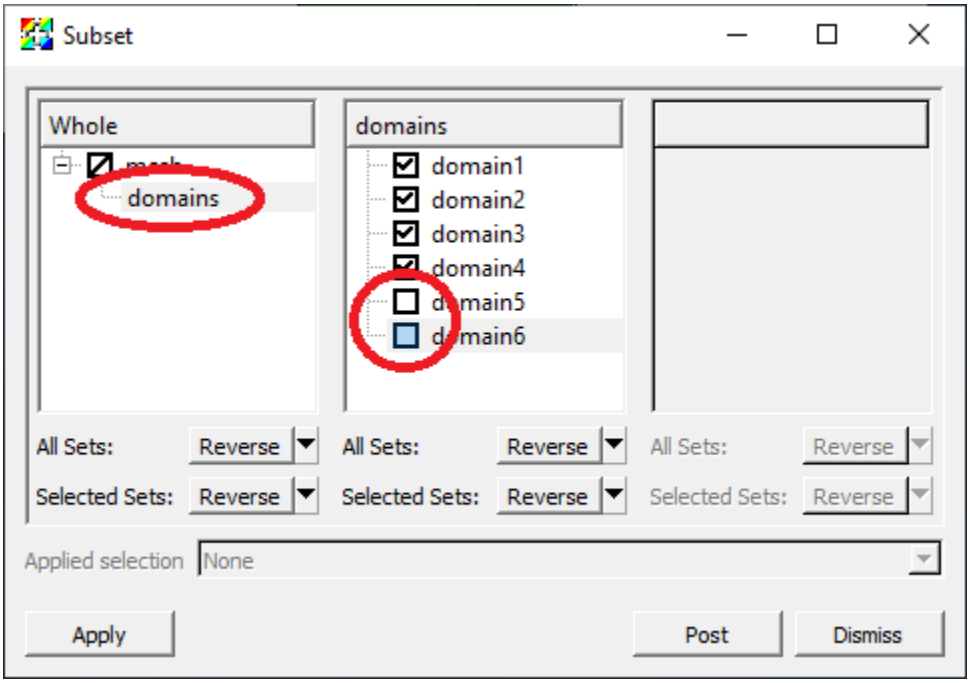


Fig. 6.136: Removing boundaries with the *Subset* window.

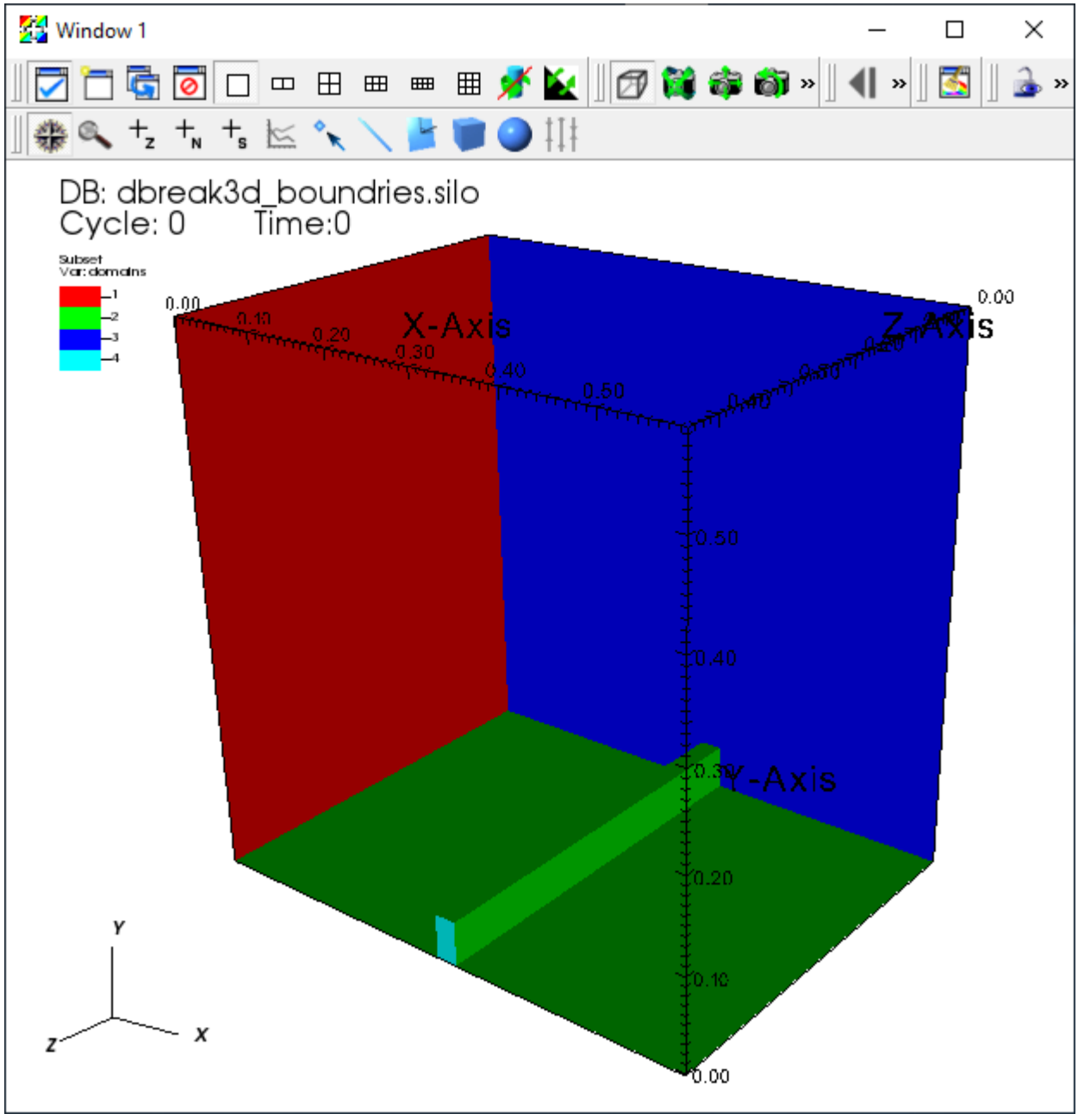


Fig. 6.137: The *Subset* plot with the boundaries removed.

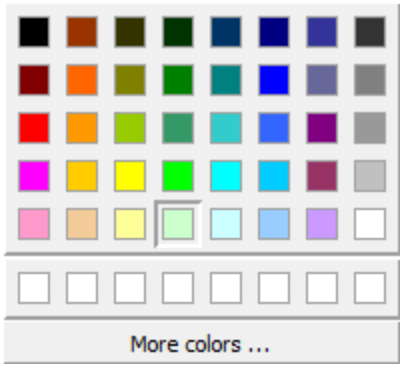


Fig. 6.138: Changing the colors of the *Subset* plot.

15. Click *Draw*.

Improve the annotations

To make the movie look more polished, we will change the window annotations, the background color, the lighting and add a time slider.

1. Go to *Controls->Annotation* to bring up the *Annotation* window.
2. Select the *General* tab.
3. Click *No annotations*.
4. Click *Apply*.
5. Select the *3D* tab.
6. Select *Show bounding box*.
7. Click *Apply*.
8. Select the *Colors* tab.
9. Set the *Foreground color* to be the same color as our tank boundaries plot.
10. Set the *Background style* to *Gradient*.
11. Set the *Gradient style* to *Radial*.
12. Set *Gradient color 1* to be light gray.
13. Set *Gradient color 2* to be very dark gray.
14. Click *Apply*.
15. Select the *Objects* tab.
16. Create a new *Time slider*.
17. Click *Ok* when it prompts you for a name.
18. Set the *Width* to *40%*.
19. Set the *Height* to *7%*.
20. Set the *Start color* to light blue.
21. Set the *End color* to a darker blue.

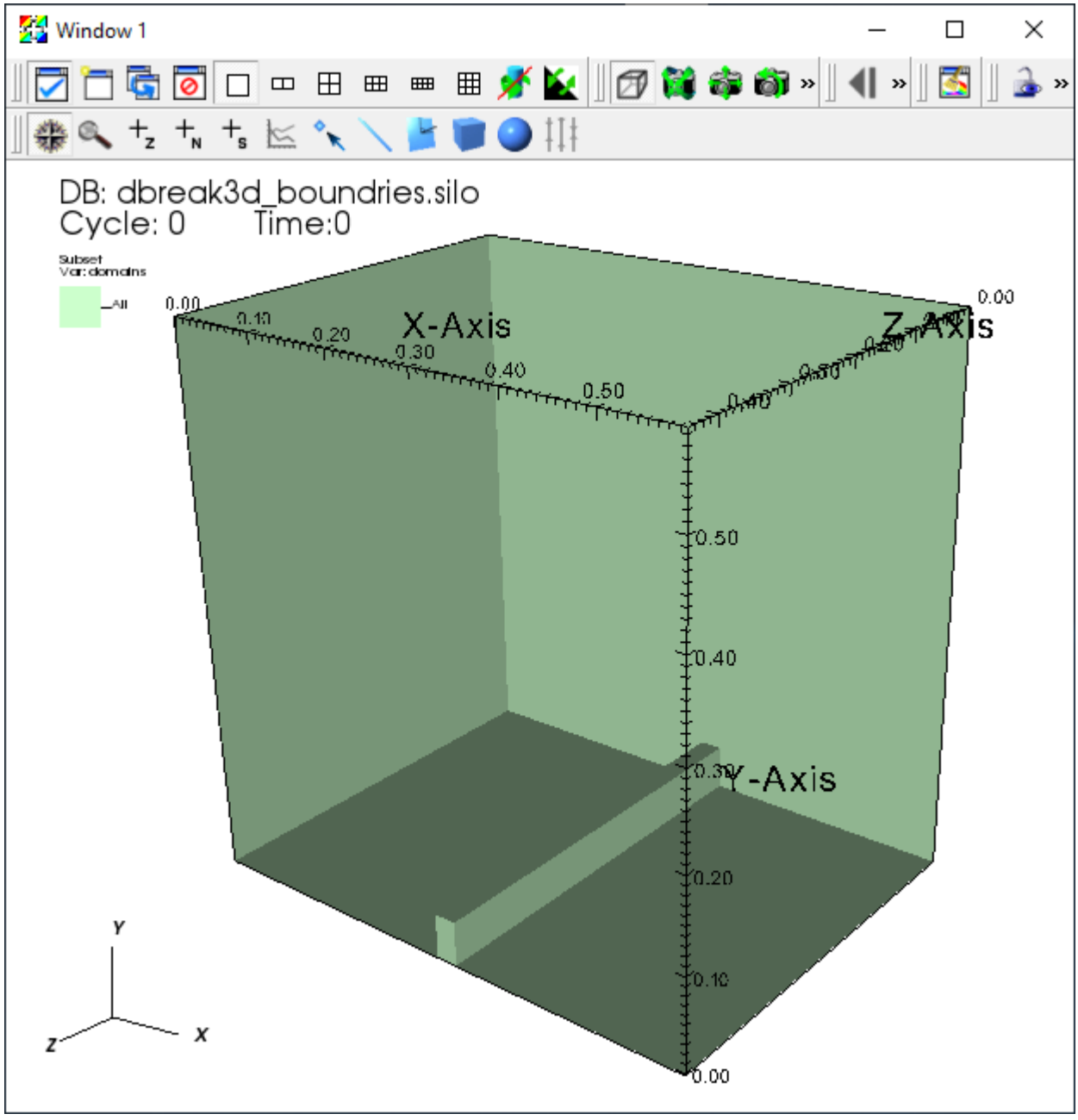


Fig. 6.139: The *Subset* plot boundaries in a single color.

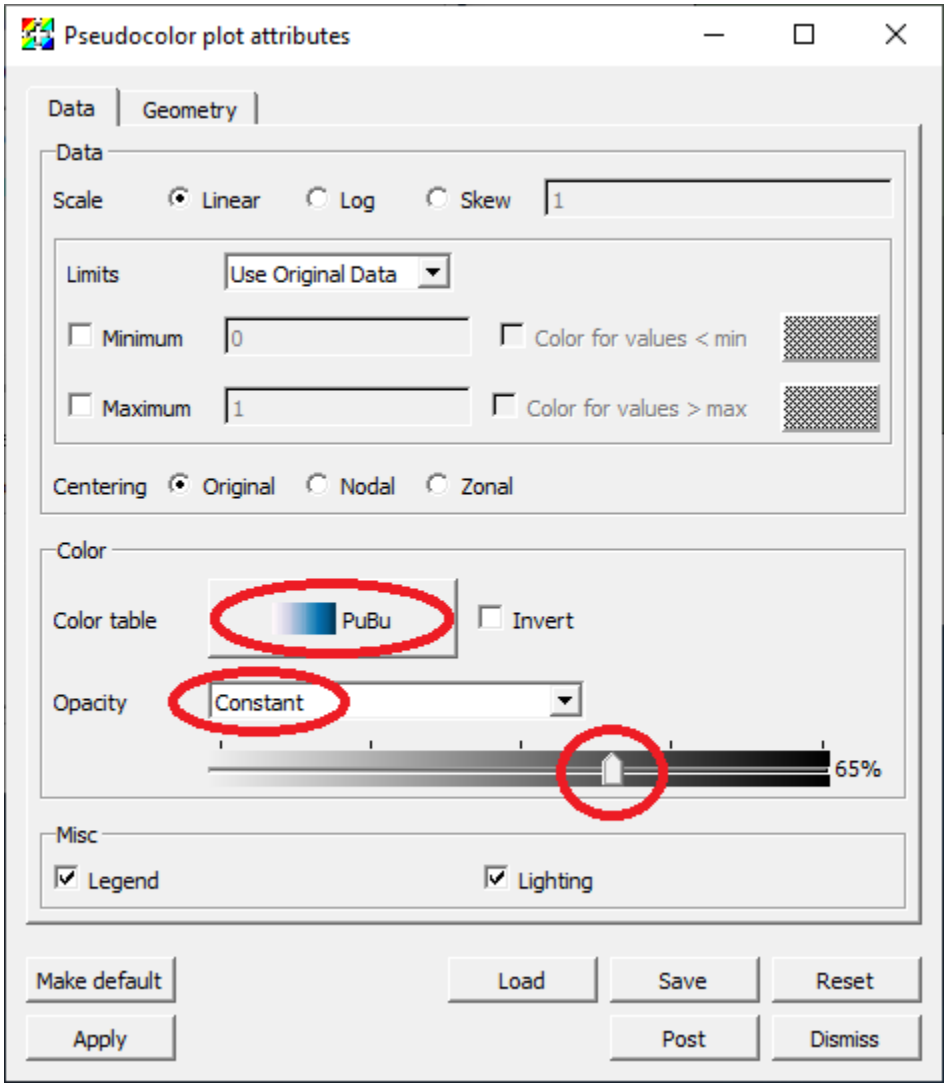


Fig. 6.140: Setting the *Pseudocolor* attributes for the water.

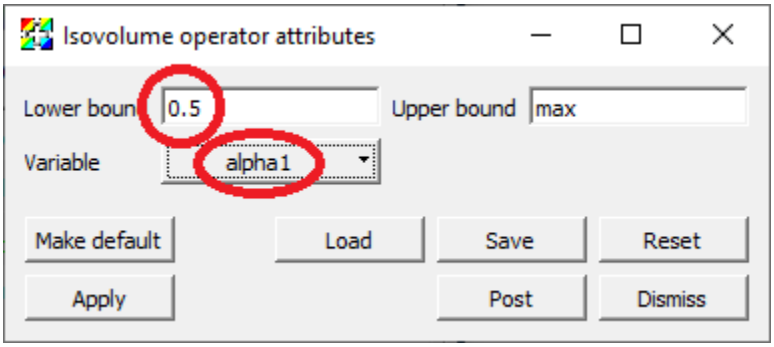


Fig. 6.141: Using the *Isovolume* operator to select the water.

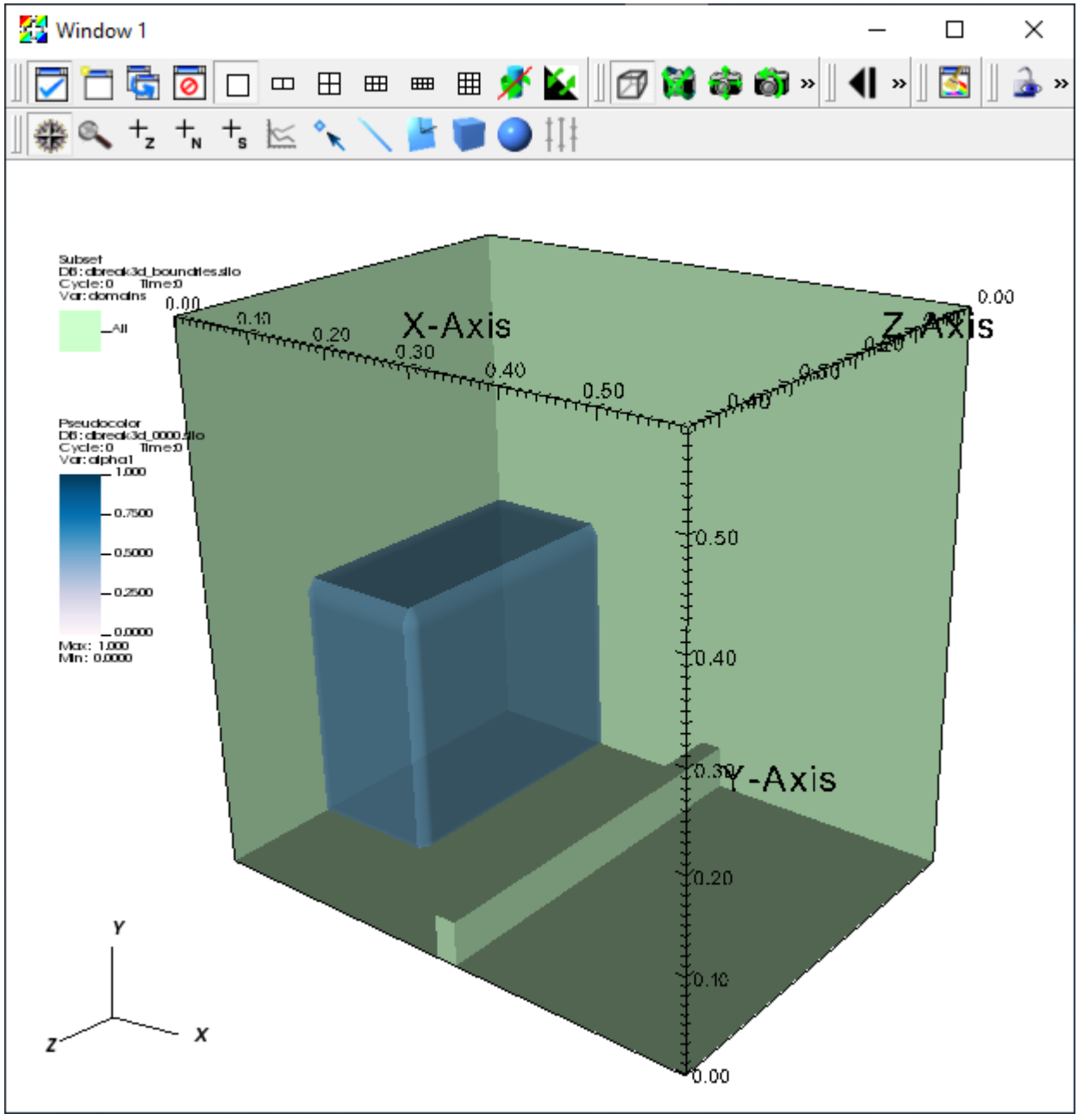


Fig. 6.142: The boundaries and the water.

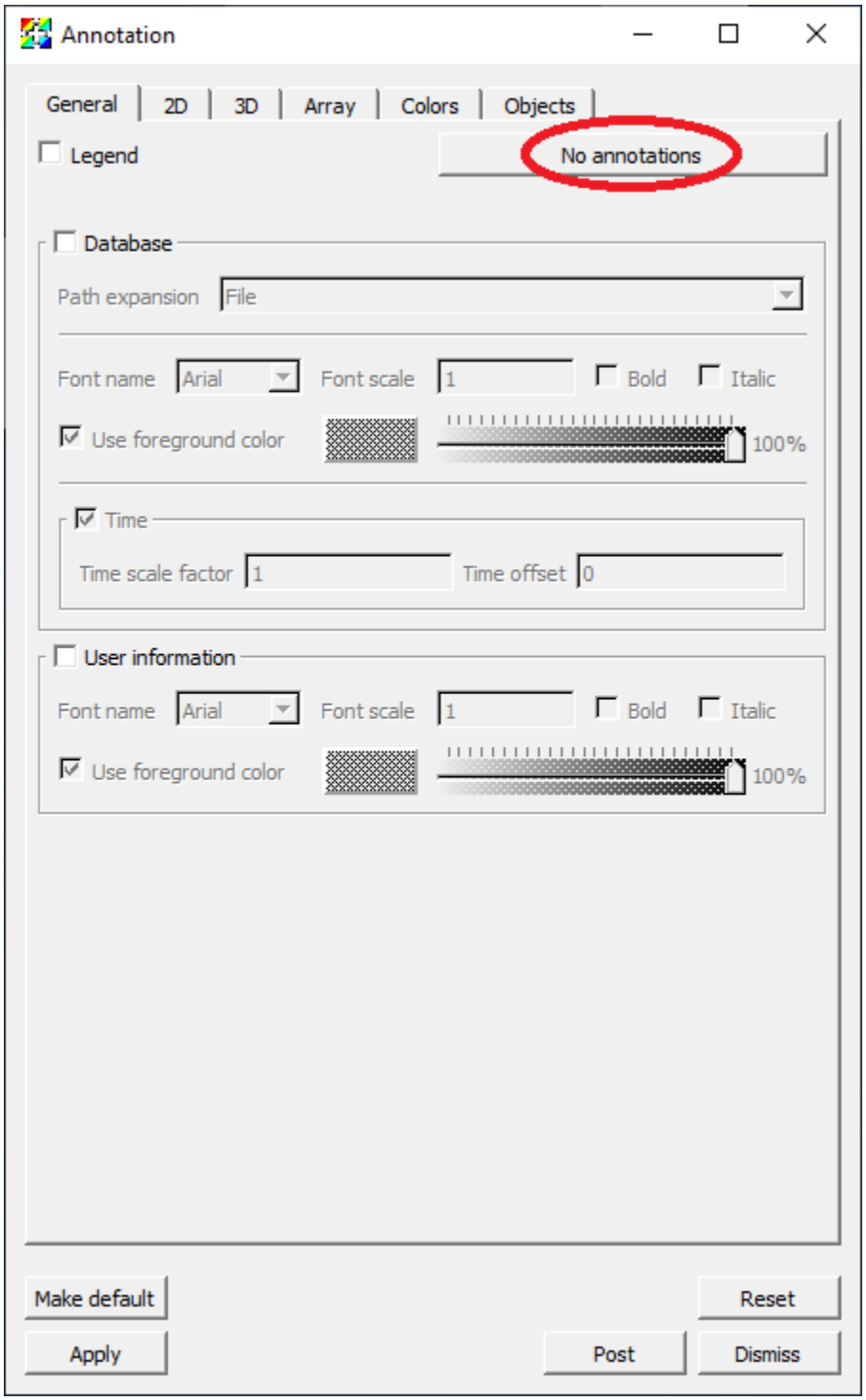


Fig. 6.143: Turning off all the annotations.

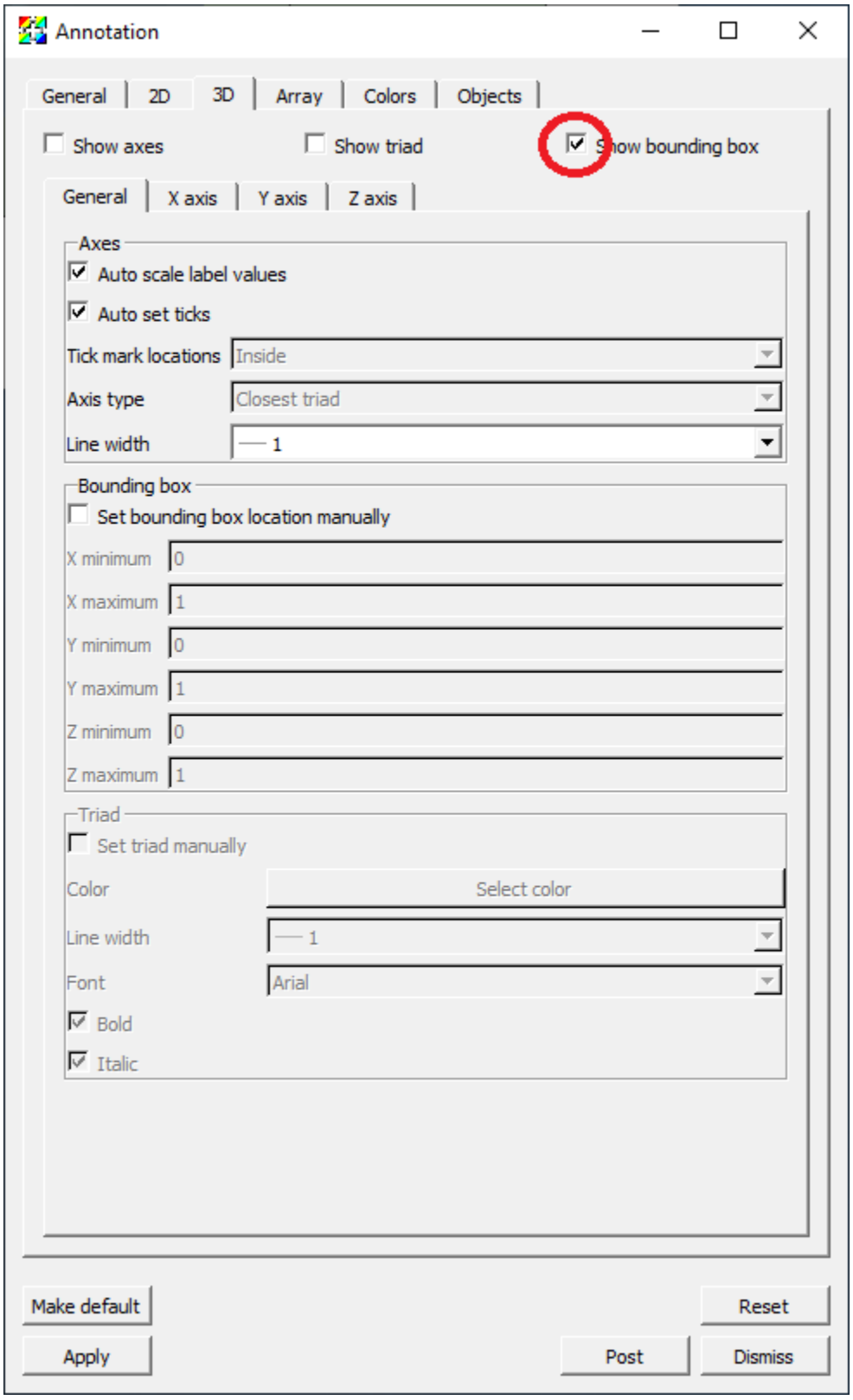


Fig. 6.144: Adding the bounding box.

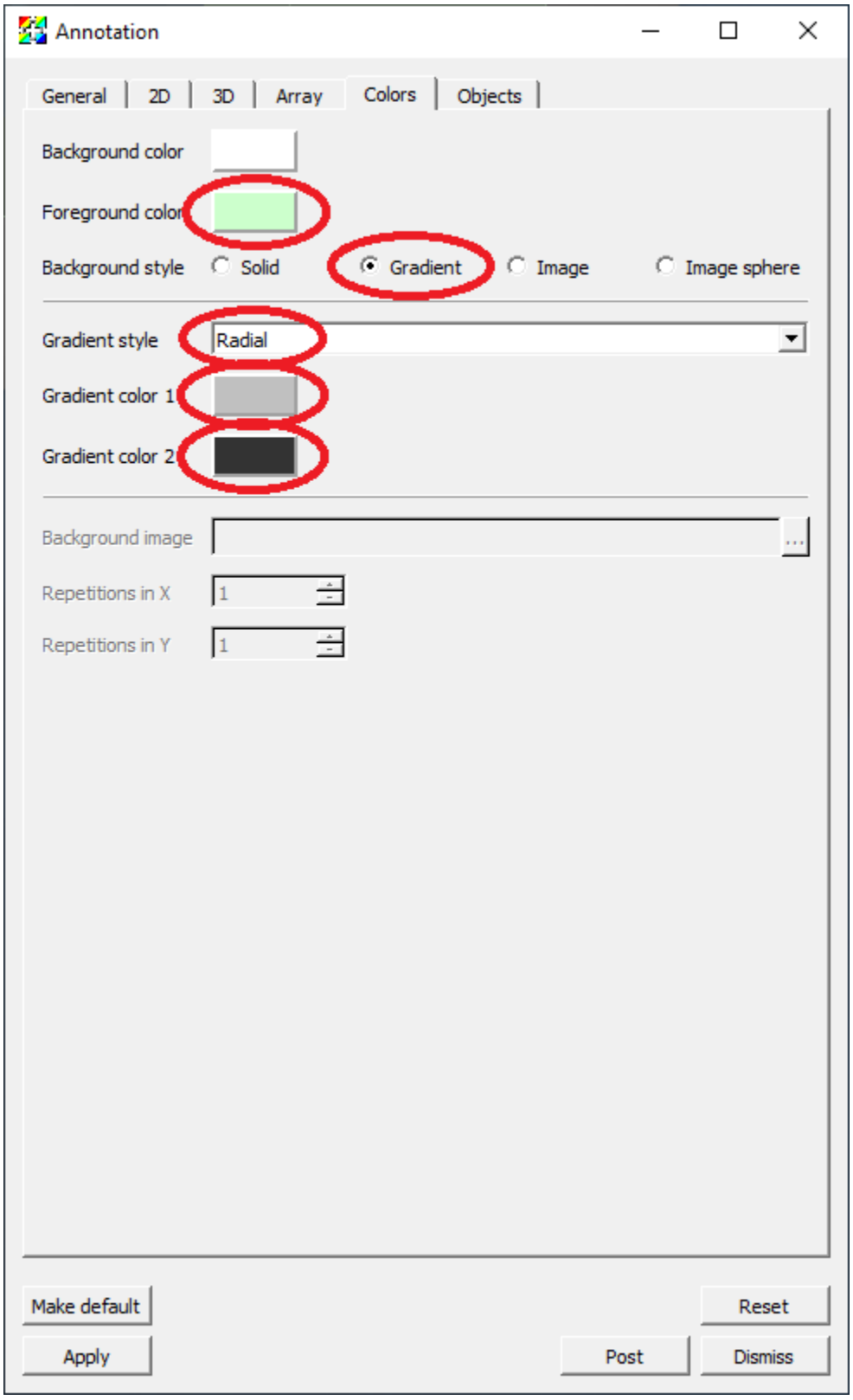


Fig. 6.145: Setting the foreground and background colors.

22. Deselect *Use foreground color*.
23. Set the *Text color* to white.
24. Click *Draw*.
25. Go to *Controls->Lighting* to bring up the *Lighting* window.
26. Move the light vector up and to the right.
27. Click *Apply*.
28. Move the time slider in the main control window to a later time state where the water is splashing up.

6.9.3 Encoding the movie with the movie wizard

1. Go to *File->Save movie* to bring up the *Save movie wizard* window.
2. Select *New simple movie* and click *Next*.
3. Select *Specify movie size*.
4. Ensure the the *lock aspect* setting is selected. While you can encode movies with a different aspect ratio than the aspect ratio of the window on the screen, it is generally not a good idea. Objects are positioned based on a zero to one coordinate system where zero represents either the left edge or the bottom of the image and the heights and widths of objects are based on fraction of the height and width. This causes objects to change position and relative size as the aspect ratio is changed.
5. Change the *Width* to *600*. The *Height* will automatically change to maintain the aspect ratio.
6. Click the right arrow button to create an entry in the *Output* list with the format and resolution information specified on the right hand side of the window. It is possible to change the format and resolution information and click the right arrow button to create additional entries in the *Output* list to encode multiple movies with different settings at once. We are just going to create a single mpeg movie.
7. Click *Next*.
8. It is possible to specify the range of time states to use for the movie, as well as specify a stride if you have too many time states saved. The wizard will automatically set the range of time states. We will use all the time states and a stride of one, so we can use the default values.
9. Click *Next*.
10. You can specify the directory and file name for the movie. We will use the current directory and name the movie *dbreak3d*.
11. Click *Next*.
12. You can have VisIt send you an e-mail when it has finished creating the movie. Since we will wait for the movie to complete, we don't need an e-mail message to be sent when the movie has been finished and can use the default values.
13. Click *Next*.
14. You can have VisIt generate the movie now using the currently allocated processors, generate the movie with a new instance of VisIt, or generate the movie at some later time. We will generate the movie now so we can use the default value.
15. Click *Finish*.
16. This may take a few minutes depending on how fast your computer is. You may want to go get a cup of coffee.
17. A command window will appear while the movie is being generated. When the movie is finished the command window will disappear.

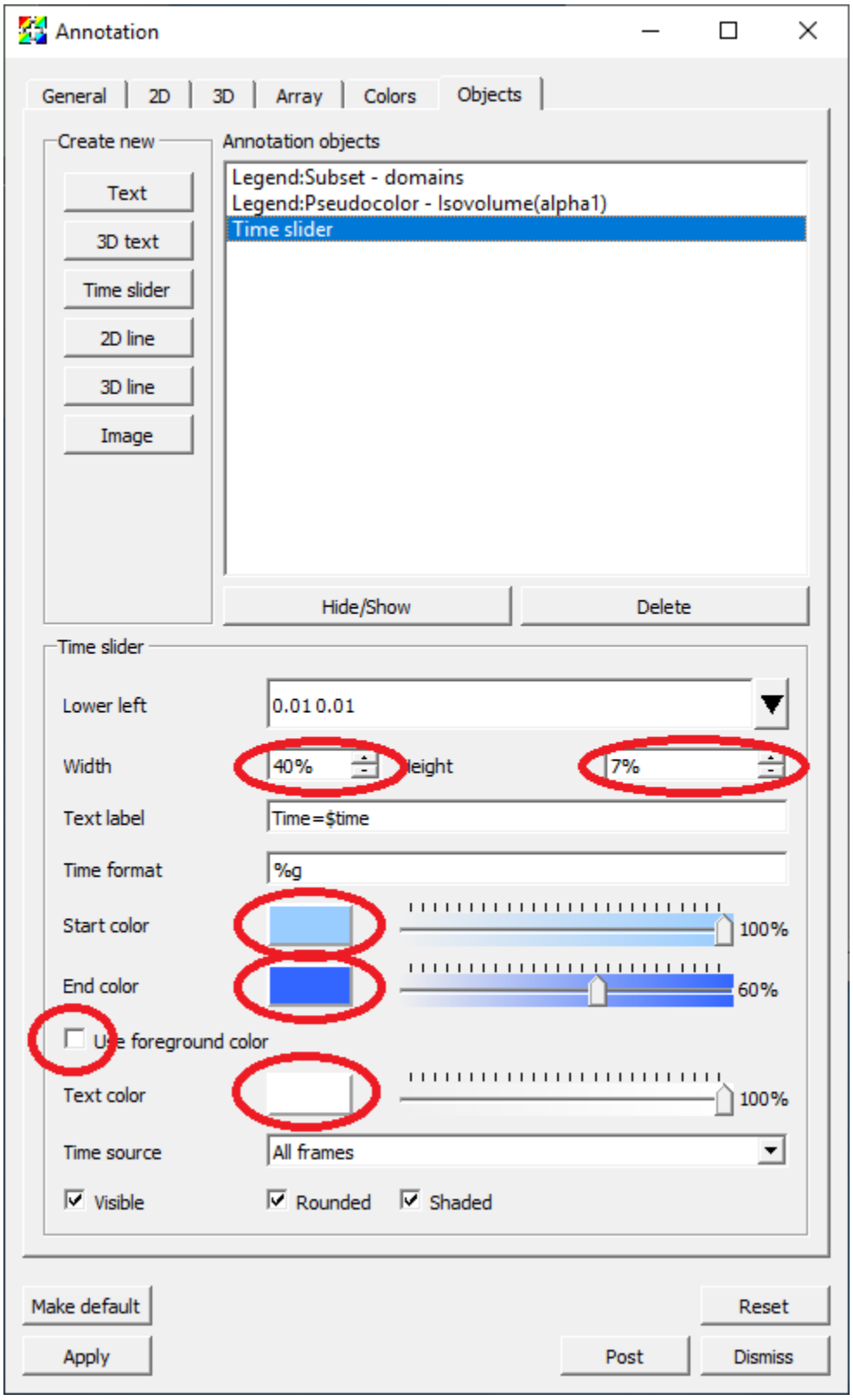


Fig. 6.146: Setting the time slider attributes.

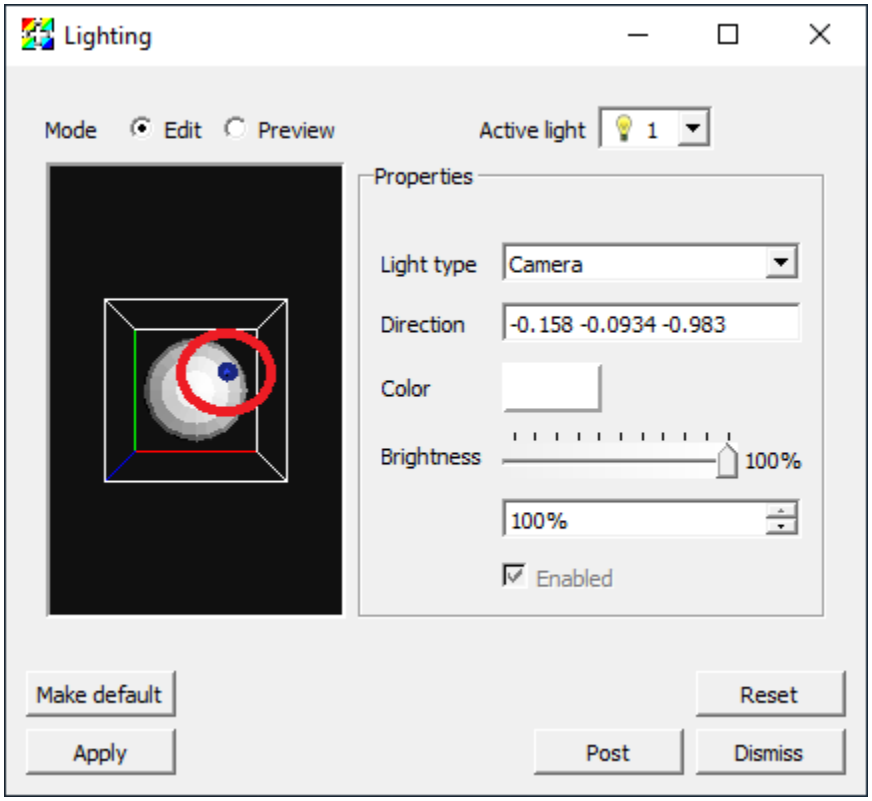


Fig. 6.147: Setting the light source position.

18. On Windows, you may get a window indicating that the VisIt Python Command Line interface has stopped working. If this happens, click on *Close program*. Your movie will have been generated properly.

Playing the movie

You can now play the movie with the native movie player on your system. On Linux you can use a player such as mplayer. On Mac macOS or Windows you can typically just double click on the icon for the movie. Note that on Windows you will need to play the movie with "Windows Media Player" and not "Movies & TV".

6.9.4 Encoding the movie with a Python script

This section of the tutorial is primarily aimed at Linux and Mac macOS systems. There are usually folder path issues on Windows that will prevent these Python code snippets from working as shown. In particular, the images from the image saving may get saved in a different folder from where the image encoding expects to find them. If you want to get this to work on Windows, you will need to specify absolute paths for the filenames. At the moment though, the image encoding won't work at all because there are issues with absolute paths and paths with spaces in them.

The first step in encoding a movie with a Python script is to create the images for encoding. The following snippet of Python code will loop over all the time states and save the images.

```
# Set the basic save options.
save_atts = SaveWindowAttributes()
save_atts.family = 0
save_atts.format = save_atts.PNG
```

(continues on next page)

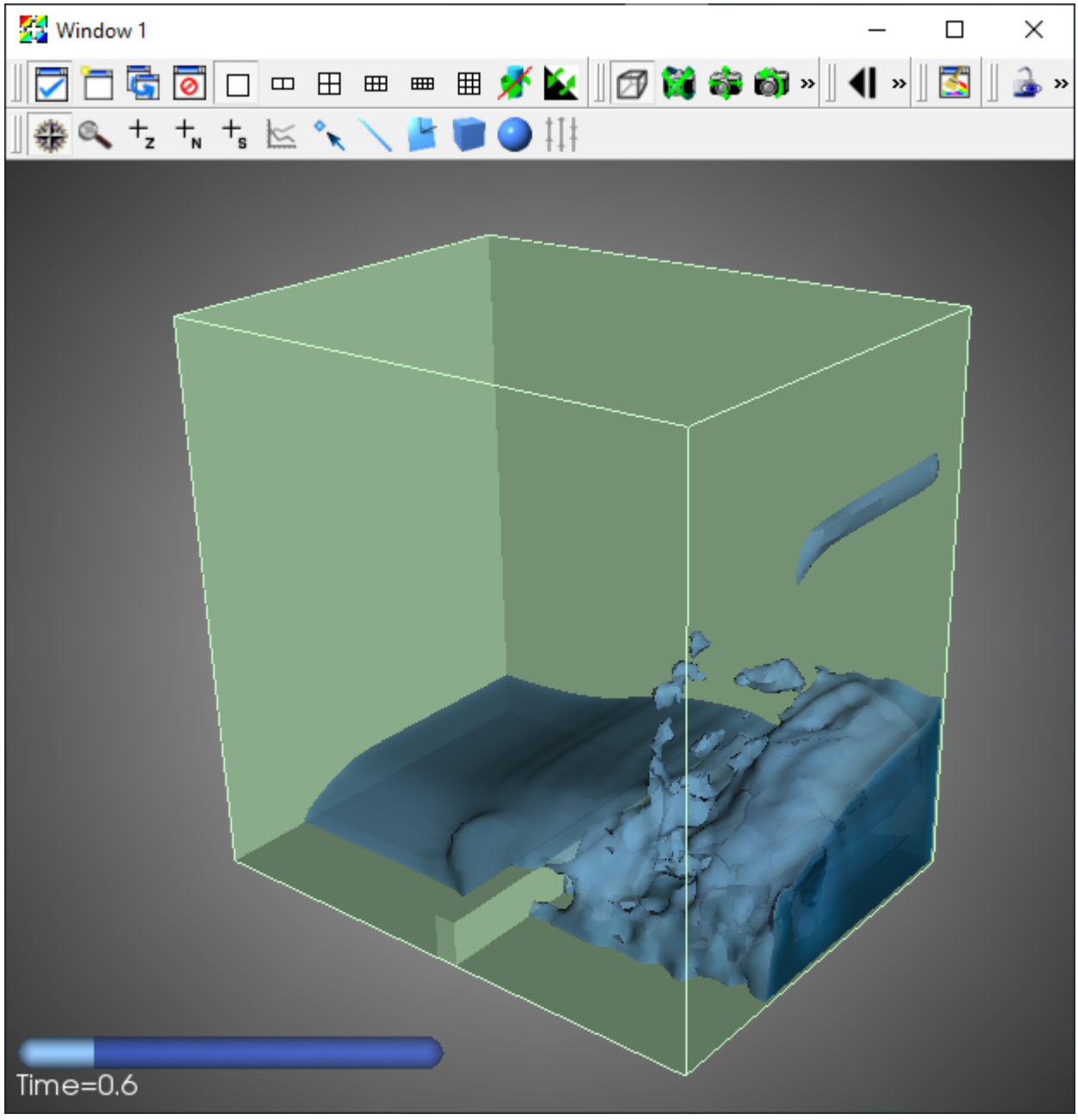


Fig. 6.148: The final result for an image in the movie.

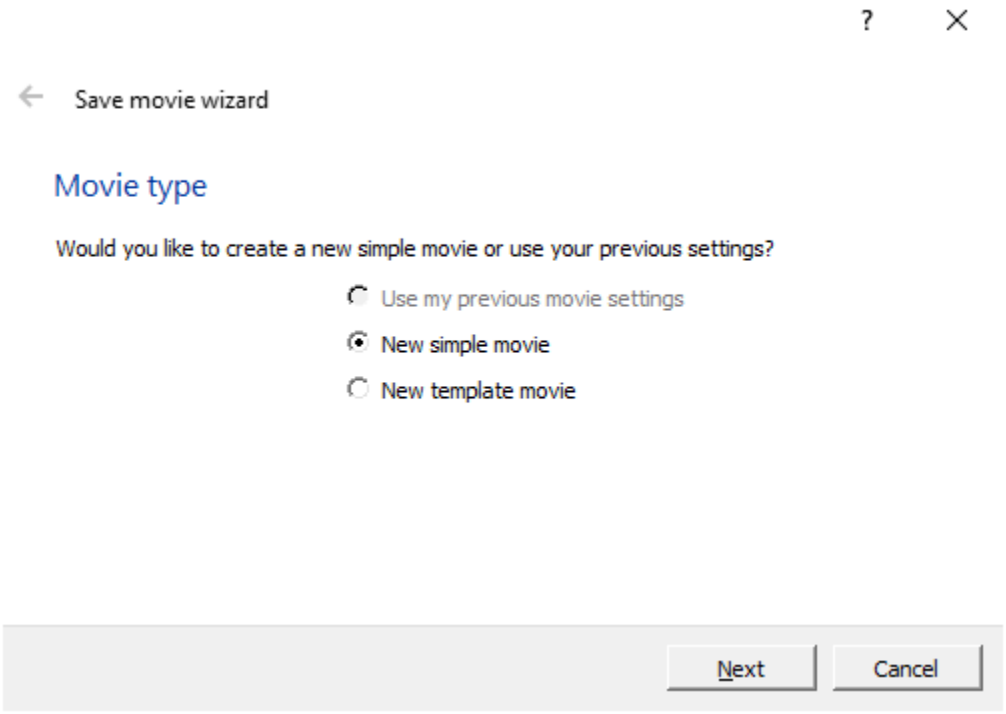


Fig. 6.149: Using the movie wizard to create a simple movie.

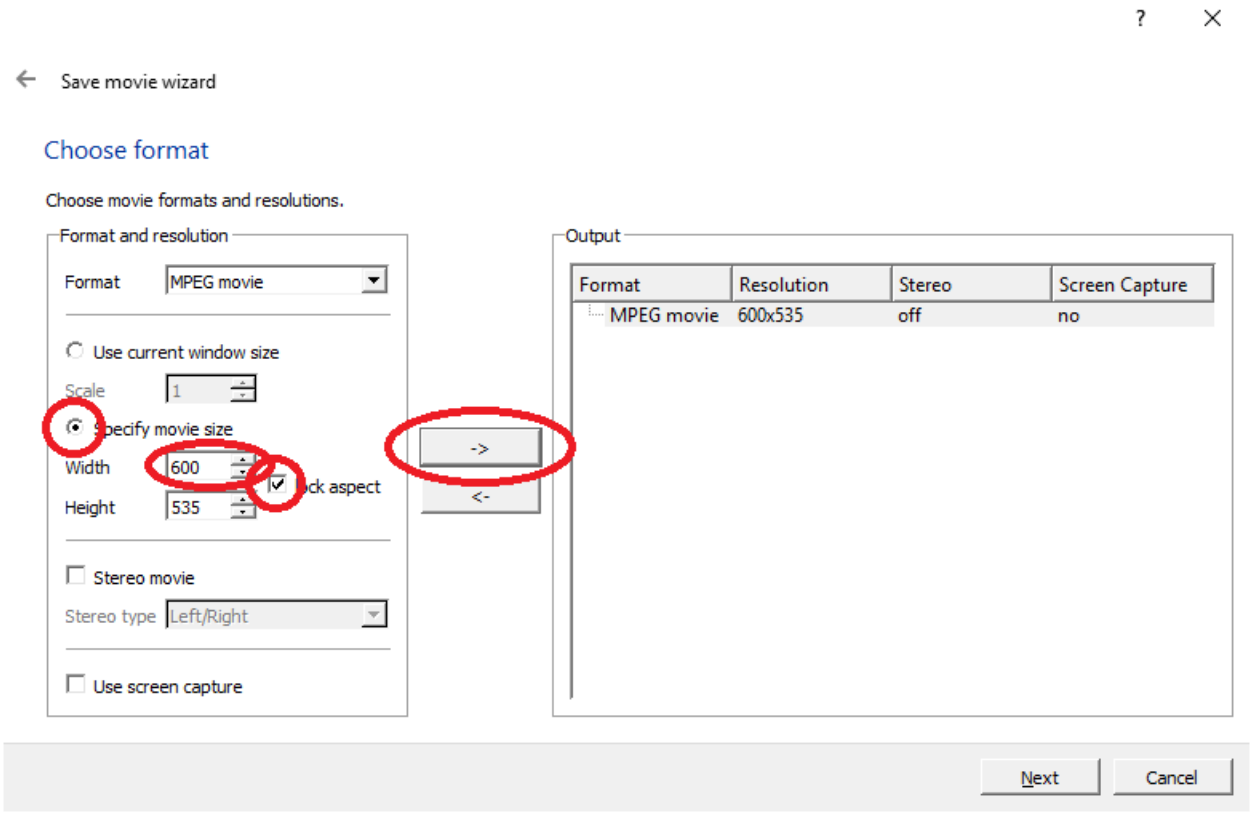


Fig. 6.150: Setting the movie format and resolution.

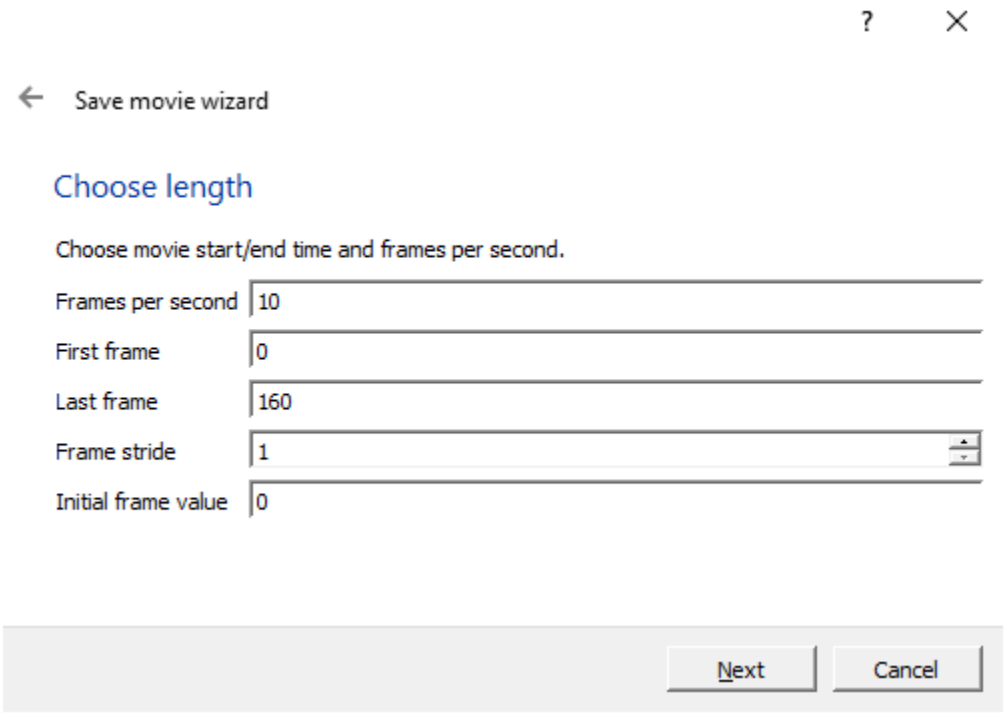


Fig. 6.151: Setting the length of the movie.

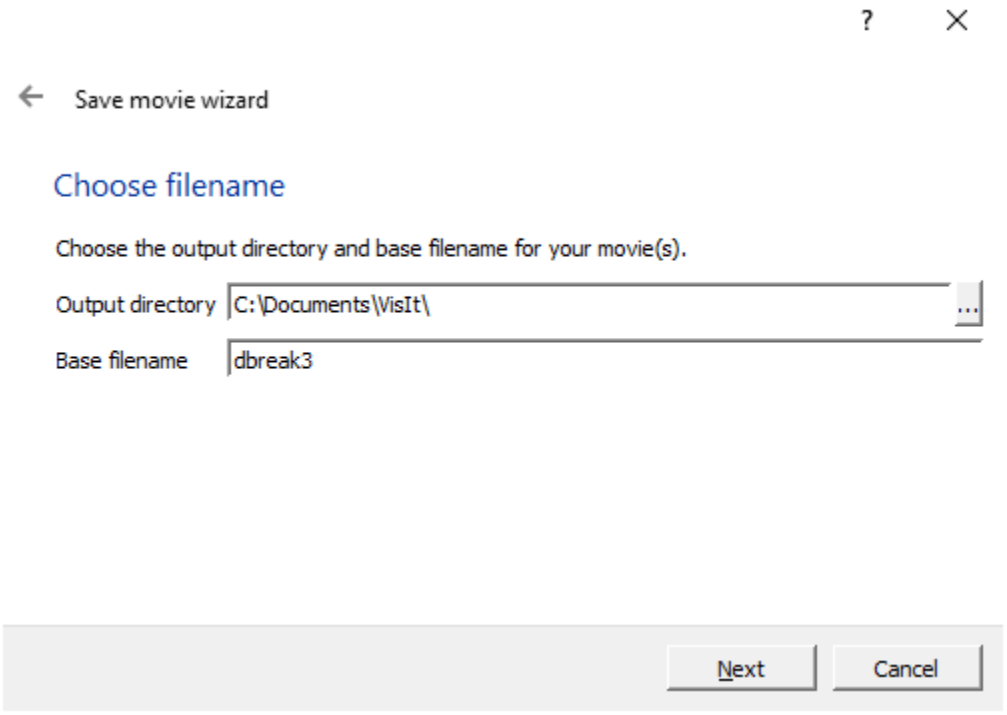


Fig. 6.152: Setting the name of the movie.

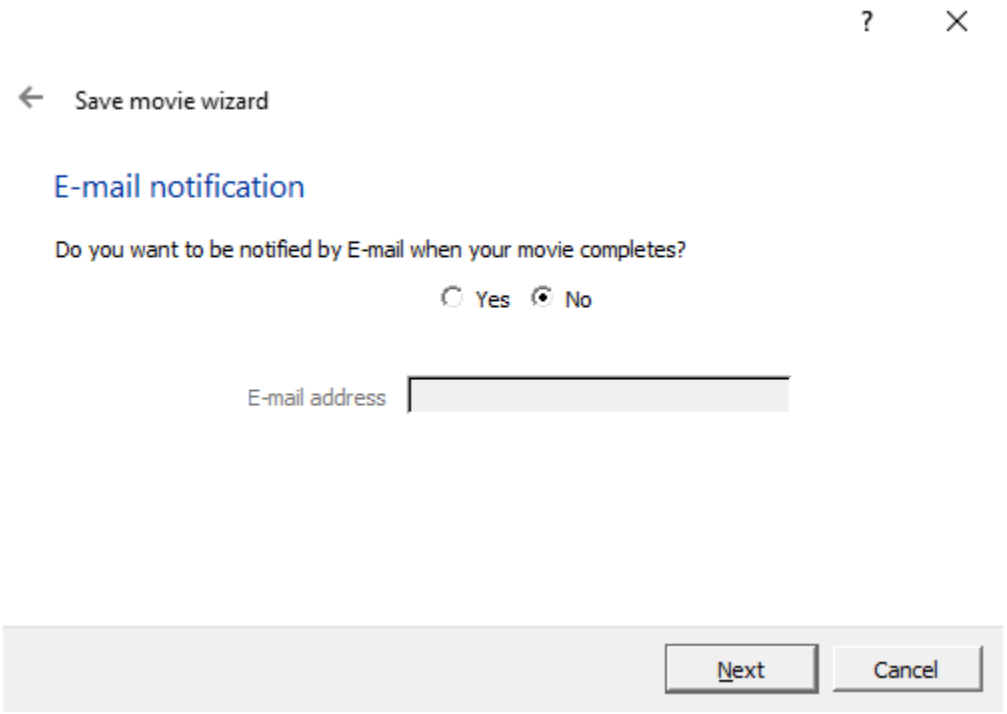


Fig. 6.153: Setting the e-mail notification for when the movie is complete.

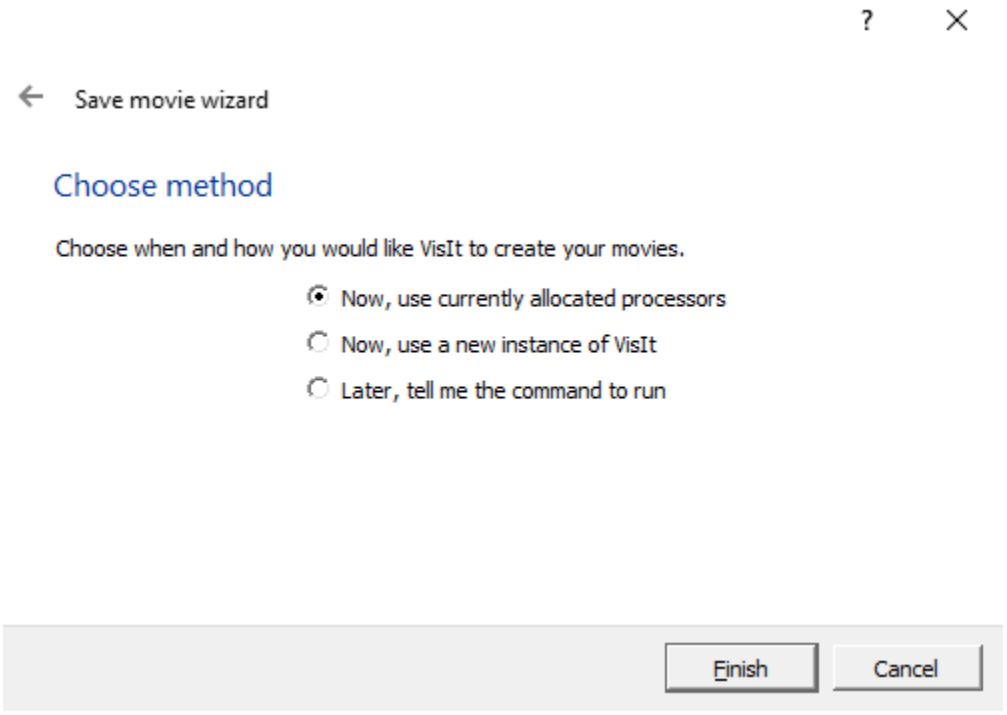


Fig. 6.154: Creating the movie with the existing processors.

(continued from previous page)

```
save_atts.resConstraint = save_atts.NoConstraint
save_atts.width = 1200
save_atts.height = 1068

# Get the number of time steps.
n_time_steps = TimeSliderGetNStates()

# Loop over the time states saving an image for each state.
for time_step in range(0,n_time_steps):
    TimeSliderSetState(time_step)
    save_atts.fileName = "dbreak3d%04d.png" % time_step
    SetSaveWindowAttributes(save_atts)
    SaveWindow()
```

1. Go to *Controls->Command* to bring up the *Commands* window.
2. Copy and paste the code snippet above into the first tab of the *Commands* window.
3. Click *Execute*.

The next step is to encode the movie using the encoder that comes with VisIt. You will need the "ffmpeg" encoder to be installed on your system and in your search path for the encoding module from visit_utils to function. The following snippet of Python code will load the visit movie encoding module and encode the movie.

```
from visit_utils import *

encoding.encode("dbreak3d%04d.png","dbreak3d.mpg",fdup=2)
```

The first argument specifies the file naming pattern for the input files. You can use the same format string used to create the images. The movie encoder doesn't support format strings that have multiple digit sequences in them, so it is best to keep the name of the input images simple, with only a single digit sequence.

The second argument is the name of the output file. The extension determines the file format to create. The available options are: "mpg", "wmv", "avi", "mov", "swf", "mp4" and "divx". "wmv" is usually the best choice and plays on most platforms (Linux, macOS and Windows). "mpg" is lower quality, but should play on any platform.

The last argument specifies the number of times each frame is duplicated. We are specifying duplicating each image twice. This option is useful if you don't have a lot of time steps and want to extend the length of the movie. Movies typically play at 30 frames per second so if you only have, for example, 60 frames, the movie will only play for about 2 seconds.

1. Copy and paste the code snippet above into the second tab of the *Commands* window.
2. Click *Execute*.

6.9.5 Other Tips for Making Quality Movies

Ensure that limits are appropriate and consistent across the entire movie

The objects in simulations typically change in size or move in position. Because of this the view that may be appropriate at the first time step isn't appropriate at later time states. For example, suppose a simulation were modeling the explosion of a supernova. As the simulation progresses the supernova grows in size and at some point most of the supernova may be outside the view. One possible solution would be to set the size based on the supernova at the last time state. If this isn't acceptable it may be necessary to zoom out at a few key points in the simulation to ensure that the supernova is still within the view.

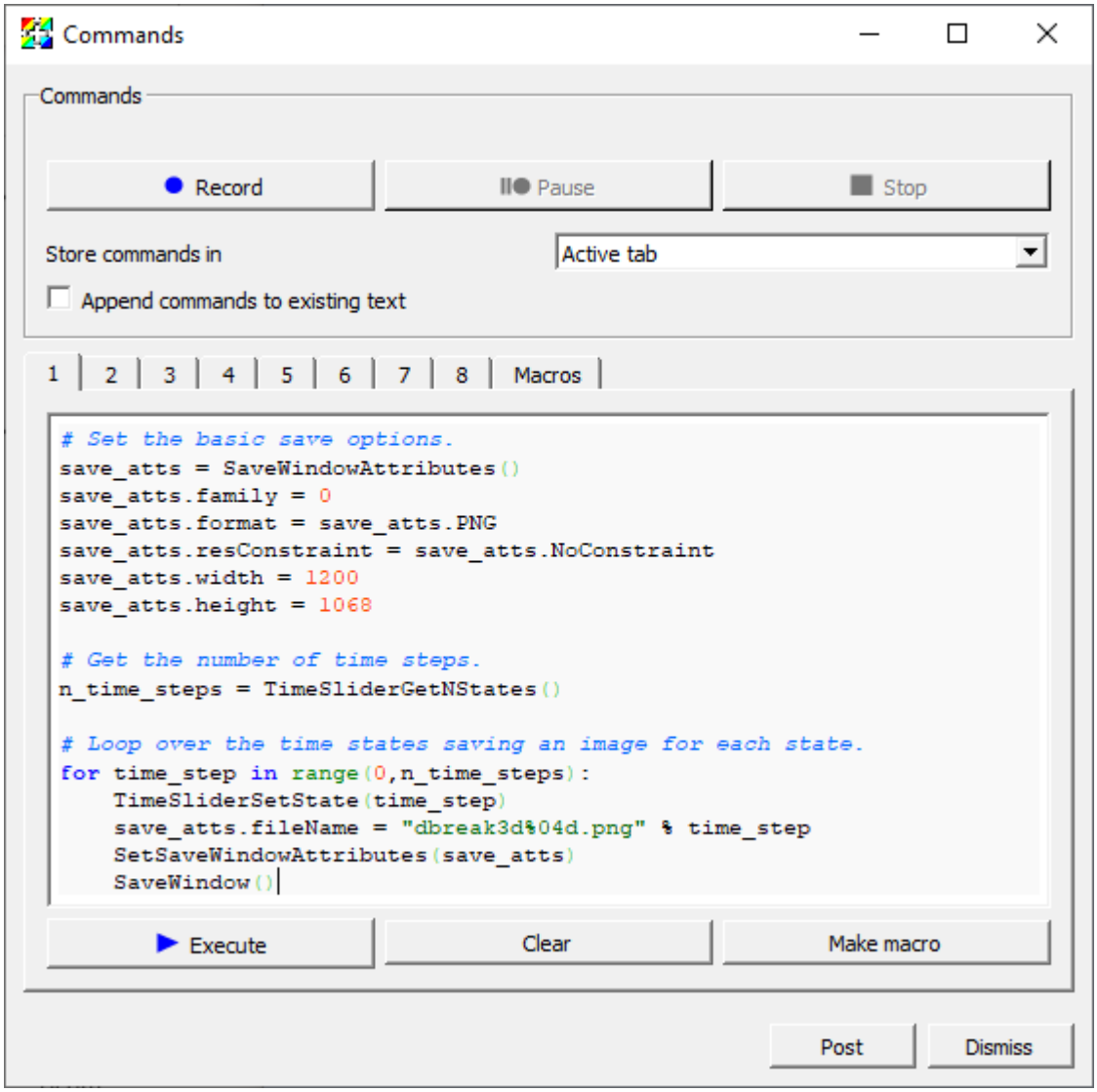


Fig. 6.155: Saving the movie images with a Python script.

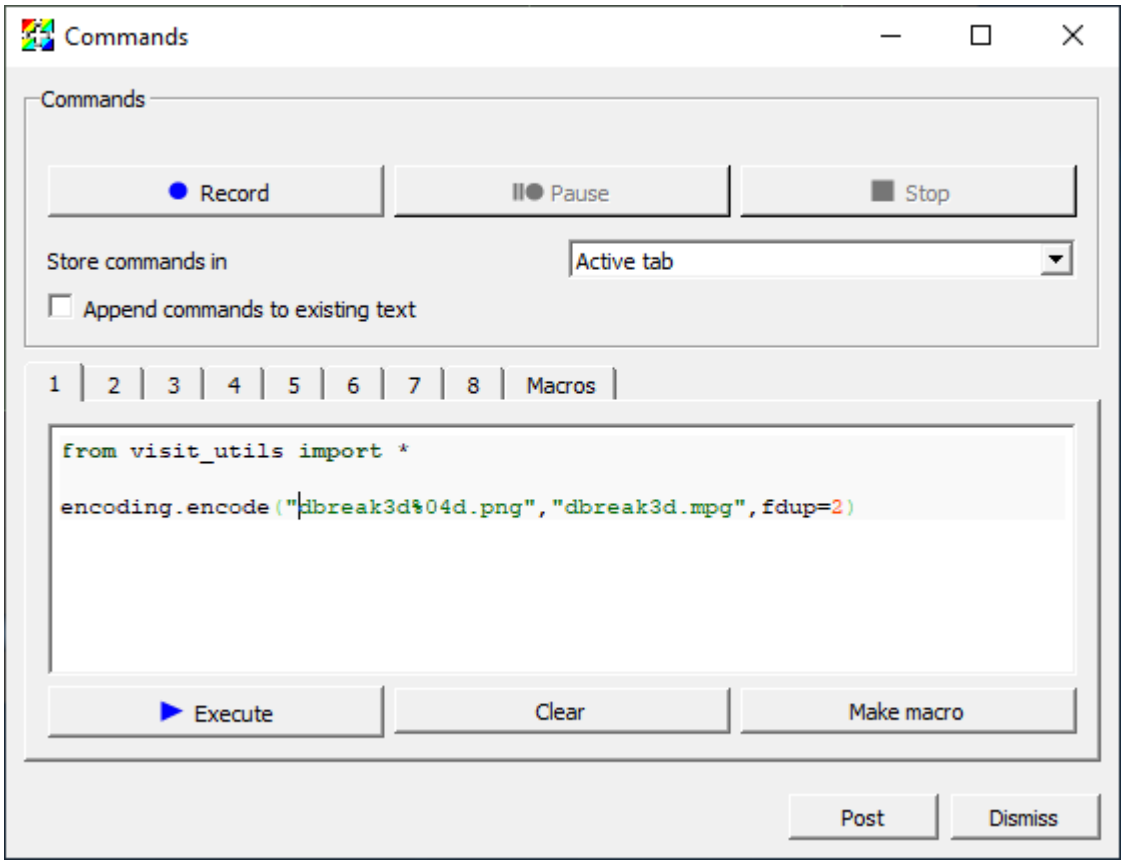


Fig. 6.156: Encoding the movie images with a Python script.

Another common issue is that VisIt by default will set the extents for things like the Pseudocolor plot based on the limits of the current time state. Typically the limits will change over time, which will result in the meaning of a specific color changing over time. This is typically not desired behavior for movies. In this case, the limits in the Pseudocolor plot should be set so that they are appropriate for the entire time series.

Selecting the resolution

You should always select an aspect ratio for your movie that shows off your content the best. One strong consideration is minimizing the amount of white space in your movie. If your simulation is primarily square then you will probably want your movie to have a roughly one-to-one aspect ratio. If it is wider than it is tall then you proably want something closer to a two-to-one or three-to-two (width-to-height) aspect ratio. Another important consideration is the type of device you will be displaying you movie on. These days monitors tend to be wide screen and a good resolution to have in mind is HDTV (1920 by 1080). It is probably best to try and add annotations to your movie to fill the white space so that you can get as close to an HDTV aspect ratio (16 x 9) as possible.

Rendering images gives the most flexibility

If you want to create a movie to show to many people or will be using it in multiple situations it is best to save images and then manually encode them using the movie encoding tools in VisIt, or if you want a really high quality movie with sound then you can use a third party movie encoding tool.

If you anticipate using your movie in multiple situations you should encode it at the highest resolution you expect to need it and then encode multiple movies at different resolutions. To create the different resolution movies, you would first resize the images to the desired size and then encode the movie. A good trick for generating higher quality anti-aliased movies is to save the images at quadruple the resolution (two times in each direction) and then resizing them to a quarter of that resolution before encoding the movie.

Resizing images

A good tool for resizing image is ImageMagick's convert tool. It is installed on most Linux and Mac macOS operating systems. If you don't have ImageMagick installed on your systems and in your search path the following code snippet will fail. The following snippet of Python code will run convert to resize the images created earlier to one half their resolution.

```
from subprocess import call

for time_step in range(0,n_time_steps,4):
    file1 = "dbreak3d%04d.png" % time_step
    file2 = "dbreak3d_600x534_%04d.png" % time_step
    call(["convert", file1, "-resize", "600x534", file2])
```

1. Copy and paste the code snippet above into the third tab of the *Commands* window.
2. Click *Execute*.

Convert can also be used to do other types of image manipulations such as cropping a flipping images. To learn more about convert google *ImageMagick convert*.

Higher quality encoding with ffmpeg

You can get higher quality encoding using *ffmpeg* instead of VisIt's built-in *mpeg2encode*. If VisIt finds *ffmpeg* in your search path it will use that instead of the built in encoder.

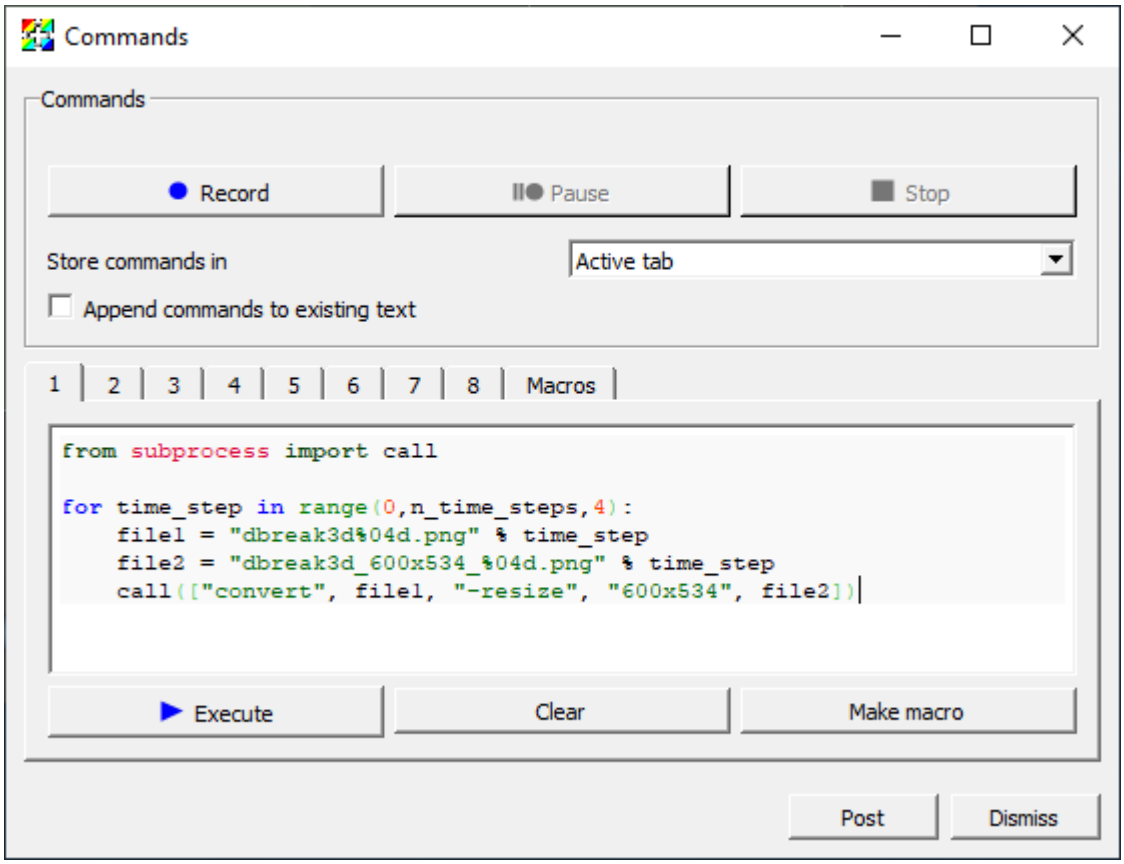


Fig. 6.157: Resizing the movie images with a Python script.

You can obtain *ffmpeg* from the ffmpeg download site. Scroll down a bit until you get to a section labeled *Get packages & executable files*. Click on the icon representing the OS you desire, this will change the text below the three logos. Choose and click one of the options and you will be taken to a page with downloads or package information.

You may be able to install *ffmpeg* on Linux with the standard package manager for the flavor of Linux you are running. For example, to install on Ubuntu:

```
apt-get update
apt-get install -y ffmpeg
```

6.10 Molecular data features

The basics of Molecular data visualization in VisIt are found in the *Molecule Plot*, the *Create Bonds operator* and the *Replicate operator*.

6.10.1 Replicate and CreateBonds Examples

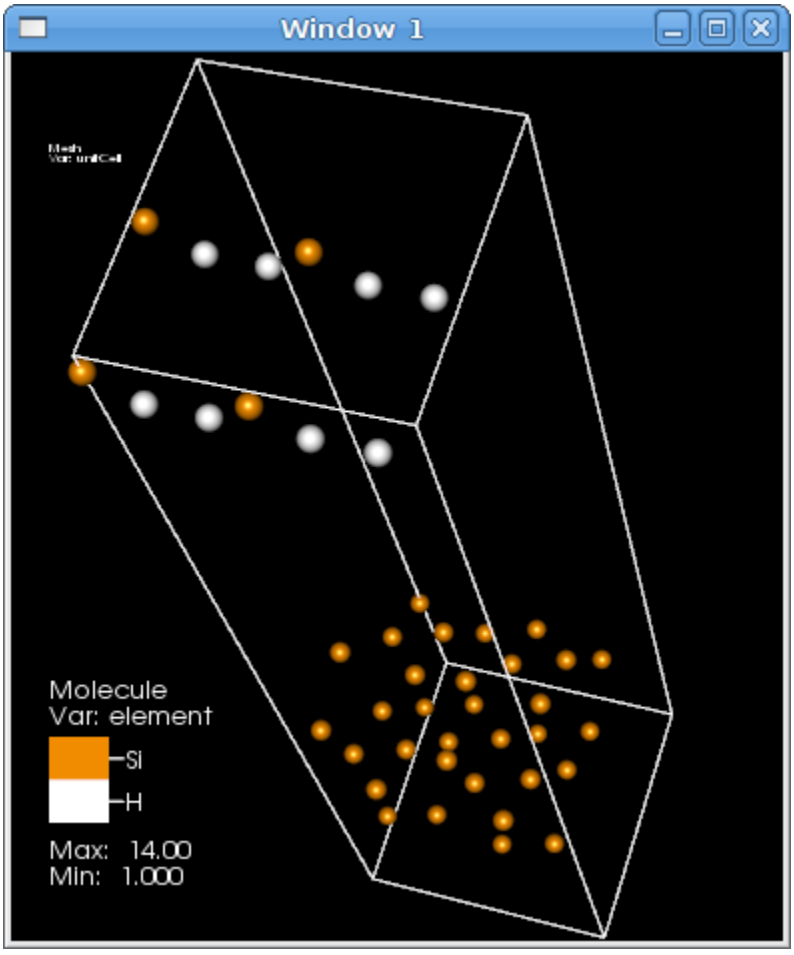


Fig. 6.158: This image shows the original data set, with the original data set's unit cell drawn. (The unit cell happens to be orthogonal, but is not actually axis-aligned). No replications and no bond creation have yet been applied.

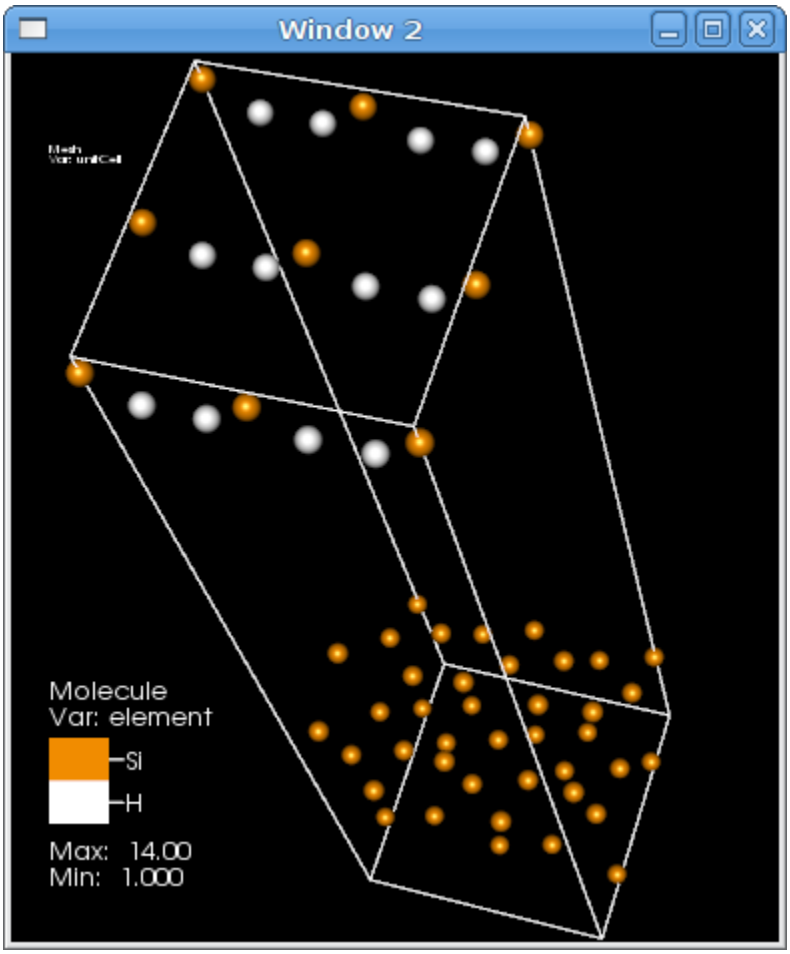


Fig. 6.159: In this image, the *Replicate operator* was applied, with no replications (i.e. X/Y/Z replication counts remaining at 1,1,1), but with the **periodically replicate atoms at unit cell boundaries** feature enabled.

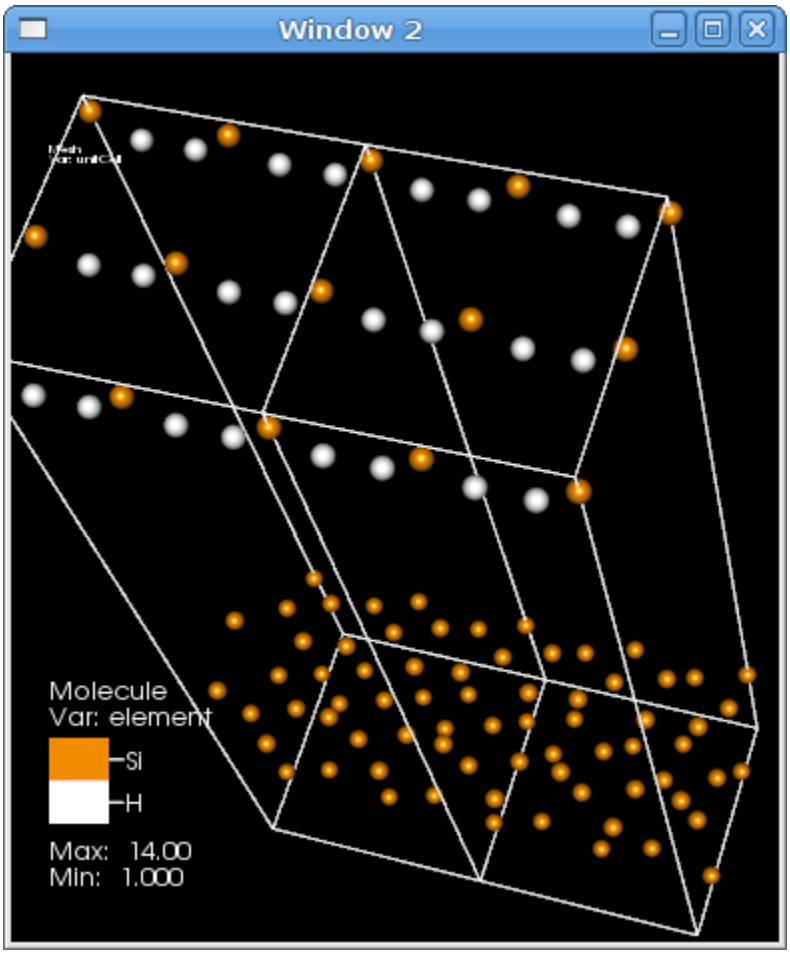


Fig. 6.160: Now the replication values have been changed in this image to “2,1,1”, with the replication vectors being used as-specified in the file to correspond to the unit cell of the problem.

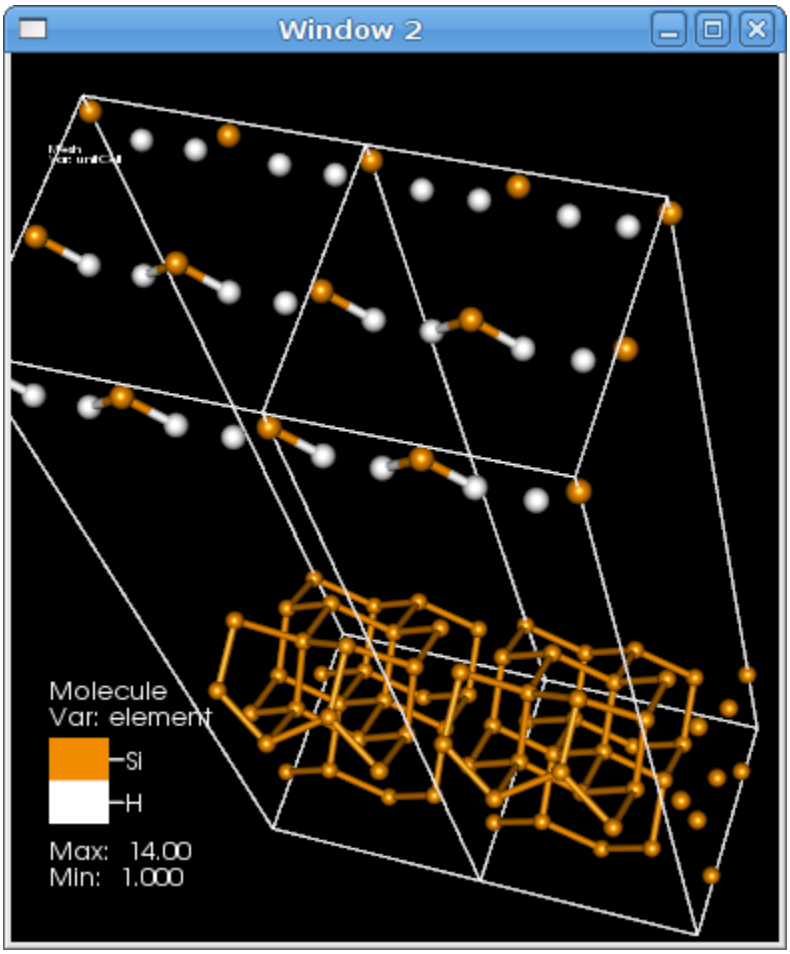


Fig. 6.161: This image shows the incorrect result (missing bonds between unit cell instances) occurring in two conditions: either the *Create Bonds operator* was applied before replication, or the *Replicate operator* did not have the **Merge into one block** box checked.

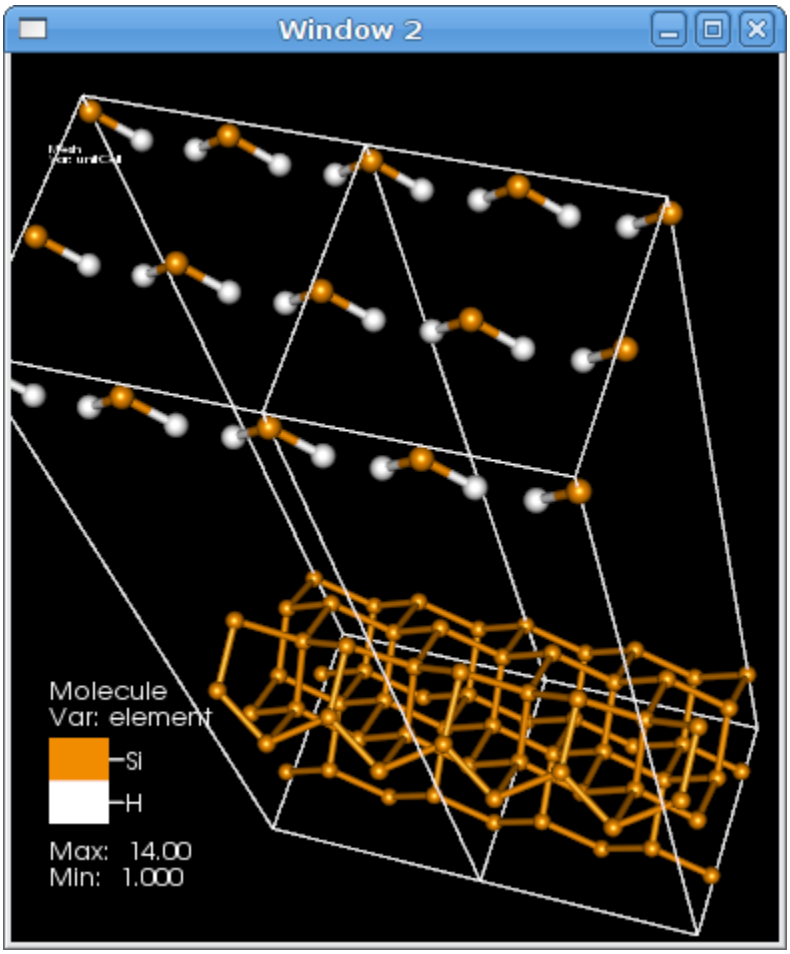


Fig. 6.162: This shows the correct behavior: the **Merge into one block** box was checked, and the *Create Bonds operator* was applied after replication, thus allowing bonds to span unit cell instances.

6.10.2 Other plots and operators

The following images show plots and operators you might use to explore your data apart from the *Molecule Plot* and related operators. These examples show charge density and force vectors associated with the raw molecular positions and species, all combined in the same window as a *Molecule Plot*.

Pseudocolor Plot and ThreeSlice Operator

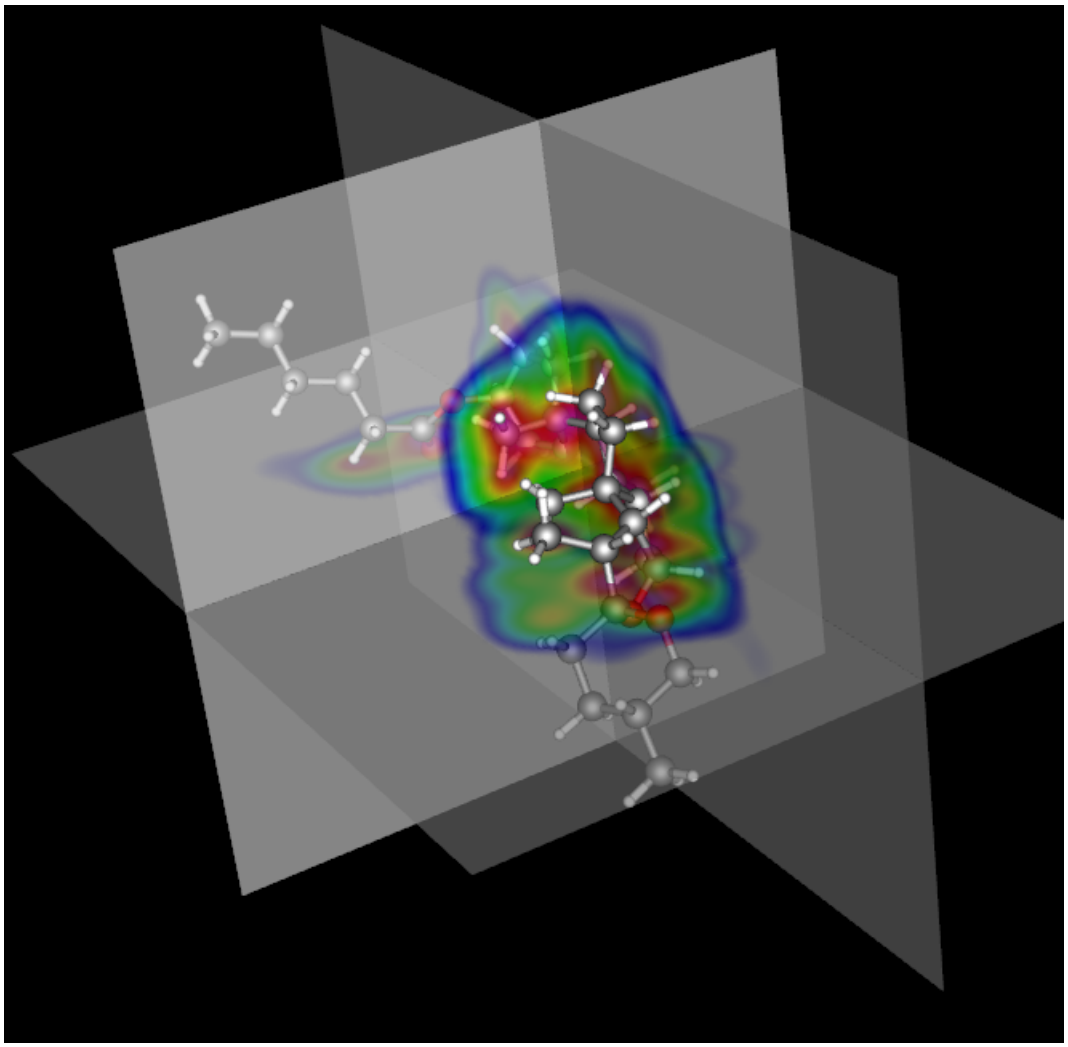


Fig. 6.163: In this image, the charge density grid is shown using the *Pseudocolor plot*, with moderate transparency, after applying the *ThreeSlice operator* to the grid around a point near the center of the molecule.

Contour Plot on a 3D Structured Grid

Volume Plot of the 3D Grid

Isocontour Lines on a Slice

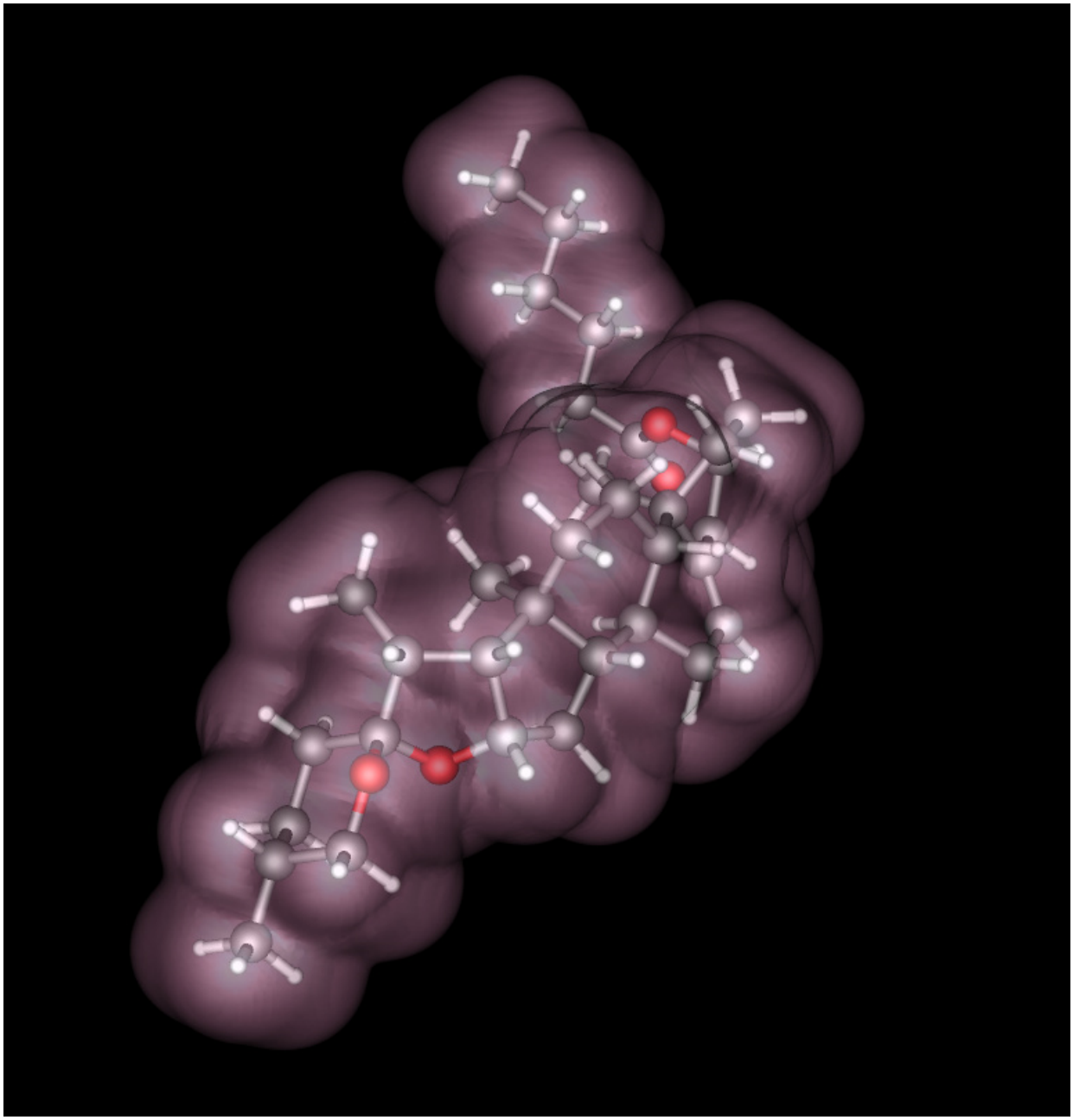


Fig. 6.164: In this image, a *Contour Plot* has been applied the charge density grid, with a single low-density value, and some transparency so that the molecule itself is still visible. Note that if you have more than one variable on your grid, for more flexibility you might choose to use the *Isosurface operator* over one variable and color using the *Pseudocolor plot* on a second variable.

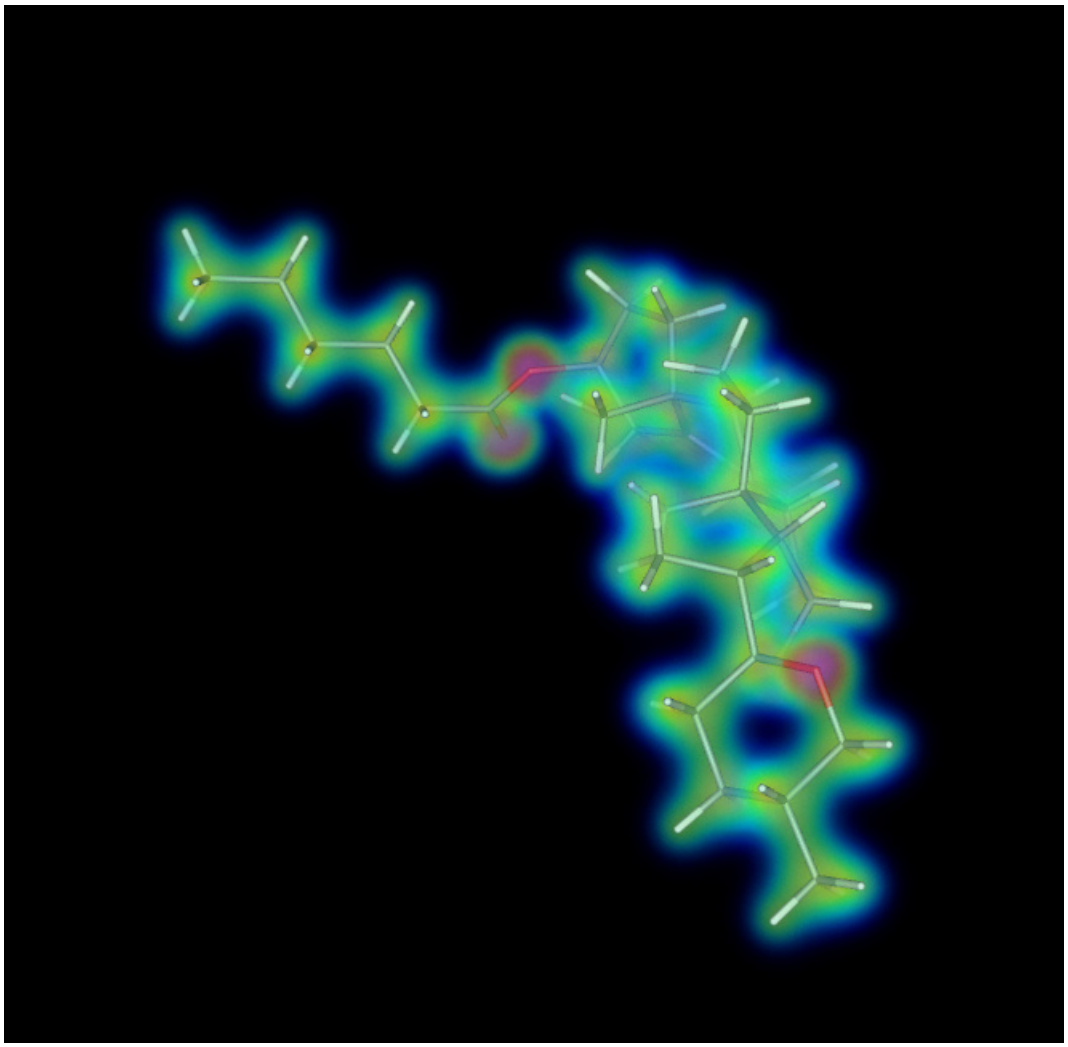


Fig. 6.165: This shows a *Volume plot* of charge density. Note that the *Volume plot* has a continuously adjustable opacity and by nature allows farther parts of the data to show through to the front, allowing the whole data set to be involved in the final picture.

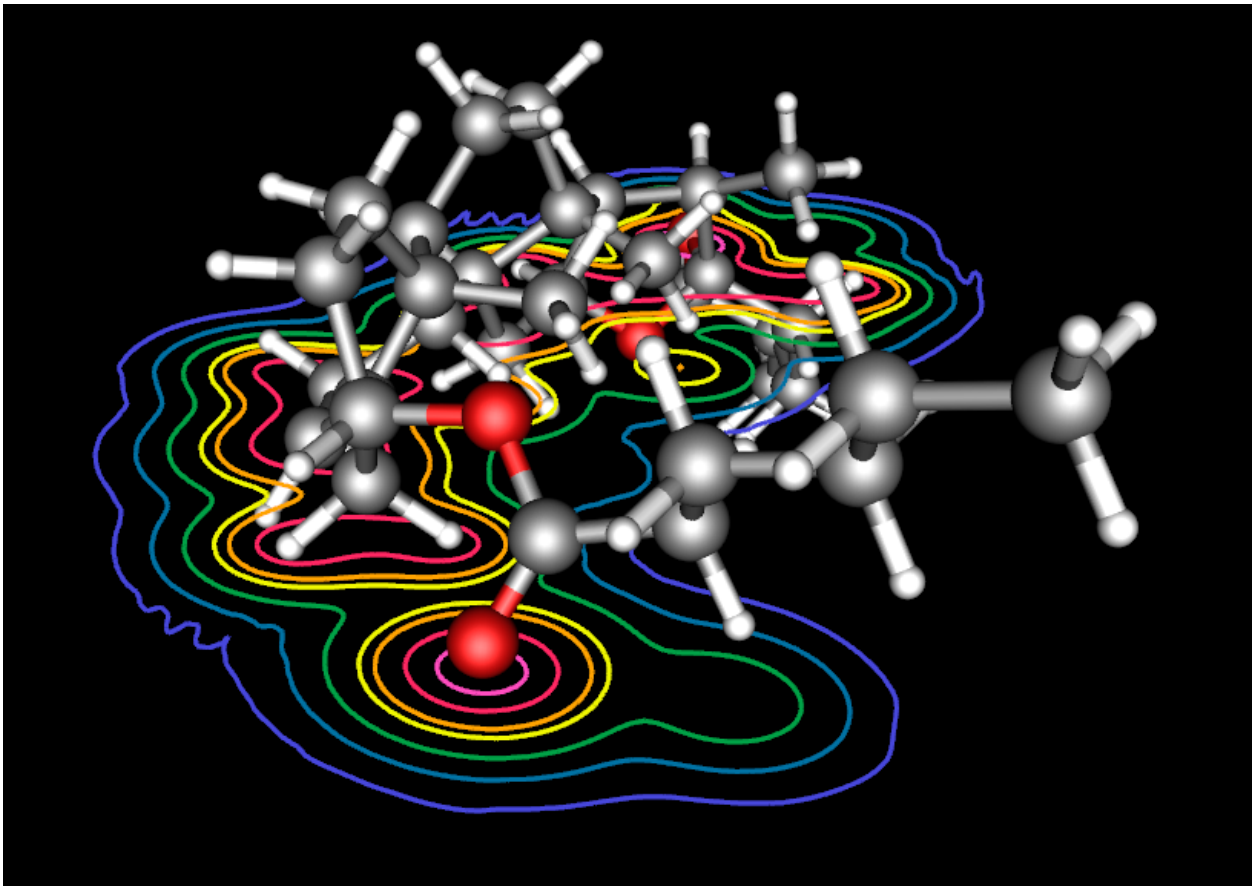


Fig. 6.166: Here we used the *Contour Plot* on a slice through the data, with a thicker line width, and a continuous color table to show the increasing charge density.

Vector Plot of Forces on Point Data

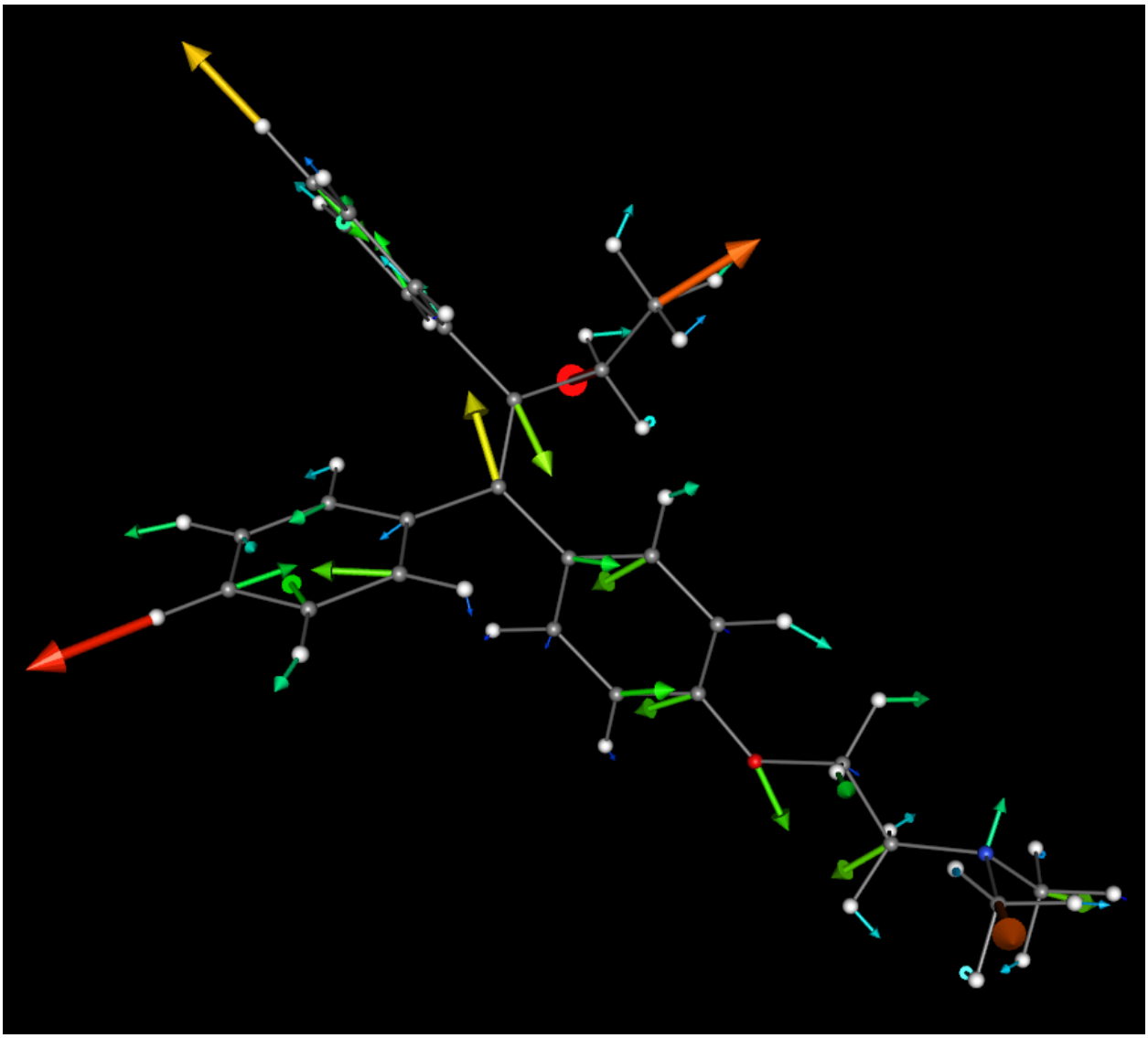


Fig. 6.167: This image shows a *Vector plot* of the force vectors on the atomic data itself. Vectors are both colored and sized using the magnitude of the force vector.

6.10.3 Analysis Capabilities

Subset Selection

The screenshot in Figure 6.168 shows the same plot in two windows, but with different subset selection. The top image shows the standard Molecule plot of a data set. The bottom shows the *Molecule Plot*, but with the "Subset" set to de-select Oxygen atoms.

Various file format readers may present a different set of subsets to the user through VisIt. For example, the *Protein Data Bank* reader presents compounds, residues, and atom type. The *VASP* reader presents only the atom type, but is

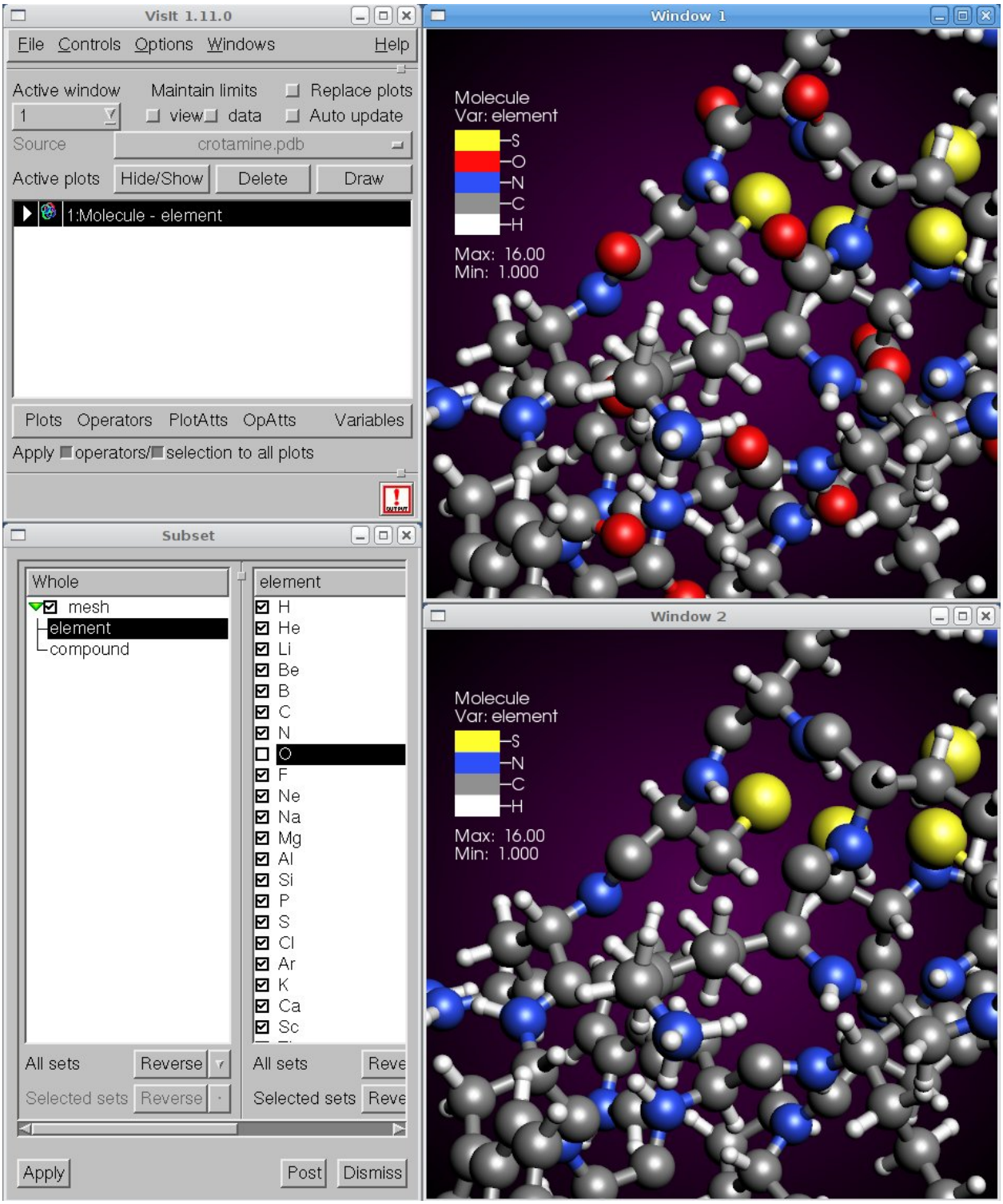


Fig. 6.168: A molecule plot and a subsetted molecule plot

smart enough to restrict the choice to only those elements actually present in the file (while the *PDB* reader presents all 100+ element types).

Atomic Color Tables

VisIt includes a variety of color tables, some for continuous variables and some for discrete variables. For molecular plots, such as ones coloring atoms by their species, VisIt includes color tables which match up with residue types or atomic numbers and have similar colors to conventional ones used. The ones included with VisIt for atomic numbers are called "cpk_rasmol" and "cpk_jmol", and for residue types are "amino_rasmol" and "amino_shapely".

However, you can also create your own. The easiest way is to start by selecting one of these, typing a new name, e.g. "my_atom_colors", and clicking the **New** button. This makes a copy of the selected color table with the new name. You can then edit the colors at will, and when you Save Settings (in the Options menu), it will keep your new color tables in future sessions.

Note that in Figure 6.169, you see one of the features of the color table editor for atomic data, which is to provide hint labels for the colors in the grid. Normally these are displayed as numbers, but for atomic color tables it will display the element's symbol instead. Note: VisIt assumes if the number of colors matches what is in the provided atomic number color tables (which is 110) that it is an atomic color table. So make sure if you're creating a new atomic color table to create one with the correct number of color values.

Expressions

Basic Expression Support

Numeric expressions, created in VisIt's Expressions window, are compatible with molecular data types. For example, if one created the variable "zcoord" as a Scalar, defined as "coords(mesh)[2]" (where "mesh" is the name of the mesh in your data file containing the atomic data), then it will create a new value, centered at the atoms, of the value of the Z coordinate of the atoms.

Enumerate Expression

One useful expression for some molecular data files is the *Enumerate Expression*. The most common use case is if your data file contains only a species type index, such as {0, 1, 2, etc.}, but does not have support for mapping this index to an actual atomic number. In this case, some molecular operations in VisIt, which require an atomic number (often called "element"), will not work. In this case, you can use the *Enumerate Expression* to map, e.g. "0" to "14" (Si), "1" to "80" (Hg), etc. Typically you want to call this new scalar variable "element" as this is the convention VisIt follows by default for this variable (though in some plots/operators you can specify a different one).

For example, the LAMMPS readers and VASP POSCAR reader do not have intrinsic knowledge of which type of atom in the file maps to which atomic number – but they do report the atom type (0,1,2...) as a variable called "species". To enable the VisIt features which use atomic number, define a new expression, called "element", of type "Scalar Mesh Variable", with the definition "enumerate(species, [14,80,8])", which maps the first type to Si, the second to Hg, and the third to O.

Enhanced Rendering

Plot Quality

Most plots have a number of options which can increase their quality at the cost of performance. Some examples follow.

Fig. 6.169: A color table for plotting molecules

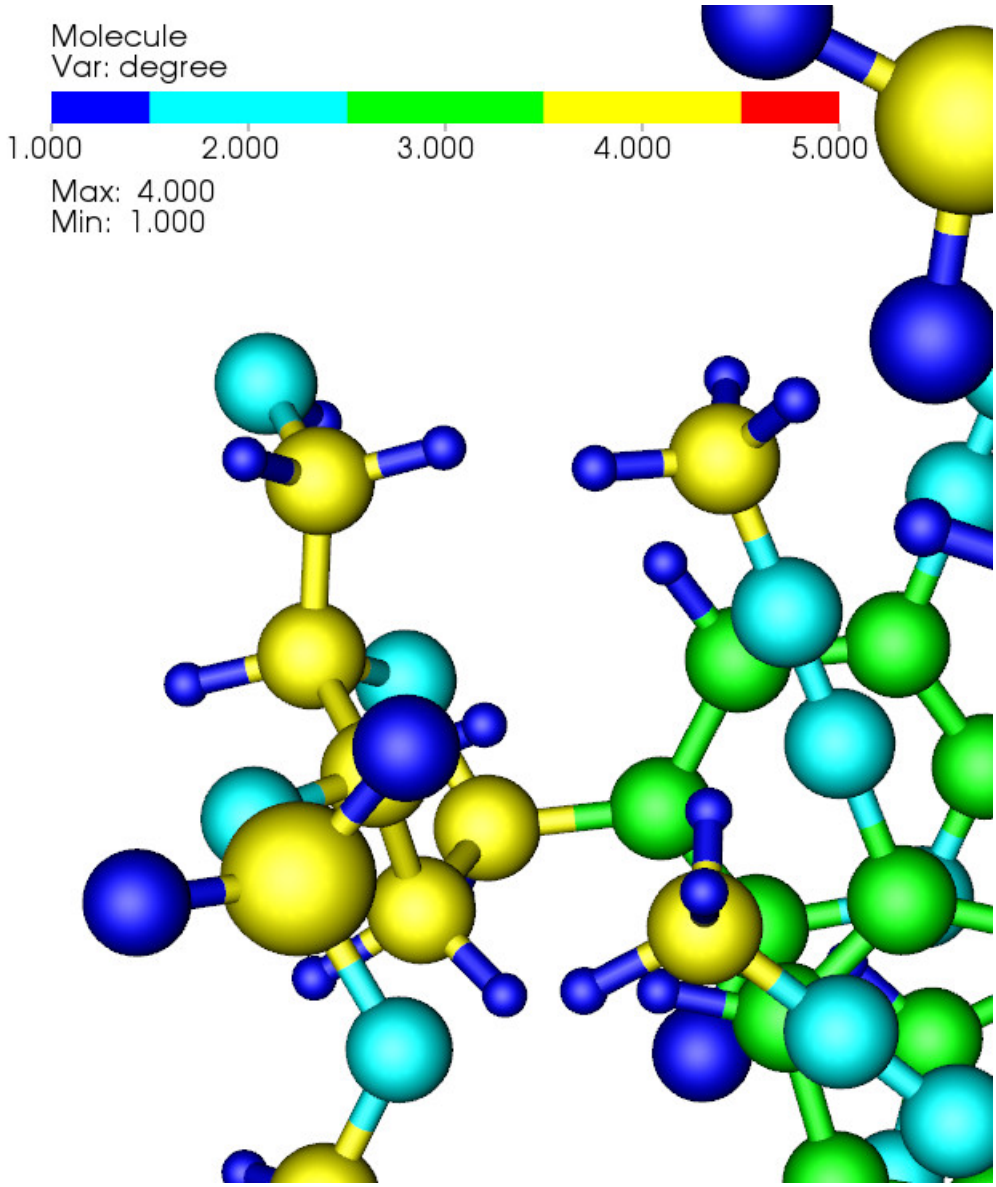


Fig. 6.170: Molecule Plot of “degree(mesh)-1” (subtracting 1 because the atom itself is a cell in VTK)

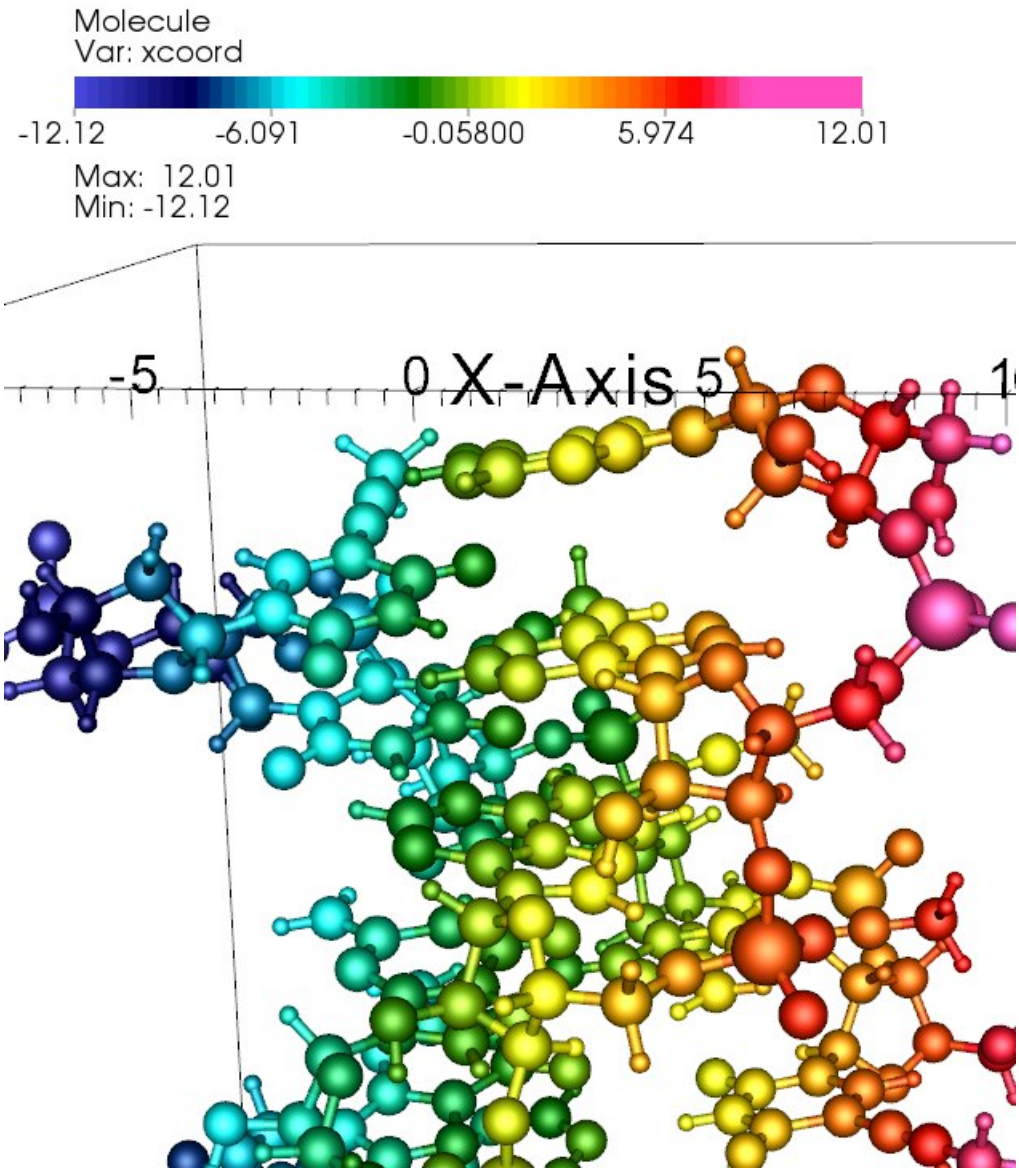


Fig. 6.171: Molecule Plot of the X coordinates of the atoms via the expression "coords(mesh)[0]"

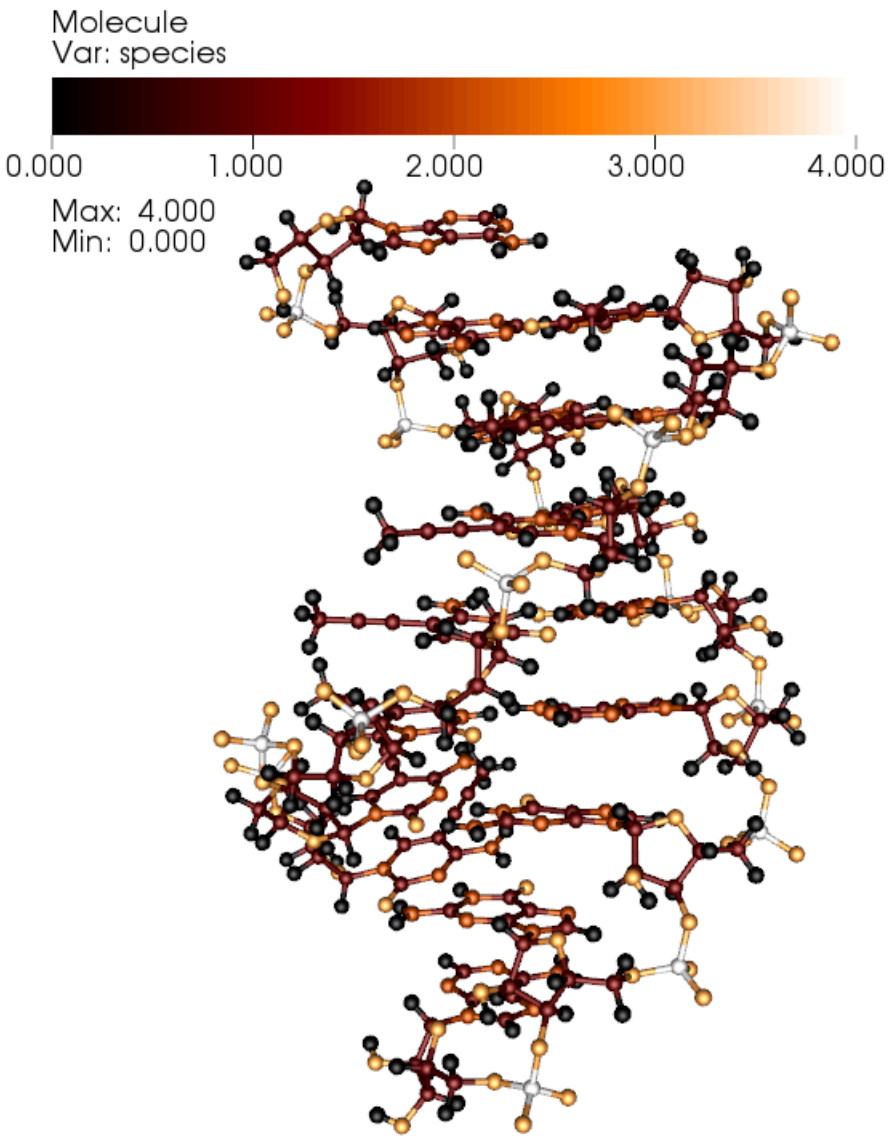


Fig. 6.172: Molecule Plot of "species" directly from file. Note that it's simply a continuous scalar field as far as VisIt is concerned, and can't be used for atomic properites

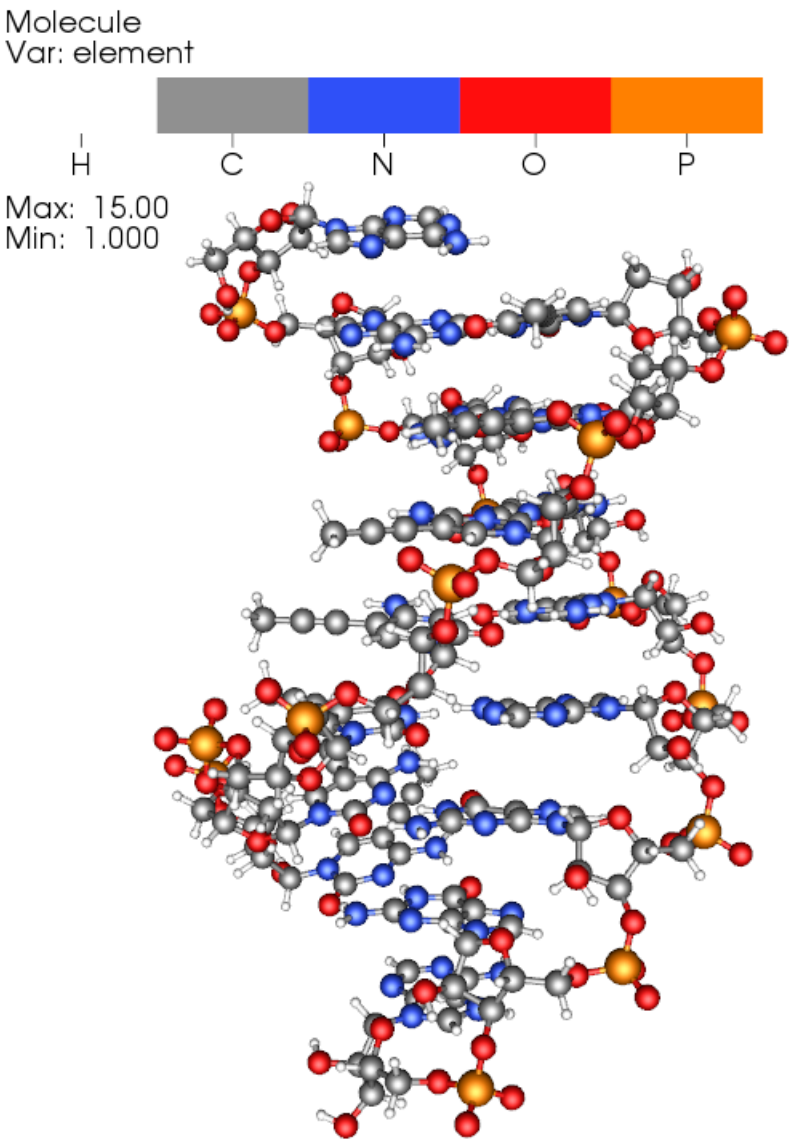


Fig. 6.173: Molecule Plot of "element" expression defined as an enumeration of "species". Note that the Molecule plot can use this element variable to determine atomic radius.

Molecule Plot Quality

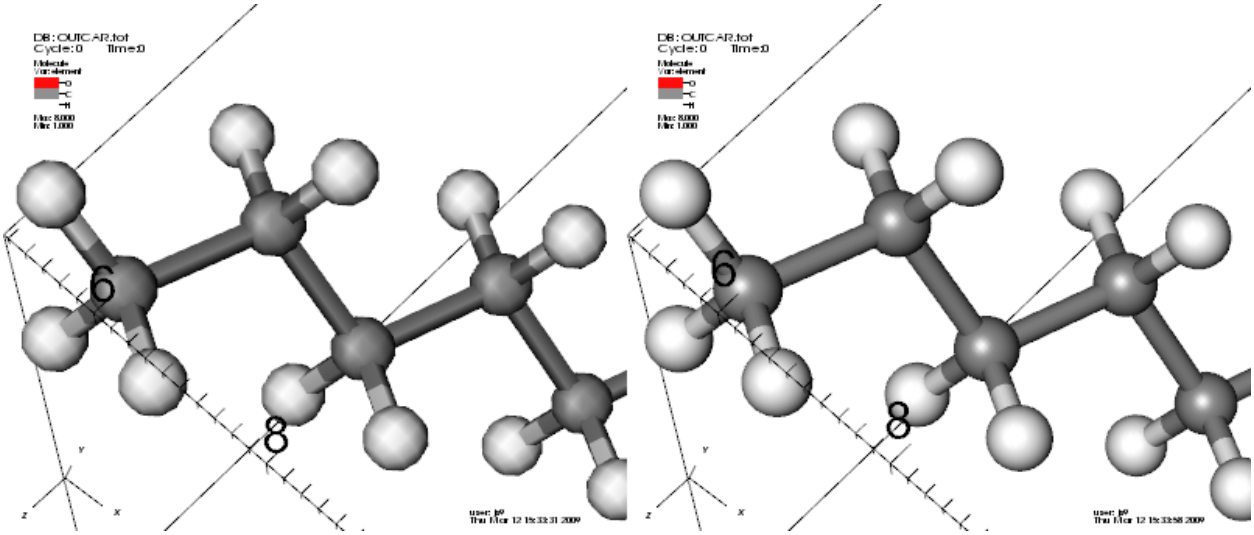


Fig. 6.174: The first example, on the left (before) vs. on the right (after), shows what increasing the atom and bond rendering quality can do in the *Molecule Plot*.

Vector Plot Quality

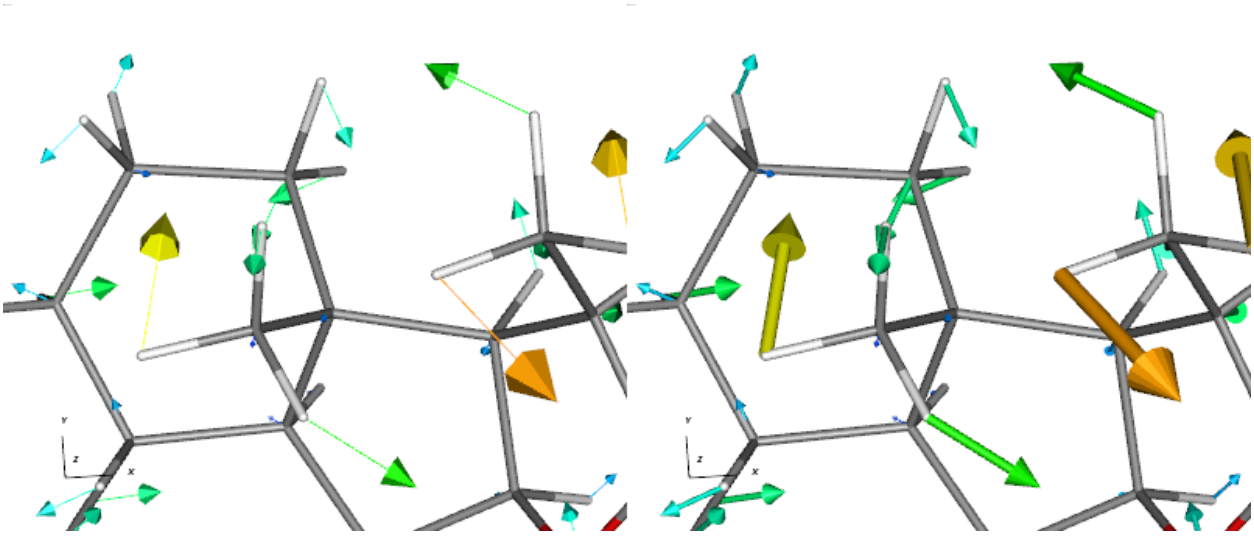


Fig. 6.175: This second example, left (before) vs. right (after), shows what using cylinders for stems, and higher polygon count vector heads, does for the *Vector plot*.

Annotations

The example in Figure 6.176 shows the same plot before and after modifying various annotation properties, such as:

- switching to a darker, gradient background

- turning off the 3D bounding box, coordinate axes, and triad
- disabling database and user information
- moving the legend, changing its orientation and size
- adding a time slider progress bar, and text showing the time value

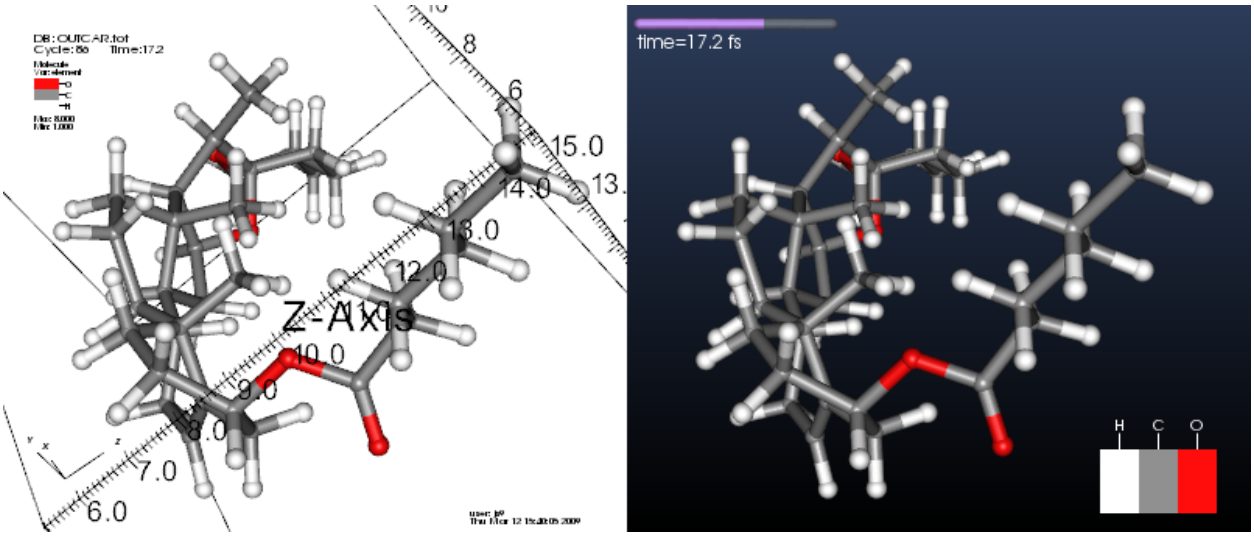


Fig. 6.176: Before (left), After (right)

File Export

VisIt has the ability to save windows, not just as image formats like PNG and JPEG, but as data files which can be imported into other tools. Some of these data types can be imported back into VisIt or other visualization and rendering tools which might have different rendering features of interest for making renderings.

POV-Ray

One of the exportable data file types in VisIt, after composing your plots in VisIt, is a set of POV-Ray scene description files, which are commented and composed in a manner intended to be tweakable by users to achieve results better than what one could get with a real-time rendering tool. See below for an example.

Data File Formats

VisIt contains readers for over 100 different scientific, code-specific, and other general file formats. Below are listed several of the most specific to molecular data.

Note that many of these formats have lax restrictions on naming, and VisIt may not automatically detect the file type. To force VisIt to try your desired file reader (as listed in quotation marks in the section header below), use that reader's name as the input to the "-assume_format" command when launching VisIt. For example, "visit -assume_format CTRL" will try the LMTO CTRL reader before reverting to its automatic detection code, and "visit -assume_format LAMMPS" will try the two LAMMPS readers first.

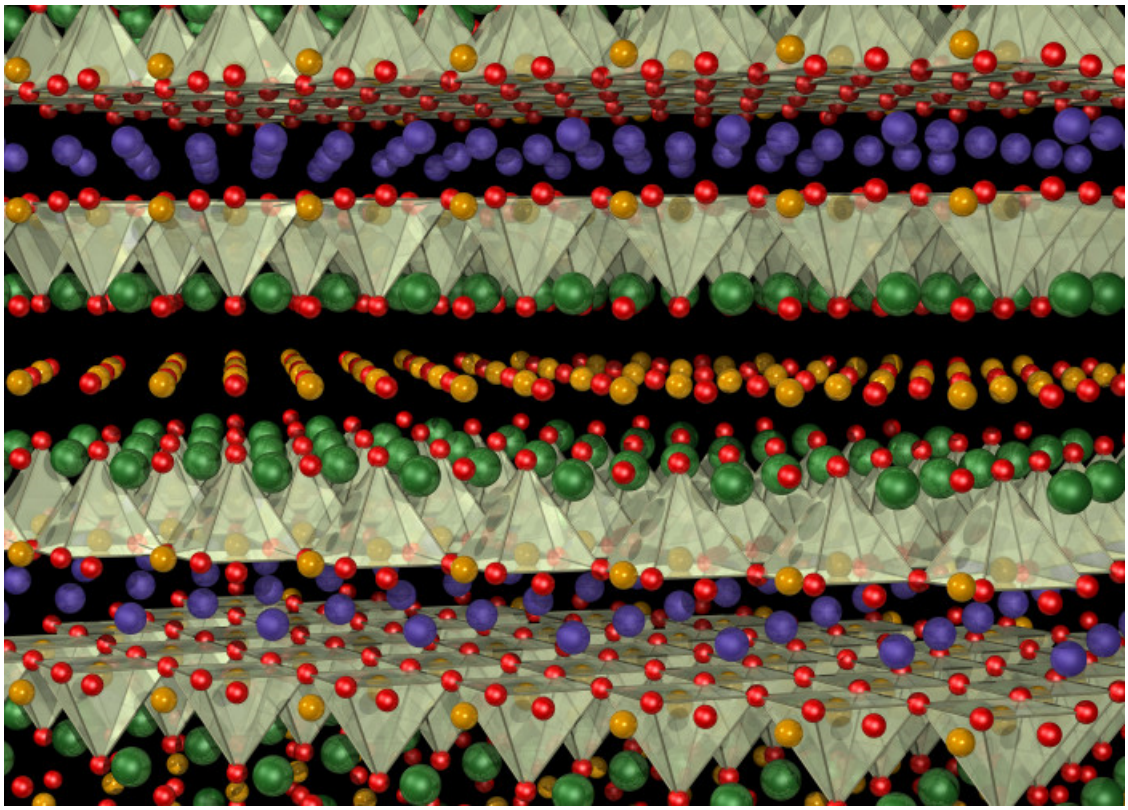


Fig. 6.177: A set of atoms and geometry rendered with POV-Ray.

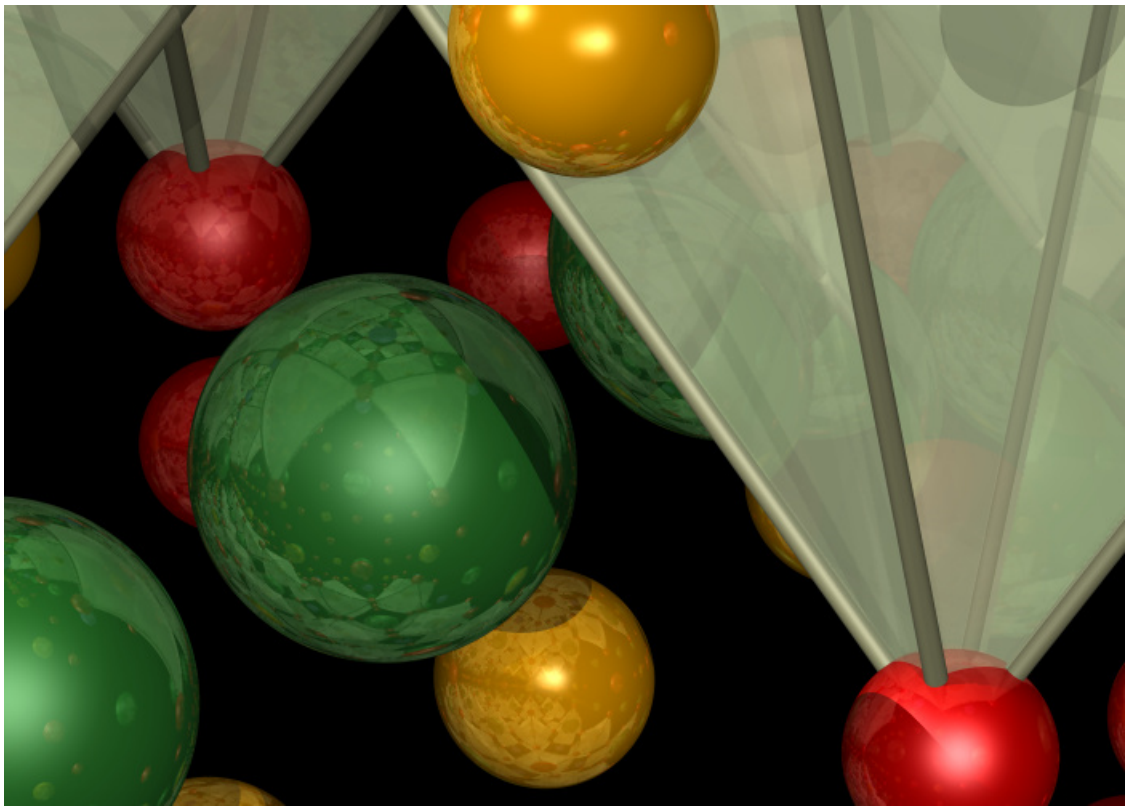


Fig. 6.178: A closeup of the previous one, showing reflection, refraction, shadows, and varying surface characteristics.

VASP (CHGCAR, POSCAR, OUTCAR) File Formats

The VASP code, as described in the link, is "a package for performing ab-initio quantum-mechanical molecular dynamics (MD) using pseudopotentials and a plane wave basis set." Its output is ASCII text in several files, and the VASP reader in VisIt supports "OUTCAR" and "POSCAR" for varieties of atomic positions and variables, and "CHGCAR" for charge density grids.

Since the charge density grids can get very large, the VisIt CHGCAR reader is actually parallelized to help speed the ASCII-binary conversion process on multi-node machines when using the MPI-enabled version of VisIt's computation engine. It will decompose the grid into as many domains as you have processors, and each will read and process its chunk of data. Since this is an ASCII format, the speedup for the I/O portion will not scale to large numbers of processors, but the decomposition will also help the rest of the pipeline scale in parallel for other compute-intensive operations.

LAMMPS (input structure and output dump) File Formats

LAMMPS is the "Large-scale Atomic/Molecular Massively Parallel Simulator". The VisIt LAMMPS reader supports two flavors of data files used with LAMMPS.

The first is the output dump file in Atom style, usually ending in ".dump". Here's a small example of that format with three variables per atom (the final three columns):

```
ITEM: TIMESTEP
1500
ITEM: NUMBER OF ATOMS
5
ITEM: BOX BOUNDS
0.0 2.0
0.0 3.0
0.0 2.5
ITEM: ATOMS
2 1  0.0 0.0 1.0  0 0 0
4 1  2.0 3.0 2.5  0 0 0
1 2  1.4 0.7 0.0  0 3 1
3 2  0.3 1.0 0.5  0 1 7
5 2  1.7 2.0 0.2  0 7 7
```

In this example, the second and fourth atoms are of the first species type, and the first, third, and fifth are of a second species. So you'll need to create an enumerate expression to create the atomic numbers needed for various molecular operations. For example, create a variable called "element", of type Scalar Mesh Variable, and define it as "enumerate(species, [1, 8])" – this maps the first species to hydrogen, and the second to oxygen.

Note that the LAMMPS Atom-style dump has changed: the ITEM line with ATOMS now specifies the columns which were be written out. To continue supporting the old atom-style dump format, the reader assumes a format string of "id type x y z" (i.e. unscaled atom coordinates) if the line only contains the word "ATOM" with no format specified. The new default is "id type xs ys zs" (scaled atom coordinates) for the updated format. See the LAMMPS documentation of the "dump" command for details.

The second format is the input format used for the LAMMPS "read_data" command. Its file extension is not standardized, but can sometimes be ".eam", ".meam", and ".rigid".

```
Position data on strange chemical

     5       atoms
     2       atom types
     0.0 2.0     xlo xhi
```

(continues on next page)

(continued from previous page)

```
     0.0 3.0     ylo yhi
     0.0 2.5     zlo zhi

Atoms

 2    1      0.0          0.0          1.0
 4    1      2.0          3.0          2.5
 1    2      1.4          0.7          0.0
 3    2      0.3          1.0          0.5
 5    2      1.7          2.0          0.2
```

(As an aside, note that there is a "proper" EAM file containing pair potentials. Though the "EAM" refers to the embedded atom potential method in both usages, these are different files.)

The ProteinDataBank (.pdb) File Format

The Protein Data Bank (PDB) archive contains molecular files in a standard ASCII format. The format, however, is used for a wide range of molecular data, not just proteins. See the docs for a full description of the file format. The PDB reader supports ATOM, HETATM, HETNAM, MODEL/ENDMDL, TITLE, SOURCE, CONECT, and COMPND directives.

This is a simple example of a 2-compound, 4-element type data file with a single model.

```
COMPND    First
ATOM      1  N   TYR A   1      27.557 -46.589  10.074  1.00  0.00           N
ATOM      2  H   TYR A   1      28.603 -46.872   9.068  1.00  0.00           H
COMPND    Second
ATOM      3  C   TYR A   1      29.675 -45.772   8.980  1.00  0.00           C
ATOM      4  O   TYR A   1      30.403 -45.678   7.992  1.00  0.00           O
```

The XYZ File Format

The .xyz file format is a simple ASCII format used for describing atom positions, species, possibly variables, and possibly with multiple time steps. Here's a simple example file:

```
``   3
Some file comment
H       22.3844     2.0352     0.0000
O       18.4512     3.5123     0.0000
Cu      14.2455     6.1056     7.3436``
```

Note that the first line lists the number of atoms, the second is a comment (or blank), and the third starts the data. In each data line, there is the element name, then the X, Y, and Z coordinates. Note that you may have several variables after the Z coordinate – VisIt will allow up to 6 extra variables. Below is an example with three extra variables, which will be called "var0" through "var2" inside VisIt, and can be combined into vectors or included in any other plotting or analysis operation VisIt supports.

```
3

H       22.3844     2.0352     0.0000      7   7.8    8
O       18.4512     3.5123     0.0000     12   1.6    9
Cu      14.2455     6.1056     7.3436     10   1.4   10
```

To support multiple timesteps in a single file, simply concatenate each timestep at the end of the previous one, with no blank lines or other separators. The VisIt XYZ reader also supports atomic numbers instead of element symbols in the first column and also supports the rather dissimilar CrystalMaker flavor of .xyz file (which we don't describe here).

Wikipedia has a page on the XYZ format, though it does not mention the possibility of extra variables or multiple timesteps, both of which are supported by VisIt.

The LMTO CTRL File Format

The CTRL file is a format used by the STUTTGART TB-LMTO program. LMTO is the linear muffin-tin orbital method used in density functional theory (DFT). This CTRL reader supports the STRUC, CLASS, SITE, ALAT, and PLAT file categories. (See this page for more details.)

Using the VTK File Format for Molecular Data

The VTK file format is well-understood by VisIt, as it is the underlying low-level data model for many of its internal data types. The VTK structure best used for molecular data is that of a "vtkPolyData" type, where the vertices are the atoms, lines are the bonds (if desired), and fields on the atoms are point data fields. An example of an approximate of a water molecule in the ASCII VTK file format is shown below:

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET POLYDATA

POINTS 3 float
1.0 0.5 1.5
0.2 0.1 0.8
0.4 0.2 2.3

LINES 2 6
2 0 1
2 0 2

VERTICES 3 6
1 0
1 1
1 2

POINT_DATA 3
SCALARS element float
LOOKUP_TABLE default
8 1 1
SCALARS somefield float
LOOKUP_TABLE default
0.687 0.262 0.185
```

If you have no bonds in the file, or would prefer to use the *Create Bonds operator* to generate them inside VisIt, simply drop the three lines of text in the "LINES" section of the file. For more detailed information about the VTK formats, see http://www.vtk.org/VTK/img/file-formats.pdf. Note that what are called the "Legacy" formats are both simpler and may be more widely supported than the more recent, and complex, XML formats

Acknowledgements

This work was supported in part by the Department of Energy (DOE) Office of Basic Energy Sciences (BES), through the Center for Nanophase Materials Sciences (CNMS) and Oak Ridge National Laboratory (ORNL), as well as the Advanced Simulation and Computing (ASC) Program through Lawrence Livermore National Laboratory (LLNL).

6.11 Python Expressions

This tutorial describes using Python expressions. Python expressions are an advanced feature and should olny be used if the needed functionality is not available in the *standard expression system*.

6.11.1 Python Expressions Overview

When the available expressions fail to provide needed functionality, users can extend VisIt's expression system in arbitrary ways using Python expressions. However, Python expressions require users to become somewhat converscent in detailed aspects of how mesh and variable data is represented in terms of VTK objects as well as how they are decomposed and processed in parallel. Nonetheless, Python expressions provide a powerful approach to filling in gaps in needed functionality.

Python expressions are run on the compute engine and will run in parallel whenever a) the compute engine is parallel and b) the input dataset is decomposed for parallel. Python expressions operate on `vtkDataSets`, which provide access to mesh variables as well as the mesh coordinates and topology. Python expression also have access to VisIt's metadata as well as access to MPI when running in parallel. Python expressions return a `vtkDataArray`, which allows returning new variables. It is not possible to change the mesh topology or coordinates *within* a Python expression. However, it is possible to *combine* Python expressions with *Cross Mesh Field Evaluation (CMFE) functions* which can have the effect of changing mesh topology and coordinates. The functionality is available through the **GUI** and the **CLI**. When using the **GUI**, they can be created in the *Expressions* window. When using the **CLI**, they can be created with the *DefinePythonExpression* function.

6.11.2 Creating a Python Expression with the GUI

We will now go through the steps required to create a Python expression using the **GUI**.

Let us start by opening a file and creating a plot.

1. Open the file `curv2d.silo`.
2. Create a Psuedocolor plot of `d`.

Now let us go ahead and create the Python expression.

3. Go to *Controls->Expressions*.
4. This brings up the Expressions window.
5. Click *New* in the *Expression list* to create a new expression.
6. Change the *Name* in the *Definition* section to "MyExpression".
7. Click on the *Python expression editor* tab in the *Definition* section.
8. Select *Insert variable...->Scalars->d* to add `d` to the *Arguments* text field.
9. Select *Insert variable...->Scalars->p* to add `p` to the *Arguments* text field. Note that the variable names are seperated by a comma. If the variable names are not separated by commas you will get a cryptic error message when you try to plot the expression.

10. Click *Load script->Template->Simple filter* to add a template of a Python expression in the *Python expression script* editor.

At this point you modify the template to create your expression. A common practice is to make modifications to the script and test it using the Pseudocolor plot. Changes to the script can be made either by modifying the script in the *Python expression script* editor or modifying it in an external text editor and then reloading the script. Generaly speaking, modifying the script in the *Python expression script* editor is easier than doing it in an external text editor except that it is difficult to tell how many spaces are at the beginning of the line since the editor does not use a fixed width font.

Developing the script in the *Python expression script* editor

The following steps can be used to iteratively develop your script using the *Python expression script* editor.

1. Edit the script.
2. Click the *Apply* button.
3. Go to *Variables->d* to change the variable to `d`. The first time you try your script this will not be necessary since the variable is already `d`.
4. Go to *Variables->MyExpression* to change the variable to "MyExpression" and execute the script.

Developing the script in a text editor

The following steps can be used to iteratively develop your script using a text editor.

Before you can modify the script you will need to save it.

1. Click *Save script* to save the script.

Now you are ready to modify the script.

2. Edit the script with your favorit editor.
3. Go to *Load script->File* to load the script.
4. Click the *Apply* button.
5. Go to *Variables->d* to change the variable to `d`. The first time you try your script this will not be necessary since the variable is already `d`.
6. Go to *Variables->MyExpression* to change the variable to "MyExpression" and execute the script.

6.11.3 Python Expression Example 1

This example adds two cell centered variables. It demonstrates accessing multiple variables and performing simple operations with them to generate a result.

Here is the example script.

```
class MyExpression(SimplePythonExpression):
    def __init__(self):
        SimplePythonExpression.__init__(self)
        self.name = "PythonExpression"
        self.description = "Add two scalar variables together"
        self.output_is_point_var  = False
        self.output_dimension = 1
    def modify_contract(self,contract):
```

(continues on next page)

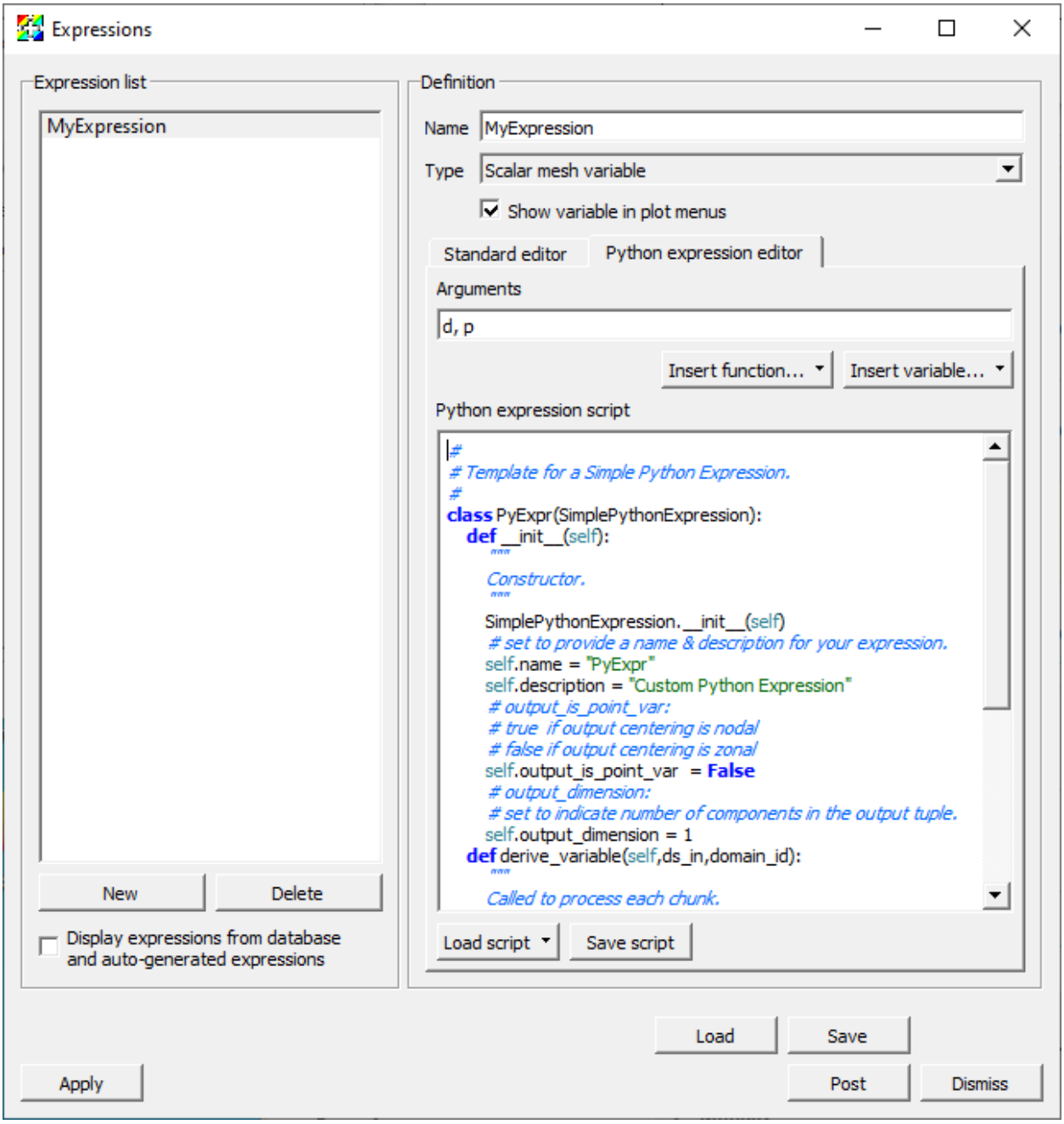


Fig. 6.179: The Expressions window with the simple filter template

(continued from previous page)

```
        pass
    def derive_variable(self,ds_in,domain_id):
        # ds_in is the input data set
        # Get the data array for the first variable
        cell_vals1 = ds_in.GetCellData().GetArray(self.input_var_names[0])
        # Get the data array for the second variable
        cell_vals2 = ds_in.GetCellData().GetArray(self.input_var_names[1])
        # Get the number of values in the variables
        ncells = ds_in.GetNumberOfCells()
        # Create a scalar float array with ncells values for the result
        res = vtk.vtkFloatArray()
        res.SetNumberOfComponents(1)
        res.SetNumberOfTuples(ncells)
        for i in range(ncells):
            # Add the i'th value from the first variable to the i'th
            # value for the second variable
            val = cell_vals1.GetTuple1(i) + cell_vals2.GetTuple1(i)
            # Store the value in the i'th value in the result
            res.SetTuple1(i, val)
        return res

py_filter = MyExpression
```

Let us start off by creating a Pseudocolor plot from the expression.

1. Copy the script into the *Python expression script* editor.
2. Click the *Apply* button.
3. Go to *Variables->MyExpression* to change the variable to the expression.

Now let us take a look at the script and see what each portion does.

The `__init__` method provides information about the expression, including

- That it inherits from `SimplePythonExpression`.
- The name of the expression.
- A description of the expression.
- A flag indicating that the output is not a point centered value.
- A flag that the output is a scalar.

```
    def __init__(self):
        SimplePythonExpression.__init__(self)
        self.name = "PythonExpression"
        self.description = "Add two scalar variables together"
        self.output_is_point_var  = False
        self.output_dimension = 1
```

The `modify_contract` method can be used to request special information for the expression from VisIt. In this case it is a no-op.

```
    def modify_contract(self,contract):
        pass
```

The `derive_variable` method performs the real work of the expression. It is passed the input `vtkDataSet` and the `domain_id`.

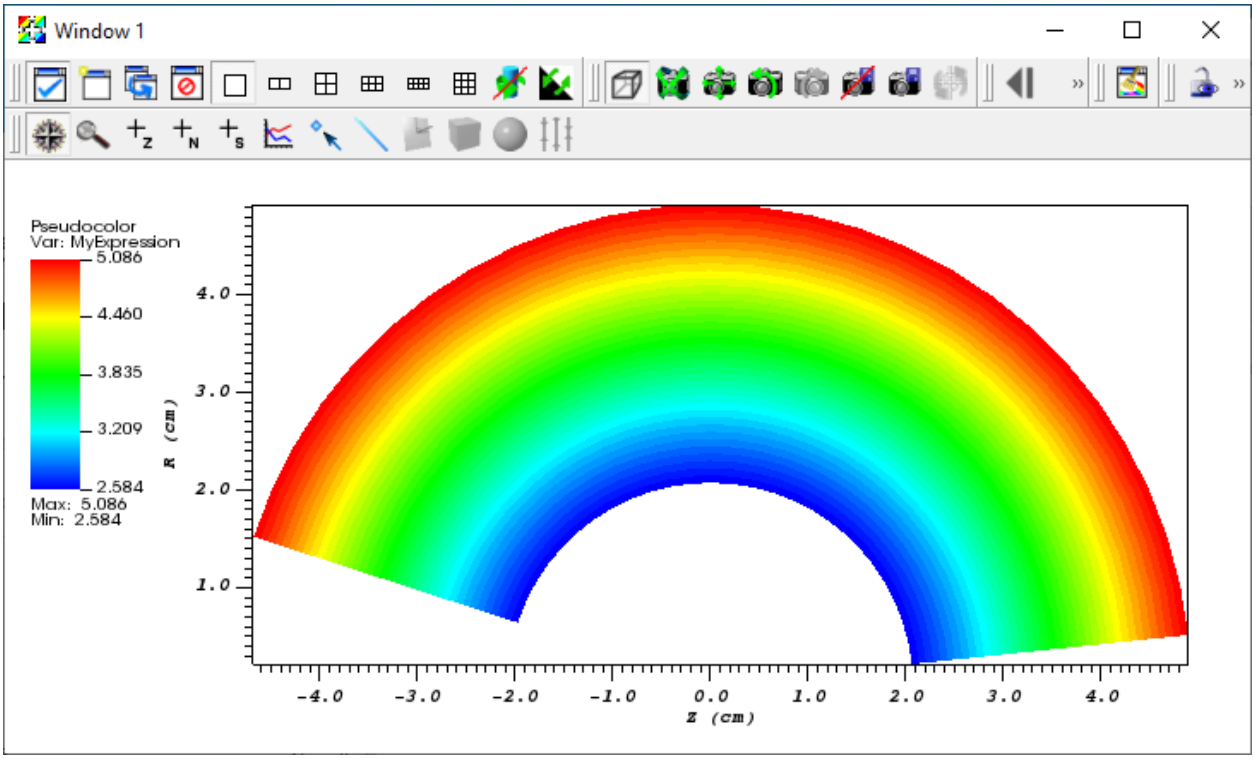


Fig. 6.180: The Pseudocolor plot of MyExpression

```
    def derive_variable(self,ds_in,domain_id):
        # ds_in is the input data set
```

The following lines get the `vtkDataArrays` for the cell values for the two variables and the number of cells.

```
        # Get the data array for the first variable
        cell_vals1 = ds_in.GetCellData().GetArray(self.input_var_names[0])
        # Get the data array for the second variable
        cell_vals2 = ds_in.GetCellData().GetArray(self.input_var_names[1])
        # Get the number of values in the variables
        ncells = ds_in.GetNumberOfCells()
```

The following lines set the output `vtkDataArray` to be an array of floats with 1 component and ncells values.

```
        # Create a scalar float array with ncells values for the result
        res = vtk.vtkFloatArray()
        res.SetNumberOfComponents(1)
        res.SetNumberOfTuples(ncells)
```

Now we loop over the cells, setting the output value for each cell.

```
        for i in range(ncells):
```

The following lines add the two variables for the current cell.

```
            # Add the i'th value from the first variable to the i'th
            # value for the second variable
            val = cell_vals1.GetTuple1(i) + cell_vals2.GetTuple1(i)
```

The following lines set the result value for the current cell.

```
            # Store the value in the i'th value in the result
            res.SetTuple1(i, val)
```

Once we have finished processing all the cells, we return the `vtkDataArray`.

```
        return res
```

Using your Python Expression with the CLI

The Python expression we just created can also be used with the **CLI**.

We will start by saving the script we just created.

1. Click *Save script* and save the script with the name `MyExpression.py`.

The following script will open `curv2d.silo` and create a Pseudocolor plot of the expression.

```
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")

DefinePythonExpression("MyExpression", ['d', 'p'], file="MyExpression.py")

AddPlot("Pseudocolor", "MyExpression")
DrawPlots()
```

6.11.4 Python Expression Example 2

This example operates on 2D meshes and takes the distance around the edges of each cell and multiplies it by the value of the cell. It demonstrates accessing the coordinates and topology of the mesh as well as a variable.

Here is the example script.

```
from math import sqrt
class MyExpression(SimplePythonExpression):
    def __init__(self):
        SimplePythonExpression.__init__(self)
        self.name = "PythonExpression"
        self.description = "Multiply the variable by sum of cell edge lengths in 2D"
        self.output_is_point_var  = False
        self.output_dimension = 1
    def modify_contract(self,contract):
        pass
    def derive_variable(self,ds_in,domain_id):
        # ds_in is the input data set
        # Get the data array for the variable
        cell_vals = ds_in.GetCellData().GetArray(self.input_var_names[0])
        # Get the number of values in the variable
        ncells = ds_in.GetNumberOfCells()
        # Create a scalar float array with ncells values for the result
        res = vtk.vtkFloatArray()
        res.SetNumberOfComponents(1)
        res.SetNumberOfTuples(ncells)
        for i in range(ncells):
            # Get the i'th cell
            cell = ds_in.GetCell(i)
```

(continues on next page)

(continued from previous page)

```
            # Get the number of edges in the cell
            nedges = cell.GetNumberOfEdges()
            # Sum up the lengths of the edges
            sum = 0.
            for j in range(nedges):
                # Get the j'th edge
                edge = cell.GetEdge(j)
                # Calculate the edge length from the end points
                pt1 = ds_in.GetPoint(edge.GetPointId(0))
                pt2 = ds_in.GetPoint(edge.GetPointId(1))
                len = sqrt((pt2[0] - pt1[0]) * (pt2[0] - pt1[0]) +
                           (pt2[1] - pt1[1]) * (pt2[1] - pt1[1]) +
                           (pt2[2] - pt1[2]) * (pt2[2] - pt1[2]))
                sum = sum + len
            # Multiply the sum by the i'th value of the variable
            sum *= cell_vals.GetTuple1(i)
            # Store the value in the i'th value in the result
            res.SetTuple1(i, sum)
        return res

py_filter = MyExpression
```

Let us start off by creating a Pseudocolor plot from the expression.

1. Copy the script into the *Python expression script* editor.
2. Click the *Apply* button.
3. Go to *Variables->MyExpression* to change the variable to the expression.

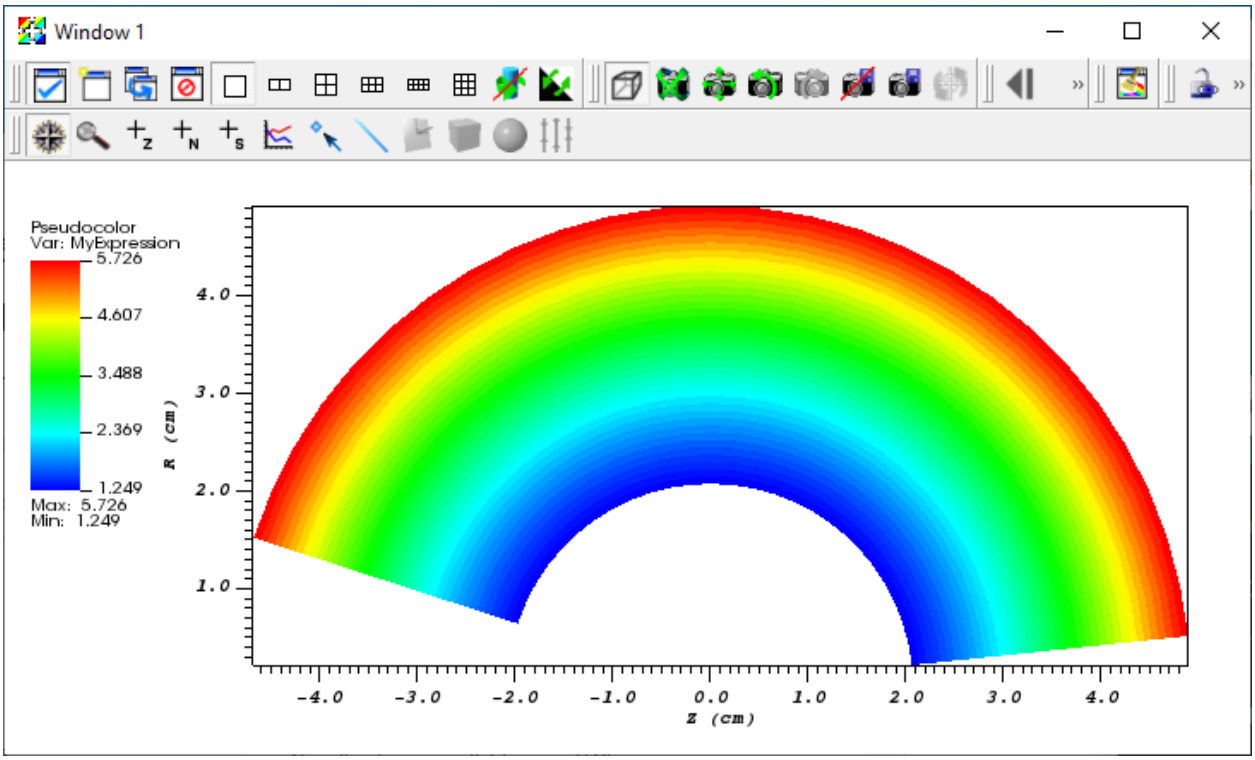


Fig. 6.181: The Pseudocolor plot of MyExpression

The __init__ and modify_contract methods are the same as the previous example, so we will only look at the derive_variable method.

```
    def derive_variable(self,ds_in,domain_id):
        # ds_in is the input data set
```

The following lines get the vtkDataArray for the cell values and the number of cells.

```
        # Get the data array for the variable
        cell_vals = ds_in.GetCellData().GetArray(self.input_var_names[0])
        # Get the number of values in the variable
        ncells = ds_in.GetNumberOfCells()
```

The following lines set the output vtkDataArray to be an array of floats with 1 component and ncells values.

```
        # Create a scalar float array with ncells values for the result
        res = vtk.vtkFloatArray()
        res.SetNumberOfComponents(1)
        res.SetNumberOfTuples(ncells)
```

Now we loop over the cells, setting the output value for each cell.

```
        for i in range(ncells):
```

The following lines get the current cell and the number of edges in the cell.

```
            # Get the i'th cell
            cell = ds_in.GetCell(i)
            # Get the number of edges in the cell
            nedges = cell.GetNumberOfEdges()
```

Now we loop over the edges, calculating the sum of the lengths of the edges.

```
            # Sum up the lengths of the edges
            sum = 0.
            for j in range(nedges):
```

We calculate the length of the edge from the 3D coordinates of the end points of the edge, which we add to the sum.

```
                # Get the j'th edge
                edge = cell.GetEdge(j)
                # Calculate the edge length from the end points
                pt1 = ds_in.GetPoint(edge.GetPointId(0))
                pt2 = ds_in.GetPoint(edge.GetPointId(1))
                len = sqrt((pt2[0] - pt1[0]) * (pt2[0] - pt1[0]) +
                           (pt2[1] - pt1[1]) * (pt2[1] - pt1[1]) +
                           (pt2[2] - pt1[2]) * (pt2[2] - pt1[2]))
                sum = sum + len
```

Once we have summed the lengths of the edges we multiply the sum by the cell value and set it in the result.

```
            # Multiply the sum by the i'th value of the variable
            sum *= cell_vals.GetTuple1(i)
            # Store the value in the i'th value in the result
            res.SetTuple1(i, sum)
```

Once we have finished processing all the cells, we return the vtkDataArray.

```
        return res
```

Using your Python Expression with the CLI

This Python expression can also be used with the **CLI**, just as the one in the first example, except the specification of the variables to use is slightly different. Since you are only passing a single variable you would use `("d")` for the list of variables.

```
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")

DefinePythonExpression("MyExpression", ("d"), file="MyExpression.py")

AddPlot("Pseudocolor", "MyExpression")
DrawPlots()
```

6.11.5 Using VTK in Python

The VTK Python interface mirrors the C++ interface.

To find out information on a particular VTK class, type the name of the class in your favorite search engine.

Here are links to some VTK classes that will be of most use to you.

- vtkCell
- vtkDataArray
- vtkDataSet
- vtkDoubleArray
- vtkFloatArray

6.12 Partitioning

This tutorial describes how to Partition meshes.

6.12.1 Partitioning overview

Partitioning meshes is commonly needed in order to evenly distribute work among many simulation ranks. VisIt provides the ability to partition meshes when exporting data in Blueprint format. Full M to N repartioning is supported for both structured and unstructured meshes.

The partitioning algorithms built into VisIt are fairly simple. The goal was not to provide sophisticated partiting algorithms but rather provide the mechanism to partition the data. In order to support more sophisticated partitioning algorithms, VisIt supports the ability to use a variable to specify the partitioning. This enables using an external repartitioning tool to define the partitions.

6.12.2 The built-in partitioning algorithm

Increasing the number of blocks

The built-in partitioning algorithm will split blocks when increasing the number of blocks. It will first sort the list of blocks from largest to smallest in terms of cell count. It will then split the largest block and insert the two new blocks into the list in the appropriate locations. It will continue splitting the largest block and inserting the new blocks into the list until the desired number of blocks is reached. In the case of an unstructured mesh it will split the block in half. In the case of a structured mesh it will split it in half along the longest dimension based on the number of cells.

Decreasing the number of blocks

The built-in partitioning algorithm will merge blocks when decreasing the number of blocks. It will first sort the list of blocks from largest to smallest in terms of cell count. It will then merge the two smallest blocks and insert the block into the list in the appropriate location. It will continue merging the two smallest blocks and inserting the new block into the list until the desired number of blocks is reached. In the case of an unstructured mesh it will merge matching points. By default, the points must match exactly to be merged. It is possible to set a merge tolerance that indicates how close two points need to be to be considered a match. In the case of a structured mesh it will try to maintain it as a structured mesh if possible. If it is not possible, it will convert them to an unstructured mesh and merge matching points as specified above.

6.12.3 An example of using the built-in partitioning algorithm

Here we will go through an example of partitioning a 3-dimensional multi-block unstructured mesh.

Let us start by opening a file and creating a plot.

1. Open the file `multi_ucd3d.silo`.
2. Create a Subset plot of `domains(mesh1)`.

Now we are ready to export the data.

3. Go to *File->Export database*.
4. This brings up the Export Database window.
5. Change the *File name* to `multi_ucd_repart`.
6. Change the *Export to* to *Blueprint*.
7. Click *Apply* and *Export*.

This will bring up the Export options for Blueprint writer window.

8. Change the *Operation* to *Partition*.
9. Change the *Partition target number of domains* to `2`.
10. Click *OK*.

Now let us plot the new partitioned mesh.

11. Click *Delete* in the main control window.
12. Open the file `multi_ucd3d_repart.cycle_000000.root`.
13. Create a Subset plot of `domains`.

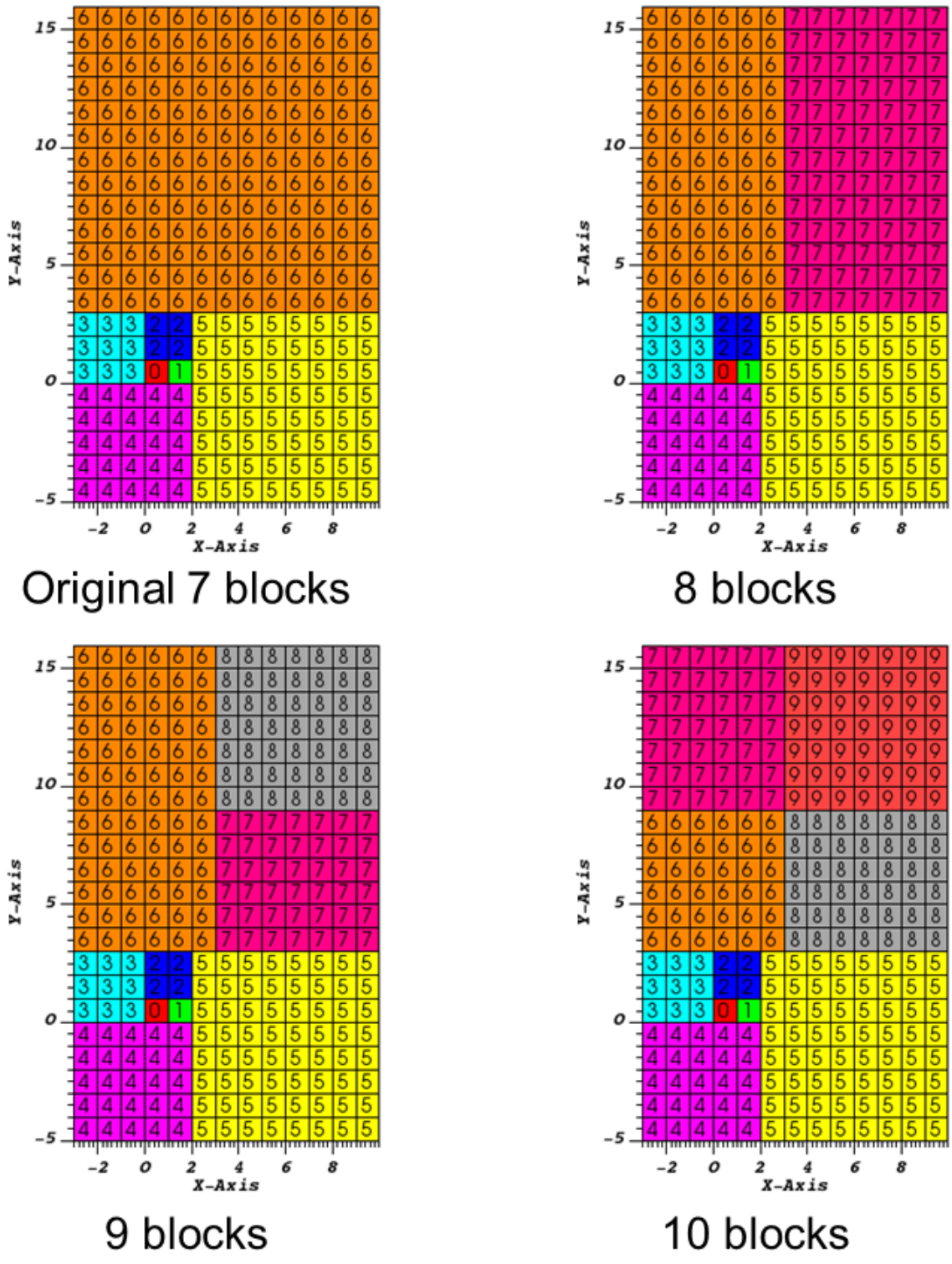


Fig. 6.182: Examples of increasing the number of blocks

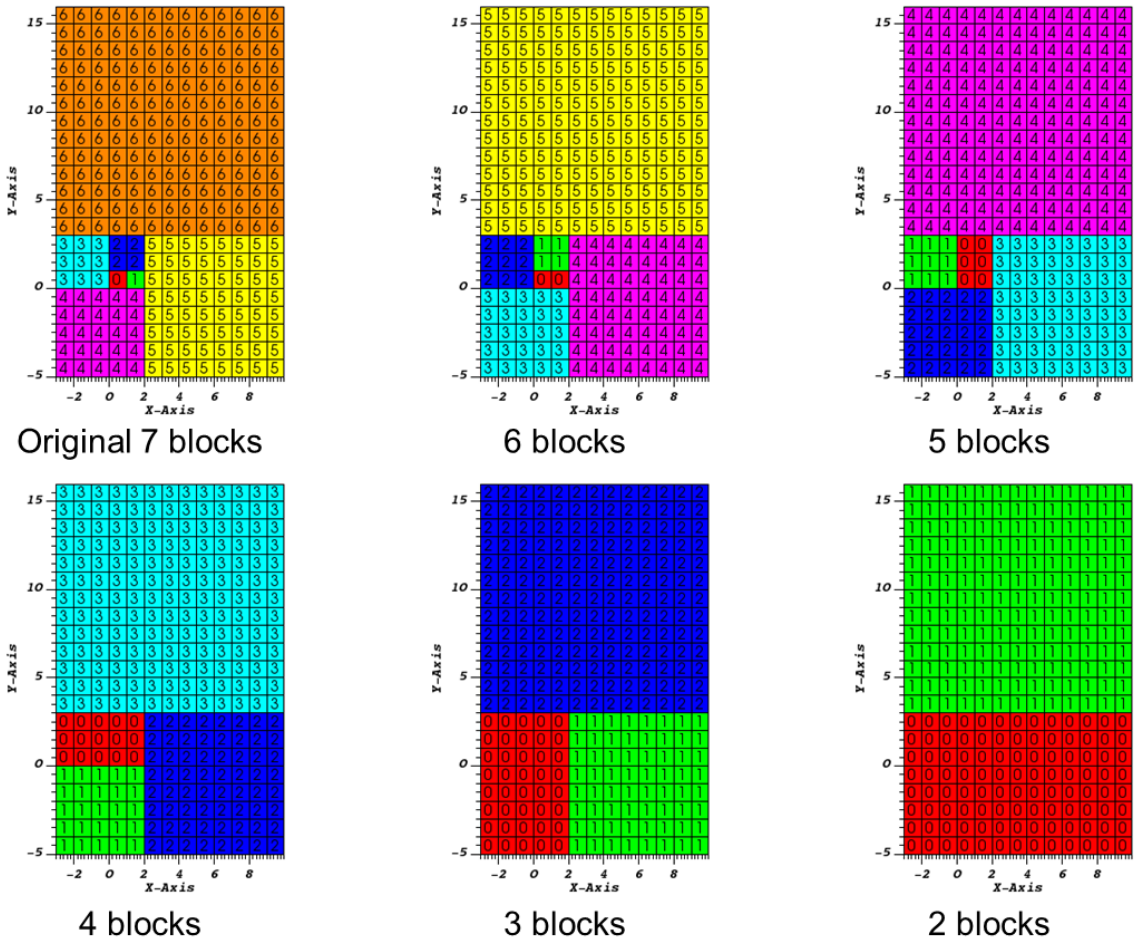


Fig. 6.183: Examples of decreasing the number of blocks

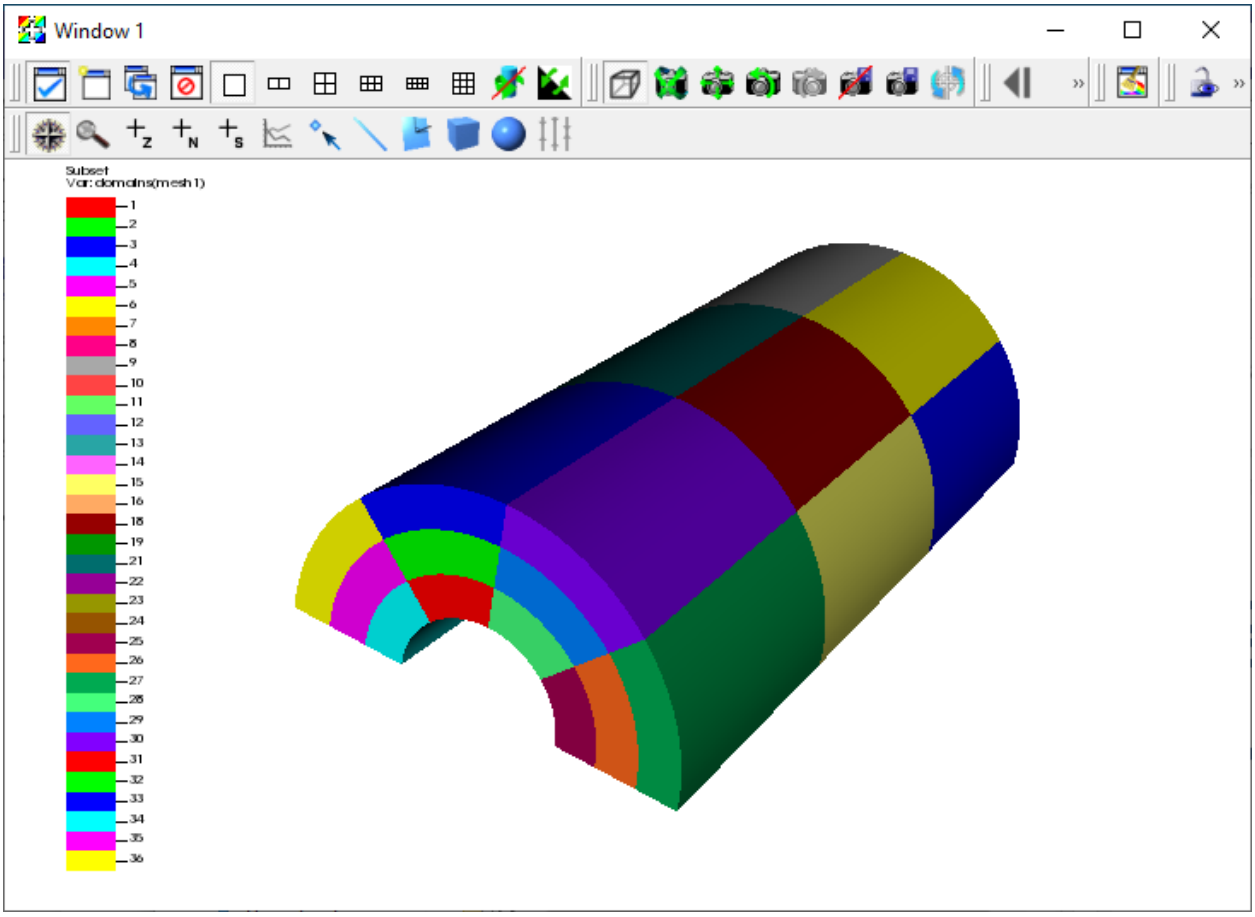


Fig. 6.184: Subset plot of the original 36 block mesh

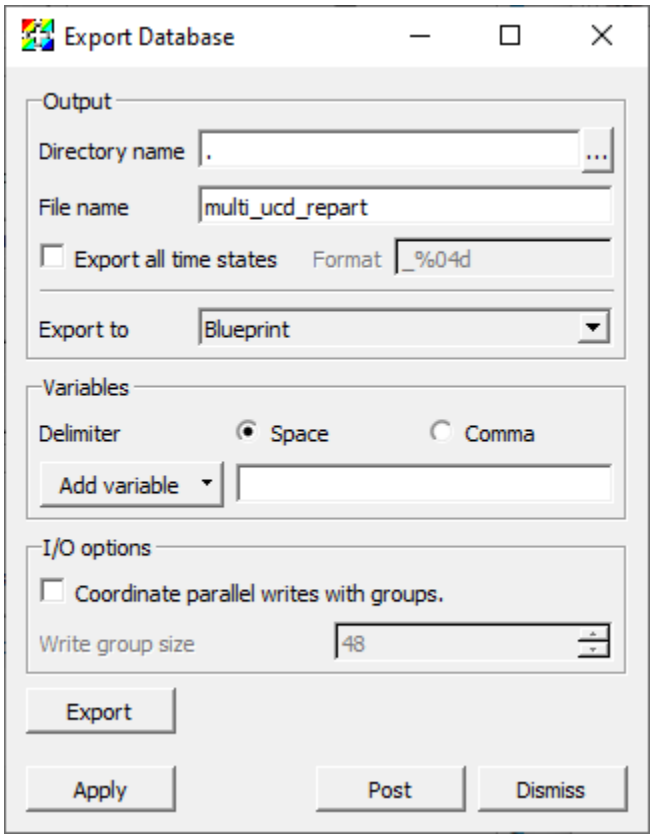


Fig. 6.185: The Export Database window.

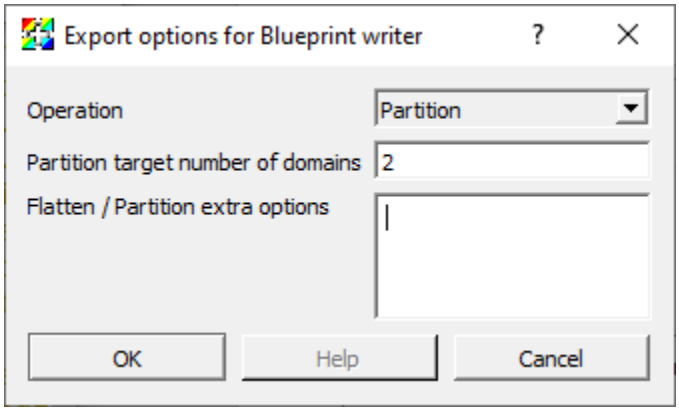


Fig. 6.186: The Blueprint writer window

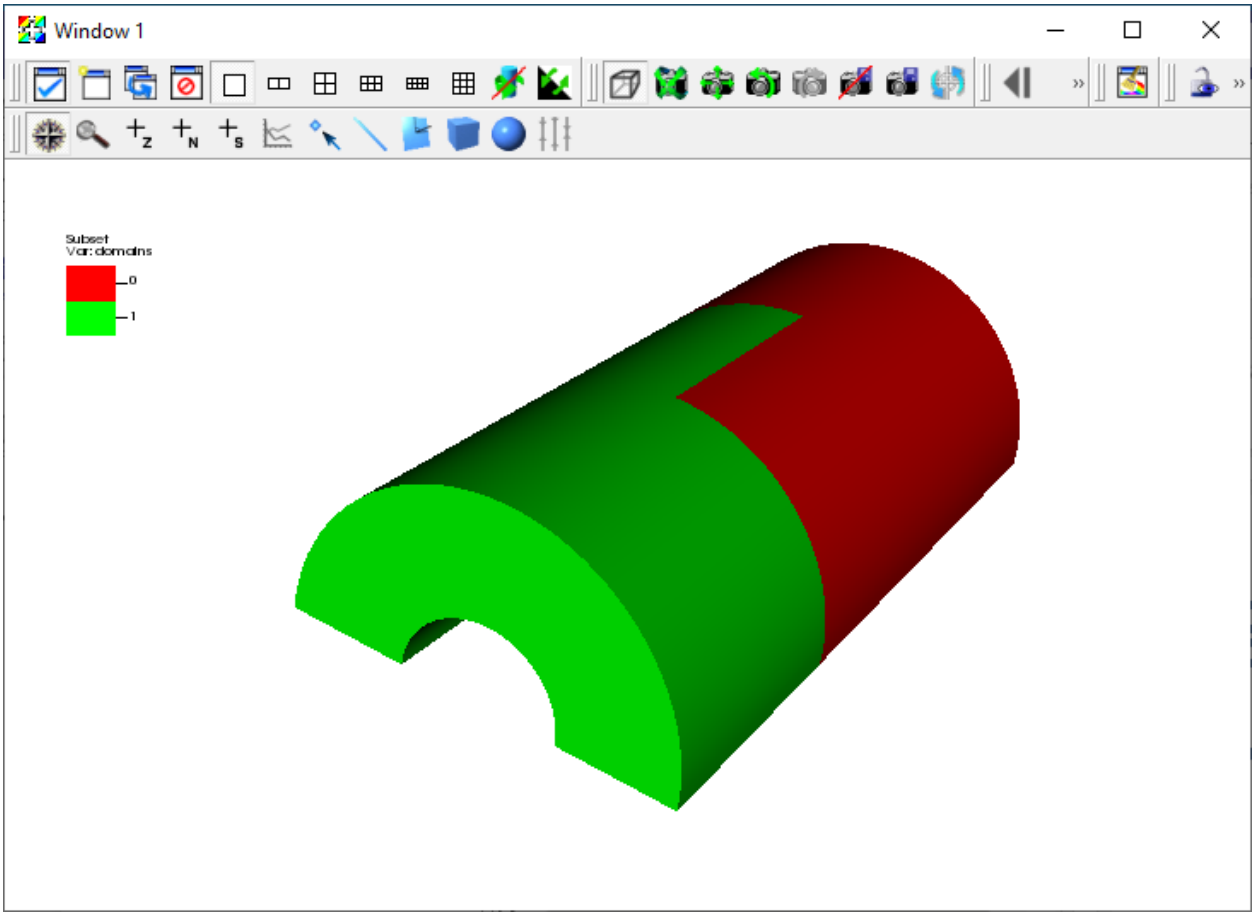


Fig. 6.187: Subset plot of the new partitioned 2 block mesh

6.12.4 An example of using a field to partition the mesh

Here we will go through an example of partitioning a 3-dimensional rectilinear mesh using a field. The field will be generated using a expression that has ones and zeros to partition the mesh into two arbitrary blocks.

Let us start by opening a file and creating a plot.

1. Open the file `noise.silo`.
2. Create a Pseudocolor plot of `hardyglobal`.

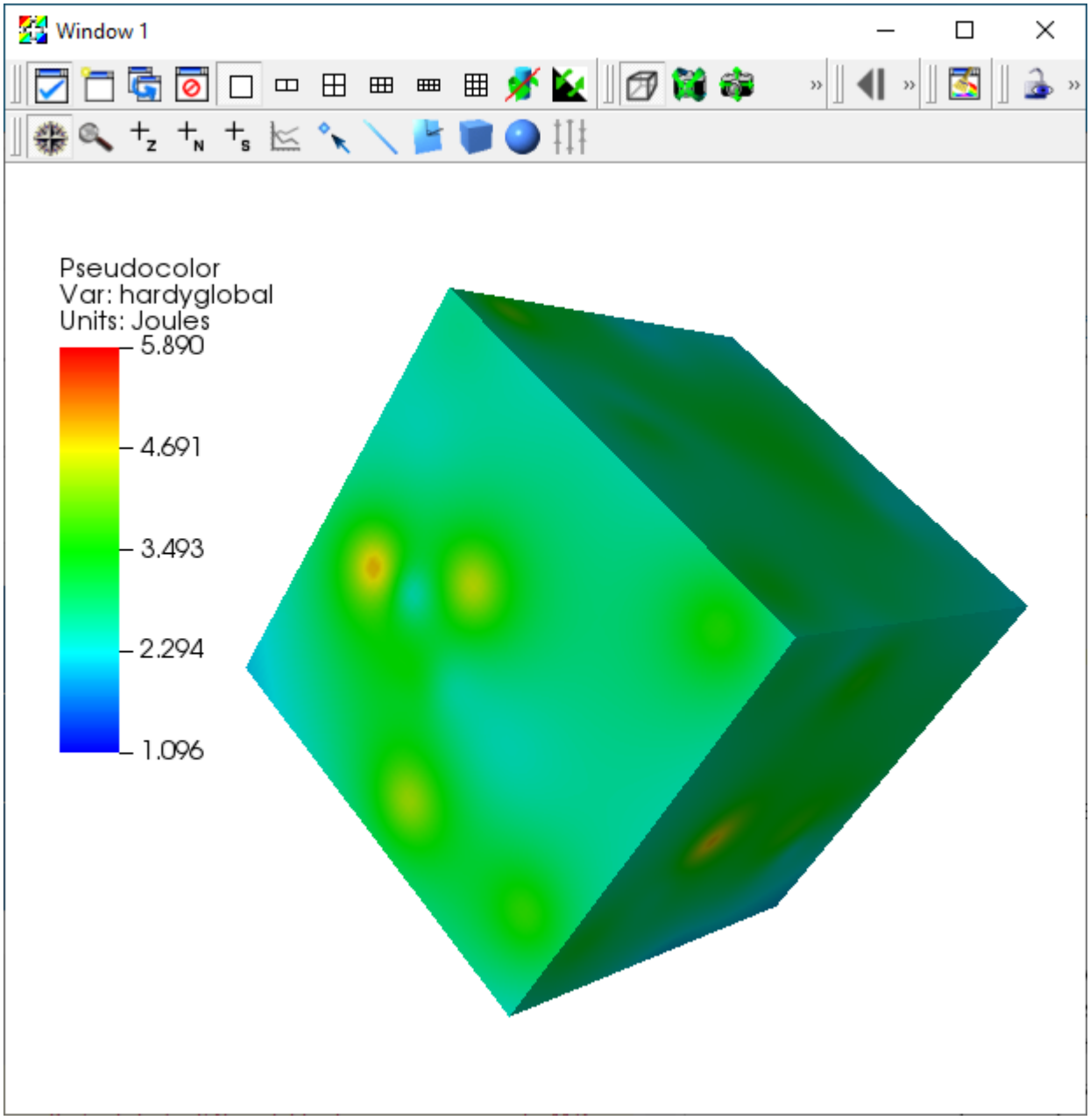


Fig. 6.188: Pseudocolor plot of hardyglobal

We will use the `hardyglobal` variable to create a variable that is one where the variable is greater than or equal to 2.8 and zero otherwise.

3. Go to *Controls->Expressions*.
4. Click *New* to create a new expression.
5. Change the *Name* text field to `part`.
6. Set the *Definition* to

```
if(ge(recenter(hardyglobal),zonal_constant(Mesh, 2.8)),zonal_constant(Mesh,1.),zonal_
↪constant(Mesh,0.))
```

7. Click *Apply* and *Dismiss*.

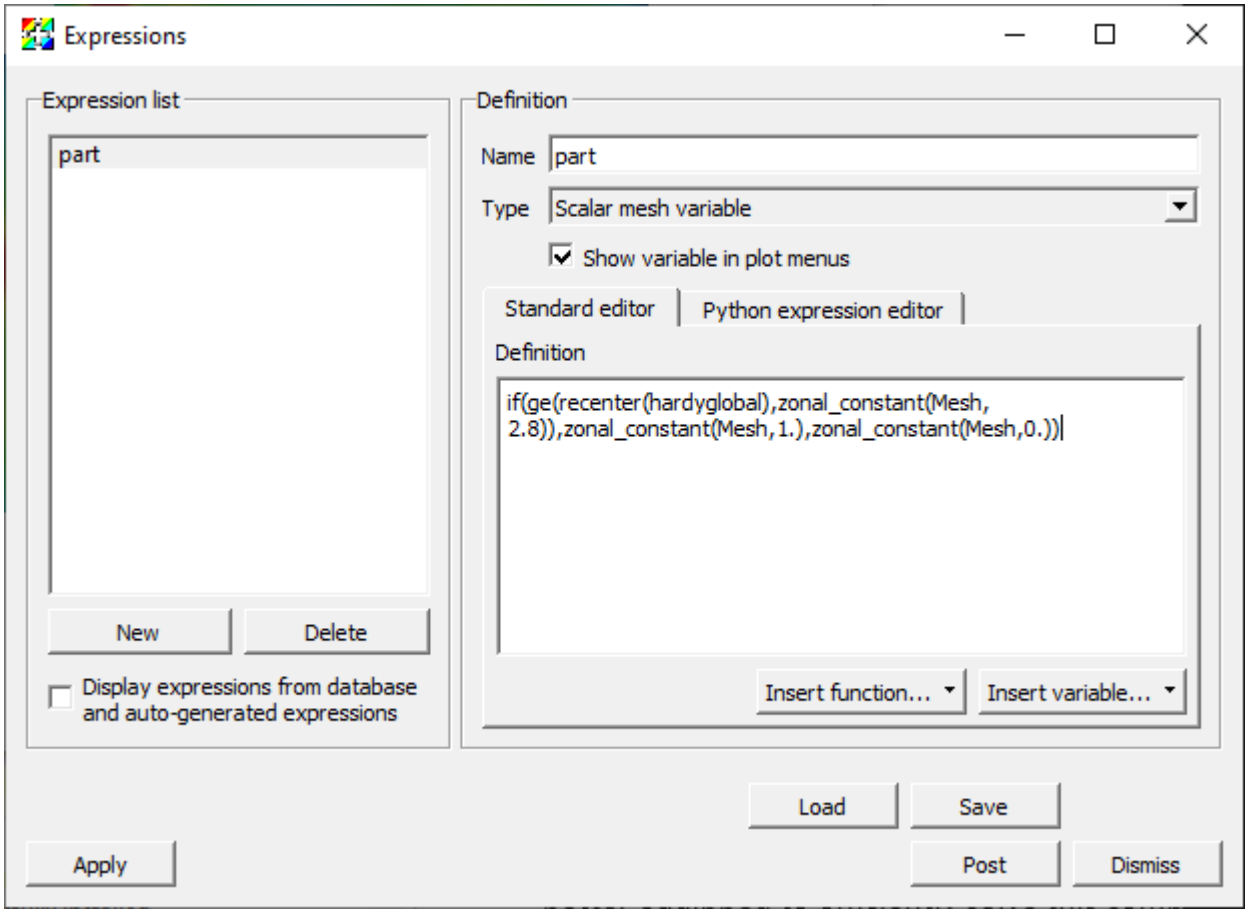


Fig. 6.189: Defining part in the Expressions window

Now let us look at the `part` variable.

8. Go to *Variables->part* to change the Pseudocolor variable to `part`.

Now we are ready to export the data. When partitioning using a variable the partitioning variable must either be the variable being plotted or listed as one of the variables in the *Variables* portion of the window. In this case we are plotting the partitioning variable, so we do not have to list it as an additional variable.

3. Go to *File->Export database*.
4. This brings up the Export Database window.
5. Change the *File name* to `noise_repart`.
6. Change the *Export to* to *Blueprint*.

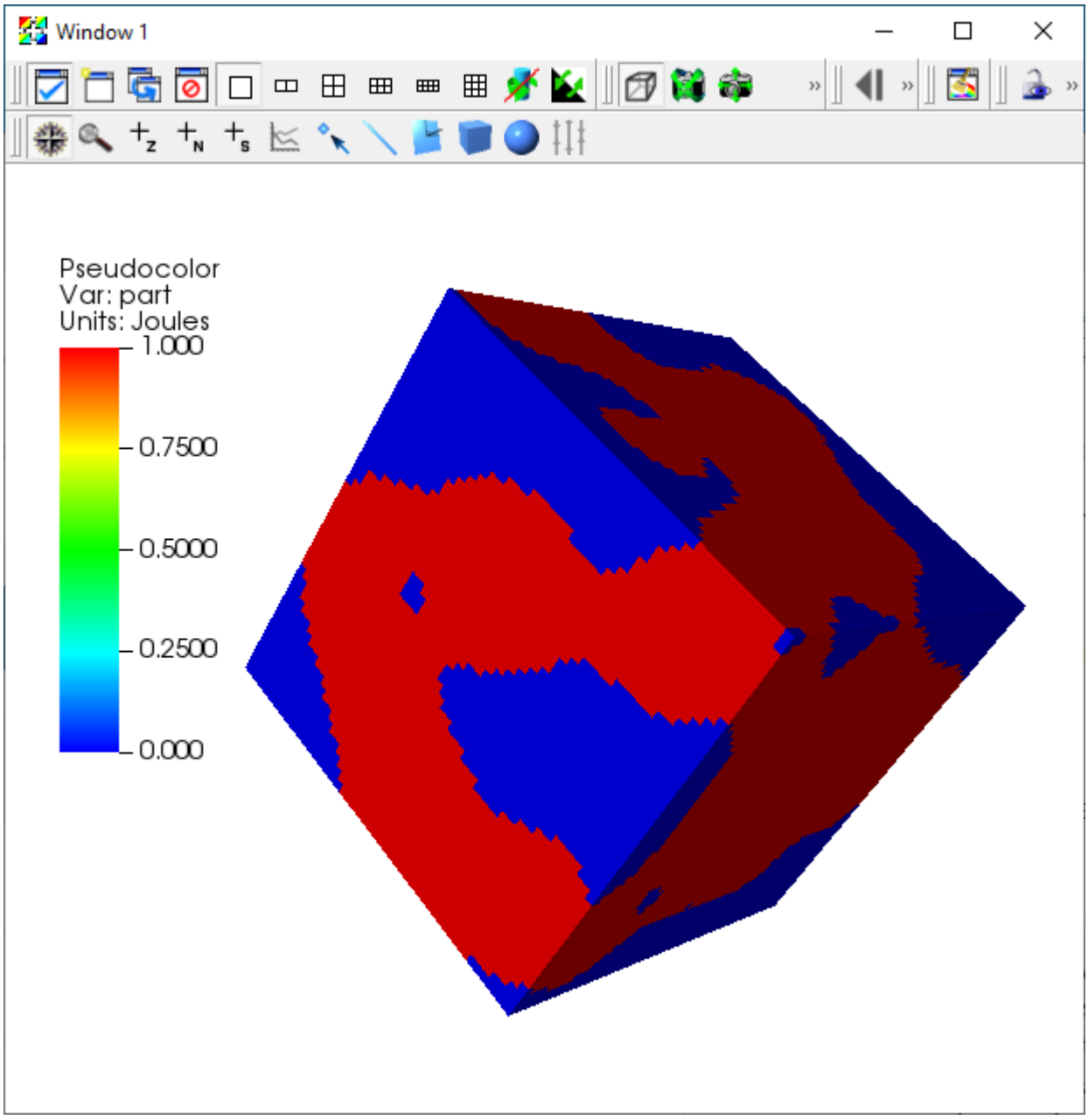


Fig. 6.190: Pseudocolor plot of part

7. Click *Apply* and *Export*.

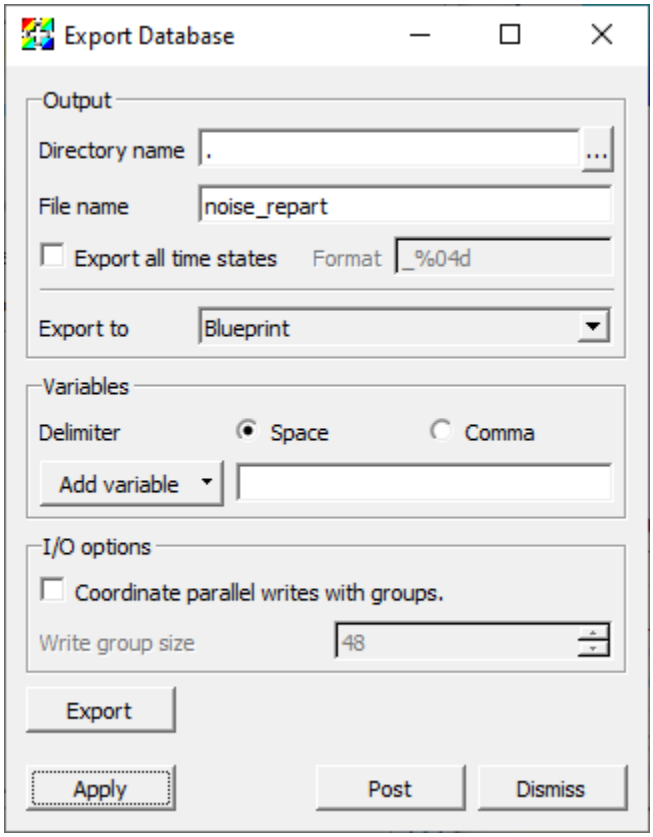


Fig. 6.191: The Export Database window.

This will bring up the Export options for Blueprint writer window.

8. Change the *Operation* to *Partition*.
9. Change the *Partition target number of domains* to `2`.
10. Change the *Flatten/Partition extra options* to

```
selections:
  -
   type: field
   domain_id: any
   field: part
```

Note that the options are in yaml and the indentation is critical.

Now let us plot the new partitioned mesh.

11. Click *Delete* in the main control window.
12. Open the file `noise_repart.cycle_000000.root`.
13. Create a Subset plot of `domains`.

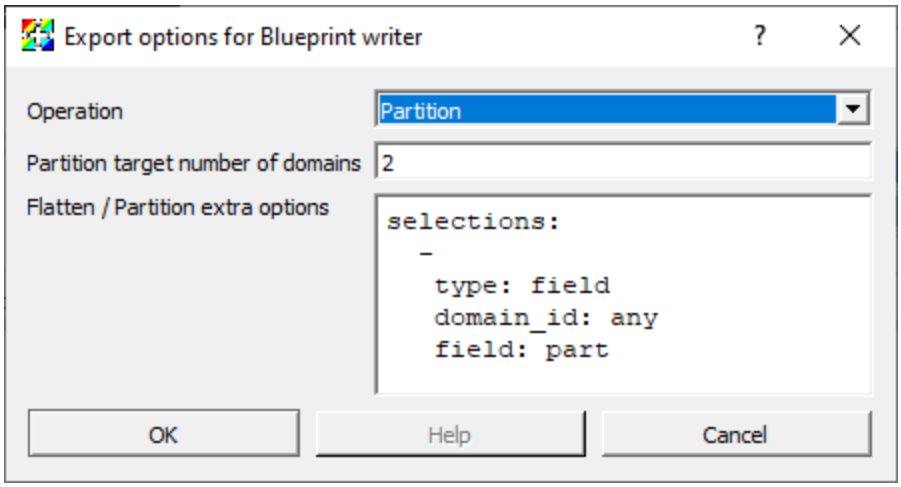


Fig. 6.192: The Blueprint writer window

6.13 Keyframe Animation

This tutorial describes how to use keyframe animations.

There is also a video that demonstrates keyframe animation creation using a different example than what is described here.

6.13.1 Keyframe animation overview

Keyframing is an advanced form of animation that allows you create animations where certain animation attributes such as view or plot attributes can change as the animation progresses. You can design an entire complex animation upfront by specifying a number of animation frames to be created and then you can tell VisIt which plots exist over the animation frames and how their time states map to the frames. You can also specify the plot attributes so they remain fixed over time or you can make individual plot and operator attributes evolve over time. With keyframing, you can make a plot fade out as the animation progresses, you can make a slice plane move, you can make the view slowly change, etc.

6.13.2 Using keyframe animation to do modal analysis

Modal analysis is the study of the dynamic properties of linear structures. It looks for the natural frequencies of a structure.

This tutorial demonstrates using animation to view the results of a modal analysis.

Creating the animation

Here we create an animation of elongating a globe using the Displace operator.

Let us start by opening a file, creating a plot, and applying the Displace operator.

1. Open the file `globe.silo`.
2. Create a Pseudocolor plot of `dx`.
3. Add the Displace operator.

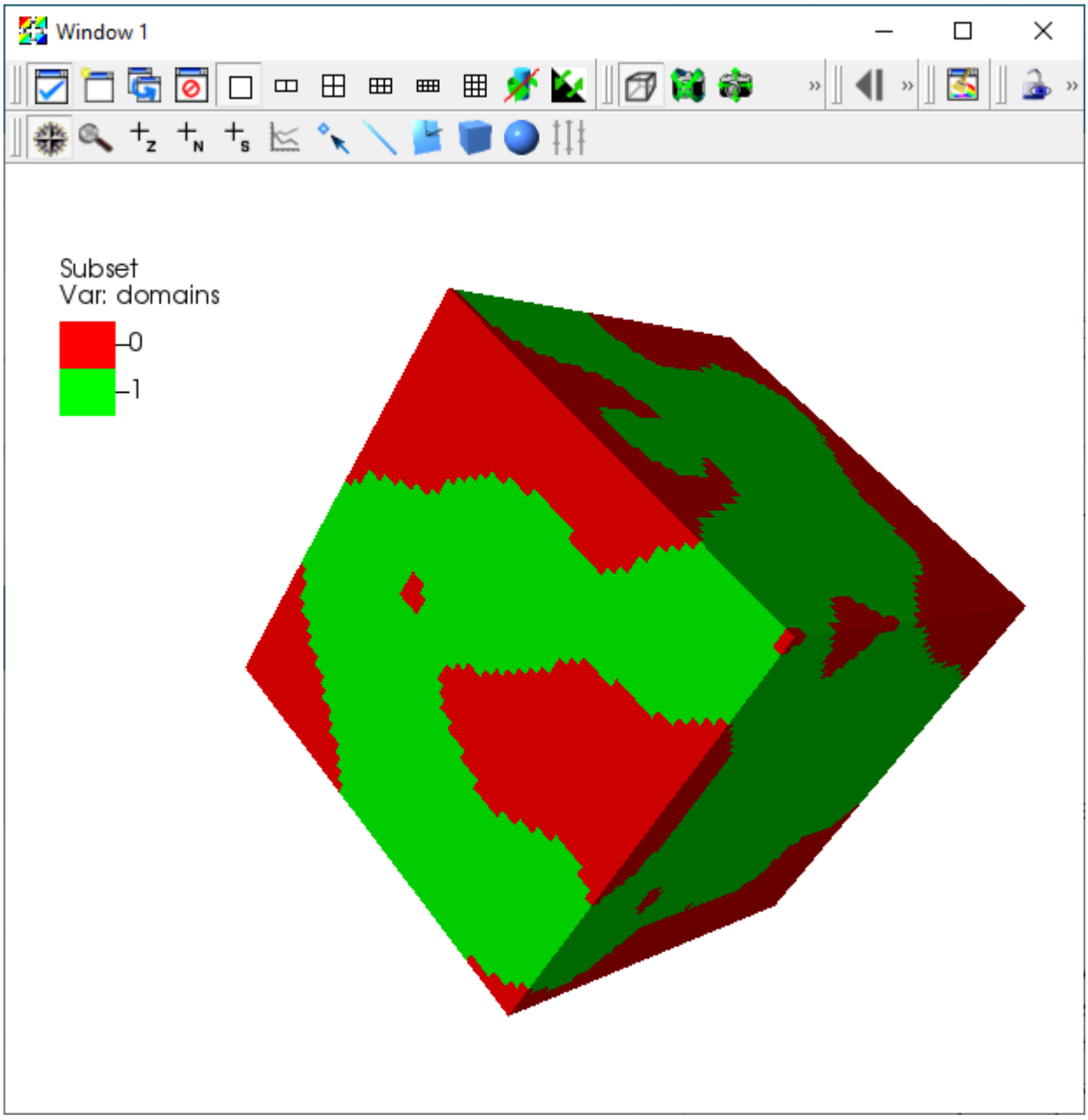


Fig. 6.193: Subset plot of the new partitioned 2 block mesh

4. Go to *OpAtts->Transforms->Displace* to bring up the Displace operator attributes window.

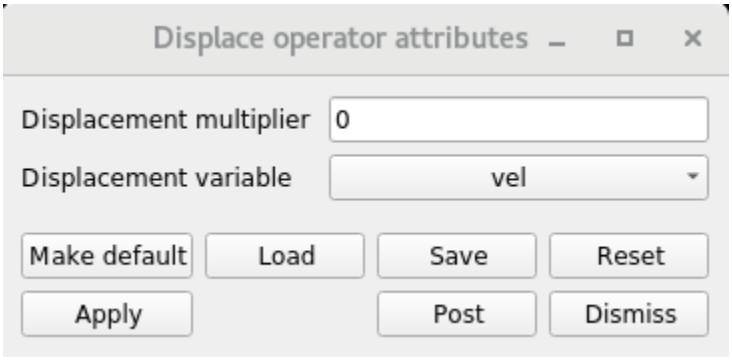


Fig. 6.194: The Displace operator attributes

5. Set the *Displacement multiplier* to `0`.
6. Set the *Displacement variable* to `vel`.
7. Click *Apply*.

We will now create a keyframe animation with 81 frames.

8. Go to *Controls->Keyframing*.
9. This will bring up the Keyframe Editor window.
10. Set the *Number of frames* to `81`.
11. Press *Enter*.
12. Toggle *Keyframing enabled* to on.
13. Click *Apply*.

We will now set the *Displacement multiplier* to range from `0` to `1` back to `0`.

14. Set the *Time* slider on the main control window to `40`.
15. Set the *Displacement multiplier* to `1`.
16. Click *Apply*.
17. This will create a Displace operator keyframe at frame 40.
18. Set the *Time* slider on the main control window to `80`.
19. Set the *Displacement multiplier* to `0`.
20. Click *Apply*.
21. This will create a Displace operator keyframe at frame 80.

Viewing the animation

Now that we have finished creating our animation we can view it.

1. Set the *Time* slider on the main control window to `0`.
2. Click the *Play* button.
3. This will play the animation once through.

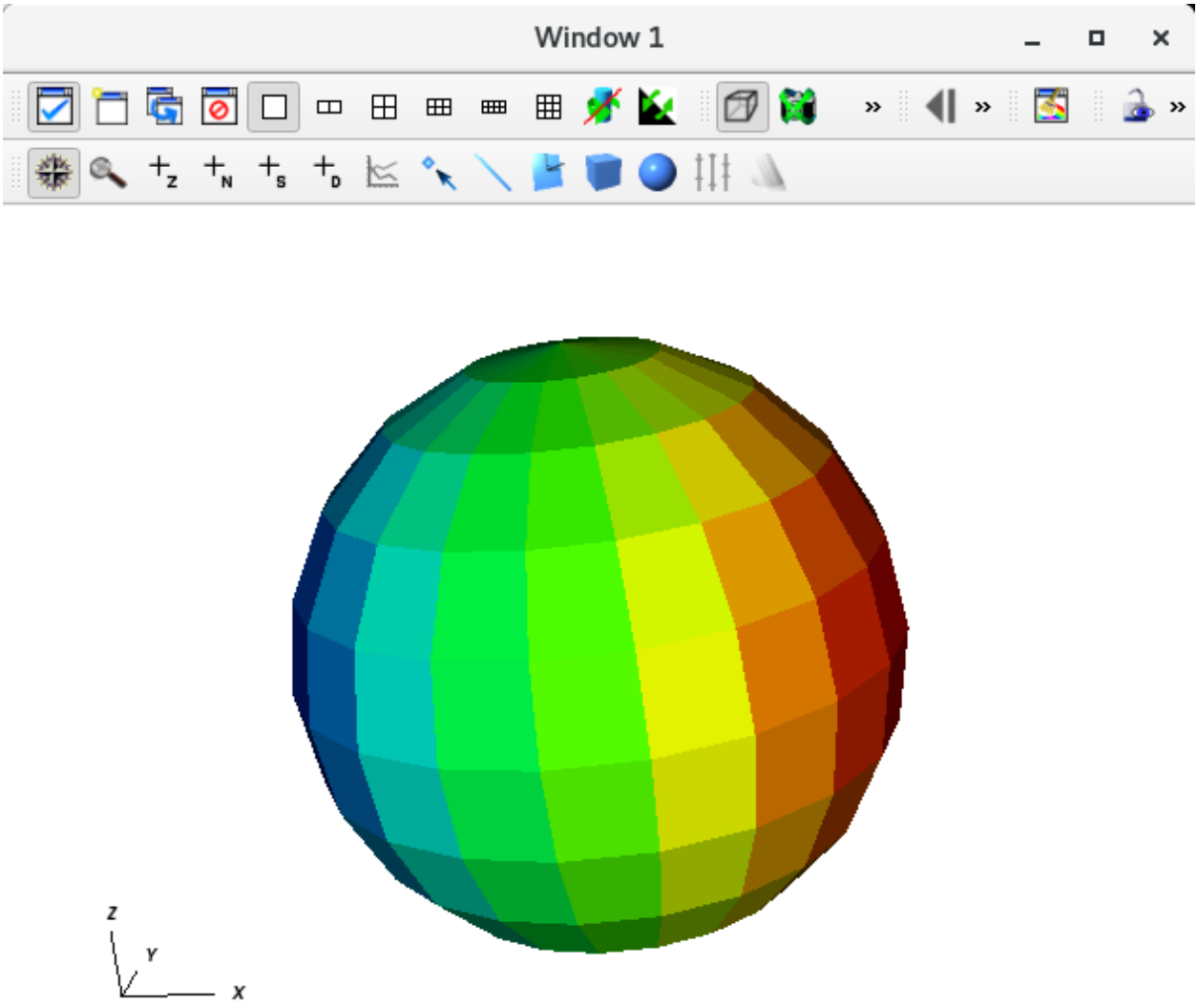


Fig. 6.195: Pseudocolor plot of dx with the Displace operator

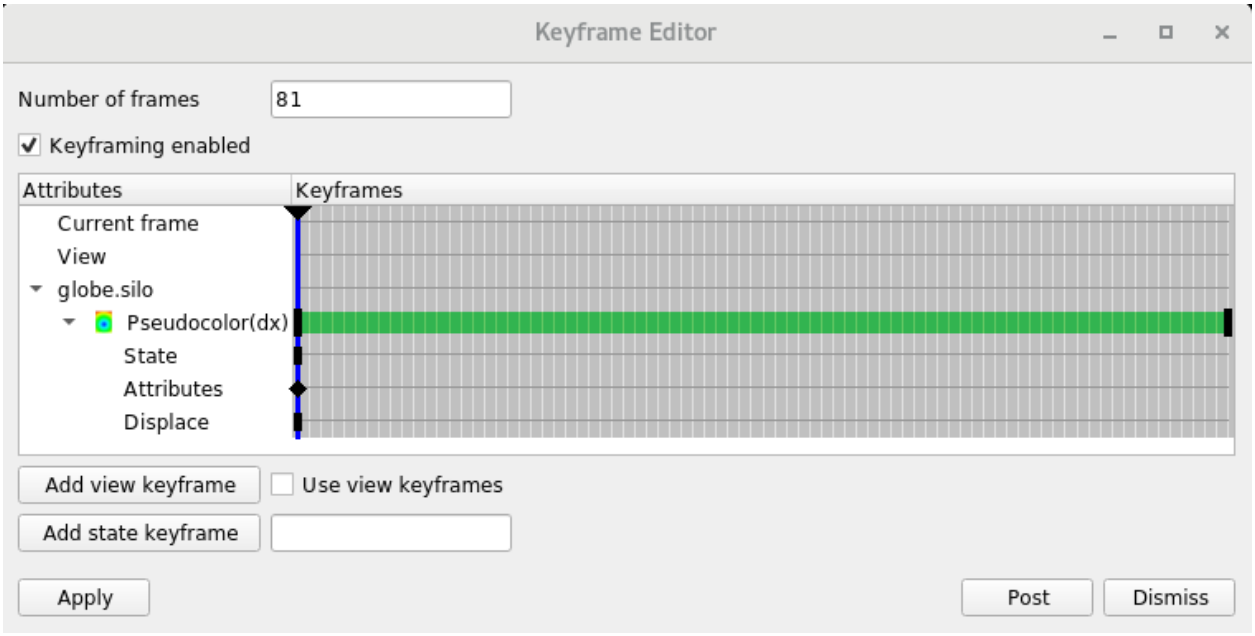


Fig. 6.196: The Keyframe Editor after creating the 81 frame animation

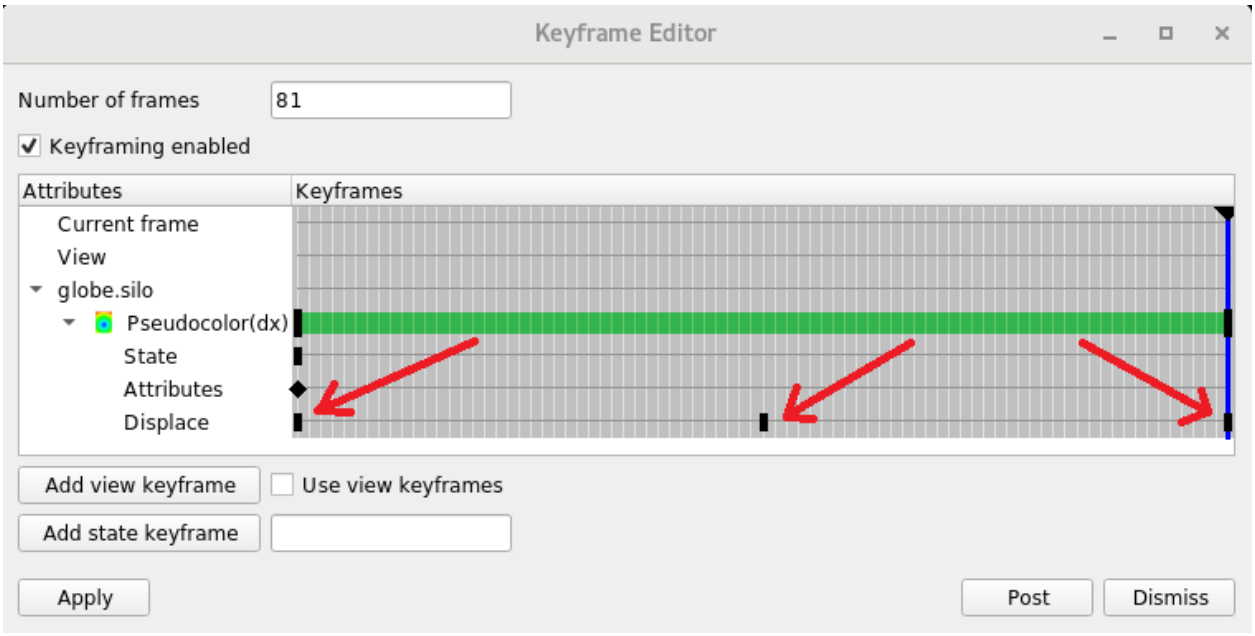


Fig. 6.197: The Keyframe Editor with the three Displace operator keyframes

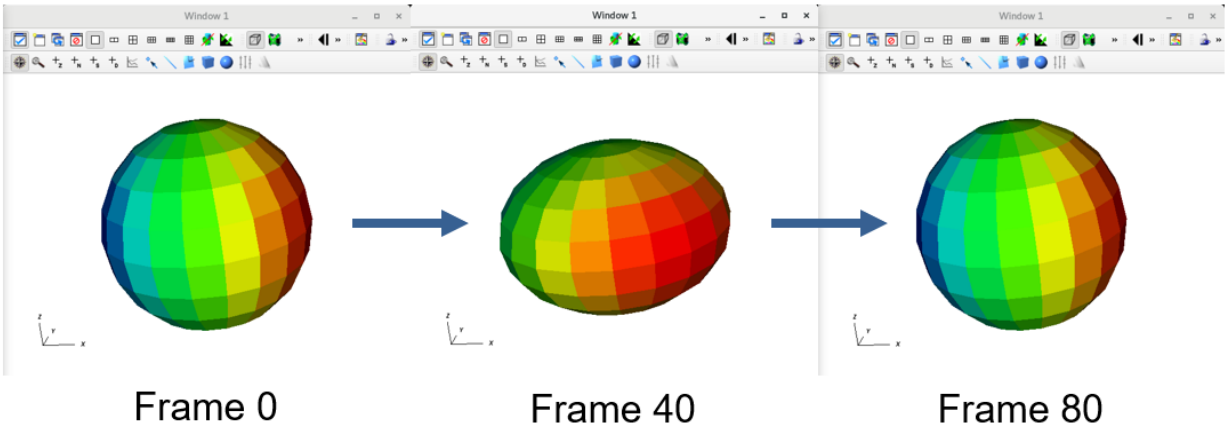


Fig. 6.198: Key frames in our animation

Saving the animation

We can also save the animation as a jpeg movie.

1. Go to *File->Save movie* to bring up the Save movie wizard.
2. Click on the *Next>* button to create a new simple movie.
3. Toggle *Specify movie size* to on and set the *Width* to `1024`.
4. Click on the -> button to move the definition to the *Output list*.
5. Click the *Next>* button to move to the next wizard pane.
6. Click the *Next>* button to save all the frames.
7. Click the *Next>* button to save the movie with the basename `movie`.
8. Click the *Next>* button to skip sending an e-mail when the movie is completed.
9. Click the *Finish* button to create the movie.

CHAPTER

SEVEN

JAVA CLIENT

VisIt's Java API allows Java applications to control VisIt's viewer. Java applications can launch VisIt's viewer window and control it programmatically, even replacing VisIt's GUI with a custom Java GUI.

Disclaimers:

> VisIt will not draw its plots inside of your Java application. VisIt's separate viewer window will be activated and controlled from your application. VisIt will not be able to plot data from your Java application without going through the file system.

The Java client for a particular version of VisIt is available as separate tarball on our downloads page. Look for `jvisit<version>.tar.gz`, e.g. `jvisit3.3.0.tar.gz`. Simply untar to the directory of your choosing to use it. Note that the tarball untars its contents to `.` and not its own directory. Its best to make a directory (e.g. `mkdir visit_java`), copy the tarball into that directory and untar it there.

7.1 Java Client API Docs

Java Client API docs

7.2 Building from source

To build the java client from a source build of VisIt, you must set the `VISIT_JAVA` CMake bool variable to `true` when configuring VisIt with CMake. Once the CMake configure step is complete, cd to the java subdirectory of the build directory and type `make`. This will compile the core parts of the Java interface. There are two other targets that can be built from the java directory: `main` and `pack`. The `main` target will build the class files for all the examples. The `pack` target causes a JAR file to be created and then packaged up with the Java source code, docs and example programs into a TAR file that can be shared. The created `visit.jar` file will be present in the build directory. The `pack` target will also build the example class files if not already built.

7.3 Examples

What follows are examples of how VisIt's Java client can be used. Source code for each example is available in the `java` subdirectory of the source code repository. There are several arguments that are common to all the examples:

`-stay`: tells the program to continue running after the `work()` method has completed instead of exiting.

`-path`: path to VisIt's bin directory, eg, `/usr/local/visit/3.3.0/bin`

`-datapath`: path to directory holding VisIt's silo example data (used by many of the examples) eg: `/usr/local/visit/3.3.0/data/`. **Note**: the trailing slash is **required**.

-dv: A shorthand for `-path ../bin`.

-nopty: Is forwarded to VisIt launcher to make VisIt prompt for passwords in the console/terminal from which Java was run.

All of the examples are available from the top (`.`) directory of the untarred `jvisit<version>.tar.gz`.

7.4 Java Classes Documentation

There is a `docs` subdirectory from the untarred `jvisit<version>.tar.gz` with a bunch of `.html` files. If you point a web browser at `index.html` there, you can find a lot of documentation on the various Java classes available and used in these examples.

7.5 Running the Examples

Assuming the current working directory is the directory where `jvisit<version>.tar.gz` was untarred, to run an example named `Example.class`, do the following...

```
java -cp .:visit.jar Example -stay -path /path/to/visit/bin -datapath /path/to/silo/
↪data/dir/
```

Note: The Java program to run, here `Example`, does not include the `.class` extension when it is specified on the command line to `java`.

The trailing slash for the `-datapath` argument is required.

All arguments *before* the name of the Java program to run are arguments to `java`.

All arguments *after* the name of the Java program to run are arguments to the program, or to VisIt. Any arguments not consumed by the Java program are forwarded to VisIt.

In client/server scenarios, if the above command line does not work or if the viewer seems to stall when connecting to the remote computer, try adding the `-nopty` argument to make VisIt prompt for passwords in the console from which Java was run. This should rarely be necessary.

7.5.1 Basic example

This program defines the `RunViewer` class, which serves as the base class for some of the other example Java programs. The `RunViewer` program does much of its initialization of the ViewerProxy, the main class for controlling VisIt's viewer, in its `run()` method. The actual VisIt-related work, however, is defined in the `work()` method and is overridden in subclasses to perform different VisIt functionality. This program's `work()` method opens the VisIt globe.silo database, sets some annotation properties, and makes a Pseudocolor plot and a Mesh plot. After the plots are drawn, the program changes the plot variables a couple of times and saves out images for each of the plotted variables.

Show/Hide Code for `RunViewer`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import llnl.visit.Axes3D;
```

(continues on next page)

(continued from previous page)

```
import llnl.visit.EventLoop;
import llnl.visit.ViewerProxy;
import llnl.visit.ColorAttribute;
import llnl.visit.AnnotationAttributes;

// ****************************************************************************
// Class: RunViewer
//
// Purpose:
//   This class implements an example program that shows how to use the
//   ViewerProxy class and control VisIt's viewer from Java.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Thu Aug 8 12:47:31 PDT 2002
//
// Modifications:
//   Brad Whitlock, Fri Nov 22 12:33:00 PDT 2002
//   Updated because of changes to AnnotationAttributes.
//
//   Brad Whitlock, Thu Dec 12 10:43:50 PDT 2002
//   Updated because of changes to color tables.
//
//   Brad Whitlock, Thu Mar 20 10:53:39 PDT 2003
//   I made it use port 5600.
//
//   Brad Whitlock, Mon Jun 6 10:18:10 PDT 2005
//   I added a little code to reduce CPU usage. I also made it use
//   GetDataPath to locate the data.
//
//   Brad Whitlock, Thu Jul 26 15:44:08 PST 2007
//   Added support for -dv instead of -vob.
//
//   Brad Whitlock, Mon Feb 25 11:07:24 PDT 2008
//   Changed to new ViewerProxy interface.
//
//   Kathleen Biagas, Tue Jan 14 08:45:32 MST 2014
//   Updated usage: changed -vob to -dv, added -datapath.
//
//   Justin Privitera, Wed May 18 11:25:46 PDT 2022
//   Changed *active* to *default* for everything related to color tables.
//
// ****************************************************************************

public class RunViewer
{
    public RunViewer()
    {
        viewer = new ViewerProxy();
    }

    public void run(String[] args)
    {
        // Pass command line options to the viewer viewer
        boolean stay = false;
        boolean sync = true;
```

(continues on next page)

(continued from previous page)

```
        boolean verbose = false;

        for(int i = 0; i < args.length; ++i)
        {
            if(args[i].equals("-stay"))
                stay = true;
            else if(args[i].equals("-dv"))
                viewer.SetBinPath("../bin");
            else if(args[i].equals("-sync"))
                sync = true;
            else if(args[i].equals("-async"))
                sync = false;
            else if(args[i].equals("-verbose"))
                verbose = true;
            else if(args[i].equals("-quiet"))
                verbose = false;
            else if(args[i].equals("-path") && ((i + 1) < args.length))
            {
                viewer.SetBinPath(args[i + 1]);
                ++i;
            }
            else if(args[i].equals("-datapath") && ((i + 1) < args.length))
            {
                viewer.SetDataPath(args[i + 1]);
                ++i;
            }
            else if(args[i].equals("-help"))
            {
                printUsage();
                return;
            }
            else
                viewer.AddArgument(args[i]);
        }

        // Set the viewer proxy's verbose flag.
        viewer.SetVerbose(verbose);

        // Try and open the viewer using the viewer proxy.
        if(viewer.Create(5600))
        {
            System.out.println("ViewerProxy opened the viewer.");

            // Set some viewer properties based on command line args.
            viewer.SetSynchronous(sync);

            // Show the windows.
            viewer.GetViewerMethods().ShowAllWindows();

            work(args);

            // If we have the -stay argument on the command line, keep the
            // viewer around so we can do stuff with it.
            if(stay)
                viewer.GetEventLoop().Execute();

            viewer.Close();
```

(continues on next page)

(continued from previous page)

```
        }
        else
            System.out.println("ViewerProxy could not open the viewer.");
    }

    protected void printUsage()
    {
        System.out.println("Options:");
        System.out.println("    -stay      Keeps the viewer around after it is done␣
↪processing commands.");
        System.out.println("    -dv        Runs the viewer located in ../bin.");
        System.out.println("    -sync      Runs the viewer in synchronous mode. This␣
↪is the default.");
        System.out.println("    -async     Runs the viewer in asynchronous mode.");
        System.out.println("    -verbose   Prints information to the console.");
        System.out.println("    -quiet     Prevents information from being printed to␣
↪the console.");
        System.out.println("    -path dir  Sets the directory that is searched for␣
↪the visit script.");
        System.out.println("    -datapath dir  Sets the directory that is searched␣
↪for the data files.");
        System.out.println("    -help      Displays options and exits program.");
    }

    protected void work(String[] args)
    {
        // Do a plot
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + "globe.silo
↪"))
        {
            viewer.GetViewerMethods().AddPlot("Pseudocolor", "u");
            viewer.GetViewerMethods().AddPlot("Mesh", "mesh1");
            viewer.GetViewerMethods().DrawPlots();
            viewer.GetViewerMethods().SaveWindow();

            // Change some annotation attributes.
            AnnotationAttributes a = viewer.GetViewerState().
↪GetAnnotationAttributes();
            a.SetBackgroundMode(AnnotationAttributes.BACKGROUNDMODE_GRADIENT);
            a.SetGradientBackgroundStyle(AnnotationAttributes.GRADIENTSTYLE_RADIAL);
            a.SetGradientColor1(new ColorAttribute(0,0,255));
            a.SetGradientColor2(new ColorAttribute(0,0,0));
            a.SetForegroundColor(new ColorAttribute(255,255,255));
            Axes3D a3d = new Axes3D(a.GetAxes3D());
            a3d.SetAxesType(Axes3D.AXES_STATICEDGES);
            a3d.SetVisible(true);
            a.SetAxes3D(a3d);
            a.Notify();
            viewer.GetViewerMethods().SetAnnotationAttributes();

            // Change the active color table
            viewer.GetViewerMethods().SetDefaultContinuousColorTable("rainbow");

            viewer.GetViewerMethods().SetActivePlot(0);
            viewer.GetViewerMethods().ChangeActivePlotsVar("v");
            viewer.GetViewerMethods().SaveWindow();
```

(continues on next page)

(continued from previous page)

```
            viewer.GetViewerMethods().ChangeActivePlotsVar("t");
            viewer.GetViewerMethods().SaveWindow();
        }
        else
        {
            System.out.println("Could not open the database!");
        }
    }

    public static void main(String args[])
    {
        RunViewer r = new RunViewer();
        r.run(args);
    }

    protected ViewerProxy viewer;
}
```

7.5.2 Controlling lighting

This example program is based on the `RunViewer` example program and it shows how to modify lighting in VisIt. The basic procedure is to obtain a handle to the state object that you want to modify, in this case, `LightList` and then modify the state object and call its Notify() method to send the changed object back to VisIt's viewer. Once the changed object has been sent back to VisIt's viewer, you call a method from ViewerMethods that tells VisIt to apply the sent object to its internal state.

Show/Hide Code for `TryLighting`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import java.lang.ArrayIndexOutOfBoundsException;
import llnl.visit.AttributeSubject;
import llnl.visit.ColorAttribute;
import llnl.visit.LightList;
import llnl.visit.LightAttributes;
import llnl.visit.SimpleObserver;
import llnl.visit.View3DAttributes;

import llnl.visit.plots.PseudocolorAttributes;

// ****************************************************************************
// Class: TryLighting
//
// Purpose:
//   This example program sets the view and turns on some colored lights. It
//   also shows how to observe the viewer's state objects so actions can be
//   performed when new state arrives.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Thu Aug 15 16:10:44 PST 2002
//
```

(continues on next page)

(continued from previous page)

```
// Modifications:
//   Brad Whitlock, Tue Sep 24 09:08:51 PDT 2002
//   I fixed the example so the lights work as intended.
//
//   Brad Whitlock, Thu Dec 12 10:44:31 PDT 2002
//   Updated because of changse to color table methods.
//
//   Eric Brugger, Wed Aug 27 09:06:38 PDT 2003
//   I modified it to use the new view interface.
//
//   Brad Whitlock, Mon Jun 6 17:25:34 PST 2005
//   I made it use GetDataPath to locate the data.
//
//   Brad Whitlock, Thu Jul 14 12:08:42 PDT 2005
//   I made it set the Pseudocolor plot atts's color table to "Default".
//
//   Brad Whitlock, Fri Sep 22 15:15:02 PST 2006
//   I fixed a problem with the brightness of lights 2,3 being set to 0.
//
//   Brad Whitlock, Mon Feb 25 11:07:24 PDT 2008
//   Changed to new ViewerProxy interface.
//
//   Justin Privitera, Wed May 18 11:25:46 PDT 2022
//   Changed *active* to *default* for everything related to color tables.
//
// ****************************************************************************

public class TryLighting extends RunViewer implements SimpleObserver
{
    public TryLighting()
    {
        super();
        doUpdate = true;

        // Make this object observe the light attributes.
        viewer.GetViewerState().GetLightList().Attach(this);
    }

    protected void work(String[] args)
    {
        // Try and open a database
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + "globe.silo
↪"))
        {
            viewer.GetViewerMethods().AddPlot("Pseudocolor", "w");

            // Set the pseudocolor attributes
            PseudocolorAttributes p = (PseudocolorAttributes)viewer.GetPlotAttributes(
↪"Pseudocolor");
            p.SetColorTableName("Default");
            p.SetOpacity(1.);
            p.Notify();
            viewer.GetViewerMethods().SetPlotOptions("Pseudocolor");
            viewer.GetViewerMethods().DrawPlots();

            // Set the colortable to one that has white at the bottom values.
            viewer.GetViewerMethods().SetDefaultContinuousColorTable("calewhite");
```

(continues on next page)

(continued from previous page)

```
        // Set the view
        View3DAttributes v = viewer.GetViewerState().GetView3DAttributes();
        v.SetViewNormal(0.456808, 0.335583, 0.823839);
        v.SetFocus(-0.927295, -1.22113, 1.01159);
        v.SetViewUp(-0.184554, 0.941716, -0.281266);
        v.SetParallelScale(15.7041);
        v.SetNearPlane(-34.641);
        v.SetFarPlane(34.641);
        v.Notify();
        viewer.GetViewerMethods().SetView3D();

        LightList ll = viewer.GetViewerState().GetLightList();
        ll.SetAllEnabled(false);

        // Create a red light
        System.out.println("Setting up red light.");
        LightAttributes newLight1 = new LightAttributes();
        newLight1.SetType(LightAttributes.LIGHTTYPE_OBJECT);
        newLight1.SetDirection(0,0,-1);
        newLight1.SetColor(new ColorAttribute(255,0,0));
        newLight1.SetBrightness(1.);
        newLight1.SetEnabledFlag(true);
        ll.SetLight0(newLight1);
        ll.Notify();
        viewer.GetViewerMethods().SetLightList();
        viewer.GetViewerMethods().SaveWindow();

        // Create a green light
        System.out.println("Setting up green light.");
        LightAttributes newLight2 = new LightAttributes();
        newLight2.SetType(LightAttributes.LIGHTTYPE_OBJECT);
        newLight2.SetDirection(-1,0,0);
        newLight2.SetColor(new ColorAttribute(0,255,0));
        newLight2.SetBrightness(1.);
        newLight2.SetEnabledFlag(true);
        ll.SetLight1(newLight2);
        ll.Notify();
        viewer.GetViewerMethods().SetLightList();
        viewer.GetViewerMethods().SaveWindow();

        // Create a blue light
        System.out.println("Setting up blue light.");
        LightAttributes newLight3 = new LightAttributes();
        newLight3.SetType(LightAttributes.LIGHTTYPE_OBJECT);
        newLight3.SetDirection(0,-1,0);
        newLight3.SetColor(new ColorAttribute(0,0,255));
        newLight3.SetBrightness(1.);
        newLight3.SetEnabledFlag(true);
        ll.SetLight2(newLight3);
        ll.Notify();
        viewer.GetViewerMethods().SetLightList();
        viewer.GetViewerMethods().SaveWindow();
    }
    else
        System.out.println("Could not open the database!");
}
```

(continues on next page)

(continued from previous page)

```
    public void Update(AttributeSubject s)
    {
        LightList ll = (LightList)s;
        printLights(ll);
    }

    public void SetUpdate(boolean val) { doUpdate = val; }
    public boolean GetUpdate() { return doUpdate; }

    protected void printLights(LightList ll)
    {
        printLight(0, ll.GetLight0());
        printLight(1, ll.GetLight1());
        printLight(2, ll.GetLight2());
        printLight(3, ll.GetLight3());
        printLight(4, ll.GetLight4());
        printLight(5, ll.GetLight5());
        printLight(6, ll.GetLight6());
        printLight(7, ll.GetLight7());
        System.out.println("");
    }

    protected void printLight(int index, LightAttributes l)
    {
        System.out.println("Light["+index+"] = {");
        System.out.println("    enabled = "+l.GetEnabledFlag());
        int t = l.GetType();
        double[] d = l.GetDirection();
        switch(t)
        {
        case LightAttributes.LIGHTTYPE_AMBIENT:
            System.out.println("    type = AMBIENT");
            break;
        case LightAttributes.LIGHTTYPE_OBJECT:
            System.out.println("    type = OBJECT");
            System.out.println("    direction = {"+d[0]+", "+d[1]+", "+d[2]+"}");
            break;
        case LightAttributes.LIGHTTYPE_CAMERA:
            System.out.println("    type = CAMERA");
            System.out.println("    direction = {"+d[0]+", "+d[1]+", "+d[2]+"}");
        }
        ColorAttribute c = l.GetColor();
        System.out.println("    color = {"+c.Red()+", "+c.Green()+", "+c.Blue()+"}");
        System.out.println("    brightness = "+l.GetBrightness());
        System.out.println("}");
    }

    public static void main(String args[])
    {
        TryLighting r = new TryLighting();
        r.run(args);
    }

    private boolean doUpdate;
}
```

7.5.3 Performing queries

This example program shows how to use some of VisIt's query capabilities to perform picks and lineouts.

Show/Hide Code for `TryQuery`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import java.lang.ArrayIndexOutOfBoundsException;
import java.util.Vector;
import llnl.visit.AttributeSubject;
import llnl.visit.SimpleObserver;
import llnl.visit.QueryAttributes;

// ****************************************************************************
// Class: TryQuery
//
// Purpose:
//   This example program does a plot and queries some values in it.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Tue Oct 1 12:51:29 PDT 2002
//
// Modifications:
//   Brad Whitlock, Thu Dec 12 10:44:31 PDT 2002
//   Updated because of changse to color table methods.
//
//   Brad Whitlock, Thu Jan 2 16:05:48 PST 2003
//   Changed because of Lineout method interface change.
//
//   Brad Whitlock, Mon Jun 6 17:25:34 PST 2005
//   I made it use GetDataPath to locate the data.
//
//   Brad Whitlock, Mon Feb 25 11:07:24 PDT 2008
//   Changed to new ViewerProxy interface.
//
//   Justin Privitera, Wed May 18 11:25:46 PDT 2022
//   Changed *active* to *default* for everything related to color tables.
//
// ****************************************************************************

public class TryQuery extends RunViewer implements SimpleObserver
{
    public TryQuery()
    {
        super();
        doUpdate = true;

        // Make this object observe the query and pick attributes.
        viewer.GetViewerState().GetQueryAttributes().Attach(this);
        viewer.GetViewerState().GetPickAttributes().Attach(this);
    }

    protected void work(String[] args)
```

(continues on next page)

(continued from previous page)

```
    {
        // Try and open a database
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + "curv2d.silo
↪"))
        {
            viewer.GetViewerMethods().AddPlot("Mesh", "curvmesh2d");
            viewer.GetViewerMethods().AddPlot("Pseudocolor", "d");
            viewer.GetViewerMethods().DrawPlots();

            // Set the colortable to one that has white at the bottom values.
            viewer.GetViewerMethods().SetDefaultContinuousColorTable("calewhite");

            // Create the variable list.
            Vector vars = new Vector();
            vars.addElement(new String("default"));

            // Do some picks.
            viewer.GetViewerMethods().Pick(300, 300, vars);
            viewer.GetViewerMethods().Pick(450, 350, vars);
            viewer.GetViewerMethods().Pick(600, 400, vars);

            // Do some lineouts.
            viewer.GetViewerMethods().Lineout(-4.01261, 1.91818, 2.52975, 3.78323,␣
↪vars);
            viewer.GetViewerMethods().SetActiveWindow(1);
            viewer.GetViewerMethods().Lineout(-3.89903, 1.79309, 2.91593, 3.40794,␣
↪vars);

            // Change the window layout.
            viewer.GetViewerMethods().SetWindowLayout(2);
        }
        else
            System.out.println("Could not open the database!");
    }

    public void Update(AttributeSubject s)
    {
        System.out.println(s.toString(""));
    }

    public void SetUpdate(boolean val) { doUpdate = val; }
    public boolean GetUpdate() { return doUpdate; }

    public static void main(String args[])
    {
        TryQuery r = new TryQuery();
        r.run(args);
    }

    private boolean doUpdate;
}
```

7.5.4 Getting metadata

This program shows how to query metadata for a database and print it to the console. In real applications, of course, you'd do something more constructive with the metadata object such as populate variable menus in a GUI.

Show/Hide Code for `GetMetaData`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import java.lang.ArrayIndexOutOfBoundsException;
import llnl.visit.avtDatabaseMetaData;

// ****************************************************************************
// Class: GetMetaData
//
// Purpose:
//   This example program opens a database and gets the metadata, printing
//   it to the console.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Mon Feb 25 12:01:43 PDT 2008
//
// Modifications:
//
// ****************************************************************************

public class GetMetaData extends RunViewer
{
    public GetMetaData()
    {
        super();
    }

    protected void work(String[] args)
    {
        // Try and open a database
        if(viewer.GetViewerMethods().RequestMetaData(viewer.GetDataPath() + "noise.
↪silo",0))
        {
            avtDatabaseMetaData md = viewer.GetViewerState().GetDatabaseMetaData();
            System.out.print(md.toString());
        }
        else
            System.out.println("Could not get the metadata for the database!");
    }

    public static void main(String args[])
    {
        GetMetaData r = new GetMetaData();
        r.run(args);
    }
}
```

7.5.5 Controlling annotations

This example program shows how to control various annotation objects via the Java API.

Show/Hide Code for `TryAnnotations`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import java.lang.ArrayIndexOutOfBoundsException;
import java.util.Vector;
import llnl.visit.Axes3D;
import llnl.visit.View3DAttributes;
import llnl.visit.AnnotationAttributes;
import llnl.visit.AnnotationObject;
import llnl.visit.AnnotationObjectList;
import llnl.visit.ColorAttribute;
import llnl.visit.SaveWindowAttributes;

// ****************************************************************************
// Class: TryAnnotations
//
// Purpose:
//   This example program shows how to create annotation objects and set
//   their properties.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Wed Feb 27 09:38:43 PDT 2008
//
// Modifications:
//   Brad Whitlock, Wed Jun 13 17:00:08 PDT 2012
//   Set some legend options.
//
// ****************************************************************************

public class TryAnnotations extends RunViewer
{
    public TryAnnotations()
    {
        super();
    }

    protected void SetCustomDefaultAnnotations()
    {
        // Change some annotation attributes.
        AnnotationAttributes a = viewer.GetViewerState().GetAnnotationAttributes();
        a.SetBackgroundMode(AnnotationAttributes.BACKGROUNDMODE_GRADIENT);
        a.SetGradientBackgroundStyle(AnnotationAttributes.GRADIENTSTYLE_RADIAL);
        a.SetGradientColor1(new ColorAttribute(0,0,255));
        a.SetGradientColor2(new ColorAttribute(0,0,0));
        a.SetForegroundColor(new ColorAttribute(255,255,255));
        Axes3D a3d = new Axes3D(a.GetAxes3D());
        a3d.SetAxesType(Axes3D.AXES_STATICEDGES);
        a3d.SetVisible(true);
        a.SetAxes3D(a3d);
        a.Notify();
        viewer.GetViewerMethods().SetAnnotationAttributes();
    }

    protected void work(String[] args)
```

(continues on next page)

(continued from previous page)

```
    {
        // Try and open a database
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + "wave*.silo␣
↪database"))
        {
            viewer.GetViewerMethods().AddPlot("Pseudocolor", "pressure");
            viewer.GetViewerMethods().AddPlot("Mesh", "quadmesh");
            viewer.GetViewerMethods().DrawPlots();

            // Set a 3D view.
            View3DAttributes v = viewer.GetViewerState().GetView3DAttributes();
            v.SetViewNormal(-0.705386, 0.57035, 0.42087);
            v.SetFocus(5, 0.353448, 2.5);
            v.SetViewUp(0.49514, 0.821357, -0.283213);
            v.SetViewAngle(30.);
            v.SetParallelScale(5.6009);
            v.SetNearPlane(-11.2018);
            v.SetFarPlane(11.2018);
            v.SetImagePan(0.0300266, 0.0519825);
            v.SetImageZoom(1.10796);
            v.SetPerspective(true);
            v.SetEyeAngle(2.);
            v.SetCenterOfRotationSet(false);
            v.SetCenterOfRotation(5, 0.353448, 2.5);
            v.Notify();
            viewer.GetViewerMethods().SetView3D();

            SetCustomDefaultAnnotations();

            AnnotationObjectList aol = viewer.GetViewerState().
↪GetAnnotationObjectList();

            //
            // Set up a time slider annotation.
            //
            viewer.GetViewerMethods().AddAnnotationObject(AnnotationObject.
↪ANNOTATIONTYPE_TIMESLIDER, "timeSlider");
            aol.SetTimeSliderOptions("timeSlider",
                0.675, 0.01, 0.3, 0.1,
                "Wave time = $time", "%1.3f",
                new ColorAttribute(255,0,0,255), new ColorAttribute(255,255,0,255),
                new ColorAttribute(0,255,0,255), false,
                0,
                true, false, false);
            aol.Notify();
            viewer.GetViewerMethods().SetAnnotationObjectOptions();
            // Advance through time to test the time slider.
            viewer.GetViewerMethods().SetTimeSliderState(30);

            System.out.println("After time slider: " + viewer.GetViewerState().
↪GetAnnotationObjectList().toString());

            //
            // Create a 2D text annotation
            //
            viewer.GetViewerMethods().AddAnnotationObject(AnnotationObject.
↪ANNOTATIONTYPE_TEXT2D, "text");
```

(continues on next page)

(continued from previous page)

```
            aol.SetText2DOptions("text", 0.4, 0.95, 0.05,
                "Wave simulation",
                new ColorAttribute(255,0,0,255), false,
                2, true, true, true, true);
            aol.Notify();
            viewer.GetViewerMethods().SetAnnotationObjectOptions();
            SaveImage("text2Dannot.png", 300, 300);
            System.out.println("After 2D text: " + viewer.GetViewerState().
↪GetAnnotationObjectList().toString());

            //
            // Create a 3D text annotation
            //
            viewer.GetViewerMethods().AddAnnotationObject(AnnotationObject.
↪ANNOTATIONTYPE_TEXT3D, "text3D");
            aol.SetText3DOptions("text3D",
                0., 2.5, 1.5,
                "Wave simulation, 3D text",
                true, 0., 3,
                true,
                15., 0., 0.,
                new ColorAttribute(255,255,0,255), false,
                true);
            aol.Notify();
            viewer.GetViewerMethods().SetAnnotationObjectOptions();
            SaveImage("text3D.png", 300, 300);
            System.out.println("After 3D text: " + viewer.GetViewerState().
↪GetAnnotationObjectList().toString());

            // Create a line 2D/arrow annotation
            //
            viewer.GetViewerMethods().AddAnnotationObject(AnnotationObject.
↪ANNOTATIONTYPE_LINE2D, "line");
            aol.SetLine2DOptions("line",
                0.5, 0.9495, 0.5, 0.6,
                2,
                0,
                2,
                new ColorAttribute(255,0,0,255), false,
                true);
            aol.Notify();
            viewer.GetViewerMethods().SetAnnotationObjectOptions();
            SaveImage("line2Dannot.png", 300, 300);
            System.out.println("After line: " + viewer.GetViewerState().
↪GetAnnotationObjectList().toString());

            //
            // Create a 3D line/arrow annotation
            //
            viewer.GetViewerMethods().AddAnnotationObject(AnnotationObject.
↪ANNOTATIONTYPE_LINE3D, "line3d");
            aol.SetLine3DOptions("line3d",
                6.0, 0.0, 0.0, // startpoint
                6.0, 3.0, 0.0, // endpoint
                1,         // lineWidth
                1,         // lineType 1=tube
                1,         //tubeQuality,
```

(continues on next page)

(continued from previous page)

```
                0.04,     //tubeRadius,
                false,    // arrow1
                16,       // arrow1Resolution
                0.120831, // arrow1Radius
                0.338327, // arrow1Height
                true,    // arrow2
                16,       // arrow2Resolution
                0.120831, // arrow2Radius
                0.338327, // arrow2Height
                new ColorAttribute(255,153,0,255), false,
                true);
            aol.Notify();
            viewer.GetViewerMethods().SetAnnotationObjectOptions();
            System.out.println("After 3D line: " + viewer.GetViewerState().
↪GetAnnotationObjectList().toString());
            SaveImage("line3Dannot.png", 300, 300);

            //
            // Save a small image to use for an annotation.
            //
            AnnotationAttributes annot = viewer.GetViewerState().
↪GetAnnotationAttributes();
            ColorAttribute transColor = new ColorAttribute(50,0,100,255);
            annot.SetBackgroundColor(transColor);
            annot.SetBackgroundMode(annot.BACKGROUNDMODE_SOLID);
            annot.Notify();
            viewer.GetViewerMethods().SetAnnotationAttributes();
            SaveImage("imageannot.png", 300, 300);
            annot.SetBackgroundColor(new ColorAttribute(0,0,0,255));
            annot.Notify();
            viewer.GetViewerMethods().SetAnnotationAttributes();

            viewer.GetViewerMethods().AddAnnotationObject(AnnotationObject.
↪ANNOTATIONTYPE_IMAGE, "image");
            aol.SetImageOptions("image",
                "imageannot.png",
                0.02, 0.63,
                1., 1., true,
                transColor, true,
                1.,
                true);
            aol.Notify();
            viewer.GetViewerMethods().SetAnnotationObjectOptions();
            System.out.println("After image: " + viewer.GetViewerState().
↪GetAnnotationObjectList().toString());


            // Set some legend attributes. You'd get the name from the PlotList object
            // but here we're just hard-coding the plot name since the Pseudocolor is
            // called Plot0000.
            SetCustomDefaultAnnotations();
            aol.SetLegendOptions("Plot0000",
                false, // managePosition,
                0.2, 0.1, // x,y
                1.5, 0.5, // scaleX, scaleY
                7, // numTicks
                true, // drawBox
```

(continues on next page)

(continued from previous page)

```
                false, // drawLabels,
                true,  // horizontalLegend,
                true, // alternateText, (false=normal text position, true=opposite␣
↪position)
                false, // drawTitle,
                false, // drawMinMax,
                true, // controlTicks,
                true, // minMaxInclusive,
                true, // drawValues
                0.03, // fontheight
                new ColorAttribute(100,255,100), false,
                2, true, true, true,
                true);
            aol.Notify();
            viewer.GetViewerMethods().SetAnnotationObjectOptions();
            SaveImage("legendChange.png", 300, 300);
        }
        else
            System.out.println("Could not open the database!");
    }

    private void SaveImage(String filename, int xres, int yres)
    {
        viewer.GetViewerMethods().SetAnnotationAttributes();
        SaveWindowAttributes saveAtts = viewer.GetViewerState().
↪GetSaveWindowAttributes();
        saveAtts.SetFileName(filename);
        saveAtts.SetWidth(xres);
        saveAtts.SetHeight(yres);
        saveAtts.SetFamily(false);
        saveAtts.SetFormat(saveAtts.FILEFORMAT_PNG);
        saveAtts.SetResConstraint(saveAtts.RESCONSTRAINT_NOCONSTRAINT);
        saveAtts.Notify();
        viewer.GetViewerMethods().SaveWindow();
    }

    public static void main(String args[])
    {
        TryAnnotations r = new TryAnnotations();
        r.run(args);
    }
}
```

7.5.6 Making host profiles

This program shows how to create a host profile, add it to the host profile list, and send it to the viewer. The program then goes on to access data on the remote computer, making use of the host profile that was created. Additional options such as how to launch the engine in parallel could be added to the host profile. Also, more profiles could be added to the host profile list before sending it to the viewer.

Show/Hide Code for `MakeHostProfile`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.
```

(continues on next page)

(continued from previous page)

```
import llnl.visit.ViewerProxy;
import llnl.visit.MachineProfile;
import llnl.visit.LaunchProfile;
import llnl.visit.HostProfileList;
import java.util.Vector;

// ****************************************************************************
// Class: MakeHostProfile
//
// Purpose:
//   This class implements an example program that shows how to use the
//   ViewerProxy class and control VisIt's viewer from Java.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Mon Aug 10 13:40:40 PDT 2009
//
// Modifications:
//   Jeremy Meredith, Thu Feb 18 17:14:38 EST 2010
//   Split host profile into machine profile and launch profile.
//   Also, added directory argument.
//
//   Kathleen Biagas, Wed Nov  9 14:30:32 PST 2016
//   Update host name to a machine that still exists.
// ****************************************************************************

public class MakeHostProfile extends RunViewer
{
    public MakeHostProfile()
    {
        super();
    }

    protected void work(String[] args)
    {
        // Change these for your remote system.
        String host = new String("pascal.llnl.gov");
        String user = new String("kbonnell");
        String remotevisitPath = new String("/usr/gapps/visit");

        // Basic, serial profile.
        LaunchProfile example = new LaunchProfile();
        example.SetProfileName("example");
        example.SetActive(true);

        // Create a new machine profile object and the serial launch profile.
        MachineProfile profile = new MachineProfile();
        profile.SetHost(host);
        profile.SetUserName(user);
        profile.SetClientHostDetermination(MachineProfile.CLIENTHOSTDETERMINATION_
↪PARSEDFROMSSHCLIENT);
        profile.SetTunnelSSH(true);
        profile.SetDirectory(remotevisitPath);
        profile.AddLaunchProfiles(example);
```

(continues on next page)

(continued from previous page)

```
        // Replace the list of host profiles and tell the viewer about the changes.␣
↪We could
        // have added to the list instead of clearing the list.
        viewer.GetViewerState().GetHostProfileList().ClearMachines();
        viewer.GetViewerState().GetHostProfileList().AddMachines(profile);
        viewer.GetViewerState().GetHostProfileList().Notify();
        System.out.println("HostProfileList = \n" +
            viewer.GetViewerState().GetHostProfileList().toString(""));

        // Do a plot of the remote data.
        String remoteData = new String(host + ":" + remotevisitPath + "/data/globe.
↪silo");
        if(viewer.GetViewerMethods().OpenDatabase(remoteData))
        {
            viewer.GetViewerMethods().AddPlot("Pseudocolor", "u");
            viewer.GetViewerMethods().AddPlot("Mesh", "mesh1");
            viewer.GetViewerMethods().DrawPlots();
        }
        else
        {
            System.out.println("Could not open the database!");
        }
    }

    public static void main(String args[])
    {
        MakeHostProfile r = new MakeHostProfile();
        r.run(args);
    }
}
```

7.5.7 Opening the VisIt GUI from Java

This program shows how to start the VisIt **GUI** from within your Java application. By altering the arguments passed to the `OpenClient()` method, you could launch other VisIt clients too. A VisIt client is a program that uses the `ViewerProxy` class to control the viewer. Examples of VisIt clients are: VisIt's **GUI**, VisIt's Python interface (**CLI**), and any program that uses the VisIt Java interface.

The important part of this code is the call to the `OpenClient()` method. The `OpenClient` method takes 3 arguments: `clientName`, `clientProgram`, `clientArgs`. The `clientName` is the internal name that will be used to identify the client inside of VisIt. You can pass any name that you want for this. The `clientProgram` argument is a string that identifies the executable for your program. The `clientArgs` argument lets you pass command line arguments to your program when it is started. When you call `OpenClient()`, the VisIt viewer will attempt to launch the specified VisIt client and then the client will be attached to VisIt and can control the VisIt viewer. Any number of VisIt clients can be connected to the VisIt viewer.

Show/Hide Code for `OpenGUI`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import llnl.visit.ViewerProxy;
import java.util.Vector;
```

(continues on next page)

(continued from previous page)

```
// ****************************************************************************
// Class: OpenGUI
//
// Purpose:
//   This class implements an example program that shows how to use the
//   ViewerProxy class and control VisIt's viewer from Java.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Mon Aug 17 13:40:40 PDT 2009
//
// Modifications:
//
// ****************************************************************************

public class OpenGUI extends RunViewer
{
    public OpenGUI()
    {
        super();
    }

    protected void work(String[] args)
    {
        // Do a plot of the data.
        String db = new String("globe.silo");
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + db))
        {
            viewer.GetViewerMethods().AddPlot("Pseudocolor", "u");
            viewer.GetViewerMethods().AddPlot("Mesh", "mesh1");
            viewer.GetViewerMethods().DrawPlots();
        }
        else
        {
            System.out.println("Could not open the database!");
        }

        // Open the VisIt GUI.
        String clientName = new String("GUI");
        String clientProgram = new String("visit");
        Vector clientArgs = new Vector();
        clientArgs.add(new String("-gui"));
        viewer.GetViewerMethods().OpenClient(clientName, clientProgram, clientArgs);
    }

    public static void main(String args[])
    {
        OpenGUI r = new OpenGUI();
        r.run(args);
    }
}
```

7.5.8 Determining which variables can be plotted

This program shows how to open a file and determine which plots can be used with the data from the file.

Each plot in VisIt responds to a certain set of variable types (scalar, vector, and so on). When you open a file, you get a list of variables in the metadata object. You must match up the variable types supported by a plot and the variables from the metadata in order to determine which plots can accept which variables from the database. This example program demonstrates a method for doing this comparison.

Note: The Java implementation does not offer a `GetVariableTypes` method in the plugin interface as it should. This is an oversight that may be corrected in a future version of VisIt. In the meantime, this program's `GetVariableTypes` method can be used to fulfill the same purpose.

Show/Hide Code for `PlotTypes`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import java.util.Hashtable;
import llnl.visit.avtDatabaseMetaData;

// ****************************************************************************
// Class: PlotTypes
//
// Purpose:
//   This is an example program that shows how to determine which plots can
//   accept variables from a file.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Fri Aug 28 09:21:06 PDT 2009
//
// Modifications:
//
// ****************************************************************************

public class PlotTypes extends RunViewer
{
    public PlotTypes()
    {
        super();
        savePlots = false;
    }

    public final static int MESH              = 0x0001;
    public final static int SCALAR            = 0x0002;
    public final static int MATERIAL          = 0x0004;
    public final static int VECTOR            = 0x0008;
    public final static int SUBSET            = 0x0010;
    public final static int SPECIES           = 0x0020;
    public final static int CURVE             = 0x0040;
    public final static int TENSOR            = 0x0080;
    public final static int SYMMETRICTENSOR   = 0x0100;
    public final static int LABEL             = 0x0200;
    public final static int ARRAY             = 0x0400;

    // The Plugin interface should have a GetVariableTypes method.
    // For now, this will suffice.
```

(continues on next page)

(continued from previous page)

```
    public int GetVariableTypes(String plotName)
    {
        Hashtable namestovar = new Hashtable();
        namestovar.put("Boundary", new Integer(MATERIAL));
        namestovar.put("Contour", new Integer(SCALAR | SPECIES));
        namestovar.put("Curve", new Integer(CURVE));
        namestovar.put("FilledBoundary", new Integer(MATERIAL));
        namestovar.put("Histogram", new Integer(SCALAR | ARRAY));
        namestovar.put("Kerbel", new Integer(MESH));
        namestovar.put("Label", new Integer(MESH | SCALAR | VECTOR | MATERIAL |␣
↪SUBSET | TENSOR | SYMMETRICTENSOR | LABEL | ARRAY));
        namestovar.put("Mesh", new Integer(MESH));
        namestovar.put("Molecule", new Integer(SCALAR));
        namestovar.put("MultiCurve", new Integer(CURVE));
        namestovar.put("ParallelCoordinates", new Integer(0)); //SCALAR | ARRAY));
        namestovar.put("Poincare", new Integer(VECTOR));
        namestovar.put("Pseudocolor", new Integer(SCALAR | SPECIES));
        namestovar.put("Scatter", new Integer(SCALAR));
        namestovar.put("Spreadsheet", new Integer(SCALAR));
        namestovar.put("Subset", new Integer(SUBSET | MESH));
        namestovar.put("Surface", new Integer(SCALAR | SPECIES));
        namestovar.put("Tensor", new Integer(TENSOR | SYMMETRICTENSOR));
        namestovar.put("Topology", new Integer(SCALAR));
        namestovar.put("Truecolor", new Integer(VECTOR));
        namestovar.put("Vector", new Integer(VECTOR));
        namestovar.put("Volume", new Integer(SCALAR | SPECIES));
        namestovar.put("WellBore", new Integer(MESH));

        return ((Integer)namestovar.get(plotName)).intValue();
    }

    protected void savePlot(String plotType, String var)
    {
        System.out.println(var);
        if(savePlots)
        {
            viewer.GetViewerMethods().AddPlot(plotType, var);
            viewer.GetViewerMethods().DrawPlots();
            viewer.GetViewerMethods().ResetView();
            viewer.GetViewerMethods().SaveWindow();
            viewer.GetViewerMethods().DeleteActivePlots();
        }
    }

    protected void work(String[] args)
    {
        System.out.println("Plots\n==================================================
↪");
        for(int i = 0; i < viewer.GetNumPlotPlugins(); ++i)
            System.out.println("Plot "+i+": name="+viewer.GetPlotName(i)+", version=
↪"+viewer.GetPlotVersion(i));

        System.out.println(
↪"Operators\n==================================================");
        for(int i = 0; i < viewer.GetNumOperatorPlugins(); ++i)
            System.out.println("Operator "+i+": name="+viewer.GetOperatorName(i)+",␣
↪version="+viewer.GetOperatorVersion(i));
```

(continues on next page)

(continued from previous page)

```
        String db = new String(viewer.GetDataPath() + "noise.silo");
        if(viewer.GetViewerMethods().RequestMetaData(db, 0))
        {
            if(savePlots)
                viewer.GetViewerMethods().OpenDatabase(db);

            avtDatabaseMetaData md = viewer.GetViewerState().GetDatabaseMetaData();
            for(int i = 0; i < viewer.GetNumPlotPlugins(); ++i)
            {
                System.out.println("\n"+viewer.GetPlotName(i) + " can accept␣
↪variables:\n=====================================");
                int vartypes = GetVariableTypes(viewer.GetPlotName(i));
                if((vartypes & MESH) > 0)
                {
                    for(int j = 0; j < md.GetNumMeshes(); ++j)
                        savePlot(viewer.GetPlotName(i), md.GetMeshes(j).GetName());
                }
                if((vartypes & SCALAR) > 0)
                {
                    for(int j = 0; j < md.GetNumScalars(); ++j)
                        savePlot(viewer.GetPlotName(i), md.GetScalars(j).GetName());
                }
                if((vartypes & MATERIAL) > 0)
                {
                    for(int j = 0; j < md.GetNumMaterials(); ++j)
                        savePlot(viewer.GetPlotName(i), md.GetMaterials(j).GetName());
                }
                if((vartypes & VECTOR) > 0)
                {
                    for(int j = 0; j < md.GetNumVectors(); ++j)
                        savePlot(viewer.GetPlotName(i), md.GetVectors(j).GetName());
                }
                if((vartypes & SPECIES) > 0)
                {
                    for(int j = 0; j < md.GetNumSpecies(); ++j)
                        savePlot(viewer.GetPlotName(i), md.GetSpecies(j).GetName());
                }
                if((vartypes & CURVE) > 0)
                {
                    for(int j = 0; j < md.GetNumCurves(); ++j)
                        savePlot(viewer.GetPlotName(i), md.GetCurves(j).GetName());
                }
                if((vartypes & TENSOR) > 0)
                {
                    for(int j = 0; j < md.GetNumTensors(); ++j)
                        savePlot(viewer.GetPlotName(i), md.GetTensors(j).GetName());
                }
                if((vartypes & SYMMETRICTENSOR) > 0)
                {
                    for(int j = 0; j < md.GetNumSymmTensors(); ++j)
                        savePlot(viewer.GetPlotName(i), md.GetSymmTensors(j).
↪GetName());
                }
                if((vartypes & LABEL) > 0)
                {
                    for(int j = 0; j < md.GetNumLabels(); ++j)
```

(continues on next page)

(continued from previous page)

```
                        savePlot(viewer.GetPlotName(i), md.GetLabels(j).GetName());
                }
                if((vartypes & ARRAY) > 0)
                {
                    for(int j = 0; j < md.GetNumArrays(); ++j)
                        savePlot(viewer.GetPlotName(i), md.GetArrays(j).GetName());
                }
            }
        }
        else
            System.out.println("Could not open the database!");
    }

    public static void main(String args[])
    {
        PlotTypes r = new PlotTypes();

        for(int i = 0; i < args.length; ++i)
        {
            if(args[i].equals("-plot"))
                r.savePlots = true;
        }

        r.run(args);
    }

    private boolean savePlots;
}
```

7.5.9 Executing Python from Java

This code example shows how to create a Java program that launches VisIt's Python **CLI** program and send Python command strings to it for interpretation. This example program also implements the SimpleObserver interface which lets us observe state objects. In this case, we observe the plot list and print it whenever we see it.

Show/Hide Code for `DualClients`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import java.lang.ArrayIndexOutOfBoundsException;
import java.lang.String;

import java.util.Vector;

import llnl.visit.AttributeSubject;
import llnl.visit.ClientMethod;
import llnl.visit.ClientInformation;
import llnl.visit.ClientInformationList;
import llnl.visit.PlotList;
import llnl.visit.SimpleObserver;


// ****************************************************************************
```

(continues on next page)

(continued from previous page)

```
// Class: DualClients
//
// Purpose:
//   This example program shows how to launch the Python client from Java
//   and send commands to it.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Tue Jan 11 09:30:41 PST 2011
//
// Modifications:
//
// ****************************************************************************

public class DualClients extends RunViewer implements SimpleObserver
{
    public DualClients()
    {
        super();
        doUpdate = true;

        // Make this object observe the plot list
        viewer.GetViewerState().GetPlotList().Attach(this);
    }

    //
    // Main work method for the program
    //
    protected void work(String[] args)
    {
        // Try and open a database
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + "noise.silo
↪"))
        {
            // Interpret some Python using the VisIt CLI
            InterpretPython("AddPlot('Pseudocolor', 'hardyglobal')");
            InterpretPython("AddPlot('Mesh', 'Mesh')");
            InterpretPython("DrawPlots()");
            InterpretPython("SaveWindow()");
        }
        else
            System.out.println("Could not open the database!");
    }

    //
    // Check all of the client information until we find a client that
    // supports the Interpret method with a string argument.
    //
    protected boolean NoInterpretingClient()
    {
        // Make a copy because the reader thread could be messing with it.
        // Need to synchronize access.
        ClientInformationList cL = new ClientInformationList(
            viewer.GetViewerState().GetClientInformationList());

        for(int i = 0; i < cL.GetNumClients(); ++i)
```

(continues on next page)

(continued from previous page)

```
        {
            ClientInformation client = cL.GetClients(i);
            for(int j = 0; j < client.GetMethodNames().size(); ++j)
            {
                String name = (String)client.GetMethodNames().elementAt(j);
                if(name.equals("Interpret"))
                {
                    String proto = (String)client.GetMethodPrototypes().elementAt(j);
                    if(proto.equals("s"))
                    {
                        // We have an interpreting client
                        return false;
                    }
                }
            }
        }
        return true;
    }

    //
    // If we don't have a client that can "Interpret" then tell the viewer
    // to launch a VisIt CLI.
    //
    protected boolean Initialize()
    {
        boolean launched = false;
        if(NoInterpretingClient())
        {
            System.out.println("Tell the viewer to create a CLI so we can execute␣
↪code.");
            Vector args = new Vector();
            args.addElement(new String("-cli"));
            args.addElement(new String("-newconsole"));
            viewer.GetViewerMethods().OpenClient("CLI",
                 "visit",
                 args);
            launched = true;

            viewer.Synchronize();

            // HACK: Wait until we have an interpreting client.
            while(NoInterpretingClient())
                viewer.Synchronize();
        }
        return launched;
    }

    //
    // Interpret a Python command string.
    //
    protected void InterpretPython(String cmd)
    {
        Initialize();

        // Send the command to interpret as a client method.
        ClientMethod method = viewer.GetViewerState().GetClientMethod();
        method.SetIntArgs(new Vector());
```

(continues on next page)

(continued from previous page)

```
        method.SetDoubleArgs(new Vector());
        Vector args = new Vector();
        args.addElement(new String(cmd + "\n"));
        method.SetStringArgs(args);
        method.SetMethodName("Interpret");
        method.Notify();
        System.out.println("Interpret: " + cmd);

        viewer.Synchronize();
    }

    //
    // SimpleObserver interface methods
    //
    public void Update(AttributeSubject s)
    {
        // Do something with the plot list.
        System.out.println(s.toString());
    }
    public void SetUpdate(boolean val) { doUpdate = val; }
    public boolean GetUpdate() { return doUpdate; }


    public static void main(String args[])
    {
        DualClients r = new DualClients();
        r.run(args);
    }

    private boolean doUpdate;
}
```

7.5.10 Plotting vectors from Java

This example program shows how to create a vector expression and then plot a Vector plot of that expression. The **Displace** operator is also used to warp the coordinate system.

Show/Hide Code for `PlotVector`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import llnl.visit.ViewerProxy;
import llnl.visit.Expression;
import llnl.visit.ExpressionList;

public class PlotVector extends RunViewer
{
    public PlotVector()
    {
        super();
    }

    protected void work(String[] args)
```

(continues on next page)

(continued from previous page)

```
    {
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + "globe.silo
↪"))
        {
            ExpressionList explist = viewer.GetViewerState().GetExpressionList();
            Expression e = new Expression();
            e.SetName("disp");
            e.SetType(Expression.EXPRTYPE_VECTORMESHVAR);
            e.SetDefinition("{speed,u,v} - coord(mesh1)");
            explist.AddExpressions(e);
            explist.Notify();
            viewer.GetViewerMethods().ProcessExpressions();

            // Add a plot of the vector
            viewer.GetViewerMethods().AddPlot("Vector", "disp");
            viewer.GetViewerMethods().AddOperator("Displace");
            viewer.GetViewerMethods().DrawPlots();
        }
        else
        {
            System.out.println("Could not open the database!");
        }
    }

    public static void main(String args[])
    {
        PlotVector r = new PlotVector();
        r.run(args);
    }
}
```

7.5.11 Changing plot attributes

This example program shows how to set plot attributes. It changes a Pseudoocolor plot to be semi-transparent.

Show/Hide Code for `PlotAtts`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.


import llnl.visit.View3DAttributes;
import llnl.visit.plots.PseudocolorAttributes;

// ****************************************************************************
// Class: PlotAtts
//
// Purpose:
//   This is an example program that shows how to set plot attributes.
//
// Notes:
//
// Programmer: Brad Whitlock
// Creation:   Thu Aug 15 16:09:03 PST 2002
```

(continues on next page)

(continued from previous page)

```
//
// Modifications:
//   Brad Whitlock, Tue Sep 24 08:05:51 PDT 2002
//   I changed it so the view is set after the plot is drawn.
//
//   Eric Brugger, Wed Aug 27 09:04:55 PDT 2003
//   I modified it to use the new view interface.
//
//   Brad Whitlock, Mon Jun 6 17:25:34 PST 2005
//   I made it use GetDataPath to locate the data.
//
//   Brad Whitlock, Thu Jul 14 12:15:42 PDT 2005
//   Updated.
//
//   Brad Whitlock, Mon Feb 25 11:07:24 PDT 2008
//   Changed to new ViewerProxy interface.
//
// ****************************************************************************

public class PlotAtts extends RunViewer
{
    public PlotAtts()
    {
        super();
    }

    protected void work(String[] args)
    {
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + "globe.silo
↪"))
        {
            // Create a plot.
            viewer.GetViewerMethods().AddPlot("Pseudocolor", "u");

            // Set the pseudocolor attributes
            PseudocolorAttributes p = (PseudocolorAttributes)viewer.GetPlotAttributes(
↪"Pseudocolor");
            // set Pseudocolor's opacity type to constant
            p.SetOpacityType(p.OPACITYTYPE_CONSTANT);
            p.SetOpacity(0.3);
            p.Notify();
            viewer.GetViewerMethods().SetPlotOptions("Pseudocolor");

            // Draw the plot
            viewer.GetViewerMethods().DrawPlots();

            // Set the view
            View3DAttributes v = viewer.GetViewerState().GetView3DAttributes();
            v.SetViewNormal(0.456808, 0.335583, 0.823839);
            v.SetFocus(-0.927295, -1.22113, 1.01159);
            v.SetViewUp(-0.184554, 0.941716, -0.281266);
            v.SetParallelScale(15.7041);
            v.SetNearPlane(-34.641);
            v.SetFarPlane(34.641);
            v.Notify();
            viewer.GetViewerMethods().SetView3D();
        }
```

(continues on next page)

(continued from previous page)

```
        else
            System.out.println("Could not open the database!");
    }

    public static void main(String args[])
    {
        PlotAtts r = new PlotAtts();
        r.run(args);
    }
}
```

7.5.12 Changing points size and shape

This example program shows how to change point size/type for Point meshes.

Show/Hide Code for `TryPointGlyphing`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.

import java.util.Vector;

import llnl.visit.ColorAttribute;
import llnl.visit.plots.MeshAttributes;

// ****************************************************************************
// Class: TryPointGlyphing
//
// Purpose:
//   This example program sets up a Mesh plot of a point mesh and modifies
//   the point size and shape (glyph) settings.
//
// Programmer: Kathleen Biagas
// Creation:   March 31, 2017
//
// Modifications:
//
// ****************************************************************************

public class TryPointGlyphing extends RunViewer
{
    public TryPointGlyphing()
    {
        super();
    }

    protected void work(String[] args)
    {
        // Try and open a database
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + "noise.silo
↪"))
        {
            viewer.GetViewerMethods().AddPlot("Mesh", "PointMesh");
            viewer.GetViewerMethods().DrawPlots();
```

(continues on next page)

(continued from previous page)

```
            // Set the pseudocolor attributes
            MeshAttributes m = (MeshAttributes)viewer.GetPlotAttributes("Mesh");
            // Sets the size for 'Point' type
            m.SetPointSizePixels(5);
            // Sets the size for all other types
            m.SetPointSize(1.0);

            m.SetMeshColorSource(1);
            m.SetMeshColor(new ColorAttribute(204,153,255,0));
            // Run through all the point types
            // 0 : Box
            // 1 : Axis
            // 2 : Icosahedron
            // 3 : Octahedron
            // 4 : Tetrahedron
            // 5 : SphereGeometry
            // 6 : Point
            // 7 : Sphere

            for (int i = 0; i < 8; ++i)
            {
                m.SetPointType(i);
                m.Notify();
                viewer.GetViewerMethods().SetPlotOptions("Mesh");

                viewer.GetViewerMethods().SaveWindow();
            }
        }
        else
            System.out.println("Could not open the database!");
    }


    public static void main(String args[])
    {
        TryPointGlyphing r = new TryPointGlyphing();
        r.run(args);
    }
}
```

7.5.13 Using Threshold operator

This example program shows how to use a Threshold operator with a Pseudocolor plot.

ThresholdAttributes needs Vector to set ZonePortions, LowerBounds, and UpperBounds because more than one variable can be used with Threshold. If more than one variable is requested (not demonstrated in this example), the first entry in the Vector contains information for the first variable, second entry contains information for the second variable and so on.

Show/Hide Code for `TryThreshold`

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.
```

(continues on next page)

(continued from previous page)

```
import java.util.Vector;
import llnl.visit.AttributeSubject;
import llnl.visit.View3DAttributes;

import llnl.visit.plots.PseudocolorAttributes;
import llnl.visit.operators.ThresholdAttributes;

// ****************************************************************************
// Class: TryThreshold
//
// Purpose:
//   This example program sets up a Pseudocolor plot with threshold operator.
//
// Notes:      Based on threshold.py of test-suite.
//
// Programmer: Kathleen Biagas
// Creation:   March 31, 2017
//
// Modifications:
//
// ****************************************************************************

public class TryThreshold extends RunViewer
{
    public TryThreshold()
    {
        super();
    }

    protected void work(String[] args)
    {
        // Try and open a database
        if(viewer.GetViewerMethods().OpenDatabase(viewer.GetDataPath() + "globe.silo
↪"))
        {
            viewer.GetViewerMethods().AddPlot("Pseudocolor", "u");
            viewer.GetViewerMethods().AddOperator("Threshold");

            // Set the pseudocolor attributes
            ThresholdAttributes t = (ThresholdAttributes)viewer.GetOperatorAttributes(
↪"Threshold");
            t.SetOutputMeshType(0);
            Vector zp = new Vector();
            zp.add(1);
            t.SetZonePortions(zp);
            Vector lb = new Vector();
            lb.add(-4.0);
            t.SetLowerBounds(lb);
            Vector ub = new Vector();
            ub.add(4.0);
            t.SetUpperBounds(ub);
            t.Notify();
            viewer.GetViewerMethods().SetOperatorOptions("Threshold");
            viewer.GetViewerMethods().DrawPlots();

            // Set the view
```

(continues on next page)

(continued from previous page)

```
            View3DAttributes v = viewer.GetViewerState().GetView3DAttributes();
            v.SetViewNormal(-0.528889, 0.367702, 0.7649);
            v.SetViewUp(0.176641, 0.929226, -0.324558);
            v.SetParallelScale(17.3205);
            v.SetPerspective(true);
            v.Notify();
            viewer.GetViewerMethods().SetView3D();

            viewer.GetViewerMethods().SaveWindow();

            // Change zone inclusion criteria
            zp.set(0, 0);
            t.SetZonePortions(zp);
            t.Notify();
            viewer.GetViewerMethods().SetOperatorOptions("Threshold");
            viewer.GetViewerMethods().SaveWindow();

            //  Threshold by a variable different than the PC coloring variable.

            zp.set(0, 1);
            t.SetZonePortions(zp);
            lb.set(0, 140.0);
            t.SetLowerBounds(lb);
            ub.set(0, 340.0);
            t.SetUpperBounds(ub);
            Vector vn = new Vector();
            vn.add("t");
            t.SetListedVarNames(vn);
            t.Notify();
            viewer.GetViewerMethods().SetOperatorOptions("Threshold");
            viewer.GetViewerMethods().SaveWindow();

        }
        else
            System.out.println("Could not open the database!");
    }


    public static void main(String args[])
    {
        TryThreshold r = new TryThreshold();
        r.run(args);
    }
}
```

7.6 Acknowledgements

This document is primarily based on visitusers.org wiki pages written by Brad Whitlock. The Java client itself and most of the examples were also initially created by Brad in 2002.

CHAPTER

EIGHT

GETTING DATA INTO VISIT

Contents:

8.1 Introduction

VisIt comes with over 100 database readers. If you are using an existing format that we support, then it is just matter of opening the file. VisIt uses file extensions to determine the type of reader to use. If your files don't have the correct extension you can force VisIt to use a specific reader. If you don't already have an existing file format or VisIt doesn't support your format, there are two routes you can take. One is to use one of the existing formats and the other is to write your own database reader.

The database readers can be categorized into the following groups:

- Simple text formats
- Simple binary formats
- Library based formats
- Specialized formats

The simple formats are characterized by the fact that they can easily be written by a program using text or binary based write statements. The advantage is primarily that they are simple and that code can easily be added to a simulation to output the files without any external dependencies. Typically they support a single type of mesh, although not necessarily. Typically simple text formats must be read completely, even when only a portion of the data in the file may be necessary to perform the desired operation.

The library based formats are characterized by the fact that they require an external library to write. These libraries typically have sophisticated data models and support many different types of meshes. If you have a complex mesh type there is a better chance that it is supported. Another advantage is that these types of file formats have metadata that allows reading only a portion of the file to determine the contents and support partial reads to only read the portion of the file that is necessary to perform the visualization. Reading only a portion of the file can provide a significant performance improvement since reading the data is typically a large portion of the time required to get the initial image displayed. The disadvantage is that it requires an external library, which adds build complexity to the simulation. It also requires learning the API for the library, which also may be complex.

The specialized formats are typically associated with a specific simulation code. In this case, the simulation code probably already has an output file that is part of some workflow and they don't want to write out another file format, possible writing out duplicate data.

Below is a list of recommended simple file formats and the mesh types they support.

Name	Type	Supported mesh types
VTK	Binary, Text	Regular, Rectilinear, Structured, Unstructured
BOV	Binary	Regular
Curve	Text	2D Curves
PlainText	Text	2D Curves, Point, 2D Rectilinear

Here is a list of recommended library based formats and the mesh types they support.

Name	Supported mesh types
Silo	Point, Regular, Rectilinear, Structured, Unstructured, Multiblock, Patch based AMR
Xdmf	Point, Regular, Rectilinear, Structured, Unstructured, Multiblock
Conduit/Blueprint	Point, Regular, Rectilinear, Structured, Unstructured, Multiblock, Patch based AMR

Files representing a single block or a single time step may also be grouped into a multi-block representation or time series using a *.visit* file.

8.2 The VTK file format

There are primarily two types of VTK files, legacy and XML. We are going to focus on the legacy files, since they are a little bit simpler.

- VTK files can represent a single timestep.
  - If you have multiple times you can group them with a *.visit* file.
- VTK files can be text or binary
  - Both text and binary files have text meta data.
  - Binary files have binary fields and coordinates imbedded in the text.
- VisIt has conventions for additional meta data that fits within the VTK specification.

The remainder of this VTK documentation consistes of a description of the file format, a simple example, and then more complex examples of the different mesh types.

The official VTK file format descriptions can be found at the VTK website.

Warning: Reading VTK files into VisIt requires *strict* file extension matching. For example, if you have a pvtu file, it must have the `.pvtu` extension or VisIt will not be able to open and read the file. This is true even when using VisIt's feature to explicitly specify the plugin to use to open the file. For example, if you know `file.foo` is a VTK file and try to open it using VisIt's VTK plugin, it will fail because the extension does not match a known extension for VTK files. For VisIt to read VTK files, the files *must have* the correct VTK extensions.

8.2.1 The visit_writer source and header files

The `visit_writer.c` and `visit_writer.h` files contain C source code with simple functions that write VTK files that can easily be added to your existing C or C++ data generating code without introducing any external dependencies. The library can write either text or binary files. The library can output point, unstructured, rectilinear, regular and curvilinear meshes.

The header and source files can be found in the VisIt repository on GitHub.

8.2.2 Binary files

Binary files contain binary coordinates and fields. This is any data where a length and data type are specified. The binary data follows immediately after the newline character of the previous ASCII keyword and parameter sequence. Binary data must be written in *big endian* format. If you are on a *little endian* system, you will need to byte swap the data before writing it. X86 and ARM processors store data in *little endian* format, so you will need to byte swap the data in that case. If you want to write your code independent of the *endianness*, you can write code to detect it and do the byte swapping if appropriate. The `visit_writer` mentioned above contains code to detect the *endianness* and do byte swapping if necessary.

8.2.3 The basic structure of a VTK file

A VTK file consists of 5 sections.

1. The first section is the file version and identifier. This section contains the single line: # vtk DataFile Version 3.0.

2. The second section is the title. The title must be on a single line and can be at most 256 characters.

3. The next section is the file format. The file format describes the type of the file, either ASCII or binary. The type must be on a single line and contain either the word ASCII or BINARY.

4. The fourth part is the mesh structure. It contains the points and the topology. This part begins with a line containing the keyword DATASET followed by a keyword describing the type of dataset. Then, depending upon the type of dataset, other keyword/data combinations define the actual data. The topology can be 1D, 2D or 3D, although only 2D and 3D topologies are supported by VisIt. If the Z-coordinates are all zero, VisIt will treat the mesh as 2D and display it in 2D.

5. The final part describes the fields. The fields can be defined on either the cells or the points. The cell or point fields can be listed in either order. The fields can be either scalars, vectors or tensors.

Mesh topology

The following mesh topologies are supported.

- Structured points
- Structured grid
- Rectilinear grid
- Polydata
- Unstructured grid

Structured points

The structured points section has the following structure.

```
DATASET STRUCTURED_POINTS
DIMENSIONS nx ny nz
ORIGIN x y z
SPACING sx sy sz
```

nx, *ny*, *nz* are the number of dimensions in the X- Y- and Z-directions. *x*, *y*, *z* is the origin of the grid. *sx*, *sy*, *sz* is the spacing in the X- Y- and Z-directions.

Structured grid

The structured grid section has the following structure.

```
DATASET STRUCTURED_GRID
DIMENSIONS nx ny nz
POINTS nPoints dataType
```

nx, *ny*, *nz* are the number of dimensions in the X- Y- and Z-directions. *nPoints* is the number of points. *nPoints* must be consistent with the dimensions. Supported data types in VisIt are *float* and *double*.

Rectilinear grid

The rectilinear grid section has the following structure.

```
DATASET RECTILINEAR_GRID
DIMENSIONS nx ny nz
X_COORDINATES nx dataType
x1 x2 ... xn
Y_COORDINATES ny dataType
y1 y2 ... yn
Z_COORDINATES nz dataType
z1 z2 ... zn
```

nx, *ny*, *nz* are the number of dimensions in the X- Y- and Z-directions. *nx*, *ny*, *nz* in the dimensions statement must be constient with the ones in the coordinates statement. Supported data types in VisIt are *float* and *double*.

Polydata

The polydata grid section has the following structure.

```
DATASET POLYDATA
POINTS nPoints dataType
x1 y1 z1
x2 y2 z2
...
xn yn zn

VERTICES nVertices size
1 p1
1 p2
...
1 pn

LINES nLines size
2 l11 l12
2 l21 l22
...
2 ln1 ln2

POLYGONS nPolygons size
n1 i11 i12 ... i1n1
n2 i21 i22 ... i2n2
...
```

(continues on next page)

(continued from previous page)

```
nn in1 in2 ... innn

TRIANGLE_STRIPS nStrips size
n1 i11 i12 ... i1n1
n2 i21 i22 ... i2n2
...
nn in1 in2 ... innn
```

Supported data types in VisIt are *float* and *double*. The vertices, lines, polygons and triangle_strips sections may or may not be present. The vertices, lines, polygons and triangle_strips sections may be in any order. *xn*, *yn* and *zn* are the coordinates of the nth point. *n1*, *n2* and *nm* are the number of indices for each cell. *in1*, *in2* and *innn* are the zero origin indices for the nth cell.

Unstructured grid

The unstructured grid section has the following structure.

```
DATASET UNSTRUCTURED_GRID
POINTS nPoints dataType
x1 y1 z1
x2 y2 z2
...
xn yn zn

CELLS nCells size
n1 i11 i12 ... i1n1
n2 i21 i22 ... i2n2
...
nn in1 in2 ... innn

CELL_TYPES nCells
t1
t2
...
tn
```

Supported data types in VisIt are *float* and *double*. The cells and cell_types sections may be in any order. *xn*, *yn* and *zn* are the coordinates of the nth point. *n1*, *n2* and *nm* are the number of indices for each cell. *in1*, *in2* and *innn* are the zero origin indices for the nth cell. *t1*, *t2* and *tn* are the cell types for each cell.

The following cell types are supported.

Type	Value	Type	Value
VTK_VERTEX	1	VTK_QUADRATIC_EDGE	21
VTK_POLY_VERTEX	2	VTK_QUADRATIC_TRIANGLE	22
VTK_LINE	3	VTK_QUADRATIC_QUAD	23
VTK_POLY_LINE	4	VTK_QUADRATIC_TETRA	24
VTK_TRIANGLE	5	VTK_QUADRATIC_HEXAHEDRON	25
VTK_TRIANGLE_STRIP	6	VTK_BIQUADRATIC_QUAD	28
VTK_POLYGON	7	VTK_BIQUADRATIC_TRIANGLE	34
VTK_PIXEL	8	VTK_CUBIC_LINE	35
VTK_QUAD	9	VTK_LAGRANGE_TRIANGLE	69
VTK_TETRA	10	VTK_LAGRANGE_QUADRILATERAL	70
VTK_VOXEL	11	VTK_LAGRANGE_TETRAHEDRON	71
VTK_HEXAHEDRON	12	VTK_LAGRANGE_HEXAHEDRON	72
VTK_WEDGE	13		
VTK_PYRAMID	14		

Cell data

The cell data section starts with a single line with the keyword *CELL_DATA* followed by the number of cells. The number of cells must match the number of cells defined by the topology. Next comes a list of fields. Fields can be either scalars, vectors or tensors.

Scalar fields

The scalar field section has the following structure.

```
SCALARS fieldName dataType 1
LOOKUP_TABLE default
s1 s2 ... sN
```

The scalar field section starts with a single line with the keyword *SCALARS* followed by the name of the field followed by the data type followed by the number of values per scalar. The number of values per scalar is optional, and if present, must be *1*. Next comes the lookup table information. The lookup table information consists of a single line with the keyword *LOOKUP_TABLE* followed by the keyword *default*. The lookup table specifies the lookup table name when mapping the scalar to colors. The lookup table is not used by VisIt. Next come the scalar values. The scalar values can be split up into lines in an arbitrary manner.

Vector fields

The vector field section has the following structure.

```
VECTORS fieldName dataType
v11 v12 v13
v21 v22 v23
...
vn1 vn2 vn3
```

The vector field section starts with a single line with the keyword *VECTORS* followed by the name of the field followed by the data type. Next come the vector values. The vector values consist of three values per point, regardless of whether it is for a 2D or 3D mesh. The vector values can be split up into lines in an arbitrary manner.

Tensor fields

The tensor field section has the following structure.

```
TENSORS fieldName dataType
t111 t112 t113
t121 t122 t123
t131 t132 t133
t211 t212 t213
t221 t222 t223
t231 t232 t233
...
tn11 tn12 tn13
tn21 tn22 tn23
tn31 tn32 tn33
```

The tensor field section starts with a single line with the keyword *TENSORS* followed by the name of the field followed by the data type. Next come the tensor values. The tensor values consist of nine values per point, regardless of whether it is for a 2D or 3D mesh. The tensor values can be split up into lines in an arbitrary manner.

Point data

The point data section starts with a single line with the keyword *POINT_DATA* followed by the number of points. The number of points must match the number of points defined by the topology. The rest of the point data section is the same as the cell data section.

Field data

Field data is an array of data arrays that can be associated with a *DATASET*, *CELL_DATA* or *POINT_DATA*.

Field data has the following general format.

```
FIELD FieldData numberOfFields
FieldName numberOfComponent numberOfTuples dataType
t11 t12 ... t1cn
t21 t22 ... t2cn
...
tn1 tn2 ... tncn
```

The field data section begines with a single line that starts with the keywords *FIELD* and *FieldData* followed by the number of fields. Next come the fields. Each field starts with a single line with the field name, the number of components, the number of tuples and the data type.

8.2.4 VisIt meta data conventions for VTK files

VisIt supports a number of conventions for storing additional data. This data is stored as FIELD data as additional information in the DATASET, CELL_DATA or POINT_DATA.

The Following meta data is stored as DATASET FIELD data.

MeshCoordType

This specifies if the mesh coordinates are 2D cylindrical coordinates. If the value is a *1*, it is an R-Z mesh. If the value is a *2*, it is a Z-R mesh.

Here is an example of specifying the mesh coordinate type.

```
MeshCoordType 1 1 int
2
```

MeshName

The mesh name is represented as a string.

Here is an example of specifying the mesh name.

```
MeshName 1 1 string
rectmesh2d
```

CYCLE

The cycle is specified as a single integer value.

Here is an example of specifying the cycle.

```
CYCLE 1 1 int
10
```

TIME

The time is specified as a single double precision value.

Here is an example of specifying the time.

```
TIME 1 1 double
10.0
```

VisItExpressions

Each string represents a single expression. The string contains the expression name, the expression type and the expression. The three properties are seperated by semicolons. The expression type consists of one of *curve*, *scalar*, *vector*, *tensor*, *array*, *material* or *species*.

Here is an example of specifying the expressions.

```
VisItExpressions 1 2 string
vel;vector;{u,v}
speed;scalar;sqrt(u*u+v*v)
```

avtGhostZones

The ghost zones specify a flag indicating if the zone is a ghost zone or a real zone. A one indicates a ghost zone. A zero indicates a real zone. The ghost zone meta data is stored as CELL_DATA FIELD data.

Here is an example of specifying ghost zones.

```
FIELD FieldData 1
avtGhostZones 1 12 unsigned_char
1 0 0 1
1 0 0 1
1 0 0 1
```

8.2.5 An example VTK file

A basic VTK file is shown here.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET RECTILINEAR_GRID
DIMENSIONS 3 2 2
X_COORDINATES 3 float
0 1 2
Y_COORDINATES 2 float
0 1
Z_COORDINATES 2 float
0 1

CELL_DATA 2
SCALARS density float 1
LOOKUP_TABLE default
1 2

POINT_DATA 12
SCALARS u float 1
LOOKUP_TABLE default
1 2 3 1 2 3 1 2 3 1 2 3
```

The line below contains the version and identifier.

```
# vtk DataFile Version 3.0
```

The line below contains the title.

```
vtk output
```

The line below contains the data type, which in this case is ASCII.

```
ASCII
```

The line below identifies the type of the mesh, which in this case is a rectilinear grid.

```
DATASET RECTILINEAR_GRID
```

The following lines provide the coordinate information for the mesh.

```
DIMENSIONS 3 2 2
X_COORDINATES 3 float
0 1 2
Y_COORDINATES 2 float
0 1
```

(continues on next page)

(continued from previous page)

```
Z_COORDINATES 2 float
0 1
```

The information provides the dimensions of the coordinate arrays of the mesh along with the coordinates in each of the three directions.

The following lines represent one scalar field defined on the cells.

```
CELL_DATA 2
SCALARS density float 1
LOOKUP_TABLE default
1 2
```

The information tells us that there are 2 values for the cell data, that it is a scalar, that the name of the variable is *density*, that it should be read in as float values, that we should use the default lookup table, and that the values consist of *1 2*.

The following lines represent one scalar field defined at the points.

```
POINT_DATA 12
SCALARS u float 1
LOOKUP_TABLE default
1 2 3 1 2 3 1 2 3 1 2 3
```

The information tells us that there are 12 values for the point data, that it is a scalar, that the name of the variable is *u*, that it should be read in as float values, that we should use the default lookup table, and that the values consist of *1 2 3 1 2 3 1 2 3 1 2 3*.

8.2.6 An example of a VTK file with extra metadata

A VTK file with extra meta data is shown here.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET RECTILINEAR_GRID
FIELD FieldData 5
MeshCoordType 1 1 int
2
MeshName 1 1 string
rectmesh2d
CYCLE 1 1 int
10
TIME 1 1 double
10.0
VisItExpressions 1 2 string
vel;vector;{u,v}
speed;scalar;sqrt(u*u+v*v)

DIMENSIONS 5 4 1
X_COORDINATES 5 float
0 1 2 3 4
Y_COORDINATES 4 float
0 1 2 3
Z_COORDINATES 1 float
```

(continues on next page)

(continued from previous page)

```
0

CELL_DATA 12
SCALARS density float 1
LOOKUP_TABLE default
1 2 3 4
5 6 7 8
9 10 11 12

FIELD FieldData 1
avtGhostZones 1 12 unsigned_char
1 0 0 1
1 0 0 1
1 0 0 1

POINT_DATA 20
SCALARS u float 1
LOOKUP_TABLE default
1 2 3 4 5 1 2 3 4 5
1 2 3 4 5 1 2 3 4 5
SCALARS v float 1
LOOKUP_TABLE default
```

The following lines represent the mesh name associated with this file.

```
MeshName 1 1 string
rectmesh2d
```

The following lines represent cycle associated with this file.

```
CYCLE 1 1 int
10
```

The following lines represent the time associated with this file.

```
TIME 1 1 double
10.0
```

The following lines represent the expressions associated with this file.

```
VisItExpressions 1 2 string
vel;vector;{u,v}
speed;scalar;sqrt(u*u+v*v)
```

The following lines represent the ghost zones associated with this file.

```
FIELD FieldData 1
avtGhostZones 1 12 unsigned_char
1 0 0 1
1 0 0 1
1 0 0 1
```

8.2.7 An example of a structured points file

A structured points file results in a regular mesh with constant spacing in each direction. The mesh can be 2D or 3D. It is defined by specifying *STRUCTURED_POINTS* as the *DATASET* type in the mesh structure portion of the file.

Here is an example of a VTK file with 3D structured points.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 3 2 2
ORIGIN 0 0 0
SPACING 1 1 1

CELL_DATA 2
SCALARS density float 1
LOOKUP_TABLE default
1 2

POINT_DATA 12
SCALARS u float 1
LOOKUP_TABLE default
1 2 3 1 2 3 1 2 3 1 2 3
SCALARS v float 1
LOOKUP_TABLE default
1 1 1 2 2 2 1 1 1 2 2 2
VECTORS velocity float
0 1 0
0 1 0
0 1 0
0 2 0
0 2 0
0 2 0
0 1 0
0 1 0
0 1 0
0 2 0
0 2 0
0 2 0
TENSORS stress float
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
1 0 0
0 1 0
0 0 1
1 0 0
```

(continues on next page)

(continued from previous page)

```
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
```

This defines a regular mesh consiting of 12 points and 2 cells. For the cells, it defines a single scalar variable named *density*. For the points, it defines 2 scalar variables named *u* and *v*, a vector variable named *velocity*, and a tensor variable named *stress*.

Here is an example of a VTK file with 2D structured points.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 3 2 1
ORIGIN 0 0 0
SPACING 1 1 1

CELL_DATA 2
SCALARS density float 1
LOOKUP_TABLE default
1 2

POINT_DATA 6
SCALARS u float 1
LOOKUP_TABLE default
1 2 3 1 2 3
SCALARS v float 1
LOOKUP_TABLE default
1 1 1 2 2 2
VECTORS velocity float
0 1 0
0 1 0
0 1 0
0 2 0
0 2 0
0 2 0
TENSORS stress float
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
```

(continues on next page)

(continued from previous page)

```
0 0 1
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
```

This defines a mesh similar to the previous one except that it has one row of points in the Z-direction. Note that the number of dimensions in the Z-direction is 1 and that Z origin is zero. The spacing in the Z-direction is one, but it could be any value.

8.2.8 An example of a structured grid file

A structured grid file results in a structured mesh where the position of each node in the mesh is specified. The mesh can be 2D or 3D. It is defined by specifying *STRUCTURED_GRID* as the *DATASET* type in the mesh structure portion of the file.

Here is an example of a VTK file with a 3D structured grid.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET STRUCTURED_GRID
DIMENSIONS 3 2 2
POINTS 12 float
0 0 0
1 1 0
2 1 0
0 1 0
1 2 0
2 2 0
0 0 1
1 1 1
2 1 1
0 1 1
1 2 1
2 2 1

CELL_DATA 2
SCALARS density float 1
LOOKUP_TABLE default
1 2

POINT_DATA 12
SCALARS u float 1
LOOKUP_TABLE default
1 2 3 1 2 3 1 2 3 1 2 3
SCALARS v float 1
LOOKUP_TABLE default
1 1 1 2 2 2 1 1 1 2 2 2
VECTORS velocity float
```

(continues on next page)

(continued from previous page)

```
0 1 0
0 1 0
0 1 0
0 2 0
0 2 0
0 2 0
0 1 0
0 1 0
0 1 0
0 2 0
0 2 0
0 2 0
TENSORS stress float
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
```

It defines a structured mesh that is the same as the structured point example except that the mesh points are no longer regular. This defines a structured mesh consiting of 12 points and 2 cells. For the cells, it defines a single scalar variable named *density*. For the points, it defines 2 scalar variables named *u* and *v*, a vector variable named *velocity*, and a tensor variable named *stress*.

Here is an example of a VTK file with a 2D structured grid.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET STRUCTURED_GRID
DIMENSIONS 3 2 1
POINTS 6 float
0 0 0
1 1 0
2 1 0
0 1 0
1 2 0
2 2 0

CELL_DATA 2
SCALARS density float 1
LOOKUP_TABLE default
1 2

POINT_DATA 6
SCALARS u float 1
LOOKUP_TABLE default
1 2 3 1 2 3
SCALARS v float 1
LOOKUP_TABLE default
1 1 1 2 2 2
VECTORS velocity float
0 1 0
0 1 0
0 1 0
0 2 0
0 2 0
0 2 0
TENSORS stress float
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
```

This defines a mesh similar to the previous one except that it has one row of points in the Z-direction. Note that the number of dimensions in the Z-direction is 1 and that the Z values are all zero.

8.2.9 An example of a rectilinear grid file

A rectilinear grid file results in a structured mesh where the coordinates along each axis are specified as a 1-D array of values. The mesh can be 2D or 3D. It is defined by specifying *RECTILINEAR_GRID* as the *DATASET* type in the mesh structure portion of the file.

Here is an example of a VTK file with a 3D rectilinear grid.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET RECTILINEAR_GRID
DIMENSIONS 3 2 2
X_COORDINATES 3 float
0 1 2
Y_COORDINATES 2 float
0 1
Z_COORDINATES 2 float
0 1

CELL_DATA 2
SCALARS density float 1
LOOKUP_TABLE default
1 2

POINT_DATA 12
SCALARS u float 1
LOOKUP_TABLE default
1 2 3 1 2 3 1 2 3 1 2 3
SCALARS v float 1
LOOKUP_TABLE default
1 1 1 2 2 2 1 1 1 2 2 2
VECTORS velocity float
0 1 0
0 1 0
0 1 0
0 2 0
0 2 0
0 2 0
0 1 0
0 1 0
0 1 0
0 2 0
0 2 0
0 2 0
TENSORS stress float
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
2 0 0
0 2 0
0 0 2
```

(continues on next page)

(continued from previous page)

```
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
```

It defines a rectilinear mesh that is the same as the structured point example.

Here is an example of a VTK file with a 2D rectilinear grid.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET RECTILINEAR_GRID
DIMENSIONS 3 2 1
X_COORDINATES 3 float
0 1 2
Y_COORDINATES 2 float
0 1
Z_COORDINATES 1 float
0

CELL_DATA 2
SCALARS density float 1
LOOKUP_TABLE default
1 2

POINT_DATA 6
SCALARS u float 1
LOOKUP_TABLE default
1 2 3 1 2 3
SCALARS v float 1
LOOKUP_TABLE default
1 1 1 2 2 2
VECTORS velocity float
0 1 0
0 1 0
0 1 0
```

(continues on next page)

(continued from previous page)

```
0 2 0
0 2 0
0 2 0
TENSORS stress float
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
2 0 0
0 2 0
0 0 2
```

This defines a mesh similar to the previous one except that it has one row of points in the Z-direction. Note that in the Z-direction there is 1 value and that it is zero.

8.2.10 An example of a polydata file

A polydata file results in a points, lines and surface cells. The mesh can be 2D or 3D. It is defined by specifying *POLYDATA* as the *DATASET* type in the mesh structure portion of the file.

Here is an example of a VTK file with polydata.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET POLYDATA
POINTS 17 float
-5 -5  5
 0 -5  5
 5 -5  5
 0  0 -5
 0  0  5
 0 -5  0
 0  5  0
-5  0  0
 5  0  0
-5 -5 -5
-5  5 -5
 5  5 -5
 5 -5 -5
-5 -5 -5
 5 -5 -5
-5 -5  5
 5 -5  5
```

(continues on next page)

(continued from previous page)

```
VERTICES 3 6
1 0
1 1
1 2

LINES 3 9
2 3 4
2 5 6
2 7 8

POLYGONS 1 5
4 9 10 11 12

TRIANGLE_STRIPS 1 5
4 13 14 15 16

CELL_DATA 8
SCALARS density float 1
LOOKUP_TABLE default
1 2 3 4 5 6 7 8

POINT_DATA 17
SCALARS u float 1
LOOKUP_TABLE default
0 0 0 0.5 0.5 1 1 1.5 1.5 2 2 2 2 3 3 3 3
SCALARS v float 1
LOOKUP_TABLE default
0 0 0 0.05 1 0.05 1 0.05 1 0.5 0.5 1 1 0.5 1 0.5 1
```

It supportes points, lines, polygons and triangle strips.

8.2.11 An example of a unstructured grid file

An unstructured grid file results in an arbitrary combination of cell types. The points are explicitly specified. The mesh can be 2D or 3D. It is defined by specifying *UNSTRUCTURED_GRID* as the *DATASET* type in the mesh structure portion of the file.

Here is an example of a VTK file with a 3D unstructured grid.

```
# vtk DataFile Version 3.0
vtk output
ASCII
DATASET UNSTRUCTURED_GRID
POINTS 48 float
0 0 0
1 0 0
2 0 0
3 0 0
4 0 0
0 1 0
1 1 0
2 1 0
3 1 0
4 1 0
```

(continues on next page)

(continued from previous page)

```
0 2 0
1 2 0
2 2 0
3 2 0
4 2 0
0 3 0
1 3 0
0 4 0
0 5 0
1 4 0
0 4 1
1 4 0
2 4 0
1 4 1
2 4 1
1 5 0
2 5 0
1 5 1
2 5 1
2 4 0
3 4 0
3 4 1
2 4 1
2 5 0
3 5 0
3 5 1
2 5 1
3 4 0
3 4 1
3 5 0
4 4 0
4 4 1
4 5 0
4 4 0
5 4 0
5 4 1
4 4 1
4 5 0

CELLS 17 82
1 0
1 1
3 2 3 4
2 5 6
2 6 7
3 7 8 9
3 5 10 6
3 10 6 11
4 6 11 7 12
4 7 12 13 8
4 8 9 13 14
4 10 15 16 11
4 17 18 19 20
8 21 22 23 24 25 26 27 28
8 29 30 31 32 33 34 35 36
6 37 38 39 40 41 42
5 43 44 45 46 47
```

(continues on next page)

(continued from previous page)

```
CELL_TYPES 17
1
1
2
3
3
4
5
5
6
7
8
9
10
11
12
13
14

CELL_DATA 17
SCALARS density float 1
LOOKUP_TABLE default
1 1 2 3 3 4 5 5 6 7 8 9 10 11 12 13 14
```

It contains all the linear element types.

8.3 The BOV file format

The BOV (Brick Of Values) format is a binary format used to represent a single variable defined on a regular mesh. The BOV file consists of two files, a file with the binary data, and a small text file describing it.

8.3.1 The structure of a BOV file

The BOV header file consists of a text file with keyword / value pairs, one per line. The BOV header file may be interspersed with comment lines that begin with a #. The supported keywords are:

BRICK_ORIGIN

This is the origin of your grid.

Here is an example of specifying the origin of your grid.

```
BRICK_ORIGIN: 0. 0. 0.
```

BRICK_SIZE

This is the size of your grid.

Here is an example of specifying the size of your grid.

```
BRICK_SIZE: 10. 10. 10.
```

The *i* component varies the fastest, then *j*, then *k*. This means that if *BRICK_SIZE* is *2. 2. 2.*, the first float in the data file corresponds to *[0,0,0]*, the second to *[1,0,0]*, the third to *[0,1,0]*, the fourth to *[1,1,0]*, the fifth to *[0,0,1]*, and so on.

BYTE_OFFSET

This is a byte offset into the file to start reading the data. It lets you specify some number of bytes to skip at the front of the file. This can be useful for skipping the 4-byte header that Fortran tends to write to files. If your file does not have a header then DO NOT USE BYTE_OFFSET.

Here is an example of specifying the size of your grid.

```
BYTE_OFFSET: 4
```

CENTERING

This is the centering of your variable. Valid values are *ZONAL* and *NODAL*.

Here is an example of specifying the centering of your variable.

```
CENTERING: ZONAL
```

DATA_BRICKLETS

Specifies the size of the chunks when *DIVIDE_BRICK* is true. The values chosen for *DATA_BRICKLETS* must be factors of the numbers used for *DATA_SIZE*.

Here is an example of specifying the data bricklets of your grid.

```
DATA_BRICKLETS: 5 5 5
```

DATA_COMPONENTS

Specifies the number of components in your variable. 1 specifies a scalar, 2 specifies a complex number, 3 specifies a vector, and any value beyond 3 specifies an array variable. You can use *COMPLEX* instead of *2* for complex numbers. When your data consists of multiple components, all components for a cell or node are written sequentially to the file before going to the next cell or node.

Here is an example of specifying the data bricklets of your grid.

```
DATA_COMPONENTS: 1
```

DATA_ENDIAN

This is the Endian representation of your data. Valid values are *BIG* and *LITTLE*. Data from Intel processors is *LITTLE*, while most other processors are *BIG*.

Here is an example of specifying the Endian representation of your variable.

```
DATA_ENDIAN: LITTLE
```

DATA_FILE

This is the name of the file with the binary data.

Here is an example of specifying the name of the data file.

```
DATA_FILE: density.bof
```

DATA_FORMAT

This is the type of your data. Valid values are *BYTE*, *SHORT*, *INT*, *FLOAT* and *DOUBLE*.

Here is an example of specifying the type of your variable.

```
DATA_FORMAT: FLOAT
```

DATA_SIZE

These are the dimensions of your variable.

Here is an example of specifying the dimensions of your variable.

```
DATA_SIZE: 10 10 10
```

DIVIDE_BRICK

A flag indicating if the array should be divided into chunks that can be processed in parallel. Valid values are *TRUE* and *FALSE*. When DIVIDE_BRICK is true, the values stored in DATA_BRICKLETS specifies the size of the chunks.

Here is an example of specifying divide brick.

```
DIVIDE_BRICK: TRUE
```

TIME

This is the time associated with the variable.

Here is an example of specifying the time.

```
TIME: 10.
```

VARIABLE

This is the name of your variable.

Here is an example of specifying the name of your variable.

```
VARIABLE: density
```

8.3.2 An example of a BOV file

Here is an example header file representing a 10 by 10 by 10 array of density values.

```
TIME: 10.
DATA_FILE: density.bof
DATA_SIZE: 10 10 10
DATA_FORMAT: FLOAT
VARIABLE: density
DATA_ENDIAN: LITTLE
CENTERING: ZONAL
BRICK_ORIGIN: 0. 0. 0.
BRICK_SIZE: 10. 10. 10.
```

Here is a sample python script that creates the 10 by 10 by 10 array of density values.

```
import math
from array import array
nx = 10
ny = 10
nz = 10
vals = array('f')
for iz in range(nz):
    z = float(iz)
    for iy in range(ny):
        y = float(iy)
        for ix in range(nx):
            x = float(ix)
            vals.append(math.sqrt(x*x+y*y+z*z))

f = open("density.bof", "wb")
vals.tofile(f)
f.close()
```

8.4 The Curve file format

A curve file is a text file for specifying 1D curves. It consists of one or more comment lines with meta data about the curve followed by the coordinates. The comment lines start with either a # or % character. Comment lines consist of keyword / value pairs. Supported keywords are *CYCLE* and *TIME*. The last comment line contains the name of the curve. The coordinates are provided one point per line. Both Cartesian and Polar coordinates are supported. Cartesion coordinates are assumed by default. This can be overriden with options to the Curve plot. A curve file may contain multiple curves per file.

Here is a sample curve file.

```
#TIME 2.5
#CYCLE 25
#Max density vs time
0. 1.
1. 1.1
2. 1.2
3. 1.3
4. 1.4
5. 1.6
#TIME 2.5
```

(continues on next page)

(continued from previous page)

```
#CYCLE 25
#Max pressure vs time
0. 2.
1. 2.2
2. 2.4
3. 2.5
4. 2.6
5. 2.8
```

8.5 The PlainText file format

Standard CSV (Comma Separated Values) files are read using the `PlainText` reader.

Plain text files are text files with columns of data. A *single* space, comma or tab character separates (e.g. *delimits*) values in each line of the file belonging to the different columns. The `PlainText` reader automatically detects the separator character in use. The file can include an arbitrary number of lines at the *beginning* of the file to be skipped. Following any skipped lines, the file may include an optional *header* line holding the names associated with each column. Plain text files can be used to represent the following types of data:

- A collection of curves, all defined on the same (explicit or implicit) domain.
- Points in 2D or 3D with variables defined on the points.
- A single variable defined on a 2D, uniform grid.

8.5.1 Defining curves with a `PlainText` file

The first line can be an optional list of variable names. The remaining lines consist of rows, where each row represents one point in each of the curves. In this example, the values on each row are separated by commas.

Here are the first 10 lines of an example of a file representing curves.

```
angle,sine,cosine
0,0,1
5,0.0871557,0.996195
10,0.173648,0.984808
15,0.258819,0.965926
20,0.34202,0.939693
25,0.422618,0.906308
30,0.5,0.866025
35,0.573576,0.819152
40,0.642788,0.766044
```

Here is the Python script that created the file.

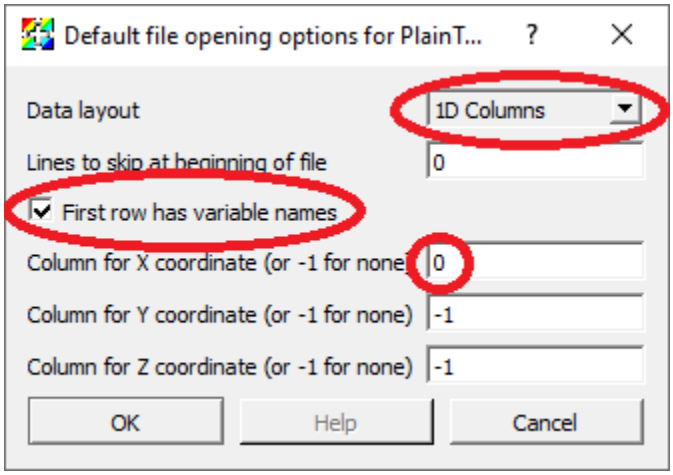
```
    with open(filename, "wt") as f:
        # create header
        f.write("angle,sine,cosine\n")
        npts = 73
        for i in range(npts):
            angle_deg = float(i) * (360. / float(npts-1))
            angle_rad = angle_deg * (3.1415926535 / 180.)
            sine = math.sin(angle_rad)
            cosine = math.cos(angle_rad)
```

(continues on next page)

(continued from previous page)

```
            # write abscissa (x value) and ordinates (y-value(s))
            f.write("%g,%g,%g\n" % (angle_deg, sine, cosine))
```

Here are the `PlainText` reader options used to read the data.



If you specify the column for the X coordinates, then that column will be used for the domain for all the curves. If you don't specify an X coordinate, then it will use the row index for the domain for all the curves.

Here is the Python code to plot this data in VisIt

```
    plainTextOpenOptions = GetDefaultOpenOptions()
    plainTextOpenOptions['First row has variable names'] = 1
    plainTextOpenOptions['Column for X coordinate (or -1 for none)'] = 0
    SetDefaultFileOpenOptions("PlainText", plainTextOpenOptions)

    OpenDatabase("curves.csv")
    AddPlot("Curve","sine")
    AddPlot("Curve","cosine")
    DrawPlots()
```

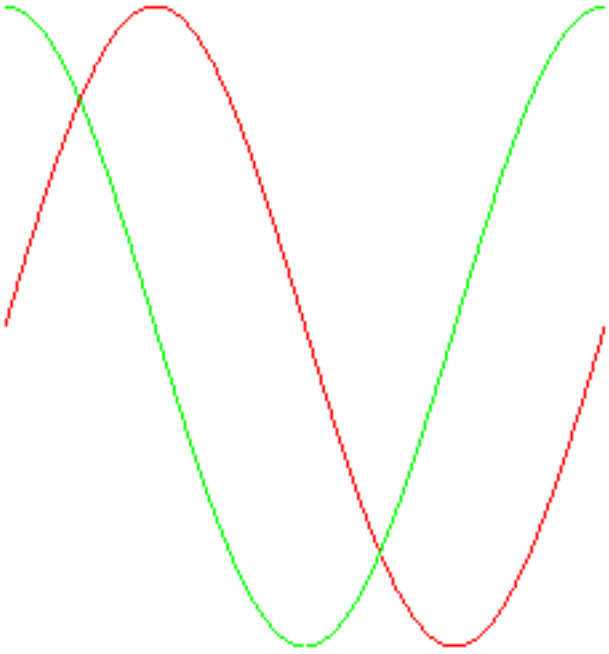
and the resulting data plotted in VisIt

8.5.2 Defining curves using row index for X coordinate

Here are the first 10 lines of an example of a file representing curves where the *abscissa* (e.g. x-coordinate) is implied by the row number (starting from 0) in the file. In this example, the values on each row are separated by commas.

```
inverse,sqrt,quadratic
100,0,0
50,10,0.01
33.3333,14.1421,0.04
25,17.3205,0.09
20,20,0.16
16.6667,22.3607,0.25
14.2857,24.4949,0.36
12.5,26.4575,0.49
11.1111,28.2843,0.64
```

Here is the Python script that created the file.



```
    with open(filename, "wt") as f:
        # create header
        f.write("inverse,sqrt,quadratic\n")
        npts = 100
        for i in range(npts):
            inv = float(100) / (float(i)+1)
            sqr = 10 * math.sqrt(i)
            quad = float(i*i) / float(100)
            f.write("%g,%g,%g\n" % (inv, sqr, quad))
```

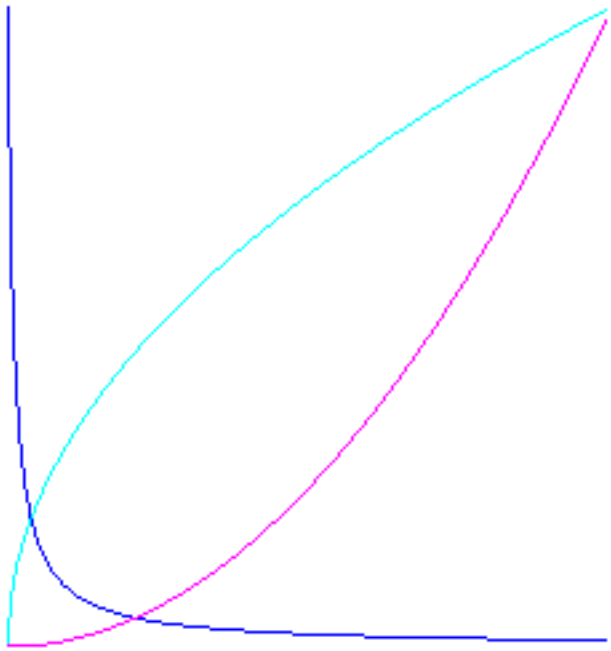
Here is the Python code to plot this data in VisIt

```
    plainTextOpenOptions = GetDefaultOpenOptions()
    plainTextOpenOptions['First row has variable names'] = 1
    SetDefaultFileOpenOptions("PlainText", plainTextOpenOptions)

    OpenDatabase("curves_nox.csv")
    AddPlot("Curve","inverse")
    AddPlot("Curve","sqrt")
    AddPlot("Curve","quadratic")
    DrawPlots()
```

and the resulting data plotted in VisIt

8.5.3 Defining 2D or 3D points with variables

The first line can be an optional list of variable names. The remaining lines consist of rows, where each row represents the coordinates and variable values for a single point. In this example, the values on each row are separated by spaces.



Here are the first 10 lines of an example of a file representing 3D points.

```
x y z velx vely velz temp
0 0 0 0 0 0 0.5
0.0959668 0.0315185 0.10101 0.10101 0.105813 0.139329 0.489899
0.162681 0.119779 0.20202 0.20202 0.23486 0.259379 0.479798
0.175775 0.246841 0.30303 0.30303 0.390843 0.35032 0.469697
0.119968 0.385819 0.40404 0.40404 0.558664 0.421475 0.459596
-0.00801311 0.504987 0.505051 0.505051 0.714204 0.505114 0.449495
-0.198223 0.572728 0.606061 0.606061 0.833862 0.637653 0.439394
-0.428209 0.56266 0.707071 0.707071 0.903623 0.826627 0.429293
-0.665597 0.45823 0.808081 0.808081 0.928962 1.04691 0.419192
```

Here is the Python script that created the file.

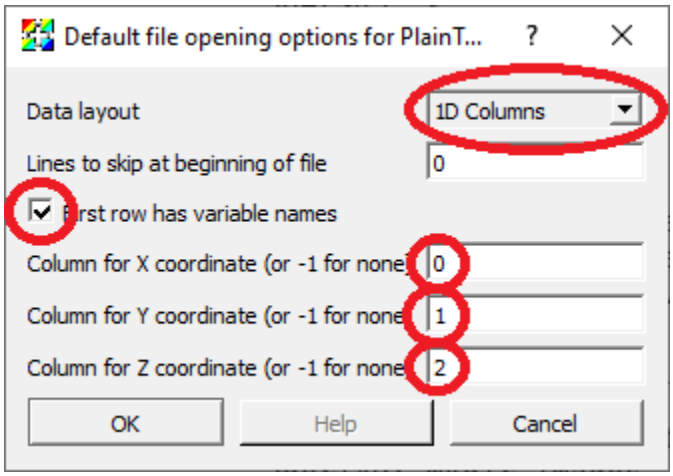
```
    with open(filename, "wt") as f:
        # write header
        f.write("x y z velx vely velz temp\n")
        n = 100
        for i in range(n):
            t = float(i) / float(n-1)
            angle = t * (math.pi * 2.) * 5.
            r = t * 10.
            x = r * math.cos(angle)
            y = r * math.sin(angle)
            z = t * 10.
            vx = math.sqrt(x*x + y*y)
            vy = math.sqrt(y*y + z*z)
            vz = math.sqrt(x*x + z*z)
```

(continues on next page)

(continued from previous page)

```
            temp = math.sqrt((t-0.5)*(t-0.5))
            # write point and value(s)
            f.write("%g %g %g %g %g %g %g\n" % (x,y,z,vx,vy,vz,temp))
```

Here are the `PlainText` reader options used to read the data.



If you specify the columns for the X and Y coordinates, the points will be defined in 2D space. If you specify the columns for the X, Y and Z coordinates, the points will be defined in 3D space.

Here is the Python code to plot this data in VisIt

```
plainTextOpenOptions = GetDefaultOpenOptions()
plainTextOpenOptions['First row has variable names'] = 1
plainTextOpenOptions['Column for X coordinate (or -1 for none)'] = 0
plainTextOpenOptions['Column for Y coordinate (or -1 for none)'] = 1
plainTextOpenOptions['Column for Z coordinate (or -1 for none)'] = 2
SetDefaultFileOpenOptions("PlainText", plainTextOpenOptions)
OpenDatabase("points.txt")

DefineVectorExpression("vel", "{velx,vely,velz}")
AddPlot("Pseudocolor", "temp")
AddPlot("Vector","vel")
DrawPlots()
```

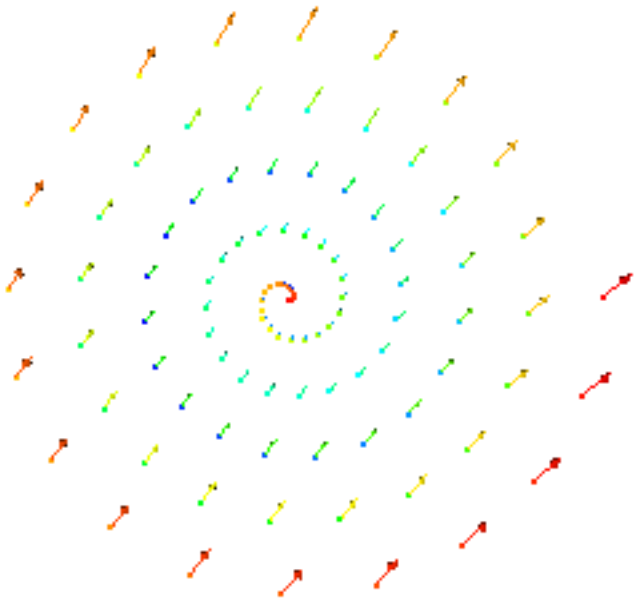
and the resulting data plotted in VisIt

8.5.4 Defining a single variable on a 2D uniform grid

The data is interpreted as a node centered variable on a uniform mesh where the row and column indices define the X and Y coordinates. The rows represent values along the X direction and the rows get *stacked* in the Y direction. Each row further *down* in the file gets stacked *up*, one upon the other in the visualized result in VisIt. This means that the row-by-row *downward* direction in the file listing is the same as the *upward* (positive Y) direction in the visualized result in VisIt.

The first line can be an optional list of variable names. The first column name will be used for the name of the variable. Other column names are ignored but nonetheless required to read the file properly. The remaining lines consist of rows, where each row represents the values for a single Y coordinate.

Here is an example of a file listing representing 3D points. In this example, the values on each row are separated by spaces.



```
density c2 c3 c4 c5 c6 c7 c8
0 1 2 3 4 5 6 7
1 1.41421 2.23607 3.16228 4.12311 5.09902 6.08276 7.07107
2 2.23607 2.82843 3.60555 4.47214 5.38516 6.32456 7.28011
3 3.16228 3.60555 4.24264 5 5.83095 6.7082 7.61577
4 4.12311 4.47214 5 5.65685 6.40312 7.2111 8.06226
5 5.09902 5.38516 5.83095 6.40312 7.07107 7.81025 8.60233
6 6.08276 6.32456 6.7082 7.2111 7.81025 8.48528 9.21954
7 7.07107 7.28011 7.61577 8.06226 8.60233 9.21954 9.89949
8 8.06226 8.24621 8.544 8.94427 9.43398 10 10.6301
9 9.05539 9.21954 9.48683 9.84886 10.2956 10.8167 11.4018
```

Here is the Python script that created the file.

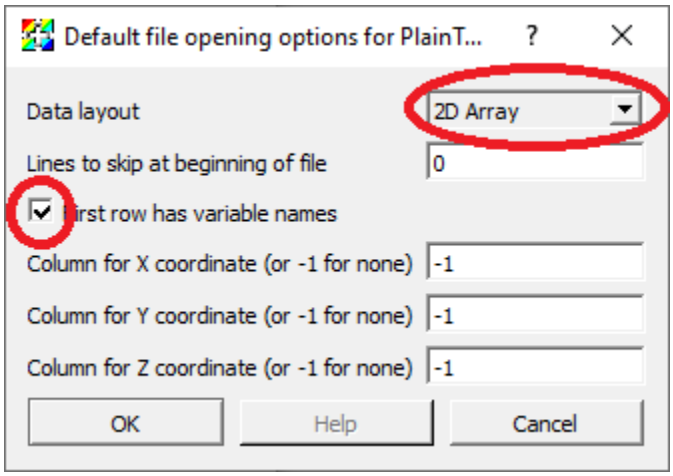
```
    with open(filename, "wt") as f:
        # Only the first column name matters.
        # The others are required but otherwise ignored.
        f.write("density c2 c3 c4 c5 c6 c7 c8\n")
        nx = 8
        ny = 10
        for iy in range(ny):
            y = float(iy)
            for ix in range(nx):
                x = float(ix)
                dist = math.sqrt(x*x + y*y)
                if (ix < nx - 1):
                   f.write("%g " % dist)
                else:
```

(continues on next page)

(continued from previous page)

```
                f.write("%g\n" % dist)
```

Here are the `PlainText` reader options used to read the data.



The columns for the X, Y and Z coordinates are not used.

Here is the Python code to plot this data in VisIt

```
plainTextOpenOptions = GetDefaultOpenOptions()
plainTextOpenOptions['First row has variable names'] = 1
plainTextOpenOptions['Data layout'] = '2D Array'
SetDefaultFileOpenOptions("PlainText", plainTextOpenOptions)

OpenDatabase("array.txt")
AddPlot("Pseudocolor", "density")
DrawPlots()
ResetView()
```

and the resulting data plotted in VisIt

Note that the reddest part of the plot (e.g. highest numerical values in the data) appears in the upper right corner of the plot whereas the highest numerical values in the file data row-by-row listing appears in the *lower* right corner.

8.6 The Silo file format

If you are writing a conversion utility or if you have a simulation code written in C, C++, or Fortran then writing out Silo files is a good choice for getting your data into VisIt. One reason for this is that among all of VisIt's plugins, the Silo plugin is likely one of the most advanced in terms of the various data features it supports. This section will illustrate how to use the Silo library to write out various types of scientific data. Since the Silo library provides bindings for multiple languages, including C, Fortran, and Python, the source code examples that demonstrate a particular topic will be given in more than one programming language, when appropriate. One goal of this section is to provide examples that are complete enough so that they can be readily adapted into working source code. This section will not necessarily explain all of the various arguments to function calls in the Silo library. You can refer to the Silo Manual for more information.

8.6.1 Using the Silo library

This subsection includes information about using Silo such as including the appropriate header files and linking with the Silo library.

Including Silo

When using any library in a program, you must tell the compiler about the symbols provided by the library. Here is what you need to include in your source code in order to use Silo:

C

```
#include <silo.h>
```

FortranFixed

```
include "silo.inc"
```

Linking with Silo

Before you can build a program that uses Silo, you must locate the Silo header and library files. Silo is distributed as part of VisIt binary distributions but is installed in those distributions differently than it would be if it was installed as a stand-alone package. For example, in Linux distros, the library file, `libsiloh5.so` (or `libsilo.so` for non-HDF5 based Silo), is in the VisIt installation's `<version>/<arch>/lib` directory and the header file, `silo.h`, is in the VisIt installation's `<version>/<arch>/include/silo` directory. A link to the most up-to-date version of the Silo library's source code can be found on Silo's Github site.

Once you download the Silo source code, building and installing it is usually only a matter of running its `configure` script and running `make`. You can even use the `build_visit` script from the VisIt Web site to build Silo with support for HDF5. An example command line to build Silo with support for HDF5 is:

```
./build_visit --console --no-visit --no-thirdparty \
--thirdparty-path /usr/local \
--silo --hdf5 --szip
```

Although Silo does not *require* HDF5, it is best to build Silo with it because the HDF5 driver supports more features than Silo's default (and built-in) Portable DataBase (PDB) driver. After you've configured, built, and installed the Silo library, your program will have to be built against the Silo library. Building against the Silo library is usually accomplished by a simple adaptation of your `Makefile` and the inclusion of `silo.h` in your C-language source code. If you used `build_visit` to install Silo and its dependant libraries in `/usr/local` then you would add the following to your `Makefile`, adjusting the values for the install location, versions, and PLATFORM accordingly:

```
PLATFORM=i386-apple-darwin10_gcc-4.2
SZIP_DIR=/usr/local/szip/2.1/$(PLATFORM)
SZIP_CPPFLAGS=-I$(SZIP_DIR)/include
SZIP_LDFLAGS=-L$(SZIP_DIR)/lib
SZIP_LIBS=-lsz
HDF5_DIR=/usr/local/hdf5/1.8.4/$(PLATFORM)
HDF5_CPPFLAGS=-I$(HDF5_DIR)/include $(SZIP_CPPFLAGS)
HDF5_LDFLAGS=-L$(HDF5_DIR)/lib $(SZIP_LDFLAGS)
HDF5_LIBS=-lhdf5 $(SZIP_LIBS) -lz
SILO_DIR=/usr/local/silo/4.6.2/$(PLATFORM)
SILO_CPPFLAGS=-I$(SILO_DIR)/include $(HDF5_CPPFLAGS)
SILO_LDFLAGS=-L$(SILO_DIR)/lib $(HDF5_LDFLAGS)
SILO_LIBS=-lsiloh5 $(HDF5_LIBS) -lm
LDFLAGS=$(LDFLAGS) $(SILO_LDFLAGS)
LIBS=$(SILO_LIBS)
CPPFLAGS=$(CPPFLAGS) $(SILO_CPPFLAGS)
```

If your `Makefile` does not use `CPPFLAGS` then you might try adding the `-I` include directives to `CFLAGS`, `F77FLAGS`, or whichever make variables are relevant for your `Makefile`.

Using Silo on Windows

When you build an application using the Silo library on Windows, you can use the precompiled Silo DLL and import library that comes with the VisIt development distribution for Windows: `visit_windowsdev_x.y.x.zip`, where `x.y.z` refers to the version, like `3.3.3`. The development distribution for Windows includes pre-built binaries (`.dlls` and import libraries) for the Third party libraries upon which VisIt depends, including Silo. Simply unzip the distribution to whichever location best suits your needs. The binaries are located in the `windowsbuild/MSVC<VERSION>` folder, with `<VERSION>` being the version of Visual Studio they were built with (eg `MSVC2017`).

If you want to build an application against the Silo library provided with VisIt, add the path to find the `silo.h` to your project file. After setting the Silo include directory to your project file, make sure that the Silo's import library is in your linker path. Next, add `silohdf5.lib` (or `siloh5.lib`, depending on the version of Silo) to the list of libraries that are linked with your program. That should be enough to get your program to build.

Before running your program, be sure to copy `silohdf5.dll`, `hdf5dll.dll`, `sziplib.dll`, and `zlib.dll` from the VisIt windows distribution into the directory where your program will execute. Note that you must configure your program to use a Multithreaded DLL version of the Microsoft runtime library or using the precompiled Silo library may result in fatal errors.

8.6.2 Inspecting Silo files

Unless it was explicitly *disabled* in the configuration, Silo includes a command line utility called *browser* that can be used to textually browse the contents of Silo files much like the Linux shell enables browsing a the Linux file system. To run the browser, type `browser` into a terminal window followed by the name of a Silo file that you want to inspect. Once the browser application opens the Silo file, type `ls` to see the contents of the Silo file, `cd` to move between Silo directories within the file, etc. From there, typing the name of any of the objects shown in the object listing will print information about that object to the console.

Silo also supports a point-n-click interface for inspecting a Silo file called *silex* as well as a python extension module for reading and writing Silo files. These additional Silo tools are built only via Silo's own tools and not via `build_visit`.

8.6.3 Silo files and parallel codes

Before we delve into examples about how to use the Silo library, let's first examine how parallel simulation codes process their data in a distributed-memory environment. Many parallel simulation codes will divide the entire simulated mesh into submeshes, called *domains*, which are assigned to parallel tasks (e.g. MPI ranks) that calculate the fields of interest on their respective domain(s). Often, the most efficient I/O strategy for the simulation code is to make each processor write its domain to a separate file. The examples that follow assume parallel simulations will write 1 file per processor. It is possible for multiple processors to append their data to a single Silo file but it requires synchronization and that technique is beyond the scope of the examples presented here.

This paradigm for handling parallel I/O with Silo is known as the Multiple Independent File (MIF) Parallel I/O Paradigm In fact, there is a header file available in Silo, `pmpio.h`, which facilitates this mode of *writing* Silo files. An example of its use can be found in Silo's test suite, pmpio_silo_test_mesh.c

8.6.4 Creating a new Silo file

The first step to saving data to a Silo file is to create the file and obtain a handle that will be used to reference the file. The handle will be passed to other Silo function calls in order to add new objects to the file. Silo creates new files using the `DBCreate` function, which takes the name of the new file, access modes, a descriptive comment, and the underlying file type as arguments. In addition to being a library, Silo is a self-describing data model, which can be implemented on top of many different underlying file formats. Silo includes drivers that allow it to read data from several different file formats, the most important of which are: Portable DataBase (PDB) (A legacy LLNL format), and HDF5 format. Silo files stored in HDF5 format often provide performance advantages so the following code to open a Silo file will create HDF5-based Silo files. You tell Silo to create HDF5-based Silo files by passing the `DB_HDF5` argument to the `DBCreate` function. If your Silo library does not have built-in HDF5 support then you can pass `DB_PDB` instead to create PDB-based Silo files.

Example for creating a new Silo file:

C

```
#include <silo.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
    DBfile *dbfile = NULL;
    /* Open the Silo file */
    dbfile = DBCreate("basic.silo", DB_CLOBBER, DB_LOCAL,
                      "Comment about the data", DB_HDF5);
```

(continues on next page)

(continued from previous page)

```
    if(dbfile == NULL)
    {
        fprintf(stderr, "Could not create Silo file!\n");
        return -1;
    }
    /* Add other Silo calls here. */
    /* Close the Silo file. */
    DBClose(dbfile);
    return 0;
}
```

FortranFixed

```
    progam main
    implicit none
    include "silo.inc"
    integer dbfile, ierr
c The 11 and 22 arguments represent the lengths of strings
    ierr = dbcreate("fbasic.silo", 11, DB_CLOBBER, DB_LOCAL,
    .               "Comment about the data", 22, DB_HDF5, dbfile)
    if(dbfile.eq.-1) then
        write (6,*) 'Could not create Silo file!\n'
        goto 10000
    endif
c Add other Silo calls here.
c Close the Silo file.
    ierr = dbclose(dbfile)
10000 stop
    end
```

In addition to using the `DBCreate` function, the previous examples also use the `DBClose` function. The `DBClose` function ensures that all data is written to the file and then closes the Silo file. You must call the `DBClose` function when you want to close a Silo file or your file may not be complete.

8.6.5 Dealing with time

A Silo file is a flexible container for storing many types of data. Silo's ability to store data hierarchically in directories can allow you to store multiple time states of your simulation data within a single data file. However, Silo is most often used to store one time state per Silo file (or ensemble of files in a parallel context) VisIt's Silo plugin is primarily designed and used to work with Silo files in this modality. Consequently, when writing data, programs that use Silo will write a new Silo file for each time step. By convention, the new file will contain an index indicating either the simulation cycle or a simple integer counter.

Listing 8.1: C-Language example for dealing with time

```
/* SIMPLE SIMULATION SKELETON */
void write_vis_dump(int cycle)
{
    DBfile *dbfile = NULL;
    /* Create a unique filename for the new Silo file*/
    char filename[100];
    sprintf(filename, "output%04d.silo", cycle);
    /* Open the Silo file */
    dbfile = DBCreate(filename, DB_CLOBBER, DB_LOCAL,
        "simulation time step", DB_HDF5);
```

(continues on next page)

(continued from previous page)

```
    /* Add other Silo calls to write data here. */
    /* Close the Silo file. */
    DBClose(dbfile);
}

int main(int, char **)
{
    int cycle = 0;
    read_input_deck();
    do
    {
        simulate_one_timestep();
        write_vis_dump(cycle);
        cycle = cycle + 1;
    } while(!simulation_done());
    return 0;
}
```

The above code listing will write out Silo files with names such as: `output0000.silo`, `output0001.silo`, `output0002.silo`, ... Each file contains the data from a particular simulation time state. It may seem like the data are less related because they are stored in different files but the fact that the files are related in time is subtly encoded in the name of each of the files. When VisIt recognizes a pattern in the names of the files such as `output????.silo`, in this case, VisIt automatically recognizes the files as a time-varying database (e.g. a *virtual* database). If you choose names for your Silo files that cannot be grouped by recognizing a numeric pattern in the trailing part of the file name then you must use a *.visit* file to tell VisIt that your files are related in time.

8.6.6 Option lists

Many of Silo's more complex functions accept an auxiliary argument called an option list. An option list is a list of option/value pairs and it is used to specify additional metadata about the data being stored. Each Silo function that accepts an option list has its options enumerated in the Silo Manual. We cover only a subset of available options here. Option lists need not be passed to the Silo functions that do support them. In fact, most of the source code examples in this manual will pass NULL instead of passing a pointer to an option list. Omitting the option list from the Silo function call in this way is not harmful; it only means that certain pieces of additional metadata will not be stored with the data.

Option lists are created using the `DBMakeOptlist` function. Although this function requires the caller to specify a (maximum) number of options, newer versions (>= version 4.9) of the Silo library handle cases where the caller adds more options than this maximum number without issue. Once an option list object is created, you can add options to it using the `DBAddOption` function. Option lists are freed using the `DBFreeOptlist` function.

Any pointers passed in a `DBAddOption` call must not be changed until after the last Silo call in which the associated option list is used is made. A common mistake is for callers to pass a pointer to an *automatic* variable in a subroutine. That pointer becomes invalid upon returning from the subroutine where it was set and when the option list is later used, the associated option is problematic.

Any pointers passed in a `DBAddOption` call must not be changed until after the last Silo call in which the associated option list is used is made. A common mistake is for callers to pass a pointer to an *automatic* variable in a subroutine. That pointer becomes invalid upon returning from the subroutine where it was set and when the option list is later used, the associated option is problematic.

Cycle and time

We've explained that a notion of time can be encoded into filenames using ranges of digits in each filename. VisIt can use the numbers in the names of related files to guess cycle number, a metric for how many times a simulation has iterated. It is also possible to use Silo's option list feature to directly encode the cycle number and the simulation time into the stored data.

Example for saving cycle and time using an option list.

C

```
/* Create an option list to save cycle and time values. */
int cycle = 100;
double dtime = 1.23456789;
DBoptlist *optlist = DBMakeOptlist(2);
DBAddOption(optlist, DBOPT_DTIME, &time);
DBAddOption(optlist, DBOPT_CYCLE, &cycle);
/* Write a mesh using the option list. */
DBPutQuadmesh(dbfile, "quadmesh", coordnames, coords, dims, ndims,
              DB_FLOAT, DB_COLLINEAR, optlist);
/* Free the option list. */
DBFreeOptlist(optlist);
```

FortranFixed

```
c Create an option list to save cycle and time values.
    integer cycle /100/
    double precision dtime /1.23456789/
    integer err, ierr, optlistid
    err = dbmkoptlist(2, optlistid)
    err = dbaddiopt(optlistid, DBOPT_CYCLE, cycle)
    err = dbadddopt(optlistid, DBOPT_DTIME, dtime)
c Write a mesh using the option list.
    err = dbputqm (dbfile, "quadmesh", 8, "xc", 2, "yc", 2,
    .              "zc", 2, x, y, DB_F77NULL, dims, ndims,
    .              DB_FLOAT, DB_COLLINEAR, optlistid, ierr)
c Free the option list.
    err = dbfreeoptlist(optlistid)
```

8.6.7 Writing a rectilinear mesh

A rectilinear mesh is a 2D or 3D mesh where all coordinates are aligned with the axes. Each axis of the rectilinear mesh can have different, non-uniform spacing, allowing for details to be concentrated in certain regions of the mesh. Rectlinear meshes are specified by lists of coordinate values for each axis. Since the mesh is aligned to the axes, it is only necessary to specify one set of values along each axis to specify all of the coordinates for the entire mesh. Figure 8.1 contains an example of a 2D rectilinear mesh. The Silo function call to write a rectlinear mesh is called `DBPutQuadmesh`.

Example for writing a 2D rectilinear mesh:

C

```
/* Write a rectilinear mesh. */
float x[] = {0., 1., 2.5, 5.};
float y[] = {0., 2., 2.25, 2.55, 5.};
int dims[] = {4, 5};
int ndims = 2;
```

(continues on next page)

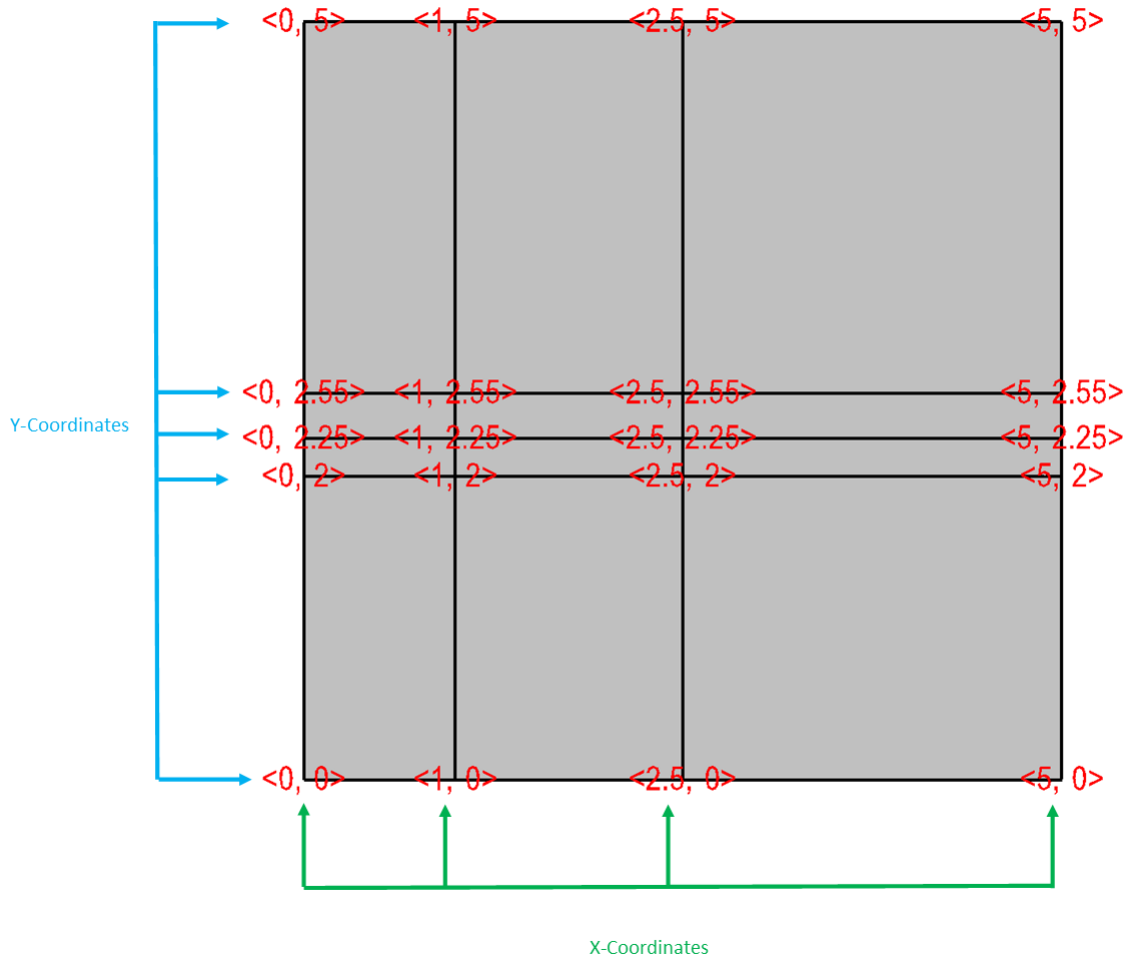


Fig. 8.1: Rectilinear mesh and its X,Y node coordinates.

(continued from previous page)

```
float *coords[] = {x, y};
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
    DB_FLOAT, DB_COLLINEAR, NULL);
```

FortranFixed

```
c Write a rectilinear mesh
    integer err, ierr, dims(2), ndims, NX, NY
    parameter (NX = 4)
    parameter (NY = 5)
    real x(NX), y(NY)
    data dims/NX, NY/
    data x/0., 1., 2.5, 5./
    data y/0., 2., 2.25, 2.55, 5./
    ndims = 2
    err = dbputqm (dbfile, "quadmesh", 8, "xc", 2, "yc", 2,
    .              "zc", 2, x, y, DB_F77NULL, dims, ndims,
    .              DB_FLOAT, DB_COLLINEAR, DB_F77NULL, ierr)
```

The previous code examples demonstrate how to write out a 2D rectilinear mesh using Silo's `DBPutQuadmesh` function (called `dbputqm` in Fortran). There are three pieces of important information passed to the `DBPutQuadmesh` function. The first important piece information is the name of the mesh being created. The name that you choose will be the name that you use when writing a variable to a Silo file and also the name that you will see in VisIt's plot menus when you want to create a Mesh plot in VisIt. After the name, you provide the coordinate arrays that contain the X and Y point values that ultimately form the set of X,Y coordinate pairs that describe the mesh. The C-interface to Silo requires that you pass pointers to the coordinate arrays in a single pointer array. The Fortran interface to Silo requires you to pass the names of the coordinate arrays, followed by the actual coordinate arrays, with a value of `DB_F77NULL` for any arrays that you do not use. The final critical pieces of information that must be passed to the `DBPutQuadmesh` function are the dimensions of the mesh, which correspond to the number of nodes, or coordinate values, along the mesh in a given dimension. The dimensions are passed in an array, along with the number of dimensions, which must be 2 or 3. Figure 8.2 shows an example of a 3D rectilinear mesh for the upcoming code examples.

Example for writing a 3D rectilinear mesh:

C

```
/* Write a rectilinear mesh. */
float x[] = {0., 1., 2.5, 5.};
float y[] = {0., 2., 2.25, 2.55, 5.};
float z[] = {0., 1., 3.};
int dims[] = {4, 5, 3};
int ndims = 3;
float *coords[] = {x, y, z};
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
    DB_FLOAT, DB_COLLINEAR, NULL);
```

FortranFixed

```
integer err, ierr, dims(3), ndims, NX, NY, NZ
parameter (NX = 4)
parameter (NY = 5)
parameter (NZ = 3)
real x(NX), y(NY), z(NZ)
data x/0., 1., 2.5, 5./
data y/0., 2., 2.25, 2.55, 5./
data z/0., 1., 3./
```

(continues on next page)

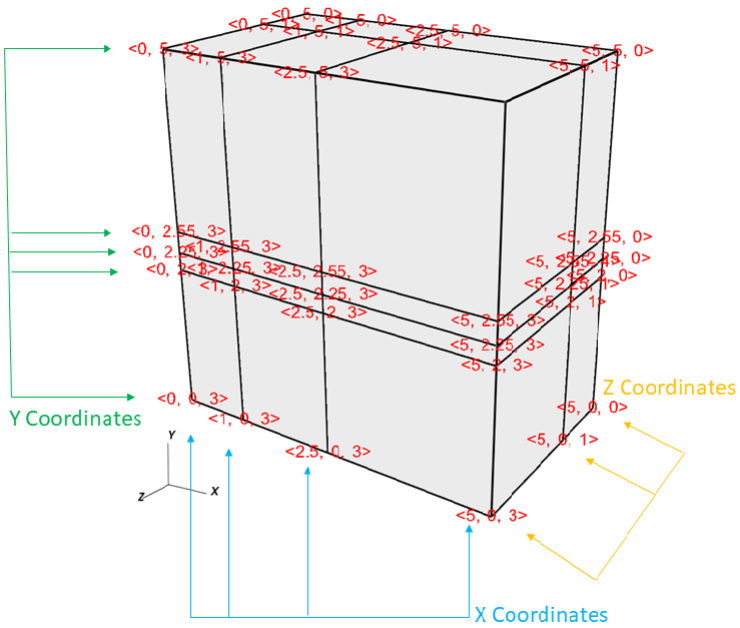


Fig. 8.2: Rectilinear mesh and its X,Y,Z coordinates

(continued from previous page)

```
ndims = 3
data dims/NX, NY, NZ/
err = dbputqm (dbfile, "quadmesh", 8, "xc", 2,
.               "yc", 2, "zc", 2, x, y, z, dims, ndims,
.               DB_FLOAT, DB_COLLINEAR, DB_F77NULL, ierr)
```

8.6.8 Writing a curvilinear mesh

A curvilinear mesh is similar to a rectlinear mesh. The main difference between the two mesh types is how coordinates are specified. Recall that in a rectilinear mesh, the coordinates are specified individually for each axis. Their cross-product determines the coordinates of any one node in the mesh. In a curvilinear mesh, you must provide an X,Y,Z value for every node in the mesh. Providing the coordinates for every point explicitly allows you to specify more complex geometries than are possible using rectilinear meshes. Note how the mesh coordinates on the mesh in Figure 8.3 allow it to assume shapes that are not aligned to the coordinate axes.

The fine line between a rectilinear mesh and a curvilinear mesh comes down to how the coordinates are specified. Silo dicates that the coordinates be specified with an array of X coordinates, an array of Y-coordinates, and an optional array of Z-coordinates. The difference, of course, is that in a curvilinear mesh, there are explicit values for each node's X,Y,Z points. Silo uses the same `DBPutQuadmesh` function to write out curvilinear meshes. The coordinate arrays are passed the same as for the rectilinear mesh, though the X,Y,Z arrays now point to larger arrays. You can pass the `DB_NONCOLLINEAR` flag to the `DBPutQuadmesh` function in order to indicate that the coordinate arrays contain values for every node in the mesh.

Example for writing a 2D curvilinear mesh:

C

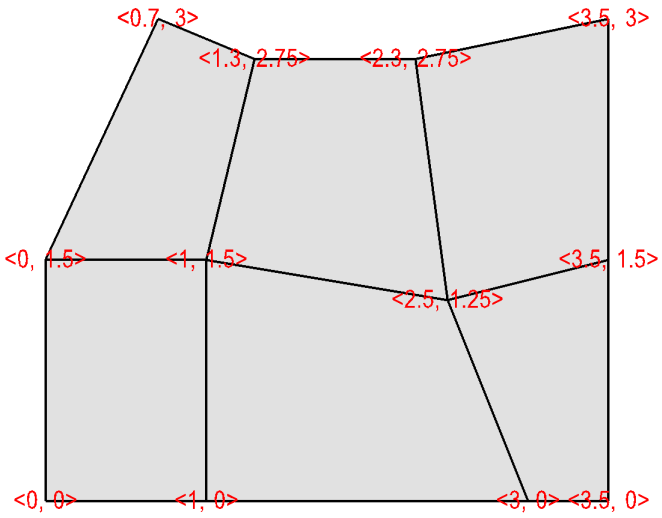


Fig. 8.3: Curvilinear mesh and its X,Y node coordinates

```
/* Write a curvilinear mesh. */
#define NX 4
#define NY 3
float x[NY][NX] = {{0., 1., 3., 3.5}, {0., 1., 2.5, 3.5},
    {0.7, 1.3, 2.3, 3.5}};
float y[NY][NX] = {{0., 0., 0., 0.}, {1.5, 1.5, 1.25, 1.5},
    {3., 2.75, 2.75, 3.}};
int dims[] = {NX, NY};
int ndims = 2;
float *coords[] = {(float*)x, (float*)y};
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
              DB_FLOAT, DB_NONCOLLINEAR, NULL);
```

FortranFixed

```
c Write a curvilinear mesh.
    integer err, ierr, dims(2), ndims, NX, NY
    parameter (NX = 4)
    parameter (NY = 3)
    real x(NX,NY), y(NX,NY)
    data x/0., 1.,  3.,  3.5,
    .    0.,   1.,  2.5, 3.5,
    .    0.7,  1.3, 2.3, 3.5/
    data y/0., 0.,   0.,   0.,
    .    1.5,  1.5,  1.25, 1.5,
    .    3.,   2.75, 2.75, 3./
    ndims = 2
    data dims/NX, NY/
    err = dbputqm (dbfile, "quadmesh", 8, "xc", 2, "yc", 2,
    .              "zc", 2, x, y, DB_F77NULL, dims, ndims,
    .              DB_FLOAT, DB_NONCOLLINEAR, DB_F77NULL, ierr)
```

Figure 8.4 shows a simple 3D curvilinear mesh that is 1 cell thick in the Z-dimension. The number of cells in a dimension is 1 less than the number of nodes in the same dimension. As you increase the number of nodes in the Z-dimension, you must also add more X and Y coordinate values because the X,Y,Z values for node coordinates must be fully specified for a curvilinear mesh.

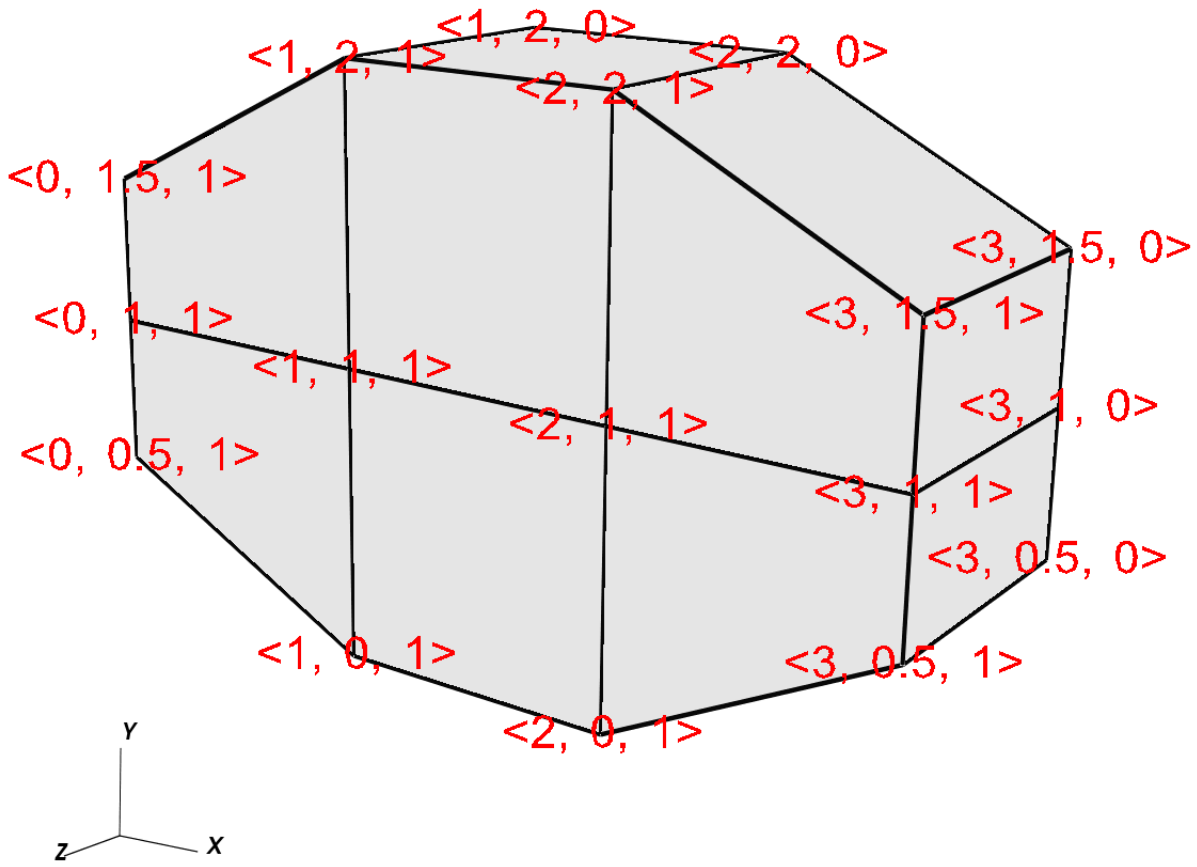


Fig. 8.4: Curvilinear mesh and its X,Y node coordinates

Example for writing a 3D curvilinear mesh:

C

```
/* Write a curvilinear mesh. */
#define NX 4
#define NY 3
#define NZ 2
float x[NZ][NY][NX] = {
    {{0.,1.,2.,3.},{0.,1.,2.,3.}, {0.,1.,2.,3.}},
    {{0.,1.,2.,3.},{0.,1.,2.,3.}, {0.,1.,2.,3.}}
```

(continues on next page)

(continued from previous page)

```
};
float y[NZ][NY][NX] = {
    {{0.5,0.,0.,0.5},{1.,1.,1.,1.}, {1.5,2.,2.,1.5}},
    {{0.5,0.,0.,0.5},{1.,1.,1.,1.}, {1.5,2.,2.,1.5}}
};
float z[NZ][NY][NX] = {
    {{0.,0.,0.,0.},{0.,0.,0.,0.},{0.,0.,0.,0.}},
    {{1.,1.,1.,1.},{1.,1.,1.,1.},{1.,1.,1.,1.}}
};
int dims[] = {NX, NY, NZ};
int ndims = 3;
float *coords[] = {(float*)x, (float*)y, (float*)z};
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
              DB_FLOAT, DB_NONCOLLINEAR, NULL);
```

FortranFixed

```
c Write a curvilinear mesh
    integer err, ierr, dims(3), ndims, NX, NY, NZ
    parameter (NX = 4)
    parameter (NY = 3)
    parameter (NZ = 2)
    real x(NX,NY,NZ), y(NX,NY,NZ), z(NX,NY,NZ)
    data x/0., 1.,2.,3.,  0.,1.,2.,3., 0., 1.,2.,3.,
    .    0.,   1.,2.,3.,  0.,1.,2.,3., 0., 1.,2.,3./
    data y/0.5,0.,0.,0.5, 1.,1.,1.,1., 1.5,2.,2.,1.5,
    .    0.5,  0.,0.,0.5, 1.,1.,1.,1., 1.5,2.,2.,1.5/
    data z/0., 0.,0.,0.,  0.,0.,0.,0., 0., 0.,0.,0,
    .    1.,   1.,1.,1.,  1.,1.,1.,1., 1., 1.,1.,1./
    ndims = 3
    data dims/NX, NY, NZ/
    err = dbputqm (dbfile, "quadmesh", 8, "xc", 2,
    .              "yc", 2, "zc", 2, x, y, z, dims, ndims,
    .              DB_FLOAT, DB_NONCOLLINEAR, DB_F77NULL, ierr)
```

8.6.9 Writing a point mesh

A point mesh is a set of 2D or 3D points where the nodes themselves also constitute the *only* cells in the mesh. Silo provides the `DBPutPointmesh` function so you can write out particle systems represented as point meshes.

Example for writing a 2D point mesh:

C

```
/* Create some points to save. */
#define NPTS 100
int i, ndims = 2;
float x[NPTS], y[NPTS];
float *coords[] = {(float*)x, (float*)y};
for(i = 0; i < NPTS; ++i)
{
    float t = ((float)i) / ((float)(NPTS-1));
    float angle = 3.14159 * 10. * t;
    x[i] = t * cos(angle);
    y[i] = t * sin(angle);
```

(continues on next page)

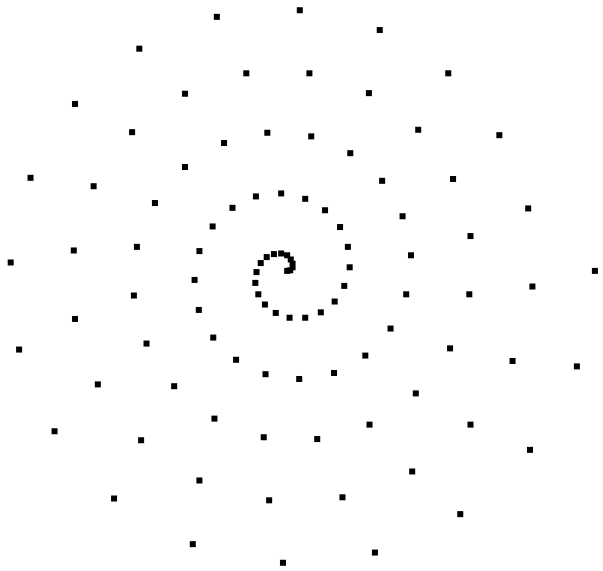


Fig. 8.5: 2D point mesh

(continued from previous page)

```
}
/* Write a point mesh. */
DBPutPointmesh(dbfile, "pointmesh", ndims, coords, NPTS,
               DB_FLOAT, NULL);
```

FortranFixed

```
c Create some points to save.
    integer err, ierr, i, ndims, NPTS
    parameter (NPTS = 100)
    real x(NPTS), y(NPTS), t, angle
    do 10000 i = 0,NPTS-1
        t = float(i) / float(NPTS-1)
        angle = 3.14159 * 10. * t
        x(i+1) = t * cos(angle);
        y(i+1) = t * sin(angle);
10000 continue
    ndims = 2
c Write a point mesh.
    err = dbputpm (dbfile, "pointmesh", 9, ndims, x, y,
    .              DB_F77NULL, NPTS, DB_FLOAT, DB_F77NULL, ierr)
```

Writing a 3D point mesh is very similar to writing a 2D point mesh with the exception that for a 3D point mesh, you must specify a Z-coordinate. Figure 8.6 shows what happens when we extend our 2D point mesh example into 3D.

Example for writing a 3D point mesh:

C

```
/* Create some points to save. */
#define NPTS 100
```

(continues on next page)

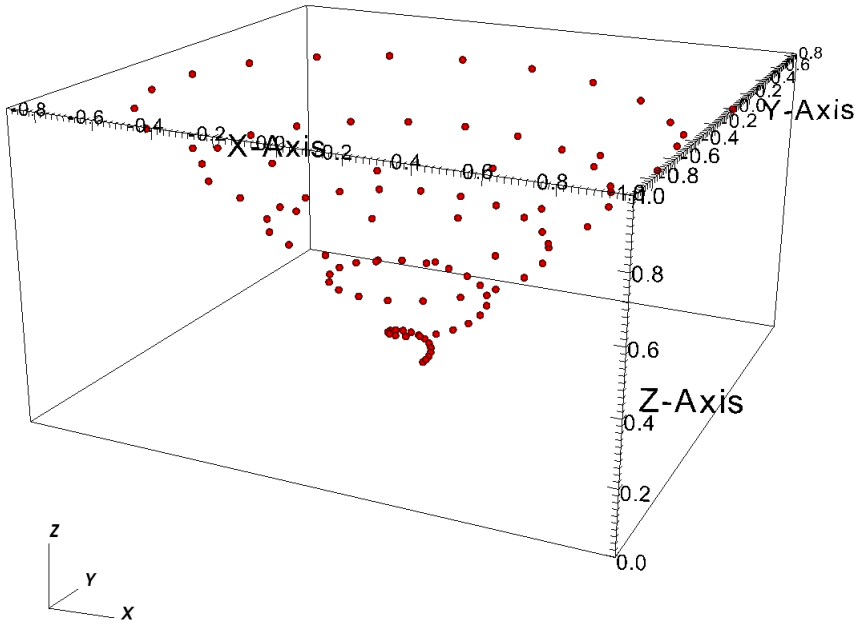


Fig. 8.6: 3D point mesh

(continued from previous page)

```
int i, ndims = 3;
float x[NPTS], y[NPTS], z[NPTS];
float *coords[] = {(float*)x, (float*)y, (float*)z};
for(i = 0; i < NPTS; ++i)
{
    float t = ((float)i) / ((float)(NPTS-1));
    float angle = 3.14159 * 10. * t;
    x[i] = t * cos(angle);
    y[i] = t * sin(angle);
    z[i] = t;
}
/* Write a point mesh. */
DBPutPointmesh(dbfile, "pointmesh", ndims, coords, NPTS,
               DB_FLOAT, NULL);
```

FortranFixed

```
c Create some points to save
    integer err, ierr, i, ndims, NPTS
    parameter (NPTS = 100)
    real x(NPTS), y(NPTS), z(NPTS), t, angle
    do 10000 i = 0,NPTS-1
        t = float(i) / float(NPTS-1)
        angle = 3.14159 * 10. * t
        x(i+1) = t * cos(angle);
        y(i+1) = t * sin(angle);
        z(i+1) = t
10000 continue
    ndims = 3
c Write a point mesh
    err = dbputpm (dbfile, "pointmesh", 9, ndims, x, y, z,
    .              NPTS, DB_FLOAT, DB_F77NULL, ierr)
```

8.6.10 Writing an unstructured mesh

Unstructured meshes are collections of different types of zones and are useful because they can represent more complex mesh geometries than structured meshes can. This section explains the Silo functions that are used to write out an unstructured mesh.

Silo supports the creation of 2D unstructured meshes composed of triangles, quadrilaterals, and polygonal cells. However, VisIt splits polygonal cells into triangles. Unstructured meshes are specified in terms of a set of nodes and then a zone list consisting of lists of nodes, called connectivity information, that make up the zones in the mesh. When creating connectivity information, be sure that the nodes in your zones are specified so that when you iterate over the nodes in the zone that a counter-clockwise pattern is observed. Silo provides the `DBPutZonelist` function to store out the connectivity information. The coordinates for the unstructured mesh itself is written out using the **DBPutUcdmesh** function.

Example for writing a 2D unstructured mesh:

C

```
/* Node coordinates */
float x[] = {0., 2., 5., 3., 5., 0., 2., 4., 5.};
float y[] = {0., 0., 0., 3., 3., 5., 5., 5., 5.};
float *coords[] = {x, y};
```

(continues on next page)

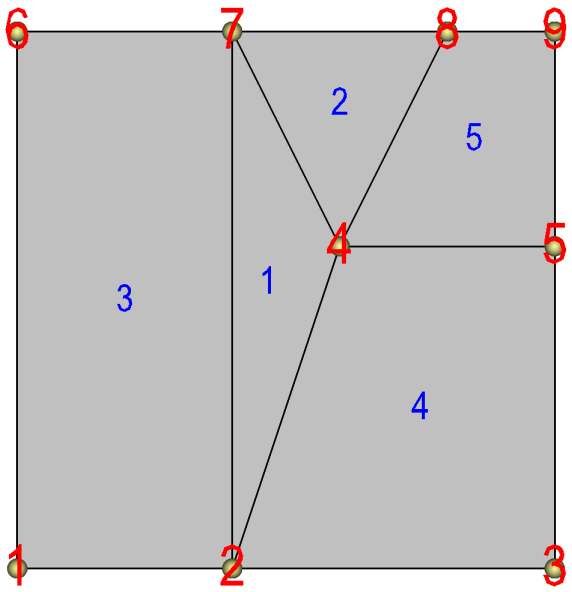


Fig. 8.7: 2D unstructured mesh composed of triangles and quadrilaterals. The node numbers are labelled red and the zone numbers are labeled blue.

(continued from previous page)

```
/* Connectivity */
int nodelist[] = {
    2,4,7, /* tri zone 1 */
    4,8,7, /* tri zone 2 */
    1,2,7,6, /* quad zone 3 */
    2,3,5,4, /* quad zone 4 */
    4,5,9,8 /* quad zone 5 */
};
int lnodelist = sizeof(nodelist) / sizeof(int);
/* shape type 1 has 3 nodes (tri), shape type 2 is quad */
int shapesize[] = {3, 4};
/* We have 2 tris and 3 quads */
int shapecounts[] = {2, 3};
int nshapetypes = 2;
int nnodes = 9;
int nzones = 5;
int ndims = 2;
/* Write out connectivity information. */
DBPutZonelist(dbfile, "zonelist", nzones, ndims, nodelist, lnodelist,
              1, shapesize, shapecounts, nshapetypes);
/* Write an unstructured mesh. */
DBPutUcdmesh(dbfile, "mesh", ndims, NULL, coords, nnodes, nzones,
             "zonelist", NULL, DB_FLOAT, NULL);
```

FortranFixed

```
    integer err, ierr, ndims, nshapetypes, nnodes, nzones
c Node coordinates
    real x(9) /0., 2., 5., 3., 5., 0., 2., 4., 5./
    real y(9) /0., 0., 0., 3., 3., 5., 5., 5., 5./
```

(continues on next page)

(continued from previous page)

```
c Connectivity
      integer LNODELIST
      parameter (LNODELIST = 18)
      integer nodelist(LNODELIST) /2,4,7,
     . 4,8,7,
     . 1,2,7,6,
     . 2,3,5,4,
     . 4,5,9,8/
c Shape type 1 has 3 nodes (tri), shape type 2 is quad
      integer shapesize(2) /3, 4/
c We have 2 tris and 3 quads
      integer shapecounts(2) /2, 3/
      nshapetypes = 2
      nnodes = 9
      nzones = 5
      ndims = 2
c Write out connectivity information.
      err = dbputzl(dbfile, "zonelist", 8, nzones, ndims, nodelist,
     . LNODELIST, 1, shapesize, shapecounts, nshapetypes, ierr)
c Write an unstructured mesh
      err = dbputum(dbfile, "mesh", 4, ndims, x, y, DB_F77NULL,
     . "X", 1, "Y", 1, DB_F77NULL, 0, DB_FLOAT, nnodes, nzones,
     . "zonelist", 8, DB_F77NULL, 0, DB_F77NULL, ierr)
```

3D unstructured meshes are created much the same way as 2D unstructured meshes are created. The main difference is that in 2D, you use triangles and quadrilateral zone types, in 3D, you use hexahedrons, pyramids, prisms, and tetrahedrons to compose your mesh. Silo also supports fully arbitrary polyhedral zones but that will not be covered here. The procedure for creating the node coordinates is the same with the exception that 3D meshes also require a Z-coordinate. The procedure for creating the zone list (connectivity information) is the same except that you specify cells using a larger number of nodes because they are 3D. The order in which the nodes are specified is also more important for 3D shapes because if the nodes are not given in the right order, the zones can become tangled. The proper zone ordering for each of the four supported 3D zone shapes is shown in the Silo_'s user manual.

Figure 8.8 shows an example of a simple 3D unstructured mesh consisting of 2 hexahedrons, 1 pyramid, 1 prism, and 1 tetrahedron.

Example for writing a 3D unstructured mesh:

C

```
/* Node coordinates */
float x[] = {0.,2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,1.,2.,4.,4.};
float y[] = {0.,0.,0.,0.,2.,2.,2.,2.,4.,4.,4.,4.,6.,0.,0.,0.};
float z[] = {2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,0.,1.,4.,2.,0.};
float *coords[] = {x, y, z};
/* Connectivity */
int nodelist[] = {
    1,2,3,4,5,6,7,8,     /* hex, zone 1 */
    5,6,7,8,9,10,11,12,  /* hex, zone 2 */
    9,10,11,12,13,       /* pyramid, zone 3 */
    2,3,16,15,6,7,       /* prism, zone 4 */
    2,15,14,6            /* tet, zone 5 */
};
int lnodelist = sizeof(nodelist) / sizeof(int);
/* shape type 1 has 8 nodes (hex) */
/* shape type 2 has 5 nodes (pyramid) */
/* shape type 3 has 6 nodes (prism) */
```

(continues on next page)

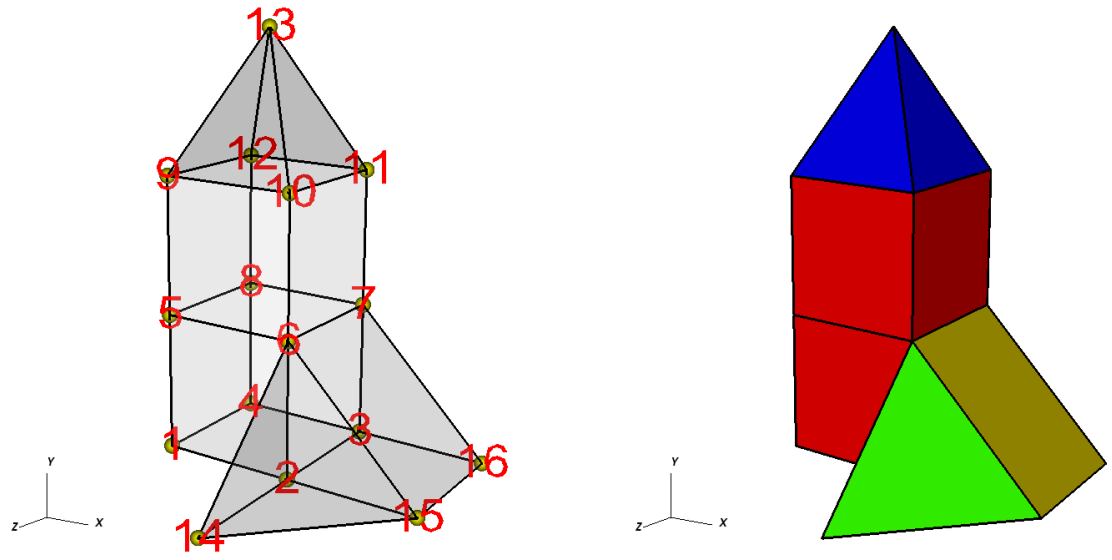


Fig. 8.8: Node numbers on the left and the mesh, colored by zone type, on the right. Hexahedron (red), Pyramid (blue), Prism (yellow), Tetrahedron (green).

(continued from previous page)

```
/* shape type 4 has 4 nodes (tet) */
int shapesize[] = {8,5,6,4};
/* We have 2 hex, 1 pyramid, 1 prism, 1 tet */
int shapecounts[] = {2,1,1,1};
int nshapetypes = 4;
int nnodes = 16;
int nzones = 5;
int ndims = 3;
/* Write out connectivity information. */
DBPutZonelist(dbfile, "zonelist", nzones, ndims, nodelist, lnodelist,
              1, shapesize, shapecounts, nshapetypes);
/* Write an unstructured mesh. */
DBPutUcdmesh(dbfile, "mesh", ndims, NULL, coords, nnodes, nzones,
             "zonelist", NULL, DB_FLOAT, NULL);
```

FortranFixed

```
    integer err, ierr, ndims, nzones
    integer NSHAPETYPES, NNODES
    parameter (NSHAPETYPES = 4)
    parameter (NN = 16)
c Node coordinates
    real x(NN) /0.,2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,1.,2.,4.,4./
    real y(NN) /0.,0.,0.,0.,2.,2.,2.,2.,4.,4.,4.,4.,6.,0.,0.,0./
    real z(NN) /2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,0.,1.,4.,2.,0./
c Connectivity
    integer LNODELIST
    parameter (LNODELIST = 31)
    integer nodelist(LNODELIST) /1,2,3,4,5,6,7,8,
    . 5,6,7,8,9,10,11,12,
    . 9,10,11,12,13,
    . 2,3,16,15,6,7,
```

(continues on next page)

(continued from previous page)

```
     . 2,15,14,6/
c Shape type 1 has 8 nodes (hex)
c Shape type 2 has 5 nodes (pyramid)
c Shape type 3 has 6 nodes (prism)
c Shape type 4 has 4 nodes (tet)
     integer shapesize(NSHAPETYPES) /8, 5, 6, 4/
c We have 2 hex, 1 pyramid, 1 prism, 1 tet
     integer shapecounts(NSHAPETYPES) /2, 1, 1, 1/
     nzones = 5
     ndims = 3
c Write out connectivity information.
     err = dbputzl(dbfile, "zonelist", 8, nzones, ndims, nodelist,
     .              LNODELIST, 1, shapesize, shapecounts, NSHAPETYPES, ierr)
c Write an unstructured mesh
     err = dbputum(dbfile, "mesh", 4, ndims, x, y, z,
     .              "X", 1, "Y", 1, "Z", 1, DB_FLOAT, NN, nzones,
     .              "zonelist", 8, DB_F77NULL, 0, DB_F77NULL, ierr)
```

Adding axis labels and axis units

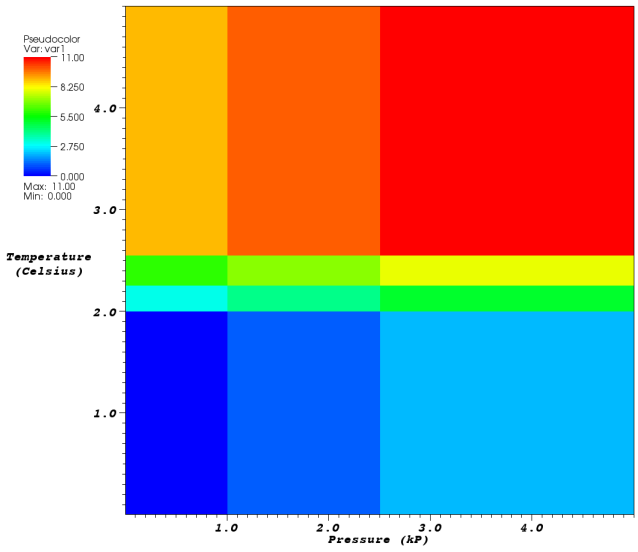


Fig. 8.9: Custom mesh labels and units along the X and Y Axes

It is possible to add additional annotations to your meshes that you store to Silo files using Silo's option list mechanism. This subsection covers how to change the axis titles and units that will be used when VisIt plots your mesh. By default, VisIt uses "X-Axis", "Y-Axis", and "ZAxis" when labelling the coordinate axes. You can override the default labels using an option list. Option lists are created with the `DBMakeOptlist` function and freed with the `DBFreeOptlist` function. All of the Silo functions for writing meshes that we've demonstrated so far can accept option lists that contain custom axis labels and units. Refer to the Silo Manual for more information on additional options that can be passed via option lists.

Adding customized labels and units for a mesh by using option lists ensures that VisIt uses your customized labels and units instead of the default values. Figure 8.9 shows how the labels and units in the previous examples show up in

VisIt's visualization window.

Example for associating new axis labels and units with a mesh:

C

```
/* Create an option list to contain labels and units. */
DBoptlist *optlist = DBMakeOptlist(4);
DBAddOption(optlist, DBOPT_XLABEL, (void *)"Pressure");
DBAddOption(optlist, DBOPT_XUNITS, (void *)"kP");
DBAddOption(optlist, DBOPT_YLABEL, (void *)"Temperature");
DBAddOption(optlist, DBOPT_YUNITS, (void *)"Degrees Celsius");
/* Write a quadmesh with an option list. */
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
              DB_FLOAT, DB_COLLINEAR, optlist);
/* Free the option list. */
DBFreeOptlist(optlist);
```

FortranFixed

```
c Create an option list to contain labels and units.
    integer err, ierr, optlistid
    err = dbmkoptlist(4, optlistid)
    err = dbaddcopt(optlistid, DBOPT_XLABEL, "Pressure", 8)
    err = dbaddcopt(optlistid, DBOPT_XUNITS, "kP", 2)
    err = dbaddcopt(optlistid, DBOPT_YLABEL, "Temperature", 11)
    err = dbaddcopt(optlistid, DBOPT_YUNITS, "Celsius", 7)
c Write a quadmesh with an option list.
    err = dbputqm (dbfile, "quadmesh", 8, "xc", 2,
    .              "yc", 2, "zc", 2, x, y, DB_F77NULL, dims, ndims,
    .              DB_FLOAT, DB_COLLINEAR, optlistid, ierr)
c Free the option list
    err = dbfreeoptlist(optlistid)
```

Another intersting feature of Silo related to structured and unstructured meshes is its ability to apply various compression algorithms including FPZIP, HZIP and ZFP to the mesh as well as its variables. See the documentation on `DBSetCompression()` in the Silo user's manual for more information.

8.6.11 Writing a scalar variable

Silo provides several different functions for writing variables; one for each basic type of mesh: quadmesh (rectilinear and curvilinear), unstructured mesh, and point mesh. Each of these functions can be used to write either zone-centered or node-centered data. This section concentrates on how to write scalar variables; vector and tensor variable components can be written as scalar variables and reassembled into vectors and tensors using expressions, covered in *Writing expressions*. This section's code examples use the rectilinear, curvilinear, point, and unstructured meshes that have appeared in previous code examples.

Zone centering vs. Node centering

VisIt supports two types of variable centering: zone-centering and node-centering. A variable's centering indicates how its values are attached to the mesh on which the variable is defined. When a variable is zone-centered, each zone is assigned a single value. If you were to plot a zone-centered value in VisIt, each zone would be drawn using a uniform color and picking anywhere in the zone would yield the same value. Arrays containing values that are to be zone-centered on a mesh must contain the same number of elements as there are zones in the mesh. Node-centered arrays, on the other hand, contain a value for every node in the mesh. When you plot a node-centered value in VisIt,

VisIt interpolates the values from the nodes across the zone's surface, usually producing a smooth gradient of values across the zone.

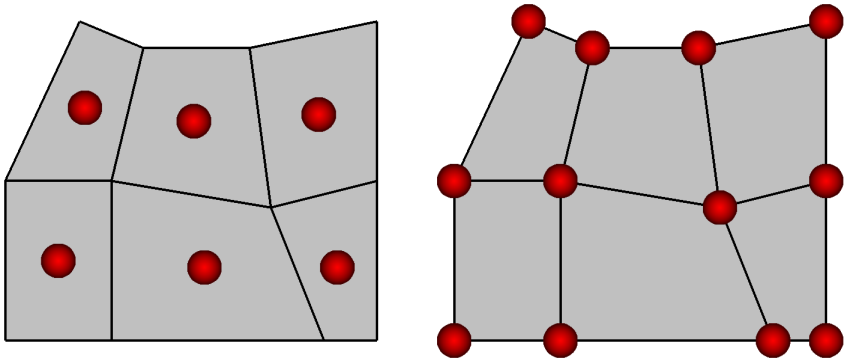


Fig. 8.10: Zone centering (left) and Node-centering (right)

API Commonality

Each of the provided functions for writing scalar variables does have certain arguments in common. For example, all of the functions must be provided the name of the variable to write out. The name that you pick is the name that will appear in VisIt's plot menus. Be careful when you pick your variable names because you should avoid characters that include punctuation marks and spaces. Variable names should contain only letters, numbers and underscores and they should begin with a letter. These guidelines are in place to assure that your data files will have the utmost compatibility with VisIt's *Expression* language.

All variables must be defined on a mesh. If you examine the code examples in this section, each Silo function that writes out a variable will be passed the name of the mesh on which the variable is to be defined.

Each of the Silo function calls will accept a pointer to the array that contains the variable's data. The data can be stored in several internal formats: `char`, `short`, `int`, `long`, `float`, and `double`. Since Silo's variable writing functions use a pointer to pass the data, you can pass a pointer that points to data in any of the mentioned types. In addition, you must pass a flag that indicates to Silo the type of data stored in the array whose address you've passed.

Most of the remaining arguments to Silo's variable writing functions are specific to the types of meshes on which the variable is defined so the rest of this section will provide examples for writing out variables that are defined on various mesh types.

Rectilinear and curvilinear meshes

Recall from sections *Writing a rectilinear mesh* and *Writing a curvilinear mesh* that the procedure for creating rectilinear and curvilinear meshes was similar and the chief difference between the two mesh types was in how their coordinates were specified. While a rectilinear mesh's coordinates could be specified quite compactly as separate X,Y,Z arrays made up of unique values along a coordinate axis, the curvilinear mesh required X,Y,Z coordinate arrays that contained the X,Y,Z values for every node in the mesh. Regardless of how the coordinates were specified, both mesh types contain (NX-1)*(NY-1)*(NZ-1) zones and NX*NY*NZ nodes. This means that the code to write a variable on a rectilinear mesh will be identical to the code to write a zone-centered variable on a curvilinear mesh! Silo provides the `DBPutQuadvar1` function to write scalar variables for both rectilinear and curvilinear meshes.

Example for writing zone-centered variables:

C

```
/* The data must be (NX-1) * (NY-1) since it is zonal. */
float var1[] = {
    0., 1., 2.,
    3., 4., 5.,
    6., 7., 8.,
    9., 10., 11.
};
double var2[] = {
    0.00, 1.11, 2.22,
    3.33, 4.44, 5.55,
    6.66, 7.77, 8.88,
    9.99, 10.1, 11.11
};
int var3[] = {
    0, 1, 2,
    3, 4, 5,
    6, 7, 8,
    9, 10, 11
};
char var4[] = {
    0, 1, 2,
    3, 4, 5,
    6, 7, 8,
    9, 10, 11
};
/* Note dims are 1 less than mesh's dims in each dimension. */
int dims[]={3, 4};
int ndims = 2;
DBPutQuadvar1(dbfile, "var1", "quadmesh", var1, dims,
              ndims, NULL, 0, DB_FLOAT, DB_ZONECENT, NULL);
/* Write a double-precision variable. */
DBPutQuadvar1(dbfile, "var2", "quadmesh", (float*)var2, dims,
              ndims, NULL, 0, DB_DOUBLE, DB_ZONECENT, NULL);
/* Write an integer variable */
DBPutQuadvar1(dbfile, "var3", "quadmesh", (float*)var3, dims,
              ndims, NULL, 0, DB_INT, DB_ZONECENT, NULL);
/* Write a char variable */
DBPutQuadvar1(dbfile, "var4", "quadmesh", (float*)var4, dims,
              ndims, NULL, 0, DB_CHAR, DB_ZONECENT, NULL);
```

FortranFixed

```
    integer err, ierr, dims(2), ndims, NX, NY, ZX, ZY
    parameter (NX = 4)
    parameter (NY = 5)
    parameter (ZX = NX-1)
    parameter (ZY = NY-1)
    real var1(ZX,ZY)
    double precision var2(ZX,ZY)
    integer var3(ZX,ZY)
    character var4(ZX,ZY)
    data var1/0., 1., 2.,
    .         3., 4., 5.,
    .         6., 7., 8.,
    .         9., 10., 11./
    data var2/0.,1.11,2.22,
    .         3.33, 4.44, 5.55,
```

(continues on next page)

(continued from previous page)

```
    .          6.66, 7.77, 8.88,
    .          9.99, 10.1, 11.11/
    data var3/0,1,2,
    .          3, 4, 5,
    .          6, 7, 8,
    .          9, 10, 11/
    data var4/0,1,2,
    .          3, 4, 5,
    .          6, 7, 8,
    .          9, 10, 11/
    data dims/ZX, ZY/
    ndims = 2
    err = dbputqv1(dbfile, "var1", 4, "quadmesh", 8, var1, dims,
    .              ndims, DB_F77NULL, 0, DB_FLOAT, DB_ZONECENT, DB_F77NULL, ierr)
c Write a double-precision variable
    err = dbputqv1(dbfile, "var2", 4, "quadmesh", 8, var2, dims,
    .              ndims, DB_F77NULL, 0, DB_DOUBLE, DB_ZONECENT,
    .              DB_F77NULL, ierr)
c Write an integer variable
    err = dbputqv1(dbfile, "var3", 4, "quadmesh", 8, var3, dims,
    .              ndims, DB_F77NULL, 0, DB_INT, DB_ZONECENT, DB_F77NULL, ierr)
c Write a char variable
    err = dbputqv1(dbfile, "var4", 4, "quadmesh", 8, var4, dims,
    .              ndims, DB_F77NULL, 0, DB_CHAR, DB_ZONECENT, DB_F77NULL, ierr)
```

Both of the previous code examples produce a data file with 4 different scalar arrays. Note that in both of the previous code examples, the same `DBPutQuadvar1` function (or `dbputqv1` in Fortran) function was used to write out data arrays of differing types.

The `DBPutQuadvar1` function can also be used to write out node centered variables. There are two differences that you must observe when writing a node-centered variable as opposed to writing a zone-centered variable. First, the data array that you pass to the `DBPutQuadvar1` function must be larger by 1 in each of its dimensions and you must pass `DB_NODECENT` instead of `DB_ZONECENT`.

Example for writing node-centered variables:

C

```
/* The data must be NX * NY since it is nodal. */
#define NX 4
#define NY 5
float nodal[] = {
    0., 1., 2., 3.,
    4., 5., 6., 7.,
    8., 9., 10., 11.,
    12., 13., 14., 15.,
    16., 17., 18., 19.
};
/* Nodal variables have same #values as #nodes in mesh */
int dims[]={NX, NY};
int ndims = 2;
DBPutQuadvar1(dbfile, "nodal", "quadmesh", nodal, dims,
              ndims, NULL, 0, DB_FLOAT, DB_NODECENT, NULL);
```

FortranFixed

```
c The data must be NX * NY since it is nodal.
      integer err, ierr, dims(2), ndims, NX, NY
      parameter (NX = 4)
      parameter (NY = 5)
      real nodal(NX, NY)
      data dims/NX, NY/
      data nodal/0., 1., 2., 3.,
     .           4., 5., 6., 7.,
     .           8., 9., 10., 11.,
     .           12., 13., 14., 15.,
     .           16., 17., 18., 19./
c Nodal variables have same #values as #nodes in mesh
      err = dbputqv1(dbfile, "nodal", 5, "quadmesh", 8, nodal,
     .               dims, ndims, DB_F77NULL, 0, DB_FLOAT, DB_NODECENT,
     .               DB_F77NULL, ierr)
```

Writing variables to 3D curvilinear and rectilinear meshes follows the same basic rules as writing variables for 2D meshes. For zone-centered variables, you must have (NX- 1)*(NY-1)*(NZ-1) data values and for node-centered variables, you must have NX*NY*NZ data values. Figure 8.11 shows what the data values look like for the Silo files produced by the examples to come.

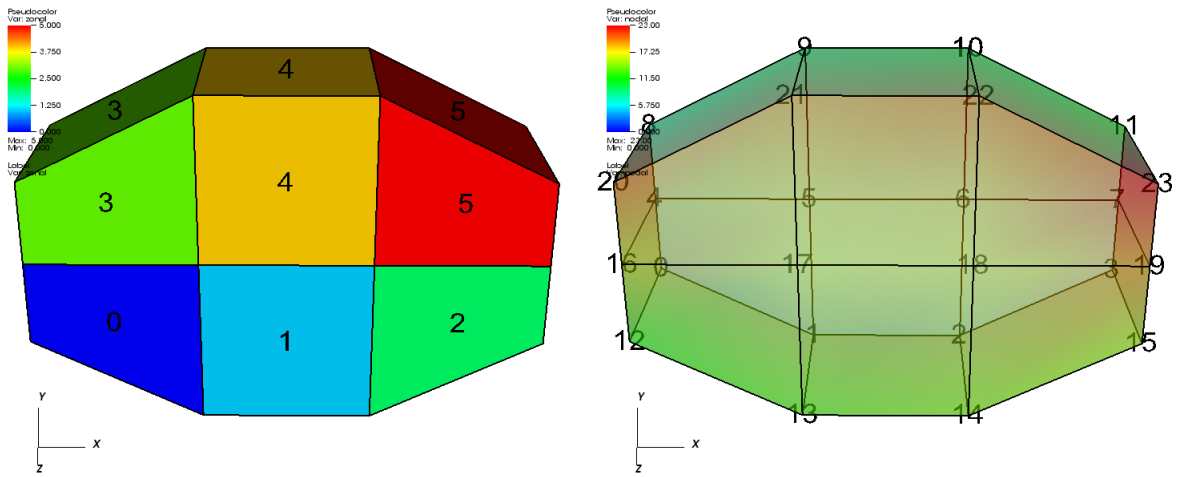


Fig. 8.11: Zone centered variable in 3D and a node-centered varaible in 3D (shown with a partially transparent plot)

Example for writing variables on a 3D mesh:

C

```
#define NX 4
#define NY 3
#define NZ 2

/* Write a zone-centered variable. */
void write_zonecent_quadvar(DBfile *dbfile)
{
    int i, dims[3], ndims = 3;
    int ncells = (NX-1)*(NY-1)*(NZ-1);
    float *data = (float *)malloc(sizeof(float)*ncells);
```

(continues on next page)

(continued from previous page)

```
    for(i = 0; i < ncells; ++i)
        data[i] = (float)i;
    dims[0] = NX-1; dims[1] = NY-1; dims[2] = NZ-1;
    DBPutQuadvar1(dbfile, "zonal", "quadmesh", data, dims,
                  ndims, NULL, 0, DB_FLOAT, DB_ZONECENT, NULL);
    free(data);
}

/* Write a node-centered variable. */
void write_nodecent_quadvar(DBfile *dbfile)
{
    int i, dims[3], ndims = 3;
    int nnodes = NX*NY*NZ;
    float *data = (float *)malloc(sizeof(float)*nnodes);
    for(i = 0; i < nnodes; ++i)
        data[i] = (float)i;
    dims[0] = NX; dims[1] = NY; dims[2] = NZ;
    DBPutQuadvar1(dbfile, "nodal", "quadmesh", data, dims,
                  ndims, NULL, 0, DB_FLOAT, DB_NODECENT, NULL);
    free(data);
}
```

FortranFixed

```
c Write a zone-centered variable.
    subroutine write_zonecent_quadvar(dbfile)
    implicit none
    integer dbfile
    include "silo.inc"
    integer err, ierr, dims(3), ndims, i,j,k,index, ZX,ZY,ZZ
    parameter (ZX = 3)
    parameter (ZY = 2)
    parameter (ZZ = 1)
    integer zonal(ZX, ZY, ZZ)
    data dims/ZX, ZY, ZZ/
    index = 0
    do 10020 k=1,ZZ
    do 10010 j=1,ZY
    do 10000 i=1,ZX
        zonal(i,j,k) = index
        index = index + 1
10000 continue
10010 continue
10020 continue
    ndims = 3
    err = dbputqv1(dbfile, "zonal", 5, "quadmesh", 8, zonal, dims,
    .              ndims, DB_F77NULL, 0, DB_INT, DB_ZONECENT, DB_F77NULL, ierr)
    end
c Write a node-centered variable.
    subroutine write_nodecent_quadvar(dbfile)
    implicit none
    integer dbfile
    include "silo.inc"
    integer err, ierr, dims(3), ndims, i,j,k,index, NZ, NY, NZ
    parameter (NX = 4)
    parameter (NY = 3)
    parameter (NZ = 2)
```

(continues on next page)

(continued from previous page)

```
    real nodal(NX, NY, NZ)
    data dims/NX, NY, NZ/
    index = 0
    do 20020 k=1,NZ
    do 20010 j=1,NY
    do 20000 i=1,NX
        nodal(i,j,k) = float(index)
        index = index + 1
20000 continue
20010 continue
20020 continue
    ndims = 3
    err = dbputqv1(dbfile, "nodal", 5, "quadmesh", 8, nodal, dims,
    .               ndims, DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL, ierr)
    end
```

Point meshes

Point meshes, which are meshes composed of a set of points can, like other mesh types, have values associated with each point. Silo provides the `DBPutPointVar1` function that you can use to write out a scalar variable stored on a point mesh. Nodes and the zones are really the same thing in a point mesh so you can consider zone-centered scalars to be the same thing as node-centered scalars.

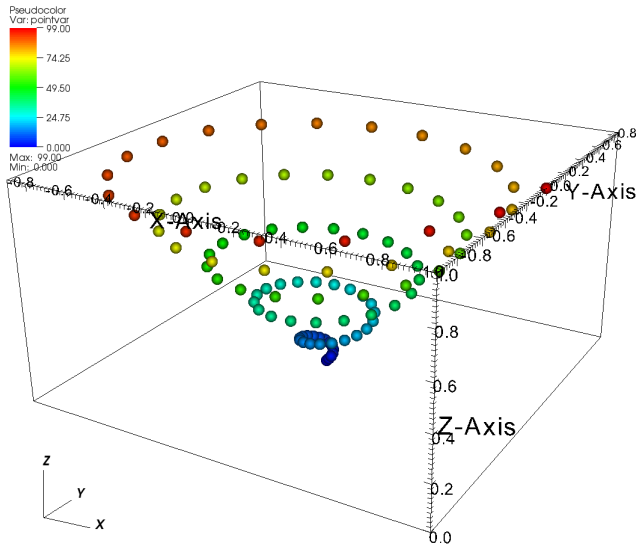


Fig. 8.12: Scalar variable defined on a point mesh

Example for writing variables on a 3D point mesh:

C

```
/* Create some values to save. */
int i;
float var[NPTS];
```

(continues on next page)

(continued from previous page)

```
for(i = 0; i < NPTS; ++i)
    var[i] = (float)i;
/* Write the point variable. */
DBPutPointvar1(dbfile, "pointvar", "pointmesh", var, NPTS,
               DB_FLOAT, NULL);
```

FortranFixed

```
c Create some values to save.
    integer err, ierr, i, NPTS
    parameter (NPTS = 100)
    real var(NPTS)
    do 10010 i = 1,NPTS
        var(i) = float(i-1)
10010 continue
c Write the point variable
    err = dbputpv1(dbfile, "pointvar", 8, "pointmesh", 9,
    .               var, NPTS, DB_FLOAT, DB_F77NULL, ierr)
```

Unstructured meshes

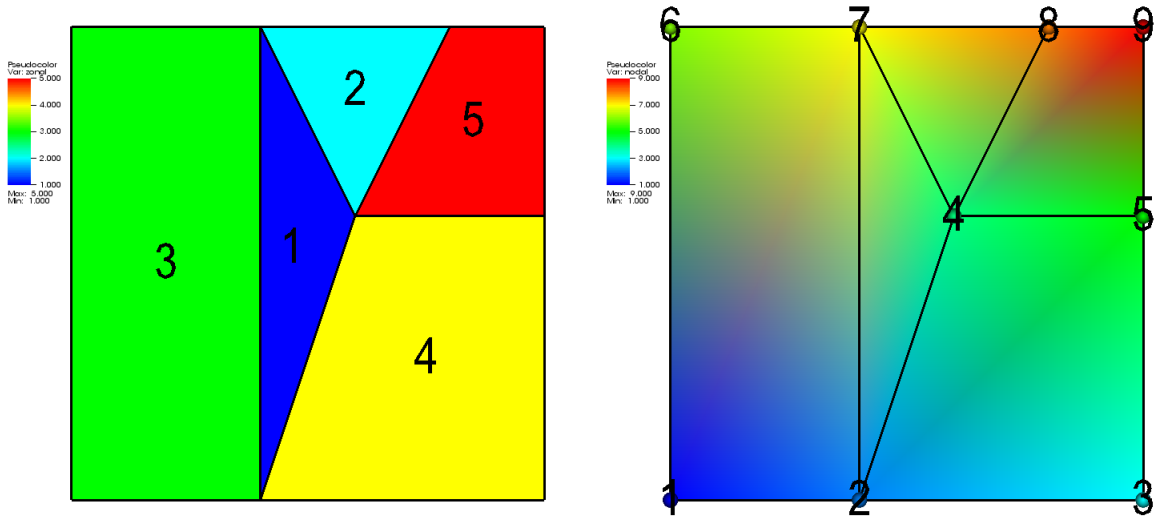


Fig. 8.13: A 2D unstructured mesh with a zonal variable (left) and a nodal varaible (right)

Writing a variable on an unstructured mesh is done following a procedure similar to that for writing a variable on a point mesh. As with other mesh types, a scalar variable defined on an unstructured grid can be zone-centered or node-centered. If the variable is zone-centered then the data array required to store the variable on the unstructured mesh must be a 1-D array with the same number of elements as the mesh has zones. If the variable to be stored is node-centered then the array containing the variable must be a 1-D array with the same number of elements as the mesh has nodes. Thinking of the data array as a 1-D array simplifies indexing since the number used to identify a particular node is the same index that would be used to access data in the variable array (assuming 0-origin in C and 1-origin in Fortran). Since the data array is always 1-D for an unstructured mesh, the code to store variables on 2D and 3D unstructured meshes is identical. Figure 8.13 shows a 2D unstructured mesh with both zonal and nodal variables. Silo provides the `DBPutUcdvar1` function for writing scalar variables on unstructured meshes.

Example for writing variables on an unstructured mesh:

C

```
float nodal[] = {1.,2.,3.,4.,5.,6.,7.,8.,9.};
float zonal[] = {1.,2.,3.,4.,5.};
int nnodes = 9;
int nzones = 5;
/* Write a zone-centered variable. */
DBPutUcdvar1(dbfile, "zonal", "mesh", zonal, nzones, NULL, 0,
             DB_FLOAT, DB_ZONECENT, NULL);
/* Write a node-centered variable. */
DBPutUcdvar1(dbfile, "nodal", "mesh", nodal, nnodes, NULL, 0,
             DB_FLOAT, DB_NODECENT, NULL);
```

FortranFixed

```
    integer err, ierr, NNODES, NZONES
    parameter (NNODES = 9)
    parameter (NZONES = 5)
    real nodal(NNODES) /1.,2.,3.,4.,5.,6.,7.,8.,9./
    real zonal(NZONES) /1.,2.,3.,4.,5./
c Write a zone-centered variable.
    err = dbputuv1(dbfile, "zonal", 5, "mesh", 4, zonal, NZONES,
    .              DB_F77NULL, 0, DB_FLOAT, DB_ZONECENT, DB_F77NULL, ierr)
c Write a node-centered variable.
    err = dbputuv1(dbfile, "nodal", 5, "mesh", 4, nodal, NNODES,
    .              DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL, ierr)
```

Adding variable units

All of the examples for writing scalar variables presented so far have focused on the basics of writing a variable array to a Silo file. Silo's option list mechanism allows a variable object to be annotated with various extra information. In the case of scalar variables, the option list passed to `DBPutQuadvar1` and `DBPutUcdvar1` can contain the units that describe the variable being stored. Refer to the Silo Manual for a complete list of the options accepted by the `DBPutQuadvar1` and `DBPutUcdvar1` functions. When a scalar variable has associated units, the units appear in the variable legend in VisIt's visualization window (see Figure 8.14).

Fig. 8.14: Plot legend with units.

If you want to add units to the variable that you write, you must create an option list to pass to the function writing your variable. You may recall that option lists are created with the `DBMakeOptlist` function and freed with the `DBFreeOptlist` function. In order to add units to the option list, you must add the `DBOPT_UNITS` option.

Example for writing a variable with units:

C

```
/* Create an option list and add "g/cc" units to it. */
DBoptlist *optlist = DBMakeOptlist(1);
DBAddOption(optlist, DBOPT_UNITS, (void*)"g/cc");
/* Write a variable that has units. */
DBPutUcdvar1(dbfile, "zonal", "mesh", zonal, nzones, NULL, 0,
             DB_FLOAT, DB_ZONECENT, optlist);
/* Free the option list. */
DBFreeOptlist(optlist);
```

FortranFixed

```
c Create an option list and add "g/cc" units to it.
    integer err, optlistid
    err = dbmkoptlist(1, optlistid)
    err = dbaddcopt(optlistid, DBOPT_UNITS, "g/cc", 4)
c Write a variable that has units.
    err = dbputuv1(dbfile, "zonal", 5, "mesh", 4, zonal, NZONES,
    .              DB_F77NULL, 0, DB_FLOAT, DB_ZONECENT, optlistid, ierr)
c Free the option list.
    err = dbfreeoptlist(optlistid)
```

8.6.12 Single precision vs. Double precision

After having written some variables to a Silo file, you've no doubt learned that you can pass a pointer to data of many different representations and precisions (char, int, float, double, etc.). When you pass data to a Silo function, you also must pass a flag that tells Silo how to interpret the data stored in your data array. For example, if you have single precision floating point data then you would tell Silo to traverse the data as such using the `DB_FLOAT` type flag in the function call to `DBPutQuadvar1`. Many of the functions in the Silo library require a type flag to indicate the type of data being passed to Silo. In fact, even the functions to write mesh coordinates can accept different data types. This means that you can use double-precision to specify your mesh coordinates, which can be immensely useful when dealing with very large or very small objects.

Listing 8.2: C-Language example for writing a mesh with double-precision coordinates

```
/* The x,y arrays contain double-precision coordinates. */
double x[NY][NX], y[NY][NX];
int dims[] = {NX, NY};
int ndims = 2;
/* Note that x,y pointers are cast to float to conform to API. */
float *coords[] = {(float*)x, (float*)y};
/* Tell Silo that the coordinate arrays are actually doubles. */
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
    DB_DOUBLE, DB_NONCOLLINEAR, NULL);
```

Writing expressions

You can plot derived quantities in VisIt by creating expressions that involve variables from your database. Sometimes, it is useful to include expression definitions in your Silo file so they are available to VisIt without users having to first create them manually. Silo provides the *DBPutdefvars'* function so you can write your expressions to a Silo file. Expression names should be valid VisIt expression names, as defined in the *Built-in expressions* section. Likewise, the expression definitions should contain only expressions that are supported by the VisIt expression language.

While VisIt's expression language can be useful for calculating a multitude of expressions, it can be particularly useful for grouping vector or tensor components into vector and tensor variables. If you store vector or tensor components as scalar variables in your Silo file then you can easily create expressions that assemble the components into real vector or tensor variables without significantly increasing your file's storage requirements. Writing out vector and tensor variables as expressions involving scalar variables also prevents you from having to use more complicated Silo functions in order to write out the vector or tensor data.

Example for writing out expression definitions:

C

```
/* Write some expressions to the Silo file. */
const char *names[] = {"velocity", "speed"};
const char *defs[] = {"{xc,yc,zc}", "magnitude(velocity)"};
int types[] = {DB_VARTYPE_VECTOR, DB_VARTYPE_SCALAR};
DBPutDefvars(dbfile, "defvars", 2, names, types, defs, NULL);
```

FortranFixed

```
    integer err, ierr, types(2), lnames(2), ldefs(2)
    integer numexpressions, oldlen
c Initialize some 20 character length strings
    character*20 names(2) /'velocity ',
    .                      'speed '/
    character*20 defs(2) /'{xc,yc,zc} ',
    .                     'magnitude(velocity) '/
c Store the length of each string
    data lnames/8, 5/
    data ldefs/10, 19/
    data types/DB_VARTYPE_VECTOR, DB_VARTYPE_SCALAR/
c Set the maximum string length to 20 since that's how long
c our strings are
    oldlen = dbget2dstrlen()
    err = dbset2dstrlen(20)
c Write out the expressions
    numexpressions = 2
    err = dbputdefvars(dbfile, "defvars", 7, numexpressions,
    .                  names, lnames, types, defs, ldefs, DB_F77NULL, ierr)
c Restore the previous value for maximum string length
    err = dbset2dstrlen(oldlen)
```

In the previous Fortran example for writing expressions, there are more functions involved than just the `dbputdefvars` function. It is critical to set the maximum 2D string length for strings in the Silo library, using the `dbset2dstrlen` function, so the Fortran interface to Silo will be able to correctly traverse the string data passed to it from Fortran. In the previous example, we used 20 characters for both the expression names and definitions. We call `dbset2dstrlen` to set the maximum allowable 2d string length to 20 characters before we pass our arrays of 20 character strings to the `dbputdefvars` function. In addition, we must also pass valid lengths for the expression name and definition strings. The lengths should be at least 1 character long but no longer than the maximum allowable string length, which we set to 20 characters in the example program. Passing valid string lengths is important so the expressions that you save to your file do not contain any extra characters, such as trailing spaces.

8.6.13 Creating a master file for parallel

When a parallel program saves out its data files, often the most efficient method of I/O is for each processor to write its own piece of the simulation, or domain, to its own Silo file. If each processor writes its own Silo file then no communication or synchronization must take place to manage access to a shared file. However, once the simulation has completed, there are many files and all of them are required to reconstitute the simulated object. Expecting a user to plot each domain file manually in VisIt would be very tedious. So, Silo provides functions to create what is known as a *master* file (or *root* file), which is a top-level file that defines cooresponding objects which list all their constituent Silo objects in the the domain files. When you open a master file in VisIt and plot variables out of it, all domains are plotted.

Master files contain what are known as multimeshes, multivars, and multimaterials. These objects are lists of filenames that contain the appropriate domain variable. They also contain some meta-information about each of the domains that helps VisIt perform better in parallel. Strategies for using metadata to improve VisIt's I/O performance will be covered shortly.

Creating a multimesh

A multimesh is an object that unites smaller domain-sized meshes into a whole mesh. The multimesh object contains a list of the filenames that contain a piece of the named mesh. When you tell VisIt to plot a multimesh, VisIt reads the named mesh in all of the required domain files and processes the mesh in each file, to produce the entire mesh.

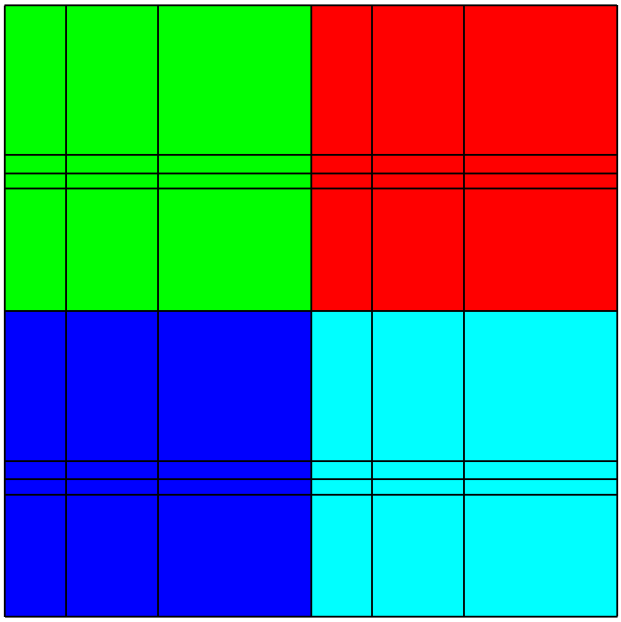


Fig. 8.15: Multimesh colored by its domain number

The example shown in Figure 8.15, uses the mesh from the 2D rectilinear mesh example program and repeats it as 4 domains. Note that the mesh forming the domains is translated in X and Y so that the edges are shared. In the given example, the meshes that make up the entire mesh are stored in separate Silo files: *multimesh.1*, *multimesh.2*, *multimesh.3*, and *multimesh.4*. The mesh and any data that may be defined on it is stored in those files. Remember that storing pieces of a single mesh is commonplace when parallel processes write their own file. Plotting each of the smaller files individually in VisIt is not neccessary when a master file has been generated since plotting the multimesh object from the master file will cause VisIt to plot each of its constituent meshes. The code that will follow shows how to use Silo's `DBPutMultimesh` function to write out a multimesh object that reassembles meshes from many domain files into a whole mesh.

The list of meshes or items in a multi-object generally take the form: `<path>:<item>` where `<path>` is the file system path to the Silo file containing the object and `<item>` is the path of the object in the Silo file. The colon character, **:** disambiguates these two parts of the object's name. Note that the path may be specified as a relative or absolute path using names valid for the file system containing the master file. However, we strongly recommend using only relative paths so the master file does not reference directories that exist only on one file system. Using relative paths makes the master files much more portable since they allow the data files to be moved. The path may also refer to subdirectories within the file being referenced since Silo files may contain directories that help to organize related data. The following examples assume that the domain files will exist in the same directory as the master file since the path includes only the names of the domain files.

Example for writing a multimesh:

C

```
void write_masterfile(void)
{
```

(continues on next page)

(continued from previous page)

```
    DBfile *dbfile = NULL;
    char **meshnames = NULL;
    int dom, nmesh = 4, *meshtypes = NULL;
    /* Create the list of mesh names. */
    meshnames = (char **)malloc(nmesh * sizeof(char *));
    for(dom = 0; dom < nmesh; ++dom)
    {
        char tmp[100];
        sprintf(tmp, "multimesh.%d:quadmesh", dom);
        meshnames[dom] = strdup(tmp);
    }
    /* Create the list of mesh types. */
    meshtypes = (int *)malloc(nmesh * sizeof(int));
    for(dom = 0; dom < nmesh; ++dom)
        meshtypes[dom] = DB_QUAD_RECT;
    /* Open the Silo file */
    dbfile = DBCreate("multimesh.root", DB_CLOBBER, DB_LOCAL,
                      "Master file", DB_HDF5);
    /* Write the multimesh. */
    DBPutMultimesh(dbfile, "quadmesh", nmesh, meshnames,
                   meshtypes, NULL);
    /* Close the Silo file. */
    DBClose(dbfile);
    /* Free the memory*/
    for(dom = 0; dom < nmesh; ++dom)
        free(meshnames[dom]);
    free(meshnames);
}
```

FortranFixed

```
    subroutine write_master()
    implicit none
    include "silo.inc"
    integer err, ierr, dbfile, nmesh, oldlen
    character*20 meshnames(4) /'multimesh.1:quadmesh',
    .                          'multimesh.2:quadmesh',
    .                          'multimesh.3:quadmesh',
    .                          'multimesh.4:quadmesh'/
    integer lmeshnames(4) /20,20,20,20/
    integer meshtypes(4) /DB_QUAD_RECT, DB_QUAD_RECT,
    .                     DB_QUAD_RECT, DB_QUAD_RECT/
c Create a new silo file
    err = dbcreate("multimesh.root", 14, DB_CLOBBER, DB_LOCAL,
    .              "multimesh root", 14, DB_HDF5, dbfile)
    if(dbfile.eq.-1) then
        write (6,*) 'Could not create Silo file!\n'
        return
    endif
c Set the maximum string length to 20 since that's how long our
c strings are
    oldlen = dbget2dstrlen()
    err = dbset2dstrlen(20)
c Write the multimesh object.
    nmesh = 4
    err = dbputmmesh(dbfile, "quadmesh", 8, nmesh, meshnames,
    .                lmeshnames, meshtypes, DB_F77NULL, ierr)
```

(continues on next page)

(continued from previous page)

```
c Restore the previous value for maximum string length
    err = dbset2dstrlen(oldlen)
c Close the Silo file
    err = dbclose(dbfile)
    end
```

Sometimes it can be advantageous to have each processor write its files to a unique subdirectory (e.g. proc-0, proc-1, proc-2, ...). You can also choose for each processor to write its files to a common directory so all files for a given time step are contained in a single place (e.g. cycle0000, cycle0001, cycle0002, ...). Generally, you will want to tailor your strategy to the strengths of your file system to spread the demands of writing files across as many I/O nodes as possible in order to increase throughput. The organization strategies mentioned so far are only suggestions and you will have to determine the optimum method for storing domain files on your computer system. Moving your domain files to subdirectories can make it easier to navigate your file system and can provide benefits later such as VisIt not having to check permissions, etc on so many files. Code to create the list of mesh names where each processor writes its data to a different subdirectory that contains all files for a given time step might look like the following:

```
int cycle = 100;
for(dom = 0; dom < nmesh; ++dom)
{
    char tmp[100];
    sprintf(tmp, "proc-%d/multimesh.%04d:quadmesh", dom, cycle);
    meshnames[dom] = strdup(tmp);
}
```

Creating a multivar

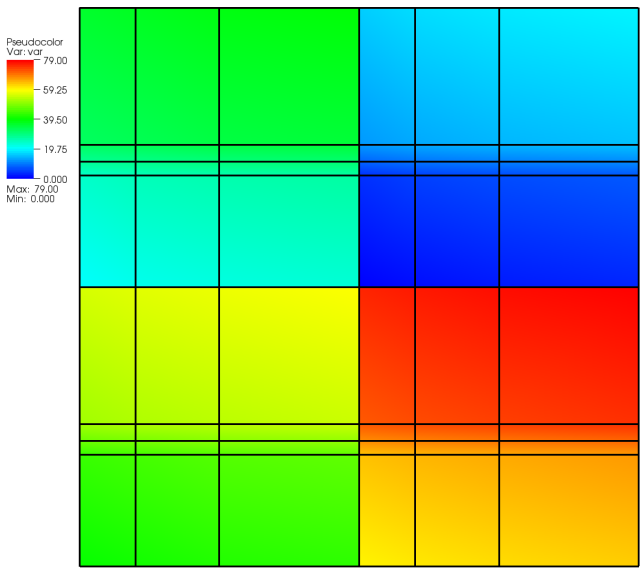


Fig. 8.16: Multivar displayed on its multimesh

A multivar object is the variable equivalent of a multimesh object. Like the multimesh object, a multivar object contains a list of filenames that make up the variable represented by the multivar object. Silo provides the `DBPutMultivar` function for writing out multivar objects.

Example for writing a multivar:

C

```
void write_multivar(DBfile *dbfile)
{
    char **varnames = NULL;
    int dom, nvar = 4, *vartypes = NULL;
    /* Create the list of var names. */
    varnames = (char **)malloc(nvar * sizeof(char *));
    for(dom = 0; dom < nvar; ++dom)
    {
        char tmp[100];
        sprintf(tmp, "multivar.%d:var", dom);
        varnames[dom] = strdup(tmp);
    }
    /* Create the list of var types. */
    vartypes = (int *)malloc(nvar * sizeof(int));
    for(dom = 0; dom < nvar; ++dom)
        vartypes[dom] = DB_QUADVAR;
    /* Write the multivar. */
    DBPutMultivar(dbfile, "var", nvar, varnames, vartypes, NULL);
    /* Free the memory*/
    for(dom = 0; dom < nvar; ++dom)
        free(varnames[dom]);
    free(varnames);
    free(vartypes);
}
```

FortranFixed

```
    subroutine write_multivar(dbfile)
    implicit none
    include "silo.inc"
    integer err, ierr, dbfile, nvar, oldlen
    character*20 varnames(4) /'multivar.1:var ',
    .                         'multivar.2:var ',
    .                         'multivar.3:var ',
    .                         'multivar.4:var '/
    integer lvarnames(4) /14,14,14,14/
    integer vartypes(4) /DB_QUADVAR,DB_QUADVAR,
    .                    DB_QUADVAR,DB_QUADVAR/
c Set the maximum string length to 20 since that's how long
c our strings are
    oldlen = dbget2dstrlen()
    err = dbset2dstrlen(20)
c Write the multivar.
    nvar = 4
    err = dbputmvar(dbfile, "var", 3, nvar, varnames, lvarnames,
    .               vartypes, DB_F77NULL, ierr)
c Restore the previous value for maximum string length
    err = dbset2dstrlen(oldlen)
    end
```

EMPTY contributions

During the course of a calculation, sometimes only a subset of processors will contribute data. This means that they will not write data files. When some processors do not write data files, creating your multi-objects can become more

complicated. Note that because of how VisIt represents its domain subsets, etc, you will want to keep the number of filenames in a multi-object equal to the number of processors that you are using (the maximum number of domains that you will generate). If the length of the list varies over time then VisIt's subsetting controls may not behave as expected. To keep things simple, if you have N processors that write N files, you will always want N entries in your multiobjects. If a processor does not contribute any data, insert the `"EMPTY"` keyword into the multi-object in place of the path and variable. The `"EMPTY"` keyword allows the size of the multi-object to remain fixed over time even as the number of processors that contribute data changes. Keeping the size of the multi-object fixed over time ensures that VisIt's subsetting controls will continue to function as expected. Note that if you use the `"EMPTY"` keyword in a multivar object then the same entry in the multimesh object for the variable must also contain the "EMPTY" keyword.

Listing 8.3: C_Language example using the EMPTY keyword.

```
/* Processors 3,4 did not contribute so use EMPTY. */
char *meshnames[] = {"proc-1/file000/mesh", "proc-2/file000/mesh",
                     "EMPTY", "EMPTY"};
int meshtypes[] = {DB_QUAD_RECT, DB_QUAD_RECT,
                   DB_QUAD_RECT, DB_QUAD_RECT};
int nmesh = 4;
/* Write the multimesh. */
DBPutMultimesh(dbfile, "mesh", nmesh, meshnames, meshtypes, NULL);
```

For really large scale problems (say more than say 10^5), explicitly listing the names of constituent domain-level objects comprising a mutli-block object can result in a performance issue. As a result, Silo provides an `sprintf`– like mechanism for generating file system paths and Silo object paths on the fly. This mechanism is known as a *namescheme*. It is a best scalability practice to use Silo nameschemes for multi-block objects. For more information, see the Silo user's manual regarding DBMakeNamescheme and the associated optlist options, `DBOPT_MB_FILE_NS` and ``DBOPT_MB_BLOCK_NS` for multi-block objects. You can find many more examples of various features of Silo by browsing source code in either VisIt's or Silo's test suites or the test data files used in VisIt's testing.

8.7 The Xdmf file format

Xdmf (eXtensible Data Model and Format) files can represent a variety of meshes including the basic simple mesh types, such as, point, regular, rectilinear, curvilinear and unstructured. They also support multi-block meshes. Xdmf files consist of an XML (eXensible Markup Language) file containing meta data that references binary data in separate files. The binary files can either be raw binary files or HDF5 files. Xdmf also supports embedding the bulk data in the XML file, but this is usually for illustrative uses or small files.

There is a high-level Xdmf library that can be used to read and write Xdmf files. It's also possible to directly write out Xdmf files as well as the files containing bulk data. The advantage to using the library is that it is easier to use. The advantage to writing out the files directly is that there are fewer external dependencies for your simulation code. In fact, if you write out raw binary files, you will have zero external dependencies.

The remainder of this Xdmf documentation consists of a description of the file format, a simple example, and then more complex examples of the different mesh types.

The official Xdmf file format specification and description can be found at xdmf.org.

8.7.1 Basic structure of an Xdmf file

The overarching structure in an Xdmf file is a *Domain*. A *Domain* consists of one or more *Grids*. A *Grid* can be either *Uniform* or a *Collection*. A Collection contains one or more *Uniform Grids*.

A *Uniform* grid is the basic unit of grid and consists of a *Topology*, *Geometry* and zero or more *Attributes*. The *Geometry* defines the coordinates of the mesh. The *Topology* defines how the coordinates are connected. The *Attributes* are the fields on the mesh.

The data values for a *Topology* and *Geometry* are stored as a *DataItem*. A *DataItem* can be used to store numeric values directly within the Xdmf file or in an external file.

Here is the structure of a basic Xdmf file.

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="3.0">
 <Domain>
   <Grid ... >
     <Topology ... />
     <Geometry ... >
       <DataItem ... >
         ...
       </DataItem>
     </Geometry>
     <Attribute ... >
       <DataItem ... >
         ...
       </DataItem>
     </Attribute>
   </Grid>
 </Domain>
</Xdmf>
```

DataItem

The *DataItem* is used to store embedded data or external binary data.

The following is an example of a *DataItem* that embeds the data directly in the Xdmf file.

```
<DataItem Format="XML" NumberType="Float" Precision="4" Dimensions="3">
    1.0 2.0 3.0
</DataItem>
```

The following is an example of a *DataItem* that references an array in an HDF5 file.

```
<DataItem Format="HDF" NumberType="Float" Precision="4" Dimensions="3">
    output.h5:/values
</DataItem>
```

The following is an example of a *DataItem* that references an array in a binary file.

```
<DataItem Format="Binary" NumberType="Float" Precision="4" Endian="Little" Seek="0"␣
↪Dimensions="3">
    output.bin
</DataItem>
```

The valid values for *Format* are:

XML	Text imbedded in the xml file
HDF	Binary data stored in an HDF5 file
Binary	Binary data stored in a binary file

The valid values for *NumberType* are:

Float	Floating point
Int	Integer
UInt	Unsigned integer
Char	Character
UChar	Unsigned character

The valid values for *Precision* are:

1	Char or UChar
2	Int or UInt
4	Float, Int or UInt
8	Float, Int or UInt

Dimensions consists of one to three values representing a 1D, 2D or 3D array.

The valid values for *Endian* are:

Native	Native endian representation on the machine
Big	Big endian representation
Little	Little endian representation

Seek is an byte offset into a binary file.

Topology

The following mesh topologies are supported.

Structured	
2DSMesh	Curvilinear
2DRectMesh	Rectilinear
2DCoRectMesh	Regular
3DSMesh	Curvilinear
3DRectMesh	Rectilinear
3DCoRectMesh	Regular

Unstructured linear	
Linear	
Polyvertex	A group of unconnected points
Polyline	A group of line segments
Polygon	
Triangle	
Quadrilateral	
Tetrahedron	
Pyramid	
Wedge	
Hexahedron	

Unstructured quadratic	
Edge_3	Quadratic line with 3 nodes
Tri_6	
Quad_8	
Tet_10	
Pyramid_13	
Wedge_15	
Hex_20	

Unstructured arbitrary	
1	POLYVERTEX
2	POLYLINE
3	POLYGON
4	TRIANGLE
5	QUADRILATERAL
6	TETRAHEDRON
7	PYRAMID
8	WEDGE
9	HEXAHEDRON
16	POLYHEDRON
34	EDGE_3
35	QUADRILATERAL_9
36	TRIANGLE_6
37	QUADRILATERAL_8
38	TETRAHEDRON_10
39	PYRAMID_13
40	WEDGE_15
41	WEDGE_18
48	HEXAHEDRON_20
49	HEXAHEDRON_24
50	HEXAHEDRON_27

Geometry

The following mesh geometries are supported.

XYZ	Interlaced locations
XY	Z is set to 0.0
X_Y_Z	X, Y and Z are separate arrays
VXVYVZ	Three arrays, one for each axis
ORIGIN_DXDYDZ	Six values, Ox, Oy, Oz + Dx, Dy, Dz
ORIGIN_DXDY	Four values, Ox, Oy + Dx, Dy

Attribute

The following *AttributeType* are supported.

Scalar	
Vector	
Tensor	9 values expected
Tensor6	6 values expected

The following *Centering* are supported.

Node	Attributes are associated with nodes
Cell	Attributes are associated with cells

8.7.2 The C++ examples

The examples consist of C++ code fragments that write out Xdmf files directly. The code fragments that write out the corresponding HDF5 data are not shown. The full C++ source code that contains all the example XDMF code shown is found here. This includes the code that generates the example mesh data and the code that writes out the binary mesh data to the HDF5 file.

8.7.3 An example of a point mesh

A point mesh consists of an unstructured mesh made up of a collection of points. The mesh can be 2D or 3D. It is defined by a *Polyvertex* topology.

Here is the code that writes a 3D point mesh.

```
void
create_point3d()
{
    FILE *xmf = 0;

    /*
     * Open the file and write the header.
     */
    xmf = fopen("point3d.xmf", "w");
    fprintf(xmf, "<?xml version=\"1.0\" ?>\n");
    fprintf(xmf, "<!DOCTYPE Xdmf SYSTEM \"Xdmf.dtd\" []>\n");
    fprintf(xmf, "<Xdmf Version=\"2.0\">\n");

    /*
     * Write the mesh description and the variables defined on the mesh.
     */
    fprintf(xmf, " <Domain>\n");

    fprintf(xmf, "   <Grid Name=\"points\" GridType=\"Uniform\">\n");
    fprintf(xmf, "     <Topology TopologyType=\"Polyvertex\" Dimensions=\"%d\"␣
↪NodesPerElement=\"1\">\n", (NX+1)*(NY+1)*(NZ+1));
    fprintf(xmf, "       <DataItem Format=\"HDF\" Dimensions=\"%d\" NumberType=\"Int\
↪">\n", (NX+1)*(NY+1)*(NZ+1));
    fprintf(xmf, "        mesh.h5:/Indexes\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Topology>\n");
    fprintf(xmf, "     <Geometry Type=\"XYZ\">\n");
    fprintf(xmf, "       <DataItem Format=\"HDF\" Dimensions=\"%d 3\" NumberType=\
↪"Float\">\n", (NX+1)*(NY+1)*(NZ+1));
```

(continues on next page)

(continued from previous page)

```
    fprintf(xmf, "        mesh.h5:/XYZ\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Geometry>\n");
    fprintf(xmf, "     <Attribute Name=\"VelocityZ\" AttributeType=\"Scalar\" Center=\
↪"Node\">\n");
    fprintf(xmf, "       <DataItem Format=\"HDF\" Dimensions=\"%d\" NumberType=\"Int\
↪">\n", (NX+1)*(NY+1)*(NZ+1));
    fprintf(xmf, "        mesh.h5:/VelocityZ\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "   </Grid>\n");

    fprintf(xmf, " </Domain>\n");

    /*
     * Write the footer and close the file.
     */
    fprintf(xmf, "</Xdmf>\n");
    fclose(xmf);
}
```

Here is the resultant Xdmf file.

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="2.0">
 <Domain>
   <Grid Name="points" GridType="Uniform">
     <Topology TopologyType="Polyvertex" Dimensions="7161" NodesPerElement="1">
       <DataItem Format="HDF" Dimensions="7161" NumberType="Int">
        mesh.h5:/Indexes
       </DataItem>
     </Topology>
     <Geometry Type="XYZ">
       <DataItem Format="HDF" Dimensions="7161 3" NumberType="Float">
        mesh.h5:/XYZ
       </DataItem>
     </Geometry>
     <Attribute Name="VelocityZ" AttributeType="Scalar" Center="Node">
       <DataItem Format="HDF" Dimensions="7161" NumberType="Int">
        mesh.h5:/VelocityZ
       </DataItem>
     </Attribute>
   </Grid>
 </Domain>
</Xdmf>
```

8.7.4 An example of a regular mesh file

A regular mesh consists of a structured mesh with constant spacing in each direction. The mesh can be 2D or 3D. It is defined by a *2DCoRectMesh* or *3DCoRectMesh* topology.

Here is the code that writes a 3D regular mesh.

```
void
create_corect3d()
{
    FILE *xmf = 0;

    /*
     * Open the file and write the header.
     */
    xmf = fopen("corect3d.xmf", "w");
    fprintf(xmf, "<?xml version=\"1.0\" ?>\n");
    fprintf(xmf, "<!DOCTYPE Xdmf SYSTEM \"Xdmf.dtd\" []>\n");
    fprintf(xmf, "<Xdmf Version=\"2.0\">\n");

    /*
     * Write the mesh description and the variables defined on the mesh.
     */
    fprintf(xmf, " <Domain>\n");

    fprintf(xmf, "   <Grid Name=\"mesh\" GridType=\"Uniform\">\n");
    fprintf(xmf, "     <Topology TopologyType=\"3DCoRectMesh\" NumberOfElements=\"%d
↪%d %d\"/>\n", NZ+1, NY+1, NX+1);
    fprintf(xmf, "     <Geometry GeometryType=\"Origin_DxDyDz\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d\" NumberType=\"Float\" Precision=\
↪"4\" Format=\"HDF\">\n", 3);
    fprintf(xmf, "        mesh.h5:/Origin\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d\" NumberType=\"Float\" Precision=\
↪"4\" Format=\"HDF\">\n", 3);
    fprintf(xmf, "        mesh.h5:/DxDyDz\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Geometry>\n");
    fprintf(xmf, "     <Attribute Name=\"Pressure\" AttributeType=\"Scalar\" Center=\
↪"Cell\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d %d\" NumberType=\"UInt\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NZ, NY, NX);
    fprintf(xmf, "        mesh.h5:/Pressure\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "     <Attribute Name=\"VelocityZ\" AttributeType=\"Scalar\" Center=\
↪"Node\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d %d\" NumberType=\"Int\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NZ+1, NY+1, NX+1);
    fprintf(xmf, "        mesh.h5:/VelocityZ\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "   </Grid>\n");

    fprintf(xmf, " </Domain>\n");

    /*
     * Write the footer and close the file.
     */
    fprintf(xmf, "</Xdmf>\n");
    fclose(xmf);
}
```

Here is the resultant Xdmf file.

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="2.0">
 <Domain>
   <Grid Name="mesh" GridType="Uniform">
     <Topology TopologyType="3DCoRectMesh" NumberOfElements="11 21 31"/>
     <Geometry GeometryType="Origin_DxDyDz">
       <DataItem Dimensions="3" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/Origin
       </DataItem>
       <DataItem Dimensions="3" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/DxDyDz
       </DataItem>
     </Geometry>
     <Attribute Name="Pressure" AttributeType="Scalar" Center="Cell">
       <DataItem Dimensions="10 20 30" NumberType="UInt" Precision="4" Format="HDF">
        mesh.h5:/Pressure
       </DataItem>
     </Attribute>
     <Attribute Name="VelocityZ" AttributeType="Scalar" Center="Node">
       <DataItem Dimensions="11 21 31" NumberType="Int" Precision="4" Format="HDF">
        mesh.h5:/VelocityZ
       </DataItem>
     </Attribute>
   </Grid>
 </Domain>
</Xdmf>
```

Here is the code that writes a 2D regular mesh.

```
void
create_corect2d()
{
    FILE *xmf = 0;

    /*
     * Open the file and write the header.
     */
    xmf = fopen("corect2d.xmf", "w");
    fprintf(xmf, "<?xml version=\"1.0\" ?>\n");
    fprintf(xmf, "<!DOCTYPE Xdmf SYSTEM \"Xdmf.dtd\" []>\n");
    fprintf(xmf, "<Xdmf Version=\"2.0\">\n");

    /*
     * Write the mesh description and the variables defined on the mesh.
     */
    fprintf(xmf, " <Domain>\n");

    fprintf(xmf, "   <Grid Name=\"mesh\" GridType=\"Uniform\">\n");
    fprintf(xmf, "     <Topology TopologyType=\"2DCoRectMesh\" NumberOfElements=\"%d
↪%d\"/>\n", NY+1, NX+1);
    fprintf(xmf, "     <Geometry GeometryType=\"Origin_DxDy\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d\" NumberType=\"Float\" Precision=\
↪"4\" Format=\"HDF\">\n", 2);
    fprintf(xmf, "        mesh.h5:/Origin2\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d\" NumberType=\"Float\" Precision=\
↪"4\" Format=\"HDF\">\n", 2);
```

(continues on next page)

(continued from previous page)

```
    fprintf(xmf, "        mesh.h5:/DxDy\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Geometry>\n");
    fprintf(xmf, "     <Attribute Name=\"Pressure\" AttributeType=\"Scalar\" Center=\
↪"Cell\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d\" NumberType=\"UChar\"␣
↪Precision=\"1\" Format=\"HDF\">\n", NY, NX);
    fprintf(xmf, "        mesh.h5:/Pressure_2D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "     <Attribute Name=\"VelocityX\" AttributeType=\"Scalar\" Center=\
↪"Node\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d\" NumberType=\"Char\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NY+1, NX+1);
    fprintf(xmf, "        mesh.h5:/VelocityX_2D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "   </Grid>\n");

    fprintf(xmf, " </Domain>\n");

    /*
     * Write the footer and close the file.
     */
    fprintf(xmf, "</Xdmf>\n");
    fclose(xmf);
}
```

Here is the resultant Xdmf file.

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="2.0">
 <Domain>
   <Grid Name="mesh" GridType="Uniform">
     <Topology TopologyType="2DCoRectMesh" NumberOfElements="21 31"/>
     <Geometry GeometryType="Origin_DxDy">
       <DataItem Dimensions="2" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/Origin2
       </DataItem>
       <DataItem Dimensions="2" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/DxDy
       </DataItem>
     </Geometry>
     <Attribute Name="Pressure" AttributeType="Scalar" Center="Cell">
       <DataItem Dimensions="20 30" NumberType="UChar" Precision="1" Format="HDF">
        mesh.h5:/Pressure_2D
       </DataItem>
     </Attribute>
     <Attribute Name="VelocityX" AttributeType="Scalar" Center="Node">
       <DataItem Dimensions="21 31" NumberType="Char" Precision="4" Format="HDF">
        mesh.h5:/VelocityX_2D
       </DataItem>
     </Attribute>
   </Grid>
 </Domain>
</Xdmf>
```

8.7.5 An example of a rectilinear mesh

A rectilinear mesh consists of a structured mesh where the coordinates along each axis are specified as a 1-D array of values. The mesh can be 2D or 3D. It is defined by a *2DRectMesh* or *3DRectMesh* topology.

Here is the code that writes a 3D rectilinear mesh.

```
void
create_rect3d()
{
    FILE *xmf = 0;

    /*
     * Open the file and write the header.
     */
    xmf = fopen("rect3d.xmf", "w");
    fprintf(xmf, "<?xml version=\"1.0\" ?>\n");
    fprintf(xmf, "<!DOCTYPE Xdmf SYSTEM \"Xdmf.dtd\" []>\n");
    fprintf(xmf, "<Xdmf Version=\"2.0\">\n");

    /*
     * Write the mesh description and the variables defined on the mesh.
     */
    fprintf(xmf, " <Domain>\n");

    fprintf(xmf, "   <Grid Name=\"mesh\" GridType=\"Uniform\">\n");
    fprintf(xmf, "     <Topology TopologyType=\"3DRectMesh\" NumberOfElements=\"%d %d
↪%d\"/>\n", NZ+1, NY+1, NX+1);
    fprintf(xmf, "     <Geometry GeometryType=\"VXVYVZ\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d\" NumberType=\"Float\" Precision=\
↪"4\" Format=\"HDF\">\n", NX+1);
    fprintf(xmf, "        mesh.h5:/X_1D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d\" NumberType=\"Float\" Precision=\
↪"4\" Format=\"HDF\">\n", NY+1);
    fprintf(xmf, "        mesh.h5:/Y_1D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d\" NumberType=\"Float\" Precision=\
↪"4\" Format=\"HDF\">\n", NZ+1);
    fprintf(xmf, "        mesh.h5:/Z_1D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Geometry>\n");
    fprintf(xmf, "     <Attribute Name=\"Pressure\" AttributeType=\"Scalar\" Center=\
↪"Cell\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d %d\" NumberType=\"UInt\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NZ, NY, NX);
    fprintf(xmf, "        mesh.h5:/Pressure\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "     <Attribute Name=\"VelocityZ\" AttributeType=\"Scalar\" Center=\
↪"Node\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d %d\" NumberType=\"Int\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NZ+1, NY+1, NX+1);
    fprintf(xmf, "        mesh.h5:/VelocityZ\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "   </Grid>\n");
```

(continues on next page)

(continued from previous page)

```
    fprintf(xmf, " </Domain>\n");

    /*
     * Write the footer and close the file.
     */
    fprintf(xmf, "</Xdmf>\n");
    fclose(xmf);
}
```

Here is the resultant Xdmf file.

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="2.0">
 <Domain>
   <Grid Name="mesh" GridType="Uniform">
     <Topology TopologyType="3DRectMesh" NumberOfElements="11 21 31"/>
     <Geometry GeometryType="VXVYVZ">
       <DataItem Dimensions="31" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/X_1D
       </DataItem>
       <DataItem Dimensions="21" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/Y_1D
       </DataItem>
       <DataItem Dimensions="11" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/Z_1D
       </DataItem>
     </Geometry>
     <Attribute Name="Pressure" AttributeType="Scalar" Center="Cell">
       <DataItem Dimensions="10 20 30" NumberType="UInt" Precision="4" Format="HDF">
        mesh.h5:/Pressure
       </DataItem>
     </Attribute>
     <Attribute Name="VelocityZ" AttributeType="Scalar" Center="Node">
       <DataItem Dimensions="11 21 31" NumberType="Int" Precision="4" Format="HDF">
        mesh.h5:/VelocityZ
       </DataItem>
     </Attribute>
   </Grid>
 </Domain>
</Xdmf>
```

Here is the code that writes a 2D rectilinear mesh.

```
void
create_rect2d()
{
    FILE *xmf = 0;

    /*
     * Open the file and write the header.
     */
    xmf = fopen("rect2d.xmf", "w");
    fprintf(xmf, "<?xml version=\"1.0\" ?>\n");
    fprintf(xmf, "<!DOCTYPE Xdmf SYSTEM \"Xdmf.dtd\" []>\n");
    fprintf(xmf, "<Xdmf Version=\"2.0\">\n");
```

(continues on next page)

(continued from previous page)

```
    /*
     * Write the mesh description and the variables defined on the mesh.
     */
    fprintf(xmf, " <Domain>\n");

    fprintf(xmf, "   <Grid Name=\"mesh\" GridType=\"Uniform\">\n");
    fprintf(xmf, "     <Topology TopologyType=\"2DRectMesh\" NumberOfElements=\"%d %d\
↪"/>\n", NY+1, NX+1);
    fprintf(xmf, "     <Geometry GeometryType=\"VXVY\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d\" NumberType=\"Float\" Precision=\
↪"4\" Format=\"HDF\">\n", NX+1);
    fprintf(xmf, "        mesh.h5:/X_1D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d\" NumberType=\"Float\" Precision=\
↪"4\" Format=\"HDF\">\n", NY+1);
    fprintf(xmf, "        mesh.h5:/Y_1D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Geometry>\n");
    fprintf(xmf, "     <Attribute Name=\"Pressure\" AttributeType=\"Scalar\" Center=\
↪"Cell\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d\" NumberType=\"UChar\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NY, NX);
    fprintf(xmf, "        mesh.h5:/Pressure_2D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "     <Attribute Name=\"VelocityX\" AttributeType=\"Scalar\" Center=\
↪"Node\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d\" NumberType=\"Char\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NY+1, NX+1);
    fprintf(xmf, "        mesh.h5:/VelocityX_2D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "   </Grid>\n");

    fprintf(xmf, " </Domain>\n");

    /*
     * Write the footer and close the file.
     */
    fprintf(xmf, "</Xdmf>\n");
    fclose(xmf);
}
```

Here is the resultant Xdmf file.

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="2.0">
 <Domain>
   <Grid Name="mesh" GridType="Uniform">
     <Topology TopologyType="2DRectMesh" NumberOfElements="21 31"/>
     <Geometry GeometryType="VXVY">
       <DataItem Dimensions="31" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/X_1D
       </DataItem>
       <DataItem Dimensions="21" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/Y_1D
```

(continues on next page)

(continued from previous page)

```
      </DataItem>
    </Geometry>
    <Attribute Name="Pressure" AttributeType="Scalar" Center="Cell">
      <DataItem Dimensions="20 30" NumberType="UChar" Precision="4" Format="HDF">
       mesh.h5:/Pressure_2D
      </DataItem>
    </Attribute>
    <Attribute Name="VelocityX" AttributeType="Scalar" Center="Node">
      <DataItem Dimensions="21 31" NumberType="Char" Precision="4" Format="HDF">
       mesh.h5:/VelocityX_2D
      </DataItem>
    </Attribute>
  </Grid>
 </Domain>
</Xdmf>
```

8.7.6 An example of a curvilinear mesh

A curvilinear mesh consists of a structured mesh where the coordinates are specified as multi-dimensional arrays of values. The mesh can be 2D or 3D. It is defined by a *2DSMesh* or *3DSMesh* topology.

Here is the code that writes a 3D curvilinear mesh.

```
void
create_curv3d()
{
    FILE *xmf = 0;

    /*
     * Open the file and write the header.
     */
    xmf = fopen("curv3d.xmf", "w");
    fprintf(xmf, "<?xml version=\"1.0\" ?>\n");
    fprintf(xmf, "<!DOCTYPE Xdmf SYSTEM \"Xdmf.dtd\" []>\n");
    fprintf(xmf, "<Xdmf Version=\"2.0\">\n");

    /*
     * Write the mesh description and the variables defined on the mesh.
     */
    fprintf(xmf, " <Domain>\n");

    fprintf(xmf, "   <Grid Name=\"mesh\" GridType=\"Uniform\">\n");
    fprintf(xmf, "     <Topology TopologyType=\"3DSMesh\" NumberOfElements=\"%d %d %d\
↪"/>\n", NZ+1, NY+1, NX+1);
    fprintf(xmf, "     <Geometry GeometryType=\"XYZ\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d\" NumberType=\"Float\"␣
↪Precision=\"4\" Format=\"HDF\">\n", (NZ+1)*(NY+1)*(NX+1), 3);
    fprintf(xmf, "        mesh.h5:/XYZ\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Geometry>\n");
    fprintf(xmf, "     <Attribute Name=\"Pressure\" AttributeType=\"Scalar\" Center=\
↪"Cell\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d %d\" NumberType=\"UInt\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NZ, NY, NX);
    fprintf(xmf, "        mesh.h5:/Pressure\n");
```

(continues on next page)

(continued from previous page)

```
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "     <Attribute Name=\"VelocityZ\" AttributeType=\"Scalar\" Center=\
↪"Node\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d %d\" NumberType=\"Int\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NZ+1, NY+1, NX+1);
    fprintf(xmf, "        mesh.h5:/VelocityZ\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "     <Attribute Name=\"Velocity\" AttributeType=\"Vector\" Center=\
↪"Node\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d %d 3\" NumberType=\"Float\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NZ+1, NY+1, NX+1);
    fprintf(xmf, "        mesh.h5:/Velocity\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "     <Attribute Name=\"Stress\" AttributeType=\"Tensor6\" Center=\
↪"Node\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d %d 6\" NumberType=\"Float\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NZ+1, NY+1, NX+1);
    fprintf(xmf, "        mesh.h5:/Stress\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "   </Grid>\n");

    fprintf(xmf, " </Domain>\n");

    /*
     * Write the footer and close the file.
     */
    fprintf(xmf, "</Xdmf>\n");
    fclose(xmf);
}
```

Here is the resultant Xdmf file.

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="2.0">
 <Domain>
   <Grid Name="mesh" GridType="Uniform">
     <Topology TopologyType="3DSMesh" NumberOfElements="11 21 31"/>
     <Geometry GeometryType="XYZ">
       <DataItem Dimensions="7161 3" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/XYZ
       </DataItem>
     </Geometry>
     <Attribute Name="Pressure" AttributeType="Scalar" Center="Cell">
       <DataItem Dimensions="10 20 30" NumberType="UInt" Precision="4" Format="HDF">
        mesh.h5:/Pressure
       </DataItem>
     </Attribute>
     <Attribute Name="VelocityZ" AttributeType="Scalar" Center="Node">
       <DataItem Dimensions="11 21 31" NumberType="Int" Precision="4" Format="HDF">
        mesh.h5:/VelocityZ
       </DataItem>
     </Attribute>
```

(continues on next page)

(continued from previous page)

```
     <Attribute Name="Velocity" AttributeType="Vector" Center="Node">
       <DataItem Dimensions="11 21 31 3" NumberType="Float" Precision="4" Format="HDF
↪">
        mesh.h5:/Velocity
       </DataItem>
     </Attribute>
     <Attribute Name="Stress" AttributeType="Tensor6" Center="Node">
       <DataItem Dimensions="11 21 31 6" NumberType="Float" Precision="4" Format="HDF
↪">
        mesh.h5:/Stress
       </DataItem>
     </Attribute>
   </Grid>
 </Domain>
</Xdmf>
```

Here is the code that writes a 2D curvilinear mesh.

```
void
create_curv2d()
{
    FILE *xmf = 0;

    /*
     * Open the file and write the header.
     */
    xmf = fopen("curv2d.xmf", "w");
    fprintf(xmf, "<?xml version=\"1.0\" ?>\n");
    fprintf(xmf, "<!DOCTYPE Xdmf SYSTEM \"Xdmf.dtd\" []>\n");
    fprintf(xmf, "<Xdmf Version=\"2.0\">\n");

    /*
     * Write the mesh description and the variables defined on the mesh.
     */
    fprintf(xmf, " <Domain>\n");

    fprintf(xmf, "   <Grid Name=\"mesh\" GridType=\"Uniform\">\n");
    fprintf(xmf, "     <Topology TopologyType=\"2DSMesh\" NumberOfElements=\"%d %d\"/>
↪\n", NY+1, NX+1);
    fprintf(xmf, "     <Geometry GeometryType=\"XY\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d\" NumberType=\"Float\"␣
↪Precision=\"4\" Format=\"HDF\">\n", (NY+1)*(NX+1), 2);
    fprintf(xmf, "        mesh.h5:/XY\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Geometry>\n");
    fprintf(xmf, "     <Attribute Name=\"Pressure_2D\" AttributeType=\"Scalar\"␣
↪Center=\"Cell\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d\" NumberType=\"UChar\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NY, NX);
    fprintf(xmf, "        mesh.h5:/Pressure_2D\n");
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "     <Attribute Name=\"VelocityX_2D\" AttributeType=\"Scalar\"␣
↪Center=\"Node\">\n");
    fprintf(xmf, "       <DataItem Dimensions=\"%d %d\" NumberType=\"Char\"␣
↪Precision=\"4\" Format=\"HDF\">\n", NY+1, NX+1);
    fprintf(xmf, "        mesh.h5:/VelocityX_2D\n");
```

(continues on next page)

(continued from previous page)

```
    fprintf(xmf, "       </DataItem>\n");
    fprintf(xmf, "     </Attribute>\n");
    fprintf(xmf, "   </Grid>\n");

    fprintf(xmf, " </Domain>\n");

    /*
     * Write the footer and close the file.
     */
    fprintf(xmf, "</Xdmf>\n");
    fclose(xmf);
}
```

Here is the resultant Xdmf file.

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="2.0">
 <Domain>
   <Grid Name="mesh" GridType="Uniform">
     <Topology TopologyType="2DSMesh" NumberOfElements="21 31"/>
     <Geometry GeometryType="XY">
       <DataItem Dimensions="651 2" NumberType="Float" Precision="4" Format="HDF">
        mesh.h5:/XY
       </DataItem>
     </Geometry>
     <Attribute Name="Pressure_2D" AttributeType="Scalar" Center="Cell">
       <DataItem Dimensions="20 30" NumberType="UChar" Precision="4" Format="HDF">
        mesh.h5:/Pressure_2D
       </DataItem>
     </Attribute>
     <Attribute Name="VelocityX_2D" AttributeType="Scalar" Center="Node">
       <DataItem Dimensions="21 31" NumberType="Char" Precision="4" Format="HDF">
        mesh.h5:/VelocityX_2D
       </DataItem>
     </Attribute>
   </Grid>
 </Domain>
</Xdmf>
```

8.8 The Conduit/Blueprint file format

Conduit is a library for intuitively describing hierarchical scientific data in C++/C, Fortran, and Python. It is used for coupling between packages in-core, serialization, and I/O tasks. The Conduit *Node* is the basic abstraction for describing data. The *Node* supports hierarchical construction.

Here is a simple example of using Conduit in Python.

```
import conduit

n = conduit.Node()
n["my"] = "data"
n["a/b/c"] = "d"
n["a"]["b"]["e"] = 64.0
```

(continues on next page)

(continued from previous page)

```
print(n)

print("total bytes: %d" % n.total_strided_bytes())
```

Here is the output.

```
my: "data"
a:
  b:
    c: "d"
    e: 64.0

total bytes: 15
```

Blueprint is a set of higher-level conventions for describing meshes and fields defined on those meshes. The Conduit library can be used to describe in-memory arrays defining a mesh and fields conforming to the Blueprint data model and then writing them to disk. Blueprint can be used to check if a Conduit Node conforms to the Blueprint specification.

Here is a simple example writing a Blueprint uniform mesh in Python.

```
import conduit
import conduit.relay.io
import conduit.blueprint.mesh
import numpy

mesh = conduit.Node()

# Create the coordinate set.
mesh["coordsets/coords/type"] = "uniform"
mesh["coordsets/coords/dims/i"] = 3
mesh["coordsets/coords/dims/j"] = 3
mesh["coordsets/coords/origin/x"] = -10.0
mesh["coordsets/coords/origin/y"] = -10.0
mesh["coordsets/coords/spacing/dx"] = 10.0
mesh["coordsets/coords/spacing/dy"] = 10.0

# Add the topology.
mesh["topologies/topo/type"] = "uniform"
mesh["topologies/topo/coordset"] = "coords"

# Add a simple element-associated field.
mesh["fields/ele_example/association"] = "element"
mesh["fields/ele_example/topology"] = "topo"

edata = numpy.array([1, 2, 3, 4], dtype=numpy.float64)
mesh["fields/ele_example/values"] = edata

# Add a simple vertex-associated field.
mesh["fields/vert_example/association"] = "vertex"
mesh["fields/vert_example/topology"] = "topo"
vdata = numpy.array([1, 1, 1, 2, 2, 2, 3, 3, 3], dtype=numpy.float64)
mesh["fields/vert_example/values"] = vdata

# Verify that the mesh conforms to the specification.
verify_info = conduit.Node()
if not conduit.blueprint.mesh.verify(mesh, verify_info):
    print("Verify failed")
```

(continues on next page)

(continued from previous page)

```
    print(verify_info)

print(mesh)

conduit.relay.io.blueprint.write_mesh(mesh, "blueprint_example", "json")
```

Here is the output.

```
coordsets:
  coords:
    type: "uniform"
    dims:
      i: 3
      j: 3
    origin:
      x: -10.0
      y: -10.0
    spacing:
      dx: 10.0
      dy: 10.0
topologies:
  topo:
    type: "uniform"
    coordset: "coords"
fields:
  ele_example:
    association: "element"
    topology: "topo"
    values: [1.0, 2.0, 3.0, 4.0]
  vert_example:
    association: "vertex"
    topology: "topo"
    values: [1.0, 1.0, 1.0, 2.0, 2.0, 2.0, 3.0, 3.0, 3.0]

PyRelay_io_blueprint_write_mesh
```

The complete documentation for Blueprint can be found here.

8.9 Advanced Topics

This section elaborates on some of the advanced topics involved in creating files that VisIt can read. Most applications should be able to write out all of their data using information contained in the previous sections. This section introduces advanced topics such as incorporating metadata to accelerate VisIt's performance as well as some less common data representations. Many of the examples in this chapter use the Silo library, which was introduced previously. For more information on getting started with the Silo library, see *Silo file format*.

8.9.1 Writing Vector Data

The components of vector data are often stored to files as individual scalar variables and VisIt uses an expression to compose the scalars back into a vector field. If you use the Silo library, you can always choose instead to store your vector data as a multi-component variable. *Silo file format* provided several examples that use the Silo library to write scalar variables on rectilinear, curvilinear, point, and unstructured meshes. The functions that were used to write the

scalars were simplified forms of the functions that are used to write vector data. The scalar functions that were used to write data for a specific mesh type as well as the vector function equivalents are listed in the following table:

Mesh type	Scalar function	Vector function
Rectilinear mesh	DBPutQuadvar1	DBPutQuadVar
Curvilinear mesh	DBPutQuadvar1	DBPutQuadVar
Point mesh	DBPutPointvar1	DBPutPointVar
Unstructured mesh	DBPutUcdvar1	DBPutUcdvar

The differences between a scalar function and a vector function are small. In fact, the argument lists for a scalar function and a vector function are nearly identical in the Silo library's C-Language interface. The chief difference is that the vector functions take two additional arguments and the meaning of one existing argument is modified. The first new argument is an integer indicating the number of components contained by the variable to be written. The next difference is that you must pass an array of pointers to character strings that represent the names of each individual component. Finally, the argument that was used to pass the data to the **DBPutQuadvar1** function, now in the **DBPutQuadvar** function, accepts an array of pointers to the various arrays that contain the variable components. For more complete information on each of the arguments to the functions that Silo uses to write multi-component data, refer to the Silo Manual.

Silo's Fortran interface does not provide functions to write out multi-component data such as vectors. If you use the Fortran interface to Silo, you will have to write out the vector components as separate scalar variables and then write an expression to your Silo file that composes the components into a single vector variable.

Example for writing vector data using Silo.

C

```
int i, dims[3], ndims = 3;
int nnodes = NX*NY*NZ;
float *comp[3];
char *varnames[] = {"nodal_comp0","nodal_comp1","nodal_comp2"};
comp[0] = (float *)malloc(sizeof(float)*nnodes);
comp[1] = (float *)malloc(sizeof(float)*nnodes);
comp[2] = (float *)malloc(sizeof(float)*nnodes);
for(i = 0; i < nnodes; ++i)
{
    comp[0][i] = (float)i; /*vector component 0*/
    comp[1][i] = (float)i; /*vector component 1*/
    comp[2][i] = (float)i; /*vector component 2*/
}
dims[0] = NX; dims[1] = NY; dims[2] = NZ;
DBPutQuadvar(dbfile, "nodal", "quadmesh",
             3, varnames, comp, dims,
             ndims, NULL, 0, DB_FLOAT, DB_NODECENT, NULL);
free(comp[0]);
free(comp[1]);
free(comp[2]);
```

FortranFixed

```
    subroutine write_nodecent_quadvar(dbfile)
    implicit none
    integer dbfile
    include "silo.inc"
    integer err, ierr, dims(3), ndims,i,j,k,index,NX,NY,NZ
    parameter (NX = 4)
    parameter (NY = 3)
```

(continues on next page)

(continued from previous page)

```
    parameter (NZ = 2)
    real comp0(NX,NY,NZ), comp1(NX,NY,NZ), comp2(NX,NY,NZ)
    data dims/NX,NY,NZ/
    index = 0
    do 20020 k=1,NZ
        do 20010 j=1,NY
            do 20000 i=1,NX
                comp0(i,j,k) = float(index)
                comp1(i,j,k) = float(index)
                comp2(i,j,k) = float(index)
                index = index + 1
20000 continue
20010 continue
20020 continue
    ndims = 3
    err = dbputqv1(dbfile, "n_comp0", 11, "quadmesh", 8, comp0, dims, ndims,
    .              DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL, ierr)
    err = dbputqv1(dbfile, "n_comp1", 11, "quadmesh", 8, comp1, dims, ndims,
    .              DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL, ierr)
    err = dbputqv1(dbfile, "n_comp2", 11, "quadmesh", 8, comp2, dims, ndims,
    .              DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL, ierr)
    end
    subroutine write_defvars(dbfile)
    implicit none
    integer dbfile
    include "silo.inc"
    integer err, ierr, types(2), lnames(2), ldefs(2), oldlen
c Initialize some 20 character length strings
    character*40 names(2) /'zonalvec ' ,
    .                      ' nodalvec ' /
    character*40 defs(2) /'{z_comp0,z_comp1,z_comp2} ',
    .                     '{n_comp0,n_comp1,n_comp2} '/
c Store the length of each string
    data lnames/8, 8/
    data ldefs/37, 37/
    data types/DB_VARTYPE_VECTOR, DB_VARTYPE_VECTOR/
c Set the maximum string length to 40 since that is how long our
c strings are
    oldlen = dbget2dstrlen()
    err = dbset2dstrlen(40)
c Write out the expressions
    err = dbputdefvars(dbfile, "defvars", 7, 2, names, lnames,
    .                  types, defs, ldefs, DB_F77NULL, ierr)
c Restore the previous value for maximum string length
    err = dbset2dstrlen(oldlen)
    end
```

8.9.2 Adding metadata for performance boosts

VisIt incorporates several performance boosting strategies that make use of metadata, if it is available. Most of the metadata applies to increasing parallel performance by reducing the amount of I/O and subsequent processing that is required. The I/O reductions are realized by not reading in and processing domains that will contribute nothing to the final image on the screen. In order to prevent domains from being read in, your multi-objects must have associated metadata for each of the domains that they contain. When a Silo multi-object contains metadata about all of its constituent domains, VisIt can make worksaving decisions since it knows the properties of each domain without

having to read in the data for each domain.

This section explains how to add metadata to your Silo multi-objects using option lists. Metadata attached to multi-objects allow VisIt to determine important data characteristics such as data extents or the spatial extents of the mesh without having to first read and process all domains. Such knowledge allows VisIt to restrict the number of domains that are processed, thus reducing the amount of work and the time required to display images on your screen.

Writing data extents

Providing data extents can help VisIt only read in and process those domains that will contribute to the final image. Many types of plots and operators use data extents for each domain, when they are provided, to perform a simple upfront test to determine if a domain contains the values which will be used. If a domain is not needed then VisIt will not read that domain because it is known beforehand that the domain does not contain the desired value.

An example of a plot that uses data extents in order to save work is VisIt's *Contour plot*. The *Contour plot* creates contours (lines or surfaces where the data has the same value) through a dataset. Consider the example shown in Figure 8.17, where the entire mesh and scalar field are divided into four smaller domains where the data extents of each domain are stored to the file so VisIt can perform optimizations. Before the Contour plot executes, it tells VisIt the data values for which it will make contours. Suppose that that you wanted to see the areas where the value in the scalar field are equal to 11.5. The *Contour plot* takes that 11.5 contour value and compares it to the data extents for all of the domains to see which domains will be needed. If a domain will not be needed then VisIt will make no further effort to read the domain or process it, thus saving work and making the plot appear on the screen faster than it could if the data extents were not available in the file metadata. In the above example, the value of 11.5 is only present in domain 3, which means that the *Contour plot* will only return a result if it processes data from domain 3.

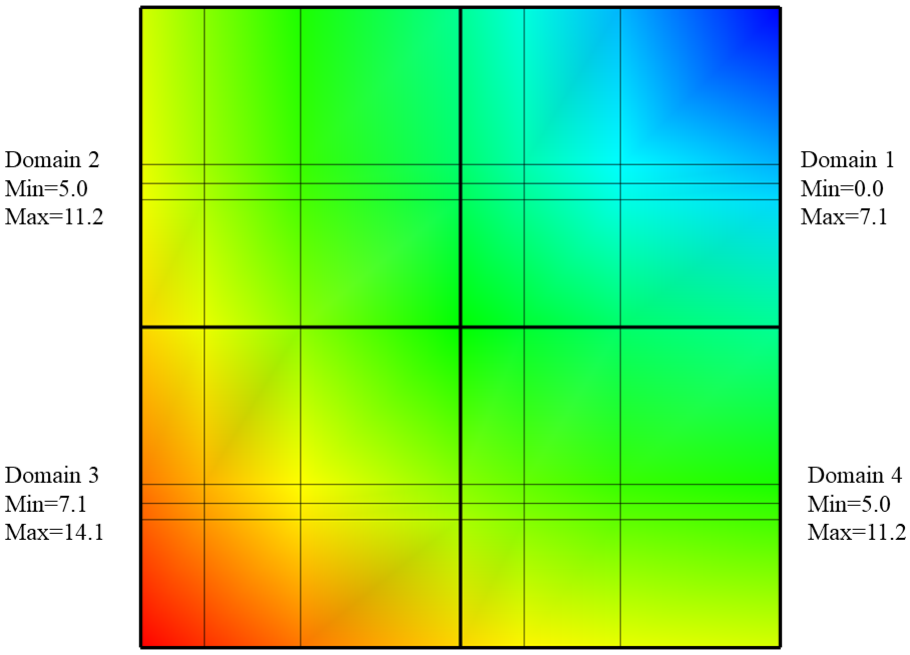


Fig. 8.17: Example Mesh and Psuedocolor plots with the data extents for each domain of the Pseudocolor plots' scalar variable.

The other domains are not processed in this case because they do not contain the required value of 11.5. After the comparisons have been made, VisIt knows which domains will have to be processed and it can divide the set of domains (just domain 3 in this case) that will contribute to the visualization among processors so they can execute the plot and return data to VisIt's viewer where it can be displayed.

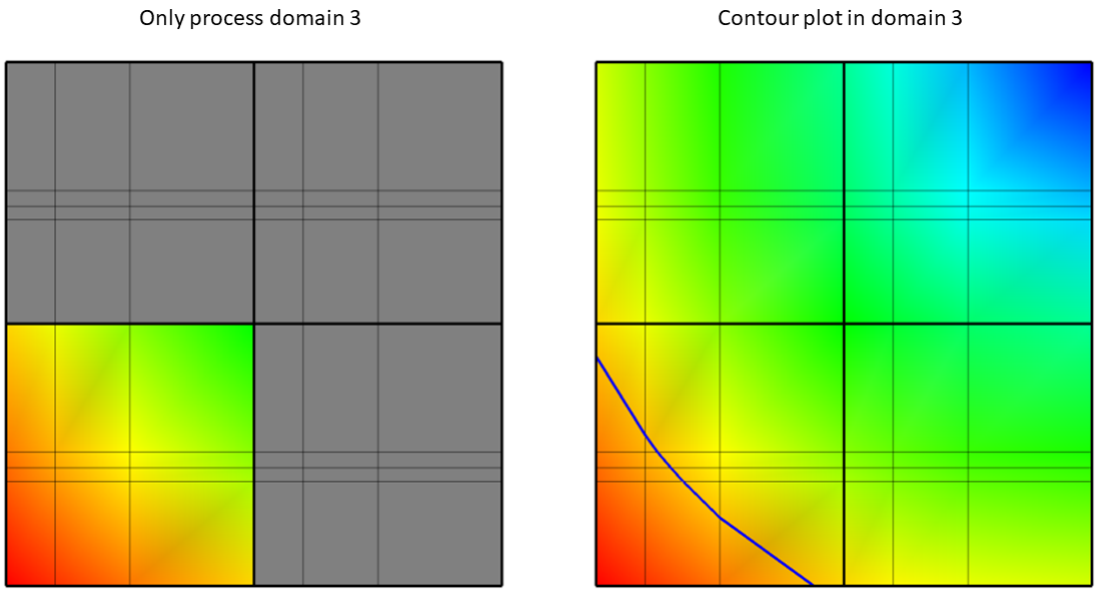


Fig. 8.18: Only process domain 3 (left) to yield the Contour plot of value 11.5 (right).

To add the data extents for each processor to the metadata using Silo, you must add the data extents to the option list that you pass to the **DBPutMultivar** function call. Having the data extents for each domain readily available in the Multivar object ensures that VisIt will have enough information to determine which domains will be necessary for operations such as Contour without having to read all of the data to determine which domains contribute to the visualization. The data extents must be stored in a double precision array that has enough entries to accommodate the min and max values for each domain in the multivar object. The layout of the min and max values within that array are as follows: *min_dom1*, *max_dom1*, *min_dom2*, *max_dom2*, . . . , *min_domN*, *max_domN*.

Example for writing data extents using Silo.

C

```
const int two = 2;
double extents[NDOMAINS][2];
DBoptlist *optlist = NULL;
/* Calculate the per-domain data extents for this variable. */
/* Write the multivar.*/
optlist = DBMakeOptlist(2);
DBAddOption(optlist, DBOPT_EXTENTS_SIZE, (void *)&two);
DBAddOption(optlist, DBOPT_EXTENTS, (void *)extents);
DBPutMultivar(dbfile, "var", nvar, varnames, vartypes, optlist);
DBFreeOptlist(optlist);
```

FortranFixed

```
    double precision extents(2,NDOMAINS)
    integer err, optlist
c Calculate the per-domain data extents for this variable.
c Write the multivar.
    err = dbmkoptlist(2, optlist)
    err = dbaddiopt(optlist, DBOPT_EXTENTS_SIZE, 2)
    err = dbadddopt(optlist, DBOPT_EXTENTS, extents)
    err = dbputmvar(dbfile, "var", 3, nvar, varnames, lvarnames,
```

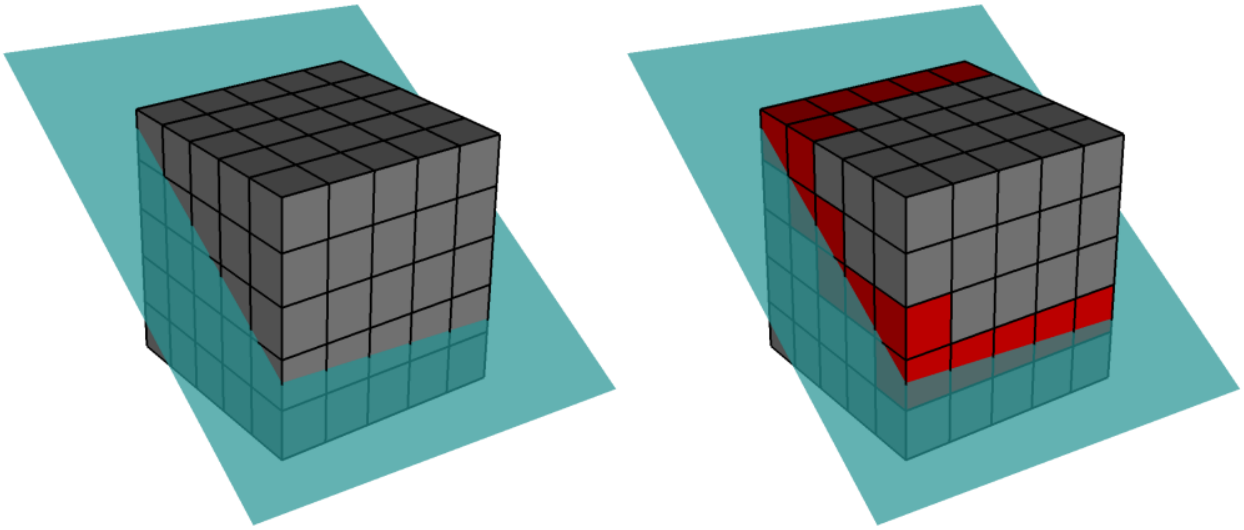
(continues on next page)

(continued from previous page)

```
    .                vartypes, optlist, ierr)
    err = dbfreeoptlist(optlist)
```

Writing spatial extents

If you provide spatial extents for each domain in your database then VisIt can use that information during spatial data reduction operations, such as slicing, to reduce the number of domains that must be read from disk and processed.



Spatial extents for a domain contain the minimum and maximum values of the coordinates within that domain, also called the domain's bounding box. The spatial extents must be stored in a double precision array that has enough entries to accommodate the min and max coordinate values for each domain in the multimesh object. The layout of the min and max values within that array for 3D domains are as follows: *xmin_dom1*, *ymin_dom1*, *zmin_dom1*, *xmax_dom1*, *ymax_dom1*, *zmax_dom1*, . . . , *xmin_domN*, *ymin_domN*, *zmin_domN*, *xmax_domN*, *ymax_domN*, *zmax_domN*. In the event that you have 2D domains then you can omit the z-components of the min and max values and tell Silo that there are 4 values per min/max tuple instead of the 6 values required to specify min and max values for 3D domains.

Example for writing spatial extents using Silo.

C

```
const int six = 6;
double spatial_extents[NDOMAINS][6];
DBoptlist *optlist = NULL;
/* Calculate the per-domain spatial extents for this mesh. */
for(int i = 0; i < NDOMAINS; ++i)
{
    spatial_extents[i][0] = xmin; /* xmin for i'th domain */
    spatial_extents[i][1] = ymin; /* ymin for i'th domain */
    spatial_extents[i][2] = zmin; /* zmin for i'th domain */
    spatial_extents[i][3] = xmin; /* xmax for i'th domain */
    spatial_extents[i][4] = ymax; /* ymax for i'th domain */
    spatial_extents[i][5] = zmax; /* zmax for i'th domain */
}
```

(continues on next page)

(continued from previous page)

```
/* Write the multimesh. */
optlist = DBMakeOptlist(2);
DBAddOption(optlist, DBOPT_EXTENTS_SIZE, (void *)&six);
DBAddOption(optlist, DBOPT_EXTENTS, (void *)spatial_extents);
DBPutMultimesh(dbfile, "mesh", nmesh, meshnames, meshtypes, optlist);
DBFreeOptlist(optlist);
```

FortranFixed

```
    double precision spatial_extents(6,NDOMAINS)
    integer optlist, err, dom
c Calculate the per-domain spatial extents for this mesh.
    do 10000 dom=1,NDOMAINS
        spatial_extents(1,dom) = xmin
        spatial_extents(2,dom) = ymin
        spatial_extents(3,dom) = zmin
        spatial_extents(4,dom) = xmin
        spatial_extents(5,dom) = ymax
        spatial_extents(6,dom) = zmax
    10000 continue
c Write the multimesh
    err = dbmkoptlist(2, optlist)
    err = dbaddiopt(optlist, DBOPT_EXTENTS_SIZE, 6)
    err = dbadddopt(optlist, DBOPT_EXTENTS, spatial_extents)
    err = dbputmmesh(dbfile, "quadmesh", 8, nmesh, meshnames,
    .                lmeshnames, meshtypes, optlist, ierr)
    err = dbfreeoptlist(optlist)
```

8.9.3 Ghost zones

Ghost zones are zones external to a domain, which correspond to zones in an adjacent domain. Ghost zones allow VisIt to ensure continuity between domains containing zonecentered data, making surfaces such as *Contour plot* continuous across domain boundaries instead of creating surfaces with ugly gaps at the domain boundaries. Ghost zones also allow VisIt to remove internal surfaces from the visualized data for plots such as *Pseudocolor plot*, which only wants to keep the surfaces that are external to the model. Removing internal surfaces results in fewer primitives that must be rendered on the graphics card and that increases interactivity with plots. See Figure 8.19 for examples of the problems that ghost zones allow VisIt to fix.

Ghost zones can be stored into the database so VisIt can read them when the data is visualized. Ghost zones can also be created on-the-fly for structured (rectilinear and curvilinear) meshes if multimesh adjacency information is provided. This section will show how to write ghost zones to the file. If you are interested in providing multimesh adjacency information so you can write smaller files and so VisIt can automatically create ghost zones then refer to the documentation for the **DBPutMultimeshadj** function in the Silo Manual.

Writing ghost zones to your files

You can write ghost zones to your files using the Silo library or you can instead write a multimesh adjacency object, covered in the Silo Manual that VisIt can use to automatically create ghost zones. This section will cover how to use the Silo library to store ghost zones explicitly in your files. The first step in creating ghost zones is to add a layer of zones around the mesh in each domain of your database where a domain boundary exists. Each zone in the layer of added ghost zones must match the location and have the same data value as the zone in the domain that it is meant to mirror in order for VisIt to be able to successfully use ghost zones to remove domain decomposition artifacts. This means that you must change your code for writing out meshes and variables so your meshes have an addition layer of

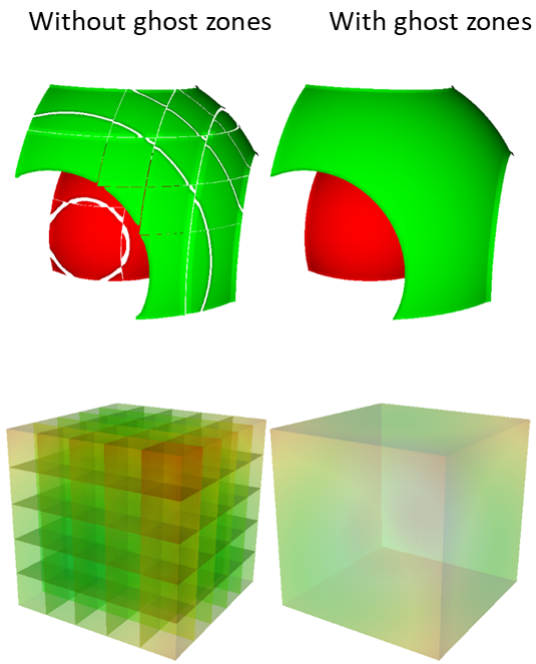


Fig. 8.19: VisIt can use ghost zones to ensure continuity and to remove internal surfaces.

zones for each domain boundary that is internal to the model. Your variables must also contain valid data values in the ghost zones since providing a domain with knowledge of the data values of its neighboring domains is the entire point of adding ghost zones. Note that you should not add ghost zones on the surface of a domain where the surface is external to the model. When ghost zones are erroneously added to external surfaces of the model, VisIt removes the external faces and this can cause plots to be invisible.

Figure 8.20 shows two domains: domain1 (red) and domain2 (green). The boundary between (blue) the two domains is the interface that would exist between the domains if there were no ghost zones. When you add a layer of ghost zones, each domain intrudes a little into the other domain's bounding box so the zones in one domain's layer of ghost zones match the zones in the other domain's external layer of zones. Of course, domains on both sides of the domain boundary have ghost zones to ensure that VisIt will know the proper zone-centered data values whether it approaches the domain boundary from the left or from the right. The first row of cells on either side of the domain boundary are ghost zones. For example, if you look at the upper left zone containing the "G" for ghost zone, the "G" is drawn in the green part of the zone, while the red part of the zone contains no "G". This means that the zone in question is a zone in domain1, the red domain, but that domain2 has a zone that exactly matches the location and values of the zone in the red domain. The corresponding zone in domain2 is a ghost zone.

Example for writing a 3D, domain-decomposed rectilinear mesh without ghost zones.

```
/* Create each of the domain meshes. */
int dom = 0, xdom, ydom, zdom;
for(zdom = 0; zdom < NZDOMS; ++zdom)
    for(ydom = 0; ydom < NYDOMS; ++ydom)
        for(xdom = 0; xdom < NXDOMS; ++xdom, ++dom)
        {
            float xc[NX], yc[NY], zc[NZ];
            float *coords[] = {xc, yc, zc};
            int index = 0;
            float xstart, xend, ystart, yend, zstart, zend;
            int xzones, yzones, zzones, nzones;
```

(continues on next page)

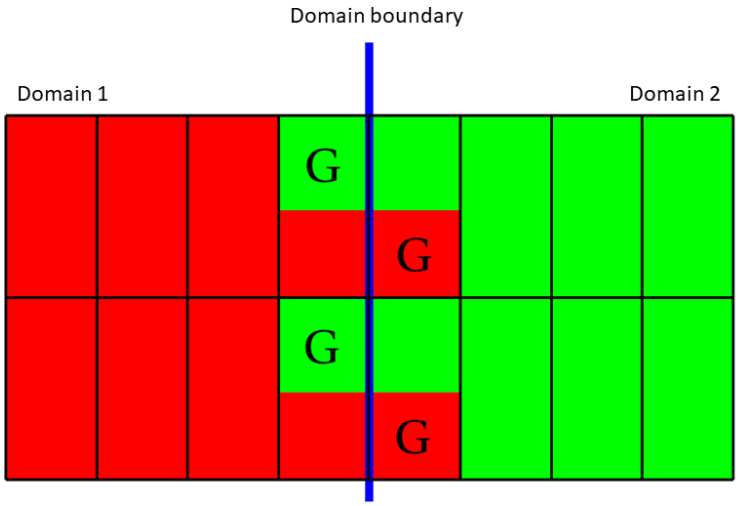


Fig. 8.20: The zone that are both read and gree are real zones in one domain and ghost zones in another. Ghost zones are designated with the label 'G'.

(continued from previous page)

```
            int xnodes, ynodes, znodes;
            /* Create a new directory. */
            char dirname[100];
            sprintf(dirname, "Domain%03d", dom);
            DBMkDir(dbfile, dirname);
            DBSetDir(dbfile, dirname);
            /* Determine default start, end coordinates */
            xstart = (float)xdom * XSIZE;
            xend = (float)(xdom+1) * XSIZE;
            xzones = NX-1;
            ystart = (float)ydom * YSIZE;
            yend = (float)(ydom+1) * YSIZE;
            yzones = NY-1;
            zstart = (float)zdom * ZSIZE;
            zend = (float)(zdom+1) * ZSIZE;
            zzones = NZ-1;
            xnodes = xzones + 1;
            ynodes = yzones + 1;
            znodes = zzones + 1;
            /* Create the mesh coordinates. */
            for(i = 0; i < xnodes; ++i
            {
                float t = (float)i / (float)(xnodes-1);
                xc[i] = (1.-t)*xstart + t*xend;
            }
            for(i = 0; i < ynodes; ++i)
            {
```

(continues on next page)

(continued from previous page)

```
            float t = (float)i / (float)(ynodes-1);
            yc[i] = (1.-t)*ystart + t*yend;
        }
        for(i = 0; i < znodes; ++i)
        {
            float t = (float)i / (float)(znodes-1);
            zc[i] = (1.-t)*zstart + t*zend;
        }
        /* Write a rectilinear mesh. */
        dims[0] = xnodes;
        dims[1] = ynodes;
        dims[2] = znodes;
        DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
                      DB_FLOAT, DB_COLLINEAR, NULL);
        /* Go back to the top directory. */
        DBSetDir(dbfile, "..");
    }
```

Once you have changed your mesh-writing code to add a layer of ghost zones, where appropriate, you must indicate that the extra layer of zones are ghost zones. If you use Silo's **DBPutQuadmesh** function to write your mesh, you can indicate which zones are ghost zones by adding **DBOPT_LO_OFFSET** and **DBOPT_HI_OFFSET** to pass arrays containing high and low zone index offsets in the option list. If you are adding ghost zones to an unstructured mesh, you would instead adjust the **lo_offset** and **hi_offset** arguments that you pass to the **DBPutZonelist2** function. The next code listing shows the additions made in order to support ghost zones in a domain-decomposed rectilinear mesh.

Example for writing a 3D, domain-decomposed rectilinear mesh with ghost zones.

```
/* Determine the size of a zone.*/
float cx, cy, cz;
cx = XSIZE / (float)(NX-1);
cy = YSIZE / (float)(NY-1);
cz = ZSIZE / (float)(NZ-1);
/* Create each of the domain meshes. */
int dom = 0, xdom, ydom, zdom;
for(zdom = 0; zdom < NZDOMS; ++zdom)
    for(ydom = 0; ydom < NYDOMS; ++ydom)
        for(xdom = 0; xdom < NXDOMS; ++xdom, ++dom)
        {
            float xc[NX], yc[NY], zc[NZ];
            float *coords[] = {xc, yc, zc};
            int index = 0;
            float xstart, xend, ystart, yend, zstart, zend;
            int xzones, yzones, zzones, nzones;
            int xnodes, ynodes, znodes;
            int hi_offset[3], lo_offset[3];
            DBoptlist *optlist = NULL;
            /* Create a new directory. */
            char dirname[100];
            sprintf(dirname, "Domain%03d", dom);
            DBMkDir(dbfile, dirname);
            DBSetDir(dbfile, dirname);
            /* Determine default start, end coordinates */
            xstart = (float)xdom * XSIZE;
            xend = (float)(xdom+1) * XSIZE;
            xzones = NX-1;
            ystart = (float)ydom * YSIZE;
```

(continues on next page)

(continued from previous page)

```
            yend = (float)(ydom+1) * YSIZE;
            yzones = NY-1;
            zstart = (float)zdom * ZSIZE;
            zend = (float)(zdom+1) * ZSIZE;
            zzones = NZ-1;
            /* Set the starting hi/lo offsets. */
            lo_offset[0] = 0;
            lo_offset[1] = 0;
            lo_offset[2] = 0;
            hi_offset[0] = 0;
            hi_offset[1] = 0;
            hi_offset[2] = 0;
            /* Adjust the start and end coordinates based on whether
             or not we have ghost zones. */
            if(xdom > 0)
            {
                xstart -= cx;
                lo_offset[0] = 1;
                ++xzones;
            }
            if(xdom < NXDOMS-1)
            {
                xend += cx;
                hi_offset[0] = 1;
                ++xzones;
            }
            if(ydom > 0)
            {
                ystart -= cy;
                lo_offset[1] = 1;
                ++yzones;
            }
            if(ydom < NYDOMS-1)
            {
                yend += cy;
                hi_offset[1] = 1;
                ++yzones;
            }
            if(zdom > 0)
            {
                zstart -= cz;
                lo_offset[2] = 1;
                ++zzones;
            }
            if(zdom < NZDOMS-1)
            {
                zend += cz;
                hi_offset[2] = 1;
                ++zzones;
            }
            xnodes = xzones + 1;
            ynodes = yzones + 1;
            znodes = zzones + 1;
            /* Create the mesh coordinates. */
            for(i = 0; i < xnodes; ++i)
            {
                float t = (float)i / (float)(xnodes-1);
```

(continues on next page)

(continued from previous page)

```
            xc[i] = (1.-t)*xstart + t*xend;
        }
        for(i = 0; i < ynodes; ++i)
        {
            float t = (float)i / (float)(ynodes-1);
            yc[i] = (1.-t)*ystart + t*yend;
        }
        for(i = 0; i < znodes; ++i)
        {
            float t = (float)i / (float)(znodes-1);
            zc[i] = (1.-t)*zstart + t*zend;
        }
        /* Write a rectilinear mesh. */
        dims[0] = xnodes;
        dims[1] = ynodes;
        dims[2] = znodes;
        optlist = DBMakeOptlist(2);
        DBAddOption(optlist, DBOPT_HI_OFFSET, (void *)hi_offset);
        DBAddOption(optlist, DBOPT_LO_OFFSET, (void *)lo_offset);
        DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
                      DB_FLOAT, DB_COLLINEAR, optlist);
        DBFreeOptlist(optlist);
        /* Go back to the top directory. */
        DBSetDir(dbfile, "..");
    }
```

There are two changes to the code in the previous listing that allow it to write ghost zones. First of all, the code calculates the size of a zone in the **cx**, **cy**, **cz** variables and then uses those sizes along with the location of the domain within the model to determine which domain surfaces will receive a layer of ghost zones. The layer of ghost zones is added by altering the start and end locations of the coordinate arrays as well as incrementing the number of zones and nodes in the dimensions that will have added ghost zones. The knowledge of which surfaces get a layer of ghost zones is recorded in the **lo_offset** and **hi_offset** arrays. By setting **lo_offset[0]** to 1, Silo knows that the first layer of zones in the X dimension will all be ghost zones. Similarly, by setting **high_offset[0]** to 1, Silo) knows that the last layer of zones in the X dimension are ghost zones. The **lo_offset** and **hi_offset** arrays are associated with the mesh by adding them to the option list that is passed to the **DBPutQuadmesh** function.

8.9.4 Materials

Many simulations use materials to define the composition of regions so the response of the materials can be taken into account during the calculation. Materials are represented as a list of integers with associated material names such as: "steel". Each zone in the mesh gets one or more material numbers to indicate its composition. When a zone has a single material number, it is said to be a "clean zone". When there is more than one material number in a zone, it is said to be a "mixed zone". When zones are mixed, they have a list of material numbers and a list of volume fractions (floating point numbers that sum to one) that indicate how much of each material is contained in a zone. VisIt provides the *Filled Boundary and Bounary plots* for plotting materials and VisIt provides the *Subset window* so you can selectively turn off certain materials.

The plot of the material object shown in Figure 8.21 and Figure 8.22 contains three materials: "Water" (1), "Membrane" (2), and "Air" (3). Materials use a **matlist** array to indicate which zones are clean and which are mixed. The **matlist** array is a zone-centered array of integers that contain the material numbers for the materials in the zone. If a zone has only one material then the **matlist** array entry for that zone will contain the material number of the material that fills the zone. If a zone contains more than one material then the **matlist** array entry for that zone will contain an index into the mixed material arrays. Indices into the mixed material arrays are equal to the negative value of the desired mixed material array entry. When creating your mixed material arrays, assume that array indices for the mixed

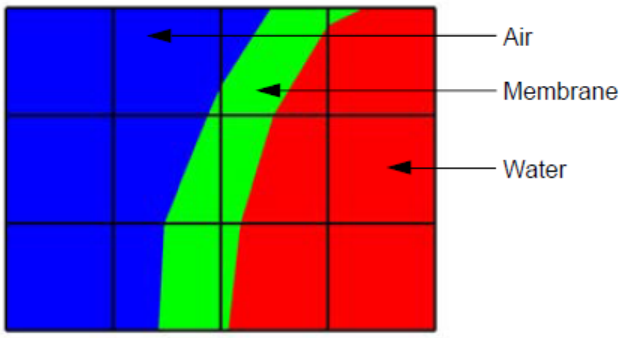


Fig. 8.21: A mesh with both clean and mixed material zones

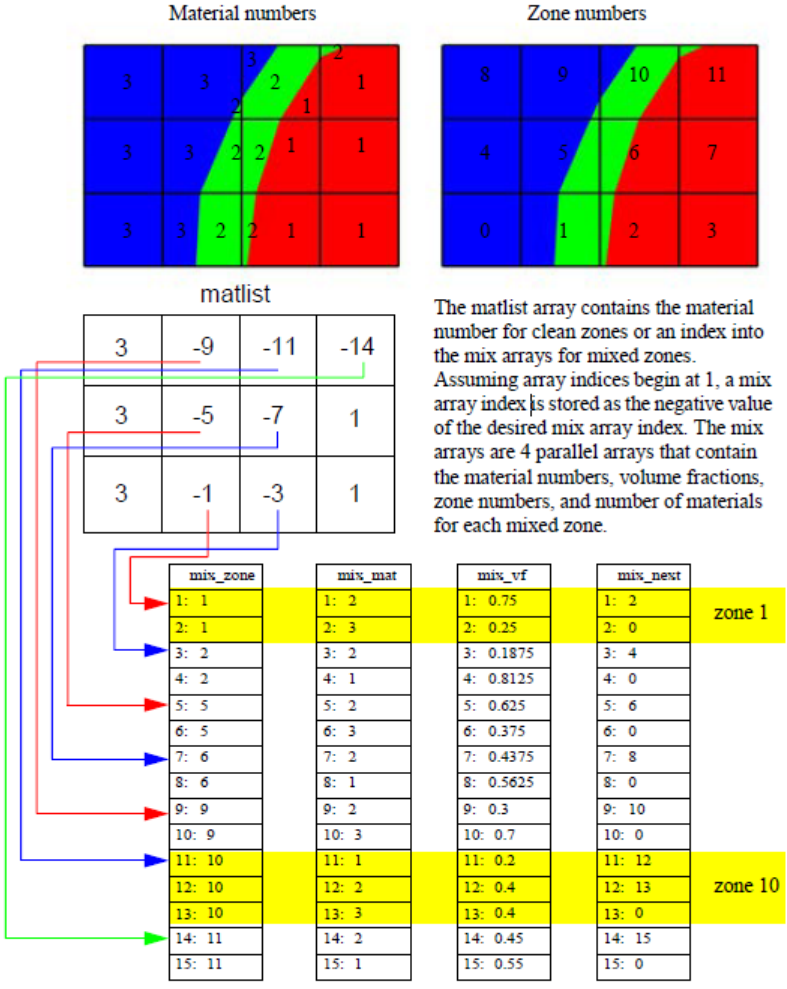


Fig. 8.22: Mixed material example

material arrays begin at 1. When you begin assigning material information into the mixed material arrays, use one array index per material in the mixed material zone. The index that you use for the beginning index for the next mixed material zone is the current index minus the number of materials in the current zone. Study the **matlist** array in Figure 8.22. The first mixed material zone is zone 1 and since it is mixed, instead of containing a material number, the **matlist** array for zone 1 contains the starting index into the mixed material arrays, or -1. If you negate the -1, you arrive at index 1, which is the starting index for zone 1 in the mixed material arrays. Since zone 1 will contain two materials, we use indices 1 and 2 in the mixed material arrays to store information for zone 1. The next available array for other zones wanting to add mixed materials to the mixed material arrays is element 3. Thus, when zone 2, which is also a mixed zone, needs to have its information added to the mixed material arrays, you store -3 into the **matlist** array to indicate that zone 2's values begin at zone 3 in the mixed material arrays.

The mixed material arrays are a set of 4 parallel arrays: **mix_zone**, **mix_mat**, **mix_vf**, and **mix_next**. All of the arrays have the number of elements but that number varies depending on how many mixed zones there are in the material object. The **mix_zone** array contains the index of the zone that owns the material information for the current array element. That is, if you examine element 14 in the **mix_zone array**, you will know that element 14 in all of the mixed material arrays contain information about zone 11.

The **mix_mat** array contains the material numbers of the materials that occupy a zone. Material numbers correspond to the names of materials (e.g. 1 = Water) and should begin at 1 and increment from there. The range of material numbers used may contain gaps without causing any problems in VisIt. However, if you create databases that have many domains that vary over time, you will want to make sure that each domain has the same list of materials at every time step. It is not necessary to use a material number in the **matlist** array or in the mixed material arrays in order to include it in a material object. Look at element 11 in the **mix_mat** array in Figure 8.22. Element 11 contains material 1, element 12 contains material 2, and element 13 contains material 3. Since those three material numbers are supposed to all be present in zone 10, they are all added to the **mix_mat** array. The same array elements in the *mix_vf* array record the amount of each material in zone 10. The values in the *mix_vf* array for zone 10 are: 0.2, 0.4, 0.4 and those numbers mean that 20% of zone 10 is filled with material 1, 40% is filled with material 2, and 40% is filled with material 3. Note that all of the numbers for a zone in the **mix_vf** array must sum to 1., or 100%.

The **mix_next** array contains indices to the next element in the mixed material arrays that contains values for the mixed material zone under consideration. The **mix_next** array allows you to construct a linked-list of material numbers for a zone within the mixed material arrays. This means that the information for one zone's mixed materials could be scattered through the mixed material arrays but in practice the mixed material information for one zone is usually contiguous within the mixed material arrays. The **mix_next** array contains the next index to use within the mixed material arrays or it contains a zero to indicate that no more information for the zone is available.

To write materials to a Silo file, you use the **DBPutMaterial** function. The **DBPutMaterial** function is covered in the Silo Manual but it is worth noting here that it can be called to write either mixed materials or clean materials. The examples so far have illustrated the more complex case of writing out mixed materials. You can pass the **matlist** array and the mixed material arrays to the **DBPutMaterial** function or, in the case of writing clean materials, you can pass only the **matlist** array and **NULL** for all of the mixed material arrays. Note that when you write clean materials, your **matlist** array will contain only the numbers of valid materials. That is, the **matlist** array does not contain any negative mixed material array indices when you write out clean material objects.

Example for writing mixed materials using Silo.

C

```
/* Material arrays */
int nmats = 2, mdims[2];
int matnos[] = {1,2,3};
char *matnames[] = {"Water", "Membrane", "Air"};
int matlist[] = {
    3, -1, -3, 1,
    3, -5, -7, 1,
    3, -9, -11, -14
};
```

(continues on next page)

(continued from previous page)

```
float mix_vf[] = {
    0.75,0.25, 0.1875,0.8125,
    0.625,0.375, 0.4375,0.56250,
    0.3,0.7, 0.2,0.4,0.4, 0.45,0.55
};
int mix_zone[] = {
    1,1, 2,2,
    5,5, 6,6,
    9,9, 10,10,10, 11,11
};
int mix_mat[] = {
    2,3, 2,1,
    2,3, 2,1,
    2,3, 1,2,3, 2,1
};
int mix_next[] = {
    2,0, 4,0,
    6,0, 8,0,
    10,0, 12,13,0, 15,0
};
int mixlen = 15;
/* Write out the material */
mdims[0] = NX-1;
mdims[1] = NY-1;
optlist = DBMakeOptlist(1);
DBAddOption(optlist, DBOPT_MATNAMES, matnames);
DBPutMaterial(dbfile, "mat", "quadmesh", nmats, matnos, matlist,
              mdims, ndims, mix_next, mix_mat, mix_zone, mix_vf, mixlen,
              DB_FLOAT, optlist);
DBFreeOptlist(optlist);
```

FortranFixed

```
    subroutine write_mixedmaterial(dbfile)
    implicit none
    integer dbfile
    include "silo.inc"
    integer NX, NY
    parameter (NX = 5)
    parameter (NY = 4)
    integer err, ierr, optlist, ndims, nmats, mixlen
    integer mdims(2) /NX-1, NY-1/
    integer matnos(3) /1,2,3/
    integer matlist(12) /3, -1, -3, 1,
    .                    3, -5, -7, 1,
    .                    3, -9, -11, -14/
    real mix_vf(15) /0.75,0.25, 0.1875,0.8125,
    .                0.625,0.375, 0.4375,0.56250,
    .                0.3,0.7, 0.2,0.4,0.4, 0.45,0.55/
    integer mix_zone(15) /1,1, 2,2,
    .                     5,5, 6,6,
    .                     9,9, 10,10,10, 11,11/
    integer mix_mat(15) /2,3, 2,1,
    .                    2,3, 2,1,
    .                    2,3, 1,2,3, 2,1/
    integer mix_next(15) /2,0, 4,0,
    .                     6,0, 8,0,
```

(continues on next page)

(continued from previous page)

```
     .                       10,0, 12,13,0, 15,0/
     ndims = 2
     nmats = 3
     mixlen = 15
c Write out the material
     err = dbputmat(dbfile, "mat", 3, "quadmesh", 8, nmats, matnos,
     .               matlist, mdims, ndims, mix_next, mix_mat, mix_zone, mix_vf,
     .               mixlen, DB_FLOAT, DB_F77NULL, ierr)
     end
```

8.10 Creating a Database reader plug-in

This section shows how to extend VisIt by writing a new database reader plug-in so you can use VisIt to access data files that you have already generated. Writing a database reader plug-in has several advantages over other approaches to importing data into VisIt such as writing a conversion program. First of all, if VisIt can natively read your file format then there is no need to convert files and consume extra disk space. Converting files may not even be possible if the data files are prohibitively large. Secondly, plug-ins offer the advantage of not having to alter a complex simulation code to write out data that VisIt can read. New plug-ins are free to read the simulation code's native file format. While many approaches to importing data into VisIt require new specialized code, when you write a database plug-in, the code that you write is external to your simulation and it is not a convertor that you have to maintain. There is no doubt that there is some maintenance involved in writing a database reader plug-in for VisIt but there is always the option of contributing your plug-in back into the VisIt source code tree where the code maintenance burden is shared among the developer community.

8.10.1 Structure of VisIt

VisIt is a parallel, distributed application that consists of four component processes that work in tandem to produce your visualizations. The two components that you may already be familiar with are the client and the viewer. VisIt has GUI, Python interface, and Java clients that control the visualization operations performed by the viewer, which is the central state repository and graphics rendering component. The other components, which are not immediately visible, are the database server and the compute engine. The database server (sometimes called the meta-data server) is responsible for browsing the file system and letting you know which files can be opened. Once you decide on a file to open, the database server attempts to open that file, loading an appropriate database reader plug-in to do so. Once the database server has opened a file, it sends file metadata such as the list of available variables to the client and the viewer. The compute engine comes into play when you want to create a plot to process your data into a form that can be rendered on the screen. The compute engine, like the database server, loads a plug-in to read a data file and does the actual work of reading the problem-sized data from the file and translating it into Visualization Toolkit (VTK) objects that VisIt can process. Once the data has been read, it is fed through the visualization pipeline and returned to the viewer component where it can be displayed.

Plug-ins

VisIt supports three types of plug-ins: plot plug-ins, operator plug-ins, and database reader plug-ins. This chapter explores database reader plug-ins as a method of importing data from new file formats into VisIt. A database reader plug-in is made of three shared libraries, which are dynamically loaded by the appropriate VisIt components when data from a file must be read. The VisIt components involved in reading data from a file are the database server and the compute engine. Each database reader plug-in has a database server component, a compute engine component, and an independent component, for a total of three shared libraries (*libM*, *libE*, *libI*).

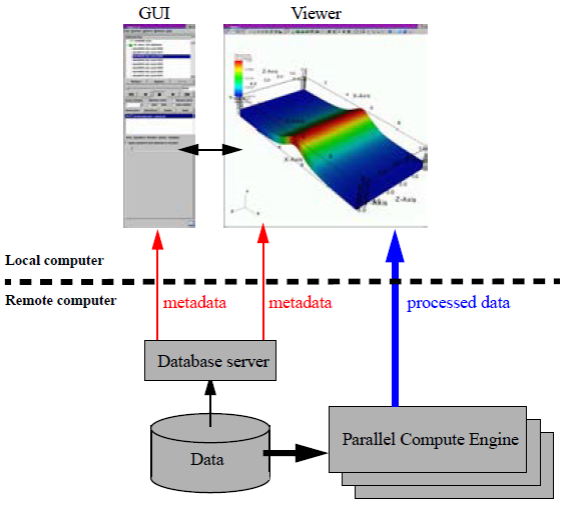


Fig. 8.23: VisIt's architecture

The independent plug-in component, or *libI* plug-in component, is a very lightweight shared library containing little more than the name and version of a plug-in as well as the file extensions that should be associated with it. When the database server and compute engine initialize at runtime, one of their first actions is to scan VisIt's plug-in directories for available *libI* plug-ins and then load all of the *libI* plug-ins to assemble an internal list of known plug-ins along with the table of file extensions for each file.

When VisIt needs to open a file, the filename is first passed to the database server, which tries to extract a file extension from the end of the filename so an appropriate plug-in can be selected from the list of available plug-ins. Once one or more matches are made, the database factory object in the database server loads the *libM* plug-in component for the first plug-in in the list of matching plug-ins. The *libM* plug-in component is the piece of the plug-in used by the database server and it is used to read the metadata from the file in question. If the plug-in cannot open the file then it should throw an exception to make the database factory attempt to open the file using the next matching plug-in. If there are no plug-ins that match the file's file extension then a default database plug-in is used. If that plug-in cannot open the file then VisIt issues an error message. Once the *libM* plug-in has read the metadata from the file, that information is sent to the VisIt clients where it can be used to populate variable menus, etc.

When you add a plot in VisIt and click the **Draw** button, the first step that the compute engine takes to process your request is to open the file that contains the data. The procedure for opening the file that contains the data in the compute engine is the same as that for the database server. In fact, the same database factory code is used internally. However, the database factory in the compute engine loads the *libE* plug-in component. The *libE* and *libM* plug-in components are essentially the same except that, when possible, database server plug-in components do less work. Both the *libE* and *libM* plug-in components contain code to read a file's metadata and both contain code to read variables and create meshes. The difference between the two plug-in types is that the code to read the variables and create meshes is only called from the *libE* plug-in component.

8.10.2 Picking a database reader plug-in interface

Database reader plug-ins have 4 possible interfaces, which affect how files are mapped to plug-in file format objects. The 4 possible interfaces are shown in the table below:

	SD	MD
ST	STSD - Single time state per file and it contains just 1 domain.	STMD - Single time state per file but each file contains multiple domains.
MT	MTSD - Multiple time states per file and each file conatins just 1 domain.	MTMD - Multiple time states per file and each file contains multiple domains.

In order to pick which plug-in interface is most appropriate for your particular file format, you must consider how your file format treats time and domains. If your file format contains multiple time states in each file then you have an *MT* file format; otherwise you have an *ST* file format. If your file format comes from a parallel simulation then you will often have some type of domain decomposition, which breaks up the entire simulation into smaller pieces called domains that are divided among processors. If your simulation has domains and the domains are written to a single file then you have an *MD* file format; otherwise, if your simulation processors wrote out their own files then you have an *SD* file format. When you consider both how your file format deals with time and how it deals with domains, you should be able to select which plug-in interface you will need when you write your database reader plug-in.

8.10.3 Using XMLEdit

Once you pick which database interface you will use to write your database plug-in, the next step is to use VisIt's XMLEdit tool to get started with some interface definitions. XMLEdit is a graphical application that lets you create an XML file that describes some of the basic attributes for your database reader plug-in. The XML file contains information such as the name of the plug-in, its version, which interface is used, the plug-in's list of file extensions, and any additional libraries or source code files that need to be included in the plug-in in order to build it.

To get started with building your plug-in, the first step is to create a source code directory to contain all of the files that will be created to generate your plug-in. It is best that the directory name be the name of your file format or the name of your simulation. Once you have created a directory for your plug-in files, you can run VisIt's XMLEdit program. To start XMLEdit on UNIX systems where VisIt is installed, open a command window and type `xmledit`. On Windows systems, XMLEdit should be available in the **Start** menu under VisIt's plug-in development options. Or, run from a command prompt: `C:\path\to\visit\xmledit`, substituting the correct path to the location where VisIt is installed.

Once XMLEdit is active you can see that it has a number of tabs that are devoted to various aspects of plug-in development. Most of the tabs are used for developing plot and operator plug-ins only so this section will focus on the actions that you need to take to create your database reader plug-in. First of all, you must type the name of your plug-in into the **Name** text field. The name should match the name of the source code directory that you created - be sure that you pick a name that can be used inside of C++ class names since the name is used to help generate the plug-in code skeleton that will form the basis of your database reader plug-in. Next, type in a label into the **Label** text field. The label for a database plug-in can contain a longer identifier that will be displayed when VisIt uses your plug-in to read files. The label may contain spaces and punctuation. Next, enter the version of your plug-in into the **Version** text field. The version for initial development should be *1.0*. Now, choose *Database* from the **Plugin type** combo box to tell XMLEdit that you want to build a database reader plug-in. Once you choose *Database* for your plug-in type, some additional options will become enabled. You can ignore these options for now since they contain reasonable default values.

The next step in creating your database plug-in using XMLEdit is to set the database type to either *STSD*, *STMD*, *MTSD*, or *MTMD* by selecting one of those options from the **Database type** combo box. Note that it is possible to instead choose to create a fully custom database type but do not choose that option since most formats do not need that level of customizeability. Once you have selected a database type for your plug-in, type in the list of file formats that you want to associate with your plug-in. You can enter as many spacedelimited file extensions as you want.

The information that you entered is the minimum amount of information required to create your database reader plug-in. Save your XMLEdit session to an XML file by selecting **Save** from the **File** menu. Be sure to use the same name as you used for the directory name that will contain your plug-in files and also be sure to save your XML file to that

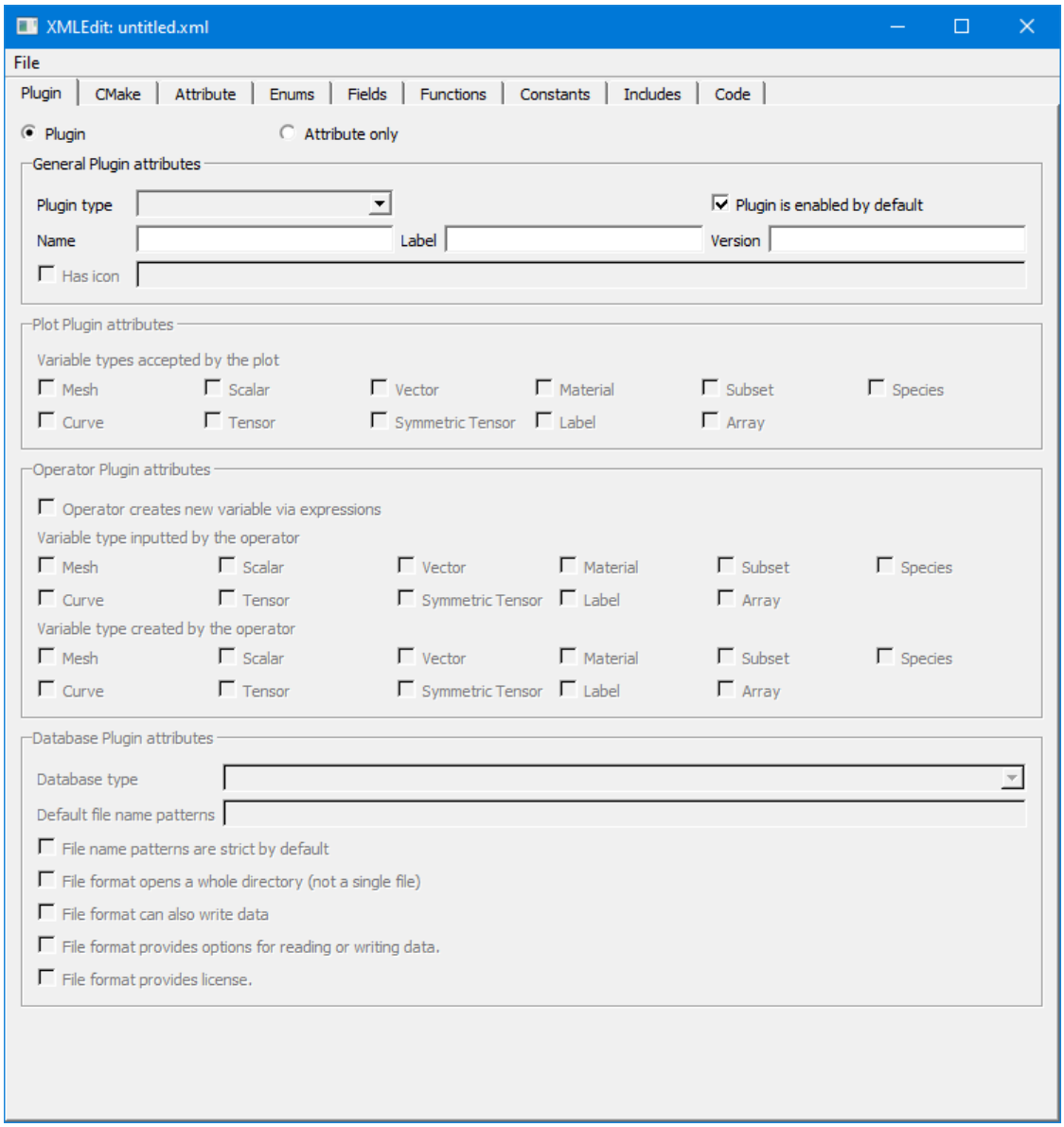


Fig. 8.24: XMLEdit plug-in tab

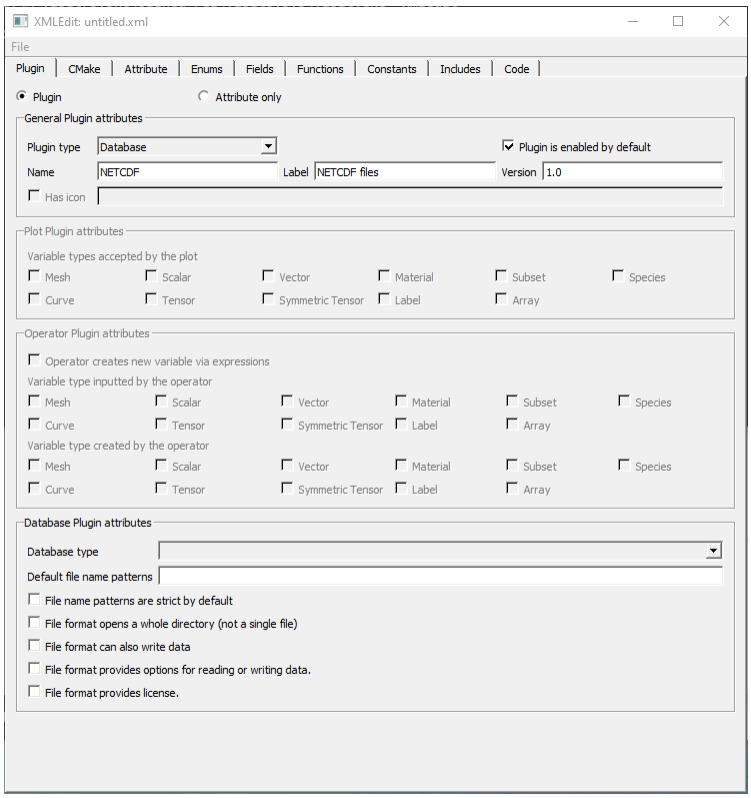


Fig. 8.25: XMLEdit plug-in tab after filling in **General Plugin Attributes**

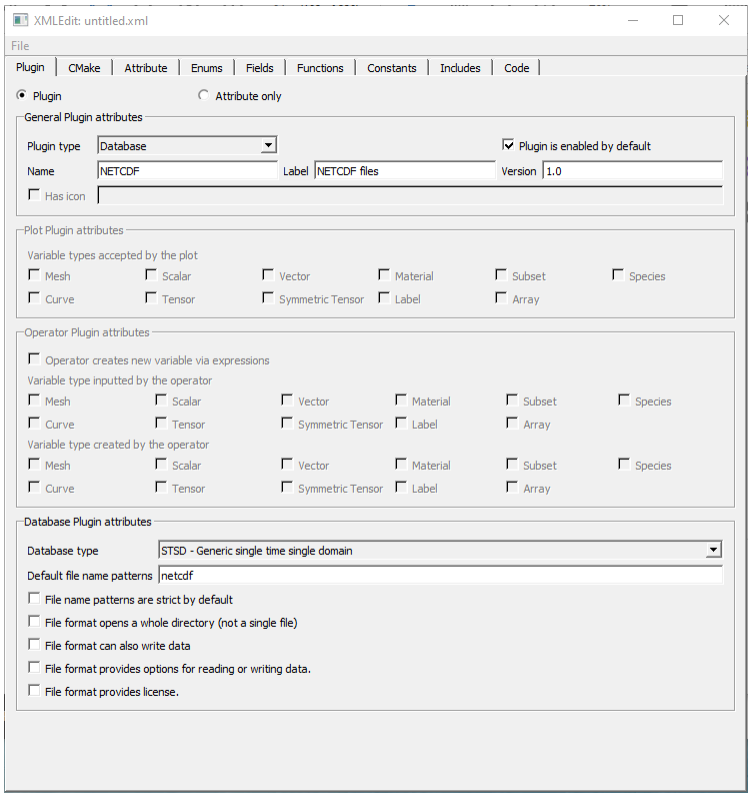


Fig. 8.26: XMLEdit plug-in tab after choosing **Database Type** and filling in **Default file name patterns**

directory. At this point, you can skip ahead to generating your plug-in code skeleton or you can continue adding options to your XML file.

CMake options

VisIt uses cmake for its build system and for the build systems of its plugins. XMLEdit contains controls on its CMake tab that allow you to add options to your XML file that will influence how your plug-in code is built when you go to compile it. For example, the **CMake** tab includes options that allow you to specify compiler options such as **CXXFLAGS**, **LDFLAGS** and **LIBS**.

Adding options to these fields can be particularly useful if your plug-in uses an external library such as NetCDF or HDF5. If you are using a library that VisIt provides (NetCDF, HDF5, CGNS, Silo, etc.) then you can use special predefined cmake variables that VisIt's build defines to locate those libraries. For example, you could use ${NETCDF_INCLUDE_DIR}, ${NETCDF_LIBRARY_DIR}, ${NETCDF_LIB} to reference the include directory, library directory, and library name for the NetCDF library. Just substitute another capitalized library name for NetCDF to use variables for other I/O libraries. It is better to use these cmake variables for libraries that VisIt provides to ensure that your plugin is linked against the right libraries.

If you are using a library that VisIt does not support, you can add the include file and library file locations to ensure that the compiler will know where to look for your external library when your plug-in is built. Be sure to use `-I/path/to/include` in the **CXXFLAGS** when you want to add include directories for your plugin. Use `-L/path/to/lib` in the **LDFLAGS** when you want to add link directories for your plugin. Finally, add the name of the library (e.g. `netcdf` instead of `-lnetcdf`) in the **LIBS** when you need to link against additional libraries. **LIBS** may contain the full-path to a library, in which case use of **LDFLAGS** to locate the library is unnecessary.

You can also add extra files to the *libE* and *libM* plug-ins by adding a list of files to the **Engine files** and **MDServer files** text fields, respectively. There are also rarely needed MDServer-specific or Engine-specific *defines*, *cxxflags*, or *ldflags*. The engine options are further broken down into serial and parallel versions.

If you change any of these options, shown in Figure 8.27, be sure to save your XML file before quitting XMLEdit.

8.10.4 Generating a plug-in code skeleton

Once you save your work from XMLEdit, you will find an XML file containing the options that you provided in the directory where you store your plug-in files. VisIt provides more XML tools to generate the necessary code skeleton for your plug-in. The important tools when building a database plug-in are: `xml2cmake`, `xml2info` and `xml2plugin`. The `xml2plugin` program is actually a script that automates calling the required xml2* programs. In order to generate your plug-in code skeleton, open a command window, go to the directory containing your XML file, and run `xml2plugin`. The command that you will run is:

```
xml2plugin -clobber FILE.xml
```

Be sure to replace `FILE.xml` with the name of your own XML file. Once you run the `xml2plugin` program, if you look in your directory, you will see several new files.

You can also generate the plugin skeleton code from XMLEdit by Choosing **Generate code** from the **File menu**. You can check all the options, the ones that don't apply to database plugins (C++, Java, Python, Window) will be ignored.

For database reader plug-ins, there are essentially three classes of files that `xml2plugin` creates. First of all, `xml2plugin` creates the plug-in code skeleton, which includes the plug-in entry points that are used to load the plug-in dynamically at runtime. These files have *Info* in their name and they are generated by the `xml2info` program. If you change the name, version, or file extensions that your plug-in uses then you should re-run `xml2info` instead of running `xml2plugin`. The next set of files are the AVT file format source and header files. The AVT file format source code files are C++ source code files that you will complete using new code to read your file format. Finally, `xml2cmake` created *CMakeLists.txt* file that cmake can use to generate a build system for your plug-in. If you run `cmake .` at the command prompt and you are on a UNIX system such as Linux or MacOS X, cmake will

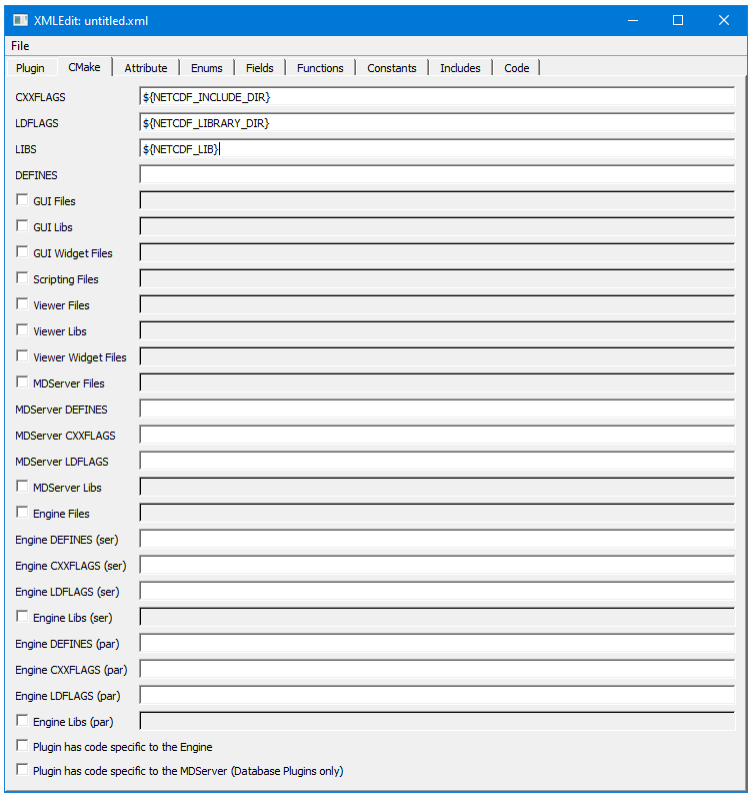


Fig. 8.27: XMLEdit CMake tab

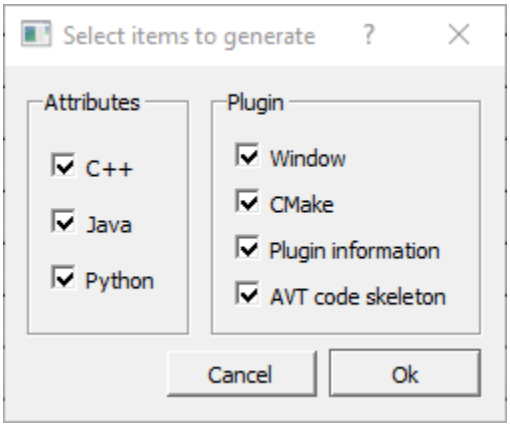


Fig. 8.28: Generate code options

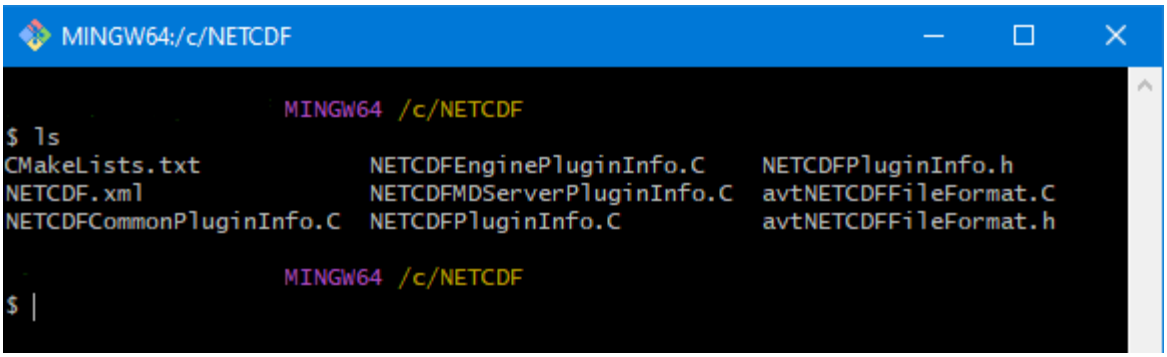


Fig. 8.29: Listing of files after running xml2plugin.

generate a Makefile for your plug-in. In that case, all you have to do in order to build your plug-in is type: `make` at the command prompt.

8.10.5 Building your plug-in

So far, we have created an XML file using the XMLEdit program and then used the XML file with VisIt's XML tools to generate plug-in source code. The static portions of the generated source code is complete but there are still some pieces that you need to write yourself in order to make VisIt read your data files. The automatically generated files that are called *avtXXXXFileFormat.C* and *avtXXXXFileFormat.h*, where XXXX is the name of your plug-in, are incomplete. These two AVT files contain a derived class of one of the *STSD*, *STMD*, *MTSD*, *MTMD* file format classes that VisIt provides for reading different file types. Your job is to fill in the missing code in the methods for the AVT classes so they can read data from your file format and translate that data into VTK objects. By default, the AVT files contain some messages in the source code like *YOU MUST IMPLEMENT THIS* , which are meant to prevent the source code from compiling and to call attention to areas of the plug-in that you need to implement. An example of this message is shown in Figure **??**.

The first step in building a plug-in is to make sure that the automatically generated source code compiles. Open the AVT files and look for instances of the *YOU MUST IMPLEMENT THIS* message and, when you find them, write down a note of where they appear. Comment out each of the messages in the C++ source code and add `return 0;` statements (See Figure 4-8). By commenting out the offending messages, the automatically generated source code will compile when you attempt to compile the plug-in. You will also have a list of some of the plug-in methods that you will have to write later when you really begin developing your plug-in.

Fig. 8.30: Commented-out *IMPLEMENT THIS* message

Once you have changed the AVT files so there are no stray messages about implementing a plug-in feature, go back to your command terminal and type `cmake -DCMAKE_BUILD_TYPE:STRING=Debug` so cmake will generate a build system for your plug-in. The generated build system is most commonly a Makefile, allowing you to use the `make` command for your system. The `make` command takes the automatically generated Makefile that was generated by cmake and starts building your plug-in against the installed version of VisIt.

For Windows OS and Visual Studio, you will need to tell CMake which generator and toolset to use, and should be the same as that used to compile VisIt itself. The cmake-gui makes this easy. See *Configuring With CMake GUI* for

more information. Your entry for **Where is the source code** will be the same directory as your .xml file. You entry for **Where to build the binaries** can be the same as source, but choosing a separate build folder is the better option, as it won't clutter your source folder with build files.

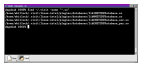
If you encounter compilation errors, such as syntax errors, then you most likely need to make further changes to your AVT files before trying to build your plug-in. A good C++ language reference can help you understand the types of errors that may be printed to your command window in the event that you have not successfully changed the AVT files. If your source code seems to compile but fails due to missing libraries such as NetCDF or HDF5 then you can edit your XML file so it points to the right library installation locations. Note that if you edit your XML file, you will need to regenerate the *CMakeLists.txt* file using `xml2cmake`. It is also a good idea that you remove the CMakeCache.txt file before rerunning cmake if you have changed the path to any libraries in your XML file.

Once your plug-in is built, it will be stored in a platform-specific subdirectory of the *.visit* directory in your home directory (`~/.visit`). If you type: `find ~/.visit -name "*.so"` into your command window, you will be able to locate the *libE*, *libI*, and *libM* files that make up your compiled plug-in (see Figure **??**).

If you develop for MacOS X, you should substitute `*.dylib` for `*.so` in the previous command because shared libraries on MacOS X have a *.dylib* file extension instead of a *.so* file extension.

If you develop on Windows, the files will be in your profile directory (generally `C:\users\<yourname>`) in a `VisIt` folder. The file extension is `.dll`.

Note that when a parallel compute engine is available in the installed version of VisIt, you will get two *libE* plug-ins; one with a *_ser* suffix and one with a *_par* suffix. The *libE* files that have a *_ser* suffix are loaded by the serial compute engine and the *_par libE* file is loaded by the parallel compute engine and may contain parallel function calls, such as calls to the MPI library.



When VisIt's database server and compute engine execute, they look in your *~/.visit* directory for available plug-ins and load any that are available. This means that even if you build plug-ins against the installed version of VisIt, it will still be able to find your private plug-ins.

It is recommended that while you develop your plug-ins, you only install them in your *~/.visit* directory so other VisIt users will not be affected. However, if you develop your plug-in on MacOS X, you will have to make sure that your plug-ins are installed publicly so that they can be loaded at runtime. You can also choose to install your plug-ins publicly once you have completed development. To install plug-ins publicly, first remove the files that were installed to your *~/.visit* directory by typing the `make clean` command in your command window. Next, re-run the `xml2cmake` program like this:

```
xml2cmake -public -clobber FILE.xml
```

Adding the `-public` argument on the command line causes make to install your plug-in files publicly so all VisIt users can access them. Don't forget to rerun `cmake` and `make` after running `xml2cmake`.

8.10.6 Calling your plug-in for the first time

Once you have completed building your plug-in for the first time, all that you need to do is run VisIt and try to open one of your files. When you open one of your files, the database server should match the file extension of the file that you tried to open with the list of file extensions that your plug-in accepts, causing your plug-in to be loaded and used for opening the file. You can verify that VisIt used your plug-in by opening the **File Information** window (see Figure **??**) in the VisIt GUI and looking for the name of your plug-in in the listed information.

If your plug-in wasn't used by VisIt, it may mean that other formats can read the same extensions as your plugin. In that case, you would need to select your plugin from the **Open file as type:** dropdown option in the **File open** window to make VisIt choose your plugin.

Note that at this stage, the database server should be properly loading your database reader plug-in but since no code to actually read your files has yet been added to the AVT source code files, no plottable meshes or variables will be available.

8.10.7 Implementing your plug-in

Now that you have built a working plug-in framework, you are ready to begin adding code to your plug-in that will make it capable of opening your file format, reading data, and translating that data into VTK objects. This section explores the details of writing the AVT code for your database reader plug-in, providing necessary background and then diving into specific topics such as how to return data for a particular mesh type. Before starting, remember that building a plug-in is an incremental process and you should proceed in small steps, saving your work, building, and testing your plug-in each step of the way.

Required plug-in methods

Most of the code in a VisIt database plug-in is automatically generated and, for the most part, the only code that you need to modify is the AVT code. The AVT code contains a class definition and implementation for a derived type of the *STSD*, *STMD*, *MTSD*, or *MTMD* file format classes and your job as a plug-in developer is to write the required methods for your derived file format class so that VisIt can read your file. There are many methods in the file format class interface that you can override to make your plug-in perform specialized operations. The only methods that you absolutely must implement are:

PopulateDatabaseMetaData VisIt calls the *PopulateDatabaseMetaData* method when file metadata is needed. File metadata is returned in a pass-by-reference *avtDatabaseMetaData* object. File metadata consists of the list of names of meshes, scalar variables, vector variables, tensor variables, label variables, array variables, expressions, cycles, and times contained in the file. These lists of variables and meshes let VisIt know the names of the objects that can be plotted from your file. The metadata is used primarily to populate the plot menus in the GUI and viewer components. The *PopulateDatabaseMetaData* method is called by both the *libM* and *libE* plugins.

GetMesh VisIt calls the *GetMesh* method in a *libE* plug-in when it needs to plot a mesh. This method is the first method to return "problem-sized" data, meaning that the mesh data can be as large as the data in your file. The *GetMesh* method must return a mesh object in the form of one of the VTK dataset objects (*vtkRectilinearGrid*, *vtkStructuredGrid*, *vtkUnstructuredGrid*, *vtkPolyData*).

GetVar VisIt calls the *GetVar* method in a *libE* plug-in when it needs to read a scalar variable. Like the *GetMesh* method, this method returns "problem-sized" data. *GetVar* reads data values from the file format, possibly performing calculations to alter the data, and stores the data into a derived type *vtkDataArray* object such as *vtkFloatArray* or *vtkDoubleArray*. If your file format does not need to return scalar data then you can leave the `return 0;` implementation that you added in order to get your plug-in to build.

GetVectorVar VisIt calls the *GetVectorVar* method in a *libE* plug-in when it needs to read a vector or tensor variable. *GetVectorVar* performs the same function as *GetVar* but returns *vtkFloatArray* or *vtkDoubleArray* objects that have more than one value per tuple. A tuple is the equivalent of a value associated with a zone or node but it can store more than one value. If your file format does not need to return vector data then you can leave the `return 0;` implementation that you added in order to get your plug-in to build.

Debugging your plug-in

Before beginning to write code for your plug-in, you should know a few techniques for debugging your plug-in since debugging VisIt can be tricky because of its distributed architecture. See *Debugging Tips* for detailed information.

Opening your file

When VisIt receives a list of files to open, it tries to determine which plug-in should be loaded to access the data in those files. The match is performed by comparing the file extension of the files against the known file extensions or patterns for all database reader plug-ins. Each plug-in in the list of matches is loaded and VisIt creates instances of the plug-in's AVT file format classes that are then used to access the data in the files. If the plugin's file format classes can be successfully constructed then VisIt tries to get the file's metadata. It is very important that your file format's constructor do as little work as possible, and try at all costs to avoid opening the files. Remember, VisIt could be creating a long list of your file format objects and opening the file in the constructor will really slow down the process of opening a file. It is better to instead add a boolean *initialized* member to your class and an *Initialize* method that reads the file to check its contents. Then override the *ActivateTimestep* method for your file format class and call your *Initialize* method from it. We make *Initialize* its own method so we can call it from other methods such as *GetMesh* or *GetVar* just in case.

In the event that your Initialize method cannot open the file if the file is not the right type, or if it contains errors, or if it cannot be accessed for some other reason, the constructor must throw an *InvalidDBTypeException* exception. When the *InvalidDBTypeException* exception is thrown, VisIt's database factory catches the exception and then tries to open the file with the next matching plug-in. This procedure ccontinues until the file is opened by a suitable plug-in or the file cannot be opened at all.

Example for identifying a file.

```
// NOTE - This code is incomplete and is for example purposes only.

#include <InvalidDBTypeException.h>

avtXXXXFileFormat::avtXXXXFileFormat(const char *filename)
      : avtSTSDFileFormat(filename)
{
    initialized = false;
}
// Override this method in your reader
void
avtXXXXFileFormat::ActivateTimestep()
{
    Initialize();
}
// Provide this method in your reader
void
avtXXXXFileFormat::Initialize())
{
    if(!initialized)
    {
        bool okay = false;
        // Open the file specified by the filename argument here using
        // your file format API. See if the file has the right things in
        // it. If so, set okay to true.
        YOU MUST IMPLEMENT THIS
        // If your file format API could not open the file then throw
        // an exception.
        if (!okay)
        {
            EXCEPTION1(InvalidDBTypeException,
                       "The file could not be opened");
        }
        initialized = true;
    }
}
```

If your database reader plug-in uses a unique file extension then you have the option of deferring any file opens until later when metadata is required. This is the preferred approach because VisIt may create many instances of your file format class and doing less work in the constructor makes opening files faster.

Once you decide whether your file format can defer opening a file or whether it must open the file in the constructor, you can begin adding code to your AVT class. Since opening files can be a costly operation, you might want to open a file and keep it open if you have a random access file format. If you open a file in one method and want to keep the file open so it is available to multiple plug-in methods, you will need to add a new class member to your AVT class to contain the handle to your open file. If your file format consists of sequential text then you might consider reading the file once and keeping the data in memory in a format that you can conveniently translate into VTK objects. Both approaches require the addition of a new class member - either a handle to the file or a pointer to data that was read from the file.

Returning file metadata

Once your you have decided how your plug-in will manage access to the file that it must read, the next step in writing your database reader plug-in is to implement the *PopulateDatabaseMetaData* method. The *PopulateDatabaseMetaData* method is called by VisIt's database infrastructure when information about a file's meshes and variables must be obtained. The *PopulateDatabaseMetaData* method is usually called only the first time that a file format's metadata is being read, though some time-varying formats can have time-varying metadata, which requires that *PopulateDatabaseMetaData* is called each time VisIt requests data for a new time state. However, most file formats call *PopulateDatabaseMetaData* once.

The *PopulateDatabaseMetaData* method arguments can vary, depending on whether your file format is *STSD*, *STMD*, *MTSD*, or *MTMD* but in all cases the first argument is an *avtDatabaseMetaData* object. The *avtDatabaseMetaData* object is a class that is pervasively used in VisIt; it contains information about the files that you plot such as the number of domains, times, meshes, and variables that the files can provide. When you implement your plug-in's *PopulateDatabaseMetaData* method, you must populate the *avtDatabaseMetaData* object with the list of meshes and variables, etc. that you want VisIt to be able to plot. You can hard-code a fixed list of meshes and variables if your file format always contains the same entities or you can open your file and provide a dynamic list of meshes and variables. This section covers how to add meshes and various variable types to the *avtDatabaseMetaData* object so your file format's data will be exposed in VisIt. For a complete listing of the *avtDatabaseMetaData* object's methods, see the *avtDatabaseMetaData.h* header file. It is worth noting that the following code examples create metadata objects and manually add them to the metadata object instead of using convenience functions. This is done because the convenience functions used in automatically generated plug-in code do not provie support for less often used metadata settings such as units and labels.

Returning mesh metadata

In order for you to be able to plot any data from your file format, your database reader plug-in must add at least one mesh to the *avtDatabaseMetaData* object that is passed into the *PopulateDatabaseMetaData* method. Adding information about a mesh to the *avtDatabaseMetaData* object is done by creating an *avtMeshMetaData* object, populating its important members, and adding it to the *avtDatabaseMetaData*. At a minimum, each mesh must have a name, spatial dimension, topological dimension, and a mesh type. The mesh's name is the identifier that will be displayed in VisIt's plot menus and it is also the name that will be passed later on into the plug-in's *GetMesh* method.

The spatial dimension attribute corresponds to how many dimensions are needed to specify the coordinates for the points that make up your mesh. If your mesh exists in a 2D plane then choose 2, otherwise choose 3. Note that when you create the points for your mesh later in the *GetMesh* method, you will always create points that contain X,Y,Z points.

The topological dimension attribute describes the number of logical dimensions used by your mesh, regardless of the dimension of the space that it sits in. For example, you may have a planar surface of triangles sitting in 3D space. Such a mesh would be topologically 2D even though it sits in 3D space. The rule of thumb that VisIt follows is that if

your mesh's cells are points then you have a mesh that is topologically 0D, lines are 1D, surfaces are 2D, and volumes are 3D. This point is illustrated in Figure 8.31.

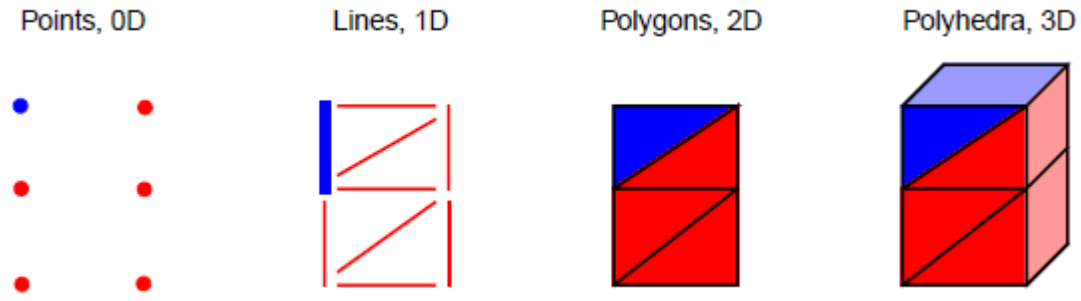


Fig. 8.31: Topological dimesnions. One zone is highlighted blue.

Once you have set the other basic attributes for your mesh object, consider which type of mesh you have. VisIt supports several different mesh types and the value that you provide in the metadata allows VisIt to tailor how it applies filters that process your data. If you have a mesh composed entirely of particles then choose *AVT_POINT_MESH*. If you have a structured mesh where the coordinates are specified by small vectors of values for each axis and the rest of the coordinates are implied then you probably have a rectilinear mesh and you should choose *AVT_RECTILINEAR_MESH*. If you have a structured mesh and every node has its own specific location in space then you probably have a curvilinear mesh and you should choose *AVT_CURVILINEAR_MESH*. If you have a mesh for which you specify a large list of nodes and then create cells using indices into that list of nodes then you probably have an unstructured mesh and you should choose *AVT_UNSTRUCTURED_MESH* for the mesh type. If you have a mesh that adaptively refines then choose *AVT_AMR_MESH*. Finally, if your mesh is specified using shapes such as cones and spheres that are unioned or differenced using boolean operations then you have a constructive solid geometry mesh and you should choose *AVT_CSG_MESH* for your mesh's mesh type.

If your mesh consists of multiple domains then you will need to set the number of domains into the *numBlocks* member of the *avtMeshMetaData* object. Remember that the number of domains tells VisIt how many pieces make up your mesh and it is especially important to specify this number if your plug-in is derived from an *MD* file format interface. You may also choose to tell VisIt what the domains are called for your file format. Some file formats use the word: "domains" while others use "brick" or "block". If you choose to set the name that VisIt uses for domains then that term will be used in parts of VisIt's GUI such as the **Subset** window. Set the *blockPieceName* member of the *avtMeshMetaData* object to a suitable term that describes a domain in the context of your simulation code. Alternatively, you can provide proper names by providing a vector of strings containing the names by setting the *blockNames* member.

Now that the most important attributes of the *avtMeshMetaData* object have been specified, you can add extra information such as the names or units of the coordinate dimensions. Once all attributes are set to your satisfaction, you must add the *avtMeshMetaData* object to the *avtDatabaseMetaData* object.

Example for returning mesh metadata.

```
// NOTE - This code is incomplete and is for example purposes only.
void
avtXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a point mesh to the metadata. Note that this example will
    // always expose a mesh called "particles" to VisIt. A real
    // plug-in may want to read a list of meshes from the data
    // file.
    avtMeshMetaData *mmd = new avtMeshMetaData;
```

(continues on next page)

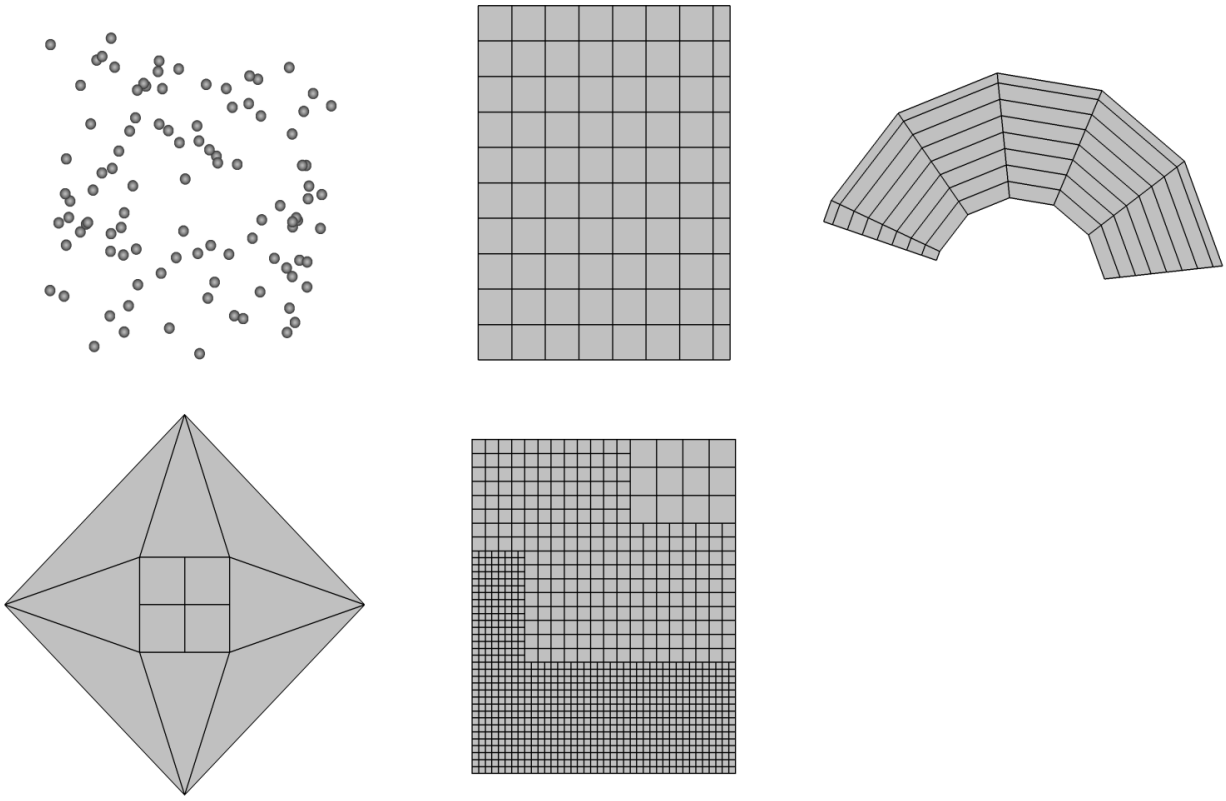


Fig. 8.32: AVT mesh types (AVT_CSG_MESH not pictured)

(continued from previous page)

```
    mmd->name = "particles";
    mmd->spatialDimension = 3;
    mmd->topologicalDimension = 0;
    mmd->meshType = AVT_POINT_MESH;
    mmd->numBlocks = 1;
    md->Add(mmd);
    // Add other objects to the metadata object.
}
```

Returning scalar metadata

Once you have exposed a mesh to VisIt by adding mesh metadata to the *avtDatabaseMetaData* object, you can add scalar field metadata. A scalar field is a set of floating point values defined for all cells or nodes of a mesh. You can expose as many scalar variables as you want on any number of meshes. The list of scalar fields that a plug-in exposes is often determined by the data file being processed. Like mesh metadata, scalar metadata requires a name so the scalar can be added to VisIt's menus. The name that you choose is the same name that later is passed to the *GetVar* plug-in method. Once you select a name for your scalar variable, you must indicate the name of the mesh on which the variable is defined by setting the *meshName* member of the *avtScalarMetaData* object. Once you have set the *name* and *meshName* members, you can set the *centering* member. The *centering* member of the *avtScalarMetaData* object can be set to *AVT_NODECENT* or *AVT_ZONECENT*, indicating that the data is defined on the nodes or at the zone centers, respectively. If you want to indicate units that are associated with the scalar variable, set the *hasUnits* member to *true* and set the *units* string to the appropriate unit names.

Example for returning scalar metadata.

```
// NOTE - This code is incomplete and is for example purposes only.
void
avtXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a mesh called "mesh" to the metadata object.
    // Add a scalar to the metadata. Note that this plug-in will
    // always expose a scalar called "temperature" to VisIt. A real
    // plug-in may want to read a list of scalars from the data
    // file.
    avtScalarMetaData *smd = new avtScalarMetaData;
    smd->name = "temperature";
    smd->meshName = "mesh";
    smd->centering = AVT_ZONECENT;
    smd->hasUnits = true;
    smd->units = "Celsius";
    md->Add(smd);
    // Add other objects to the metadata object.
}
```

Returning vector metadata

The procedure for returning vector metadata is similar to that for returning scalar metadata. In fact, if you change the object type that you create from *avtScalarMetaData* to *avtVectorMetaData* then you are almost done. After you set the basic vector metadata attributes, you must set the *varDim* member to *2* if you have a 2-component vector or *3* if you have a 3-component vector.

Example for returning vector metadata.

```
// NOTE - This code is incomplete and is for example purposes only.
void
avtXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a mesh called "mesh" to the metadata object.
    // Add a vector to the metadata. Note that this plug-in will
    // always expose a vector called "velocity" to VisIt. A real
    // plug-in may want to read a list of vectors from the data
    // file.
    avtVectorMetaData *vmd = new avtVectorMetaData;
    vmd->name = "velocity";
    vmd->meshName = "mesh";
    vmd->centering = AVT_ZONECENT;
    vmd->hasUnits = true;
    vmd->units = "m/s";
    vmd->varDim = 3;
    md->Add(vmd);
    // Add other objects to the metadata object.
}
```

Returning material metadata

Like the other types of mesh variables that we have seen so far, a material is defined on a specific mesh. However, unlike the other variables types, materials can be used to name regions of the mesh and can also be used by VisIt to break the mesh down into smaller pieces that can be turned on and off using the **Subset** window. Material metadata is stored in an *avtMaterialMetaData* object and it consists of: the name of the material object, the mesh on which it is defined, the number of materials, and the names of the materials. If you had a material called "mat1" defined on "mesh" and "mat1" was composed of: "Steel", "Wood", "Glue", and "Air" then the metadata object needed to expose "mat1" to VisIt would look like the following code listing:

Example for material mesh metadata.

```
// NOTE - This code is incomplete and is for example purposes only.
void
avtXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a mesh called "mesh" to the metadata object.
    // Add a material to the metadata. Note that this plug-in will
    // always expose a material called "mat1" to VisIt. A real
    // plug-in may want to use from the data file to construct
    // a material.
    avtMaterialMetaData *matmd = new avtMaterialMetaData;
    matmd->name = "mat1";
    matmd->meshName = "mesh";
    matmd->numMaterials = 4;
    matmd->materialNames.push_back("Steel");
    matmd->materialNames.push_back("Wood");
    matmd->materialNames.push_back("Glue");
    matmd->materialNames.push_back("Air");
    md->Add(matmd);
    // Add other objects to the metadata object.
}
```

Returning expressions

VisIt provides support for defining expressions to calculate new data based on the data in your file. VisIt provides the **Expression** window in the GUI for managing expression definitions. It can be convenient for users in certain fields, where custom expressions are used frequently, to store the expression definitions directly in the file format or to encode the custom expressions directly in the file metadata so they are always available when a given file is visualized. VisIt's *avtDatabaseMetaData* object can contain custom expressions. Thus you can add custom expressions to the *avtDatabaseMetaData* object inside of your database reader plug-in. Custom expressions are added to the *avtDatabaseMetaData* object by creating *Expression* (defined in Expression.h) objects and adding them by calling the *avtDatabaseMetaData::AddExpression* method. The *Expression* object lets you provide the name and definition of an expression as well as the expression's expected return type (scalar, vector, tensor, etc.) and whether the expression should be hidden from the user. Hidden expressions can be useful if you build a complex expression that makes use of smaller sub-expressions that do not need to be exposed in the VisIt user interface.

Example for returning expression metadata.

```
// NOTE - This code is incomplete and is for example purposes only.
#include <Expression.h>

void
avtXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a mesh called "mesh" to the metadata object.
    // Add scalars to the metadata object.
    // Add expression definitions to the metadata object.
    Expression *e0 = new Expression;
    e0->SetName("speed");
    e0->SetDefinition("{u,v,w}");
    e0->SetType(Expression::VectorMeshVar);
    e0->SetHidden(false);
    md->AddExpression(e0);
    Expression *e1 = new Expression;
    e1->SetName("density");
    e1->SetDefinition("mass/volume");
    e1->SetType(Expression::ScalarMeshVar);
    e1->SetHidden(false);
    md->AddExpression(e1);
    // Add other objects to the metadata object.
}
```

Returning a mesh

Once your database reader plug-in can successfully return metadata about one or more meshes, you can proceed to implementing your plug-in's *GetMesh* method. When you make a plot in VisIt, the plot is set up using the file metadata returned by your plug-in. When you click the **Draw** button in the VisIt GUI, it causes a series of requests that make the compute engine load your *libE* plug-in and call its *GetMesh* method with the name of the mesh being used by the plot as well as the time state and domain numbers (*MT* or *MD* formats only). A database reader plug-in's job is to read relevant data from a file format and translate the data into a VTK object that VisIt can process. The *GetMesh* method's job is to read the mesh information from the file and create a VTK object that describes the mesh in the data file. VisIt can process many different mesh types (See Figure 8.32) and you can return different types of VTK objects that best describe your mesh type. This section gives example code to show how you would take data read from your file format and turn it into VTK objects that describe your mesh. The details of reading data from your file format are omitted from the example code listings because those details change for each file format. The central message in this section is how to use data from a file format to construct different mesh types.

Determing which mesh to return

The *GetMesh* method is always passed a string containing the name of the mesh that should be returned from the plug-in. If your file format only ever has one mesh then you can ignore the meshname argument. However, if your file format can contain more than one mesh then you should check the name of the requested mesh before returning a VTK object so you create and return the correct mesh.

Example for which mesh to return in GetMesh

```
// NOTE - This code is incomplete and is for example purposes only.
#include <InvalidVariableException.h>
vtkDataSet *
avtXXXXFileFormat::GetMesh(const char *meshname)
{
    // Determine which mesh to return.
    if (strcmp(meshname, "mesh") == 0)
    {
        // Create a VTK object for "mesh"
        return mesh;
    }
    else if (strcmp(meshname, "mesh2") == 0)
    {
        // Create a VTK object for "mesh2"
        return mesh2;
    }
    else
    {
        // No mesh name that we recognize.
        EXCEPTION1(InvalidVariableException, meshname);
    }
    return 0;
}
```

If your database reader plug-in is derived from one of the *MT* or *MD* file format interfaces then the *GetMesh* method will have, in addition to the *meshname* argument, either a *timestate* argument, *domain* argument, or both. These extra arguments are both integers that VisIt passes to your plug-in so your plug-in can select the right mesh for the specified time state or domain. If your *GetMesh* method accepts a *timestate* argument then you can use it to return the mesh for the specified time state, which is in the range [0, NTS - 1], where NTS is the number of time states that your plug-in returned from its *GetNTimesteps* method. The range for the *domain* argument, if it is present, is [0,NDOMS - 1] where NDOMS is the number of domains that your file format added to the *numBlocks* member in the *avtMeshMetaData* object corresponding to the mesh named by the *meshname* argument.

Rectilinear Meshes

A rectilinear mesh is a 2D or 3D mesh where all coordinates are aligned with the axes. Each axis of the rectilinear mesh can have different, non-uniform spacing, allowing for details to be concentrated in certain regions of the mesh. Rectlinear meshes are specified by lists of coordinate values for each axis. Since the mesh is aligned to the axes, it is only necessary to specify one set of X, Y, and Z values to generate all of the coordinates for the entire mesh.

Once you read the X,Y, and Z coordinates from your data file, you can use them to assemble a *vtkRectilinearGrid* object. The procedure for creating a *vtkRectilinearGrid* object and returning it from *GetMesh* is shown in the next code listing. The capitalized portions of the code listing indicate incomplete code that you must replace with code to read values from your file format. The first such piece requires you to read the number of dimensions for your mesh from the file format and store the value into the *ndims* variable. Once you have done that, read the number of nodes in each of the X,Y,Z dimensions and store those values in the dims array. Finally, fill in the code for reading the X coordinate values into the xarray array and do the same for the Y and Z coordinate arrays. Once you have replaced

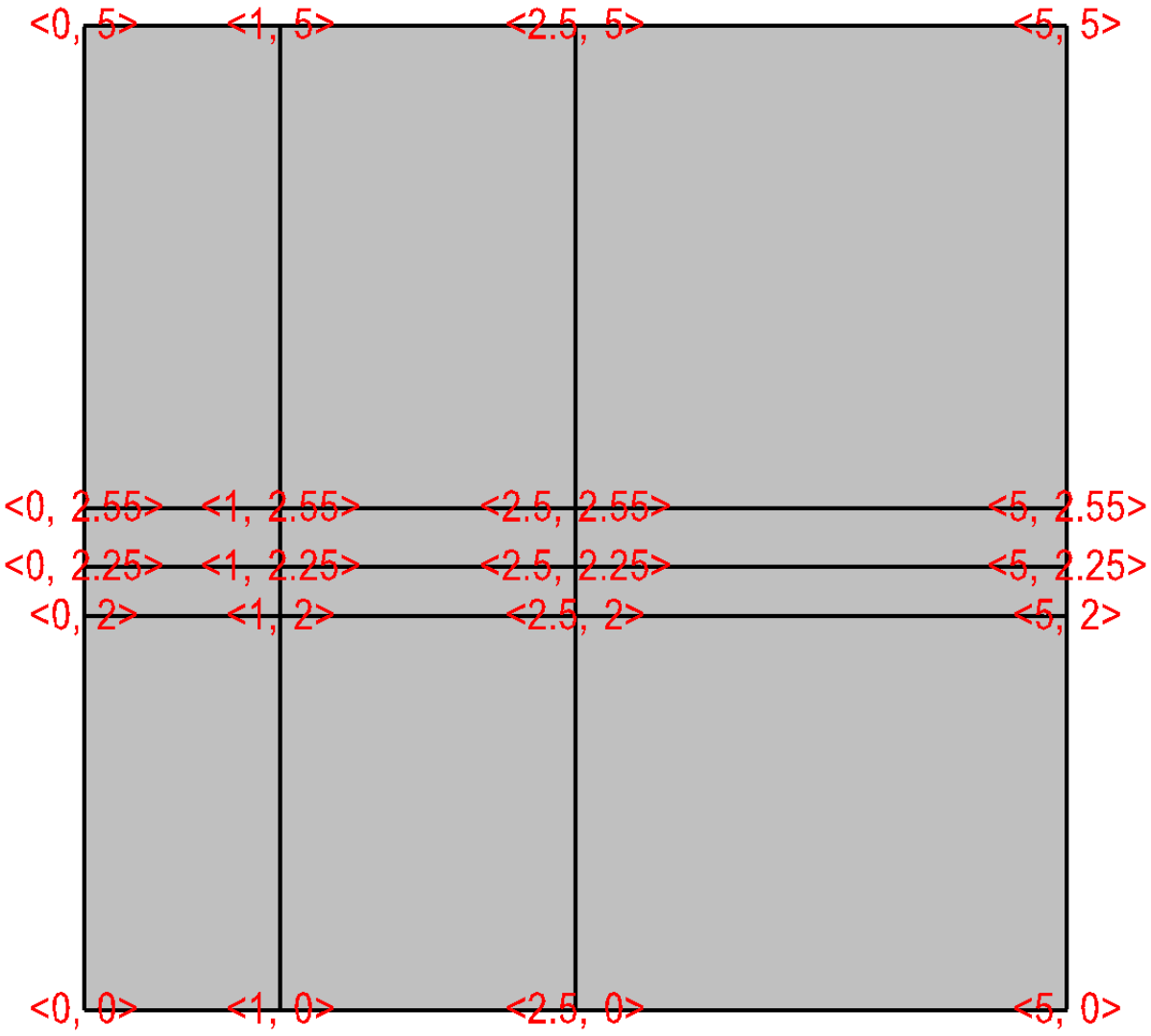


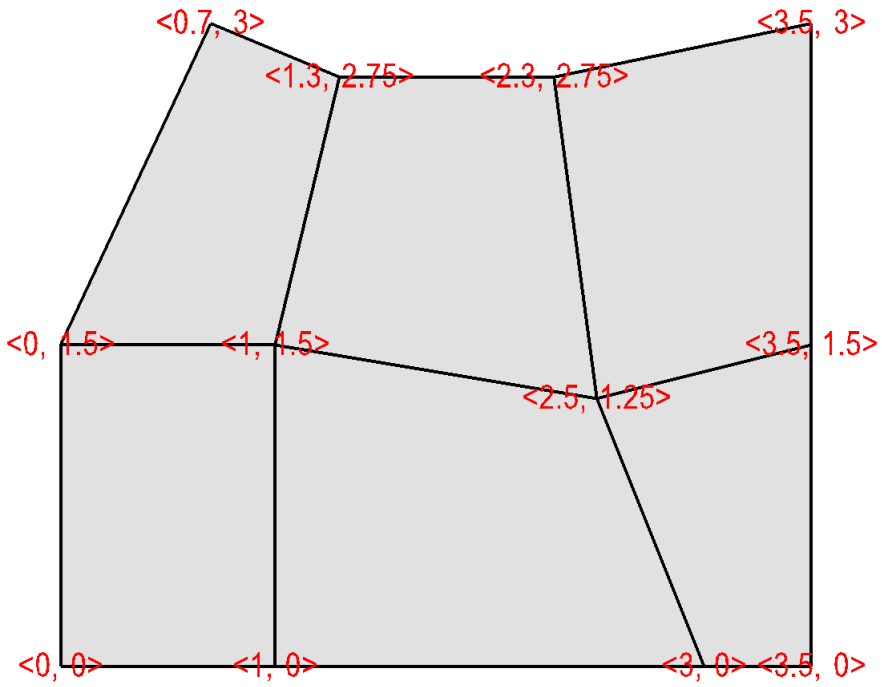
Fig. 8.33: Rectilinear mesh and its X,Y node coordinates.

the capitalized code portions with code that reads values from your file format, your plug-in should be able to return a valid *vtkRectilinearGrid* object once you rebuild it.

Example for creating vtkRectilinearGrid in GetMesh.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.
#include <vtkFloatArray.h>
#include <vtkRectilinearGrid.h>

vtkDataSet *
avtXXXFileFormat::GetMesh(const char *meshname)
{
    int ndims = 2;
    int dims[3] = {1,1,1};
    vtkFloatArray *coords[3] = {0,0,0};
    // Read the ndims and number of X,Y,Z nodes from file.
    ndims = NUMBER OF MESH DIMENSIONS;
    dims[0] = NUMBER OF NODES IN X-DIMENSION;
    dims[1] = NUMBER OF NODES IN Y-DIMENSION;
    dims[2] = NUMBER OF NODES IN Z-DIMENSION, OR 1 IF 2D;
    // Read the X coordinates from the file.
    coords[0] = vtkFloatArray::New();
    coords[0]->SetNumberOfTuples(dims[0]);
    float *xarray = (float *)coords[0]->GetVoidPointer(0);
    READ dims[0] FLOAT VALUES INTO xarray
    // Read the Y coordinates from the file.
    coords[1] = vtkFloatArray::New();
    coords[1]->SetNumberOfTuples(dims[1]);
    float *yarray = (float *)coords[1]->GetVoidPointer(0);
    READ dims[1] FLOAT VALUES INTO yarray
    // Read the Z coordinates from the file.
    coords[2] = vtkFloatArray::New();
    if(ndims > 2)
    {
        coords[2]->SetNumberOfTuples(dims[2]);
        float *zarray = (float *)coords[2]->GetVoidPointer(0);
        READ dims[2] FLOAT VALUES INTO zarray
    }
    else
    {
        coords[2]->SetNumberOfTuples(1);
        coords[2]->SetComponent(0, 0, 0.);
    }

    //
    // Create the vtkRectilinearGrid object and set its dimensions
    // and coordinates.
    //
    vtkRectilinearGrid *rgrid = vtkRectilinearGrid::New();
    rgrid->SetDimensions(dims);
    rgrid->SetXCoordinates(coords[0]);
    coords[0]->Delete();
    rgrid->SetYCoordinates(coords[1]);
    coords[1]->Delete();
    rgrid->SetZCoordinates(coords[2]);
    coords[2]->Delete();
    return rgrid;
}
```

Curvilinear meshes



Curvilinear meshes are structured meshes as are rectilinear meshes. While in a rectilinear mesh, a small set of independent X,Y,Z coordinate arrays are used to generate the coordinate values for each node in the mesh, in a curvilinear mesh, the node coordinates are explicitly given for each node in the mesh. This means that the sizes of the X,Y,Z coordinate arrays in a curvilinear mesh are all NX*NY*NZ where NX is the number of nodes in the Xdimension, NY is the number of nodes in the Y-dimension, and NZ is the number of nodes in the Z-dimension. Providing the coordinates for every node permits you to create more complex geometries than are possible using rectilinear meshes (See Figure **??**).

Curvilinear meshes are created using the *vtkStructuredGrid* class. The next code listing shows how to create a *vtkStructuredGrid* object once you have read the required information from your file format. The capitalized portions of the code listing indicate incomplete code that you will need to replace with code that can read data from your file format. First, read the number of dimensions for your mesh from the file format and store the value into the *ndims* variable. Once you have done that, read the number of nodes in each of the X,Y,Z dimensions and store those values in the *dims* array. Finally, fill in the code for reading the X coordinate values into the *xarray* array and do the same for the Y and Z coordinate arrays. Once you have replaced the capitalized code portions with code that reads values from your file format, your plug-in should be able to return a valid *vtkStructuredGrid* object once you rebuild it.

Example for creating vtkStructuredGrid in GetMesh.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

#include <vtkPoints.h>
#include <vtkStructuredGrid.h>
```

(continues on next page)

(continued from previous page)

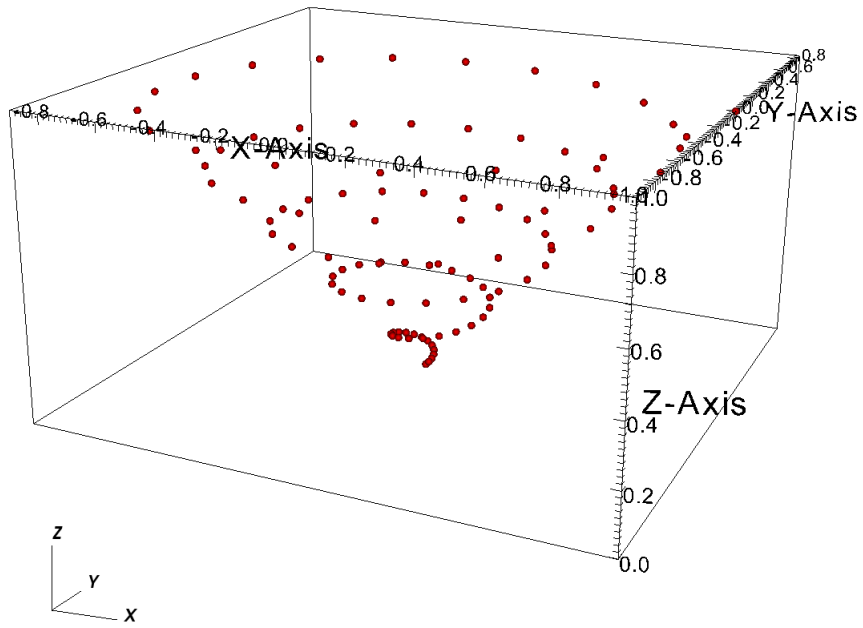
```
vtkDataSet *
avtXXXFileFormat::GetMesh(const char *meshname)
{
    int ndims = 2;
    int dims[3] = {1,1,1};
    ndims = NUMBER OF MESH DIMENSIONS;
    dims[0] = NUMBER OF NODES IN X-DIMENSION;
    dims[1] = NUMBER OF NODES IN Y-DIMENSION;
    dims[2] = NUMBER OF NODES IN Z-DIMENSION, OR 1 IF 2D;
    int nnodes = dims[0]*dims[1]*dims[2];
    // Read the X coordinates from the file.
    float *xarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO xarray
    // Read the Y coordinates from the file.
    float *yarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO yarray
    // Read the Z coordinates from the file.
    float *zarray = 0;
    if(ndims > 2  )
    {
        zarray = new float[nnodes];
        READ dims[2] FLOAT VALUES INTO zarray
    }
    //
    // Create the vtkStructuredGrid and vtkPoints objects.
    //
    vtkStructuredGrid *sgrid = vtkStructuredGrid::New();
    vtkPoints *points = vtkPoints::New();
    sgrid->SetPoints(points);
    sgrid->SetDimensions(dims);
    points->Delete();
    points->SetNumberOfPoints(nnodes);
    //
    // Copy the coordinate values into the vtkPoints object.
    //
    float *pts = (float *) points->GetVoidPointer(0);
    float *xc = xarray;
    float *yc = yarray;
    float *zc = zarray;
    if(ndims == 3)
    {
        for(int k = 0; k < dims[2]; ++k)
            for(int j = 0; j < dims[1]; ++j)
                for(int i = 0; i < dims[0]; ++i)
                {
                    *pts++ = *xc++;
                    *pts++ = *yc++;
                    *pts++ = *zc++;
                }
    }
    else if(ndims == 2)
    {
        for(int j = 0; j < dims[1]; ++j)
            for(int i = 0; i < dims[0]; ++i)
            {
                *pts++ = *xc++;
```

(continues on next page)

(continued from previous page)

```
                *pts++ = *yc++;
                *pts++ = 0.;
            }
    }
    // Delete temporary arrays.
    delete [] xarray;
    delete [] yarray;
    delete [] zarray;
    return sgrid;
}
```

Point meshes



Point meshes are collections of particle positions that can be displayed in VisIt as points or small glyphed icons. Point meshes can be returned from the *GetMesh* method as *vtkUnstructuredGrid* objects that contain the locations of the points and connectivity composed entirely of vertex cells. The next code listing shows how to create a *vtkUnstructuredGrid* object once you have read the required information from your file format. The capitalized portions of the code listing indicate incomplete code that you will need to replace with code that can read data from your file format. First, read the number of dimensions for your mesh from the file format and store the value into the *ndims* variable. Next, read the number of points that make up the point mesh into the *nnodes* variable. Finally, fill in the code for reading the X coordinate values into the *xarray* array and do the same for the Y and Z coordinate arrays. Once you have replaced the capitalized code portions with code that reads values from your file format, your plug-in should be able to return a valid *vtkUnstructuredGrid* object once you rebuild it.

Example for returning a point mesh from GetMesh

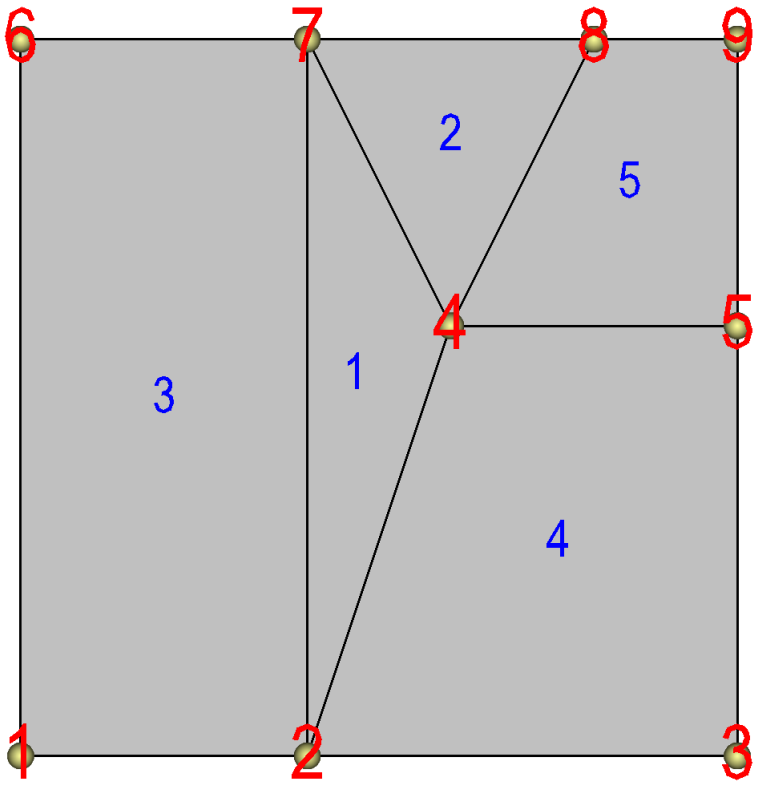
```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

#include <vtkPoints.h>
#include <vtkUnstructuredGrid.h>

vtkDataSet *
avtXXXFileFormat::GetMesh(const char *meshname)
{
    int ndims = 2;
    int nnodes;
    // Read the ndims and number of nodes from file.
    ndims = NUMBER OF MESH DIMENSIONS;
    nnodes = NUMBER OF NODES IN THE MESH;
    // Read the X coordinates from the file.
    float *xarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO xarray
    // Read the Y coordinates from the file.
    float *yarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO yarray
    // Read the Z coordinates from the file.
    float *zarray = 0;
    if(ndims > 2)
    {
        zarray = new float[nnodes];
        READ dims[2] FLOAT VALUES INTO zarray
    }
    //
    // Create the vtkPoints object and copy points into it.
    //
    vtkPoints *points = vtkPoints::New();
    points->SetNumberOfPoints(nnodes);
    float *pts = (float *) points->GetVoidPointer(0);
    float *xc = xarray;
    float *yc = yarray;
    float *zc = zarray;
    if(ndims == 3)
    {
        for(int i = 0; i < nnodes; ++i)
        {
            *pts++ = *xc++;
            *pts++ = *yc++;
            *pts++ = *zc++;
        }
    }
    else if(ndims == 2)
    {
        for(int i = 0; i < nnodes; ++i)
        {
            *pts++ = *xc++;
            *pts++ = *yc++;
            *pts++ = 0.;
        }
    }
    //
    // Create a vtkUnstructuredGrid to contain the point cells.
    //
```

(continues on next page)

(continued from previous page)

```
    vtkUnstructuredGrid *ugrid = vtkUnstructuredGrid::New();
    ugrid->SetPoints(points);
    points->Delete();
    ugrid->Allocate(nnodes);
    vtkIdType onevertex;
    for(int i = 0; i < nnodes; ++i)
    {
        onevertex = i;
        ugrid->InsertNextCell(VTK_VERTEX, 1, &onevertex);
    }
    // Delete temporary arrays.
    delete [] xarray;
    delete [] yarray;
    delete [] zarray;
    return ugrid;
}
```

Unstructured meshes



Unstructured meshes are collections of cells of various geometries that are specified using indices into an array of points. When you write your *GetMesh* method, if your mesh is best described as an unstructured mesh then you can return a *vtkUnstructuredGrid* object. Like some of the other mesh objects, the *vtkUnstructuredGrid* object also uses a *vtkPoints* object to contain its node array. In addition to the *vtkPoints* array, the *vtkUnstructuredGrid* object maintains a list of cells whose connectivity is determined by setting the cell type to one of VTK's predefined unstructured cell

types (*VTK_VERTEX*, *VTK_LINE*, *VTK_TRIANGLE*, *VTK_QUAD*, *VTK_TETRA*, *VTK_PYRAMID*, *VTK_WEDGE*, *VTK_HEXAHEDRON*, etc). More information on these cell types and more can be found in *VTK's Cell types docs <https://examples.vtk.org/site/VTKBook/05Chapter5/#54-cell-types>*. Keep in mind that VisIt may not fully support all of the higher-order cell types. When you add a cell using one of the predefined unstructured cell types, you must also provide a list of node indices that are used as the nodes for the cell. The number of nodes that each cell contains is determined by its cell type.

The next code listing shows how to create a *vtkUnstructuredGrid* object. The connectivity for an unstructured grid can be stored in a file format using a myriad of different approaches. The example code assumes that the connectivity will be stored in an integer array that contains the information for each cell, beginning with the cell type for the first cell, followed by a list of node indices that are used in the cell. After that, the cell type for the second cell appears, followed by its node indices, and so on. For example, if you wanted to store connectivity for cells 1 and 2 in the example shown in Figure **??** then the connectivity array would contain: [VTK_TRIANGLE, 2, 4, 7, VTK_TRIANGLE, 4, 8, 7, . . .]. Note that the node indices in the example begin at one so the example code will subtract one from all of the node indices to ensure that they begin at zero, the starting index for the *vtkPoints* array.

Example for returning an unstructured mesh from GetMesh.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

#include <vtkPoints.h>
#include <vtkUnstructuredGrid.h>
#include <InvalidVariableException.h>

vtkDataSet *
avtXXXFileFormat::GetMesh(const char *meshname)
{
    int ndims = 2;
    int nnodes, ncells, origin = 1;
    // Read the ndims, nnodes, ncells, origin from file.
    ndims = NUMBER OF MESH DIMENSIONS;
    nnodes = NUMBER OF NODES IN THE MESH;
    ncells = NUMBER OF CELLS IN THE MESH;
    origin = GET THE ARRAY ORIGIN (0 or 1);
    // Read the X coordinates from the file.
    float *xarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO xarray
    // Read the Y coordinates from the file.
    float *yarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO yarray
    // Read the Z coordinates from the file.
    float *zarray = 0;
    if(ndims > 2)
    {
        zarray = new float[nnodes];
        READ dims[2] FLOAT VALUES INTO zarray
    }
    // Read in the connectivity array. This example assumes that
    // the connectivity will be stored: type, indices, type,
    // indices, ... and that there will be a type/index list
    // pair for each cell in the mesh.
    int *connectivity = 0;
    ALLOCATE connectivity ARRAY AND READ VALUES INTO IT.
    //
    // Create the vtkPoints object and copy points into it.
    //
    vtkPoints *points = vtkPoints::New();
```

(continues on next page)

(continued from previous page)

```
    points->SetNumberOfPoints(nnodes);
    float *pts = (float *) points->GetVoidPointer(0);
    float *xc = xarray;
    float *yc = yarray;
    float *zc = zarray;
    if(ndims == 3)
    {
        for(int i = 0; i < nnodes; ++i)
        {
            *pts++ = *xc++;
            *pts++ = *yc++;
            *pts++ = *zc++;
        }
    }
    else if(ndims == 2)
    {
        for(int i = 0; i < nnodes; ++i)
        {
            *pts++ = *xc++;
            *pts++ = *yc++;
            *pts++ = 0.;
        }
    }
    // Delete temporary arrays.
    delete [] xarray;
    delete [] yarray;
    delete [] zarray;
    //
    // Create a vtkUnstructuredGrid to contain the point cells.
    //
    vtkUnstructuredGrid *ugrid = vtkUnstructuredGrid::New();
    ugrid->SetPoints(points);
    points->Delete();
    ugrid->Allocate(ncells);
    vtkIdType verts[8];
    int *conn = connectivity
    for(int i = 0; i < ncells; ++i)
    {
        int fileCellType = *conn++;
        // Your file's cellType will likely not match so you
        // will have to translate fileCellType to a VTK
        // cell type.
        int cellType = MAP fileCellType TO VTK CELL TYPE.
        // Determine number of vertices for each cell type.
        if(cellType == VTK_VERTEX)
            nverts = 1;
        else if(cellType == VTK_LINE)
            nverts = 2;
        else if(cellType == VTK_TRIANGLE)
            nverts = 3;
        else if(cellType == VTK_QUAD)
            nverts = 4;
        else if(cellType == VTK_TETRA)
            nverts = 4;
        else if(cellType == VTK_PYRAMID)
            nverts = 5;
        else if(cellType == VTK_WEDGE)
```

(continues on next page)

(continued from previous page)

```
            nverts = 6;
        else if(cellType == VTK_HEXAHEDRON)
            nverts = 8;
        else
        {
            delete [] connectivity;
            ugrid->Delete();
            // Other cell type - need to add a case for it.
            // In the meantime, throw exception or if you
            // know enough, skip the cell.
            EXCEPTION0(InvalidVariableException, meshname);
        }
        // Make a list of node indices that make up the cell.
        for(int j = 0; j < nverts; ++j)
            verts[j] = conn[j] - origin;
        conn += nverts;
        // Insert the cell into the mesh.
        ugrid->InsertNextCell(cellType, nverts, verts);
    }
    delete [] connectivity;
    return ugrid;
}
```

The previous code listing shows how to create an unstructured mesh in a *vtkUnstructuredGrid* object. The code listing contains capitalized portions that you must replace with working code to read the relevant data from your file format. The first instance of code that must be replaced are the lines that read *ndims*, *nnodes*, *ncells*, and *origin* from the file format. The *ndims* variable should contain 2 or 3, depending on whether your data is 2D or 3D. The *nnodes* variable should contain the number of nodes that are used in the set of vertices that describe your unstructured mesh. The *ncells* variable should contain the number of cells that will be added to your unstructured mesh. The *origin* variable should contain 0 or 1, depending on whether your connectivity indices begin at 0 or 1. Once you have set those variables to the appropriate values, you must read in the X, Y, and Z coordinate arrays from the file format and store the values into the *xarray*, *yarray*, and *zarray* array variables. If your file format keeps X,Y,Z values together in a single array then you may be able to read the coordinate values directly into the *vtkPoints* object's memory, skipping the step of copying the X,Y,Z coordinate components into the *vtkPoints* object. After reading in the coordinate values from your file format, unstructured meshes require two more changes to the code in the listing. The next change requires you to allocate memory for a connectivity array, which stores the type of cells and the nodes indices of the nodes that are used in the cells. The final change that you must make to the source code in the listing is located further down in the loop that adds cells to the *vtkUnstructuredGrid* object. The cell type read from your file format will most likely not use the same enumerated type values that VTK uses for its cell types (*VTK_VERTEX*, *VTK_LINE*, ...) so you will need to add code to translate from your cell type designation to VTK cell type numbers. After making the necessary changes and rebuilding your plug-in, your plug-in's *GetMesh* method should be capable of returning a valid *vtkUnstructuredGrid* object for VisIt to plot.

Returning a scalar variable

Now that you can successfully create a **Mesh** plot of the meshes from your file format, you can focus on other types of data such as scalars. If you exposed scalar variables in your plug-in's *PopulateDatabaseMetaData* method then those variable names will appear in the plot menus for plots that can use scalar variables (e.g. the **Pseudocolor** plot). When you create a plot of a scalar variable and click the **Draw** button in the GUI, VisIt will tell your database reader plug-in to open your file, read the mesh, and then your plug-in's *GetVar* method will be called with the name of the variable that you want to plot. The *GetVar* method, like the *GetMesh* method, takes a variable name as an argument. When you receive the variable name in the *GetVar* method you should access your file and read out the desired variable and return it in a VTK data array such as a *vtkFloatArray* or a *vtkDoubleArray*. A *vtkFloatArray* is a VTK object that encapsulates a dynamically allocated array of a given length. The length of the array that you allocate to contain your

variable must match either the number of cells in your mesh or the number of nodes in your mesh. The length is determined by the scalar variable's centering (cell-centered, node-centered).

Example for returning data from GetVar.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

#include <vtkFloatArray.h>

vtkDataArray *
avtXXXFileFormat::GetVar(const char *varname)
{
    int nvals;
    // Read the number of vaues contained in the array
    // specified by varname.
    nvals = NUMBER OF VALUES IN ARRAY NAMED BY varname;
    // Allocate the return vtkFloatArray object. Note that
    // you can use vtkFloatArray, vtkDoubleArray,
    // vtkUnsignedCharArray, vtkIntArray, etc.
    vtkFloatArray *arr = vtkFloatArray::New();
    arr->SetNumberOfTuples(nvals);
    float *data = (float *)arr->GetVoidPointer(0);
    READ nvals FLOAT NUMBERS INTO THE data ARRAY.
    return arr;
}
```

In the previous code listing, there are two capitalized areas that need to have code added to them in order to have a completed *GetVar* method. The first change that you must make is to add code to read the size of the array to be created into the *nvals* variable. The value that is read into the *nvals* variable must be either the number of cells in the mesh on which the variable is defined if you have a cell-centered variable or it must be the number of nodes in the mesh. Once you have successfully set the proper value into the *nvals* variable, you can proceed to read values from your file format into the data array, which points to storage owned by the *vtkFloatArray* object that will be returned from the *GetVar* method. Once you have made these changes, you can rebuild your plug-in and begin plotting scalar variables.

Returning a vector variable

The mechanism for returning a vector variable the same as returning a scalar variable, except in the number of components in each tuple of the *vtkFloatArray* or *vtkDoubleArray*. If you exposed vector variables in your plug-in's *PopulateDatabaseMetaData* method then those variable names will appear in the plot menus for plots that can use vector variables (e.g. the **Vector** plot). The length of the array that you allocate to contain your variable must match either the number of cells in your mesh or the number of nodes in your mesh. The length is determined by the vector variable's centering (cell-centered, node-centered). In addition to setting the length, which like a scalar variable is tied to the number of cells or nodes, you must also set the number of vector components. In VisIt, vector variables always have three components. If the third component is not needed then all values in the third component should be set to zero. The *GetVectorVar* code listing shows how to return a *vtkFloatArray* with multiple components from the *GetVectorVar* method. As with the code listing for *GetVar*, this code listing requires you to replace capitalized lines of code with code that reads data from your file format and stores the results in the variables provided.

Example for returning data from GetVectorVar.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

#include <vtkFloatArray.h>
```

(continues on next page)

(continued from previous page)

```
#include <InvalidVariableException.h>

vtkDataArray *
avtXXXFileFormat::GetVectorVar(const char *varname)
{
    int nvals, ncomps = 3;
    // Read the number of values contained in the array
    // specified by varname.
    nvals = NUMBER OF VALUES IN ARRAY NAMED BY varname;
    ncomps = NUMBER OF VECTOR COMPONENTS IN ARRAY NAMED BY varname;
    // Read component 1 from the file.
    float *comp1 = new float[nvals];
    READ nvals FLOAT VALUES INTO comp1
    // Read component 2 from the file.
    float *comp2 = new float[nvals];
    READ nvals FLOAT VALUES INTO comp2
    // Read component 3 from the file.
    float *comp3 = 0;
    if(ncomps > 2)
    {
        comp3 = new float[nvals];
        READ nvals FLOAT VALUES INTO comp3
    }
    // Allocate the return vtkFloatArray object. Note that
    // you can use vtkFloatArray, vtkDoubleArray,
    // vtkUnsignedCharArray, vtkIntArray, etc.
    vtkFloatArray *arr = vtkFloatArray::New();
    arr->SetNumberOfComponents(3);
    arr->SetNumberOfTuples(nvals);
    float *data = (float *)arr->GetVoidPointer(0);
    float *c1 = comp1;
    float *c2 = comp2;
    float *c3 = comp3;
    if(ncomps == 3)
    {
        for(int i = 0; i < nvals; ++i)
        {
            *data++ = *c1++;
            *data++ = *c2++;
            *data++ = *c3++;
        }
    }
    else if(ncomps == 2)
    {
        for(int i = 0; i < nvals; ++i)
        {
            *data++ = *c1++;
            *data++ = *c2++;
            *data++ = 0.;
        }
    }
    else
    {
        delete [] comp1;
        delete [] comp2;
        delete [] comp3;
        arr->Delete();
```

(continues on next page)

(continued from previous page)

```
        EXCEPTION1(InvalidVariableException, varname);
    }
    // Delete temporary arrays.
    delete [] comp1;
    delete [] comp2;
    delete [] comp3;

    return arr;
}
```

Using a VTK reader class

The implementations so far for the *GetMesh*, *GetVar*, and *GetVectorVar* plug-in methods have assumed that the database plug-in would do the work of interacting with the file format to read data into VTK form. Most of the work of reading a file and creating VTK objects from it can be handled at the VTK level if you wish. This means that it is possible to use an existing VTK reader class to read data into VisIt if you are willing to implement your plug-in methods so that they in turn call the VTK reader object's methods. See VisIt's VTK database reader plug-in for an example of how to call VTK reader objects from inside a VisIt database reader plug-in.

8.10.8 Advanced topics

If you've implemented your database reader plug-in using only the techniques outlined in this chapter so far then you likely have a database reader plug-in that works and correctly serves up its data to VisIt in VTK form. This part of the chapter explains some of the more advanced, though not necessarily required, techniques that you can use to enhance your plug-in. For instance, you can enhance your plug-in so it returns the correct simulation times from the data files. You can also add code to return data and spatial extents for your data, enabling VisIt to make more optimization decisions when processing files with multiple domains.

Returning cycles and times

Simulations often iterate for many thousands of cycles while they solve their systems of equations. Generally, each simulation cycle has an associated cycle number and time value. Many file formats save this information so it can be made available later to postprocessing tools such as VisIt. VisIt uses cycles and times to help you navigate through time in your database by providing the same time frame of reference that your simulation used. VisIt's can show the current time value as you scroll through time using the time slider. Cycle and time values for the current time state are often displayed in the visualization window. Returning cycle and time values from your plug-in is completely optional. In fact, returning cycle and time values for data such as CAD drawings does not make sense. Since returning cycles and times is optional in a VisIt database reader plug-in, you can choose to not implement the methods that return cycles and times. You can also implement code to return time but not cycles or vice-versa. The mechanics of returning cycles and times are a little different depending on whether you have written an *ST* or an *MT* database reader plug-in. In any case, if your plug-in implements the methods to return cycles or times then those methods will be some of the first methods called when VisIt accesses your database reader plug-in. VisIt calls the methods to get cycles and times and if the returned values appear to be valid then they are added to the metadata for your file so they can be returned to the VisIt clients and used to populate windows such as the **File Information** window.

Returning cycles and times in an ST plug-in

When VisIt creates plug-in objects to handle a list of files using an *ST* plug-in, there is one plug-in object per file in the list of files. Since each plug-in object can only ever be associated with one file, the programming interface for

returning cycles and times for an *ST* plug-in provides methods that return a single value. The methods for returning cycles and times for an *ST* plug-in are:

```
virtual bool ReturnsValidCycle() const { return true; }
virtual int GetCycle(void);
virtual bool ReturnsValidTime() const { return true; }
virtual double GetTime(void);
```

Implementing valid cycles and times can be done independently of one another and there is no requirement that you have to implement both or either of them, for that matter. The *ReturnsValidCycle* method is a simple method that you should expose if you plan to provide a custom *GetCycle* method in your database reader plug-in. If you provide *GetCycle* then the *ReturnsValidCycle* method should return *true*. The same pattern applies if you implement *GetTime* - except that you would also implement the *ReturnsValidTime* method. Replace the capitalized sections of code in the listing with code to read the correct cycle and time values from your file format.

Example for returning cyles, times from ST plug-in.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

int
avtXXXFileFormat::GetCycle(void)
{
    int cycle = OPEN FILE AND READ THE CYCLE VALUE;
    return cycle;
}

double
avtXXXFileFormat::GetTime(void)
{
    double dtime = OPEN FILE AND READ THE TIME VALUE;
    return dtime;
}
```

In the event that you implement the *GetCycle* method but no cycle value is available in the file, you can return the *INVALID_CYCLE* value to make VisIt discard your plug-in's cycle number and guess the cycle number from the filename. If you want VisIt to successfully guess the cycle number from the filename then you must implement the *GetCycleFromFilename* method.

```
int
avtXXXXFileFormat::GetCycleFromFilename(const char *f) const
{
    return GuessCycle(f);
}
```

Returning cycles and times in an MT plug-in

An *MT* database reader plug-in may return cycles and times for multiple time states so the programming interface for *MT* plug-ins allows you to return vectors of cycles and times. In addition, an *MT* database reader plug-in prefers to know upfront how many time states will be returned from the file format so in addition to *GetCycles* and *GetTimes* methods, there is a *GetNTimesteps* method that is among the first methods called from your database reader plug-in.

```
virtual void GetCycles(std::vector<int> &);
virtual void GetTimes(std::vector<double> &);
virtual int GetNTimesteps(void);
```

As with *ST* plug-ins, there is no requirement that an *MT* plug-in must provide a list of cycles or times. However, an *MT* plug-in must provide a *GetNTimesteps* method. If you are enhancing your database reader plug-in to return cycles and times then it is convenient to implement your *GetNTimesteps* method such that it just calls your *GetCycles* or *GetTimes* method and returns the length of the vector returned by those methods. This simplifies the implementation and ensures that the number of time states reported by your database reader plug-in matches the length of the cycle and time vectors returned from *GetCycles* and *GetTimes*. Replace the capitalized sections of code in the listing with code to read the correct cycles and times from your file format.

Example for returning cycles, times from MT plug-in.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

void
avtXXXFileFormat::GetCycles(std::vector<int> &cycles)
{
    int ncycles, *vals = 0;
    ncycles = OPEN FILE AND READ THE NUMBER OF CYCLES;
    READ ncycles INTEGER VALUES INTO THE vals ARRAY;
    // Store the cycles in the vector.
    for(int i = 0; i < ncycles; ++i)
        cycles.push_back(vals[i]);
    delete [] vals;
}

void
avtXXXFileFormat::GetTime(std::vector<double> &times)
{
    int ntimes;
    double *vals = 0;
    ntimes = OPEN FILE AND READ THE NUMBER OF TIMES;
    READ ntimes DOUBLE VALUES INTO THE vals ARRAY;
    // Store the times in the vector.
    for(int i = 0; i < ntimes; ++i)
        times.push_back(vals[i]);
    delete [] vals;
}

int
avtXXXXFileFormat::GetNTimesteps(void)
{
    std::vector<double> times;
    GetTimes(times);
    return times.size();
}
```

Auxiliary data

This section describes how to enable your *MD* database reader plug-in so it can provide auxiliary data such as data extents, spatial extents, and materials to VisIt if they are available in your file format. "Auxiliary data" is the generic term for many types of data that VisIt's pipeline can use to perform specific tasks such as I/O reduction or material selection. VisIt's database reader plug-in interfaces provide a method called *GetAuxiliaryData* that you can implement if you want your plug-in to be capable of returning auxiliary data. Note however that if your plug-in is *MTMD* then you will have to cache your spatial and data extents in the plug-in's variable cache in the *PopulateDatabaseMetaData* method instead of returning that information from the *GetAuxiliaryData* method. This subtle difference in how certain metadata is accessed by VisIt must be observed by an *MTMD* plug-in in order for it to return spatial and data extents.

The method arguments for the *GetAuxiliaryData* method may vary somewhat depending on whether your database reader plug-in is based on the *STSD*, *STMD*, *MTSD*, *MTMD* interfaces. There is an extra integer argument for the time state if your plug-in is *MT* and there is another integer argument for the domain if your plug-in is *MD*. Those differences aside, the *GetAuxiliaryData* method always accepts the name of a variable, a string indicating the type of data being requested, a pointer to optional data required by the type of auxiliary data being requested, and a return reference for a destructor function that will be responsible for freeing resources for the returned data. The variable name that VisIt passes to the *GetAuxiliaryData* method is the name of a variable such as those passed to the *GetVar* method when VisIt wants to read a variable's data.

Returning data extents

When an *MD* database reader plug-in provides data extents for each of its domains, VisIt has enough information to make important optimization decisions in filters that support data extents. For example, if you create a **Contour** plot using a specific contour value, VisIt can check the data extents for each domain before any domains are read from disk and determine the list of domains that contain the desired contour value. After determining which subset of the domains will contribute to the final image, VisIt's compute engine then reads and processes only those domains, saving work and accelerating VisIt's computations. For a more complete explanation of data extents, see *Writing data extents*.

In the context of returning data extents, VisIt first checks a plug-in's variable cache for extents. If the desired extents are not available then VisIt calls the plug-in's *GetAuxiliaryData* method with the name of the scalar variable for which data extents are required and also passes *AUXILIARY_DATA_DATA_EXTENTS* as the type argument, indicating that the *GetAuxiliaryData* method is being called to obtain the data extents for the specified scalar variable. If the data extents for the specified variable are not available then the *GetAuxiliaryData* method should return 0. If the data extents are available then the list of minimum and maximum values for the specified variable are assembled into an interval tree structure that VisIt uses for fast comparisons of different data ranges. Once the interval tree is constructed, as shown in the code listing, the *GetAuxiliaryData* method must return the interval tree object and set the destructor function argument to a function that can be called to later destroy the interval tree. To add support for data extents to your database reader plug-in, copy the *GetAuxiliaryData* method in the code listing and replace the capitalized lines of code with code that reads the required information from your file format.

Example for returning data extents.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

#include <avtIntervalTree.h>

// STMD version of GetAuxiliaryData.

void *
avtXXXXFileFormat::GetAuxiliaryData(const char *var,
    int domain, const char *type, void *,
    DestructorFunction &df)
{
    void *retval = 0;
    if(strcmp(type, AUXILIARY_DATA_DATA_EXTENTS) == 0)
    {
        // Read the number of domains for the mesh.
        int ndoms = READ NUMBER OF DOMAINS FROM FILE;
        // Read the min/max values for each domain of the
        // "var" variable. This information should be in
        // a single file and should be available without
        // having to read the real data.
        double *minvals = new double[ndoms];
        double *maxvals = new double[ndoms];
```

(continues on next page)

(continued from previous page)

```
        READ ndoms DOUBLE VALUES INTO minvals ARRAY.
        READ ndoms DOUBLE VALUES INTO maxvals ARRAY.
        // Create an interval tree
        avtIntervalTree *itree = new avtIntervalTree(ndoms, 1);
        for(int dom = 0; dom < ndoms; ++dom)
        {
            double range[2];
            range[0] = minvals[dom];
            range[1] = maxvals[dom];
            itree->AddElement(dom, range);
        }
        itree->Calculate(true);
        // Delete temporary arrays.
        delete [] minvals;
        delete [] maxvals;
        // Set return values
        retval = (void *)itree;
        df = avtIntervalTree::Destruct;
    }
    return retval;
}
```

Returning spatial extents

Another type of auxiliary data that VisIt supports for *MD* file formats are spatial extents. When VisIt knows the spatial extents for all of the domains that comprise a mesh, VisIt can optimize operations such as the **Slice** operator by first determining whether the slice will intersect a given domain. The **Slice** operator is thus able to use spatial extents to determine which set of domains must be read from disk and processed in order to produce the correct visualization. Spatial extents are used in this way by many filters to reduce the set of domains that must be processed. When VisIt asks the database reader plug-in for spatial extents, the *GetAuxiliaryData* method is called with its type argument set to *AUXILIARY_DATA_SPATIAL_EXTENTS*. When VisIt creates spatial extents, they are stored in an interval tree structure as they are with data extents. The main difference is the input into the interval tree. When adding information about a specific domain to the interval tree, you must provide the minimum and maximum spatial values for the domain's X, Y, and Z dimensions. The spatial extents for one domain are expected to be provided in the following order: xmin, xmax, ymin, ymax, zmin, zmax. To add support for spatial extents to your database reader plug-in, copy the *GetAuxiliaryData* method in the code listing and replace the capitalized lines of code with code that reads the required information from your file format.

Example for returning spatial extents.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

#include <avtIntervalTree.h>

// STMD version of GetAuxiliaryData.

void *
avtXXXXFileFormat::GetAuxiliaryData(const char *var,
    int domain, const char *type, void *,
    DestructorFunction &df)
{
    void *retval = 0;
    if(strcmp(type, AUXILIARY_DATA_SPATIAL_EXTENTS) == 0)
```

(continues on next page)

(continued from previous page)

```
    {
        // Read the number of domains for the mesh.
        int ndoms = READ NUMBER OF DOMAINS FROM FILE;
        // Read the spatial extents for each domain of the
        // mesh. This information should be in a single
        // and should be available without having to
        // read the real data. The expected format for
        // the data in the spatialextents array is to
        // repeat the following pattern for each domain:
        // xmin, xmax, ymin, ymax, zmin, zmax.
        double *spatialextents = new double[ndoms * 6];
        READ ndoms*6 DOUBLE VALUES INTO spatialextents ARRAY.
        // Create an interval tree
        avtIntervalTree *itree = new avtIntervalTree(ndoms, 3);
        double *extents = spatialextents;
        for(int dom = 0; dom < ndoms; ++dom)
        {
            itree->AddElement(dom, extents);
            extents += 6;
        }
        itree->Calculate(true);
        // Delete temporary array.
        delete [] spatialextents;
        // Set return values
        retval = (void *)itree;
        df = avtIntervalTree::Destruct;
    }
    return retval;
}
```

Returning materials

Materials are another type of auxiliary data that database plug-ins can provide. A material classifies different pieces of the mesh into different named subsets that can be turned on and off using VisIt's **Subset** window. In the simplest case, you can think of a material as a cell-centered variable, or matlist, defined on your mesh where each cell contains an integer that identifies a particular material such as "Steel" or "Air". VisIt's *avtMaterial* object is used to encapsulate knowledge about materials. The *avtMaterial* object contains the *matlist* array and a list of names corresponding to each unique material number in the matlist array. Materials can also be structured so that instead of providing just one material number for each cell in the mesh, you can provide multiple materials per cell with volume fractions occupied by each. So-called "mixed materials" are created using additional arrays, described in *Materials*. To add support for materials in your database reader plug-in's *GetAuxiliaryData* method, replace the capitalized lines in the code example with code that read the necessary values from your file format.

Example for returning materials.

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

#include <avtMaterial.h>

// STMD version of GetAuxiliaryData.

void *
avtXXXXFileFormat::GetAuxiliaryData(const char *var,
```

(continues on next page)

(continued from previous page)

```
    int domain, const char *type, void *,
    DestructorFunction &df)
{
    void *retval = 0;
    if(strcmp(type, AUXILIARY_DATA_MATERIAL) == 0)
    {
        int dims[3] = {1,1,1}, ndims = 1;
        // Structured mesh case
        ndims = MESH DIMENSION, 2 OR 3;
        dims[0] = NUMBER OF ZONES IN X DIMENSION;
        dims[1] = NUMBER OF ZONES IN Y DIMENSION;
        dims[2] = NUMBER OF ZONES IN Z DIMENSION, OR 1 IF 2D;
        // Unstructured mesh case
        dims[0] = NUMBER OF ZONES IN THE MESH
        ndims = 1;
        // Read the number of materials from the file. This
        // must have already been read from the file when
        // PopulateDatabaseMetaData was called.
        int nmats = NUMBER OF MATERIALS;
        // The matnos array contains the list of numbers that
        // are associated with particular materials. For example,
        // matnos[0] is the number that will be associated with
        // the first material and any time it is seen in the
        // matlist array, that number should be taken to mean
        // material 1. The numbers in the matnos array must
        // all be greater than or equal to 1.
        int *matnos = new int[nmats];
        READ nmats INTEGER VALUES INTO THE matnos ARRAY.
        // Read the material names from your file format or
        // make up names for the materials. Use the same
        // approach as when you created material names in
        // the PopulateDatabaseMetaData method.
        char **names = new char *[nmats];
        READ MATERIAL NAMES FROM YOUR FILE FORMAT UNTIL EACH
        ELEMENT OF THE names ARRAY POINTS TO ITS OWN STRING.
        // Read the matlist array, which tells what the material
        // is for each zone in the mesh.
        int nzones = dims[0] * dims[1] * dims[2];
        int *matlist = new int[nzones];
        READ nzones INTEGERS INTO THE matlist array.
        // Optionally create mix_mat, mix_next, mix_zone, mix_vf
        // arrays and read their contents from the file format.
        // Use the information to create an avtMaterial object.
        avtMaterial *mat = new avtMaterial(
            nmats,
            matnos,
            names,
            ndims,
            dims,
            0,
            matlist,
            0, // length of mix arrays
            0, // mix_mat array
            0, // mix_next array
            0, // mix_zone array
            0 // mix_vf array);
```

(continues on next page)

(continued from previous page)

```
        // Clean up.
        delete [] matlist;
        delete [] matnos;
        for(int i = 0; i < nmats; ++i)
            delete [] names[i];
        delete [] names;
        // Set the return values.
        retval = (void *)mat;
        df = avtMaterial::Destruct;
    }
    return retval;
}
```

Returning ghost zones

Ghost zones are mesh zones that should not be visible in the visualization but may provide additional information such as values along domain boundaries. VisIt uses ghost zones for ensuring variable continuity across domain boundaries, for removing internal domain boundary faces, and for blanking out specific zones. This section covers the code that must be added to make your database reader plug-in order for it to return ghost zones to VisIt.

Blanking out zones

Blanking out specific zones so they do not appear in a visualization is a common practice for creating holes in structured meshes so cells zones that overlap or tangle on top of one another can be removed from the mesh. If you want to create a mesh that contains voids where zones have been removed then you can add a special cell-centered array to your mesh before you return it from your plug-in's *GetMesh* method. The code in the listing can be used to remove zones from any mesh type and works by looking through a mesh-sized array containing on/off values for each zone and sets the appropriate values into the ghost zone array that gets added to the mesh object. Replace any capitalized code with code that can read the necessary values from your file format.

Example for returning ghost data.

Listing 8.4: Example for returning ghost data

```
// NOTE - This code is incomplete and requires capitalized portions
// to be replaced with code to read values from your file format.

#include <avtGhostData.h>
#include <vtkUnsignedCharArray.h>

vtkDataSet *
avtXXXXFileFormat::GetMesh(const char *meshname)
{
    // Code to create your mesh goes here.
    vtkDataSet *retval = CODE TO CREATE YOUR MESH;
    // Now that you have your mesh, figure out which cells need
    // to be removed.
    int nCells = retval->GetNumberOfCells();
    int *blanks = new int[nCells];
    READ nCells INTEGER VALUES INTO blanks ARRAY.
    // Now that we have the blanks array, create avtGhostZones.
    unsigned char realVal = 0, ghost = 0;
    avtGhostData::AddGhostZoneType(ghost, ZONE_NOT_APPLICABLE_TO_PROBLEM);
```

(continues on next page)

(continued from previous page)

```
    vtkUnsignedCharArray *ghostCells = vtkUnsignedCharArray::New();
    ghostCells->SetName("avtGhostZones");
    ghostCells->Allocate(nCells);
    for(int i = 0; i < nCells; ++i)
    {
        if(blanks[i])
            ghostCells->InsertNextValue(realVal);
        else
            ghostCells->InsertNextValue(ghost);
    }

    retval->GetCellData()->AddArray(ghostCells);
    retval->SetUpdateGhostLevel(0);
    ghostCells->Delete();
    // Clean up
    delete [] blanks;
    return retval;
}
```

Ghost zones at the domain boundaries

When ghost zones are used to ensure continuity across domains, an extra layer of zones must be added to the mesh boundaries where the boundary is shared with another domain. Once you have done that step, the approach for providing ghost zones is the same as for blanking out cells using ghost zones if your blanks array contains zeroes for only the zones that appear on domain boundaries. The one minor difference is that you must substitute the *DUPLICATED_ZONE_INTERNAL_TO_PROBLEM* ghost zone type for the *ZONE_NOT_APPLICABLE_TO_PROBLEM* ghost zone type in the code example.

Parallelizing your reader

VisIt is a distributed program made up of multiple software processes that act as a whole. The software process that reads in data and processes it is the compute engine, which comes in serial and parallel versions. All of the *libE* plug-ins in VisIt also have both serial and parallel versions. The parallel *libE* plug-ins can contain specialized MPI communication to support the communication patterns needed by the algorithms used. If you want to parallelize your database reader plug-in then, in most cases, you will have to use the *MD* interface or convert from *SD* to *MD*. There are some *SD* formats that can adaptively decompose their data so each processor has work (see the DDCMD plug-in) but most database plug-ins that benefit from parallelism instead are implemented as *MD* plugins. *MD* plug-ins are a natural fit for the parallel compute engine because they serve data that is already decomposed into domains. Some database reader plug-ins, such as the BOV plug-in, take single domain meshes and automatically decompose them into multiple domains for faster processing on multiple processors.

Deriving your plug-in from an *MD* interface is useful since it naturally tells VisIt to expect data from more than one domain when reading your file format. There are a number of parallel optimizations that can be made inside of your *MD* database reader plug-in. For example, you might have one processor read the metadata and broadcast it to all other processors so when you visualize your data with a large number of processors, they are not all trying to read the file that contains the metadata. VisIt's parallel compute engine can use one of two different load balancing schemes: static or dynamic. In static load balancing, each processor is assigned a fixed list of domains and each of those domains is processed one at a time in parallel visualization pipelines until the result is computed. When static load balancing is used, the same code is executed on all processors with different data and there are more opportunities for parallel, global communication. When VisIt's parallel compute engine uses dynamic load balancing, the master process acts as an executive that assigns work as needed to each processor. When a processor needs work, it requests a domain from the executive and it processes the domain in its visualization pipeline until the results for the domain have been calculated. After that, the processor asks the executive for another domain. In dynamic load balancing, each processor

can be working on very different operations so there is no opportunity to do global communication. VisIt attempts to do dynamic load balancing unless any one of the filters in its visualization pipeline requires global communication, in which case static load balancing must be used. This means that the places where global communication can occur are few.

VisIt's database plug-in interfaces provide the *ActivateTimestep* method as a location where global, parallel communication can be performed safely. If your parallel database reader needs to do parallel communication such as broadcasting metadata to all processors, or figuring out data extents in parallel then that code must be added in the *ActivateTimestep* method.

Third party library support

If your plugin depends on an I/O library not already supported by VisIt, and if the plugin will be contributed to VisIt's repo, then support for the library needs to be added to the *build_visit* script. Please see *Creating a build_visit Module* for how to add a new module to build_visit. A CMake *Find* module will also need to be added to `src/CMake` and `src/CMake/SetupThirdParty.cmake` will need to be modified to include that module. See *Adding a Find Module for Third-Party Libraries* for more information.

CHAPTER

NINE

BUILDING VISIT FROM SOURCES

In this chapter, we will discuss how to build VisIt from source code. The building of VisIt is automated with the `build_visit` script. It will build VisIt and all of VisIt's third party libraries. It can be configured to build VisIt with a minimum of third party libraries to building VisIt with all of it's third party libraries. This chapter describes how to build VisIt, starting with the most simple case and then moving to more complex use cases.

9.1 Basic Usage

9.1.1 Doing a minimal build

When using `build_visit` without any arguments it will do a minimal build of VisIt downloading the VisIt source code by making an anonymous git clone from GitHub and downloading the source code for the third party libraries from NERSC. It will build a serial version of the code without any of the optional I/O libraries. This will result in only the file readers that require no external dependencies to be built. Buiding VisIt in this fashion will give you the highest probability of success.

```
./build_visit3_0_1
```

9.1.2 Building with multiple cores

When `build_visit` is run by default it will build the code using a single core. This may take a half a day or longer. Modern computers have anywhere from 4 to 80 cores at the time of the writing of this chapter. You can speed up the build process by specify that `build_visit` use more cores. If you are using a shared resource you probably shouldn't use all the cores in consideration of other users of the system. The following example specifies using 4 cores.

```
./build_visit3_0_1 --makeflags -j4
```

9.1.3 Specifying the third party library install location

When `build_visit` is run by default it will install the third party libraries in the directory `third_party` in the current directory. If you would like to install the libraries in another directory for the purposes of sharing them with other users of the system, you can have `build_visit` install them in a different directory. The following example specifies installing the third party libraries in a another location.

```
./build_visit3_0_1 --thirdparty-path /usr/gapps/visit/third_party
```

9.1.4 Building with the HDF5 and Silo libraries

Some of the more common I/O libraries that will result in building a larger number of file readers are HDF5 and Silo. The following example specifies building HDF5 and Silo.

```
./build_visit3_0_1 --hdf5 --silo
```

9.1.5 Building the stable optional libraries

If you are feeling lucky you can have `build_visit` build all of the optional I/O libraries that have a high probability of building. The following example specifies building the more reliable of the optional I/O libraries.

```
./build_visit3_0_1 --optional
```

9.1.6 Using a VisIt source code tar file

You can also have visit use the prepackaged source code for a specific version of VisIt instead of doing a git download of the source code. The tar file should be considerably smaller than a git clone. The following example uses the VisIt source code corresponding to the official 3.0.1 release of VisIt.

```
./build_visit3_0_1 --optional --tarball visit3.0.1.tar.gz
```

9.1.7 If `build_visit` is interrupted

If `build_visit` is interrupted while it is executing, it is suggested that you remove the directories associated with the last package it was in the process of building. `build_visit` always leaves directories intact when it runs to aid with troubleshooting failures. Likewise, `build_visit` doesn't remove existing directories before starting to build a package. This can sometimes problems when `build_visit` is interrupted and you restart the build again.

9.1.8 Finishing up

Once you have successfully built VisIt, there are a couple of directions you can go. The first option is to use it in the location where it was built. The executable can run by executing the following command:

```
visit/build/bin/visit
```

if you built using a git clone.

```
visit3.0.1/build/bin/visit
```

if you built using a tar file.

The second option is to create a distribution file that you can install using `visit-install`. This can be done by executing the following command:

```
cd visit/build
make package
```

if you built using a git clone.

```
cd visit3.0.1/build
make package
```

if you built using a tar file.

9.1.9 A note about compiler versions

If you encounter problems with build_visit (especially building Qt), it may be due to the compiler version being used.

If building Qt 6, then a fully compliant c++17 compiler is required (g++8 or newer on Linux) otherwise a compiler supporting c++14 is needed, with minimum g++ set to 7.3 or newer on Linux.

Sometimes the absolute latest releases of compilers will cause compile errors during the building of third_party libraries requiring patches to the library's code base, a change to an older compiler version, or an update to a newer version of the library. If this is the case, and you would like us to support the compiler version you are using, please contact us by one of our *Contact methods*, letting us know the OS and OS-version as well as the compiler version you are attempting to use, and the version of VisIt being built, along with all compile error messages.

9.2 Advanced Usage

`build_visit` comes with many options for features such as building a parallel version, overcoming issues with OpenGL, a rendering library used by VisIt to render images, and controlling precisely what libraries VisIt is built with.

9.2.1 Building a parallel version

One of powerful capabilities of VisIt is running in parallel on large parallel clusters. VisIt runs in parallel using a library called MPI, which stands for Message Passing Interface. There are a couple of ways in which you can build a parallel version of VisIt using MPI. If your system doesn't already have MPI installed on it, which is typically the case with a desktop system or small cluster, then you can use MPICH, which is an open source implementation of MPI. The following example builds a parallel version using MPICH.

```
./build_visit3_0_1 --mpich
```

If your system already has MPI installed on it, which is typically the case with a large system at a computer center, you can set several environment variables that specify the location of the MPI libraries and header files. The following example uses a system installed MPI library.

```
 env PAR_COMPILER=/usr/packages/mvapich2/bin/mpicc \
     PAR_COMPILER_CXX=/usr/packages/mvapich2/bin/mpicxx \
     PAR_INCLUDE=-I/usr/packages/mvapich2/include \
     PAR_LIBS=-lmpl \
./build_visit3_0_1 --parallel
```

When running in parallel, the user will typically use scalable rendering for rendering images in parallel. VisIt does this through the use of the Mesa 3D graphics library. Because of this you will want to include Mesa 3D when building a parallel version. In the following example we have included building with the Mesa 3D library.

```
./build_visit3_0_1 --mpich --osmesa
```

9.2.2 Building with Mesa as the OpenGL implementation

Mesa 3D is also an implementation of OpenGL and it can be used in place of the system OpenGL when building VisIt. There are a couple of reasons you would want to use Mesa 3D instead of the system OpenGL. The first is when you don't have a system OpenGL, which typically occurs when building in a container or on a virtual machine. The second is when your system implementation of OpenGL is too old to support VTK. In the following example we use Mesa 3D instead of the system OpenGL.

```
./build_visit3_0_1 --mesagl
```

9.2.3 The difference between `--mesagl` and `--osmesa`

When you specify `--mesagl` VTK will be built against Mesa 3D. When you specify `--osmesa` VTK is built against the system OpenGL and the Mesa 3D library is substituted at run time for OpenGL when running the parallel engine to enable scalable rendering. If you specify `--mesagl` then `--osmesa` is unnecessary and ignored if specified.

9.2.4 Building Server only

Sometimes a special version of VisIt is needed for a remote server, one that only processes data and does not include any GUI elements. VisIt can be built in this fashion (and the build will be sped up considerably) by adding the `-server-comonents-only flag` to build only the server and related programs. Then you can run client/server from a desktop system running the GUI locally, and connect to the remote server to process data.

```
env PAR_COMPILER=mpicc ./build_visit3_3_3 --server-components-only --mesa --icet
```

This will do a basic build, but will probably not include the IO libraries you need (unless you only need VTK). You can specify any needed libraries individually, e.g. `--hdf5 --netcdf --conduit --mfem` to add HDF5, NetCDF, Conduit and Mfem IO libraries.

9.2.5 Building VisIt with Pre-Installed (e.g. System) Libraries

On many systems, some libraries VisIt needs (e.g. Qt, VTK, Python OpenGL, HDF5, etc.) come pre-installed. Can a user just use those pre-installed libraries to build VisIt?

Please don't! In all likelihood this will not work at all or, worse, it will only partially work and fail in subtle ways that are nearly impossible to diagnose. In the unlikely chance it appears to work upon reporting any issues our first question will be, how was VisIt configured/built? If VisIt is built in a way that is not consistent with how developers routinely build, run and test it, we will not be able to reproduce the issue, debug it, identify work-arounds or otherwise provide sufficient support.

Apart from the general issues of reproducibility and support, there are many reasons building VisIt with pre-installed libraries will likely not work. Below, we breifly summarize various compatability issues with trying to use pre-installed libraries.

Version Compatability [Pre-installed libraries are not the version VisIt requires] Often, users notice a *newer* version of a library VisIt needs is pre-installed on their system and expect VisIt will run *better* with this newer version. However, having a newer version of VTK, for example, pre-installed does not mean VisIt will build or run properly with that version. Major versions of VTK, for example, (e.g. 8.0 and 9.0) are not compatable. Incompatabilities sometimes exist even between minor versions of some libraries. Incorrect library versions may cause VisIt to either fail to build or fail to run properly.

Patch Compatability [Pre-installed libaries are missing patches VisIt requires] In some cases, the libraries VisIt needs are patched to work around various issues building or running VisIt. Such patches are almost certainly

not in any pre-installed version of the library. Missing patches may cause VisIt to either fail to build or fail to run properly.

Configuration Compatability [Pre-installed libraries are not configured in a way VisIt requires] Libraries often have many build options which enable or disable certain features. The Qt library, for example, has hundreds of build options. Some build options VisIt may not care about. Other build options, however, VisIt may require to be enabled and still other options to be disabled. Incorrect library configuration may cause VisIt to either fail to build or fail to run properly.

Dependency Compatability [Pre-installed libraries are not built with dependencies VisIt requires] Libraries often have dependencies on still other libraries. For example, Qt and VTK can both depend on OpenGL. In some cases, however, VisIt may require a specific implementation of OpenGL called MesaGL. Incorrect dependencies may cause VisIt to either fail to build or fail to run properly. Such dependencies complicate things significantly because it means all of the aforementioned compatability issues apply, recursively, to any libraries a pre-installed library depends on.

Compiler (Run-Time) Compatability [Pre-installed libraries are not built with a compiler (run-time) VisIt requires] For some situations, building VisIt and its dependencies requires a specific compiler. The compiler (run-time) used for pre-installed libraries may not be compatable with the compiler (run-time) VisIt requires.

There are likely other subtle compatability issues that can arise which we have neglected to mention here. A fully featured build of VisIt can involve 35+ libraries, many of which may come pre-installed (Qt, VTK, Python, HDF5, netCDF, OpenGL, MPI to name a few) on any particular platform. Bottom line, the number of ways pre-installed libraries can be built such that they will cause VisIt to either fail to build or fail to run properly are almost boundless. For this reason, we discourage users from attempting to build VisIt using pre-installed libraries and warn users that in all likelihood we will not have sufficient resources to help address any resulting issues that may arise.

9.2.6 Building on a system without internet access

When you want to build visit on a system without internet access, you can use `build_visit` to download the third party libraries and source code to a system that has internet access and then move those files to your machine without access. The following example downloads the optional libraries, mpich and osmesa.

```
./build_visit3_0_1 --optional --mpich --osmesa --download-only
```

Unfortunately, due to the way the code that builds Python is implemented, some Python libraries will not be downloaded. Here is the list of commands to download those additional libraries.

```
wget http://portal.nersc.gov/project/visit/releases/3.0.1/third_party/Imaging-1.1.7.
↪tar.gz
wget http://portal.nersc.gov/project/visit/releases/3.0.1/third_party/setuptools-28.0.
↪0.tar.gz
wget http://portal.nersc.gov/project/visit/releases/3.0.1/third_party/Cython-0.25.2.
↪tar.gz
wget http://portal.nersc.gov/project/visit/releases/3.0.1/third_party/numpy-1.14.1.zip
wget http://portal.nersc.gov/project/visit/releases/3.0.1/third_party/pyparsing-1.5.2.
↪tar.gz
wget http://portal.nersc.gov/project/visit/releases/3.0.1/third_party/requests-2.5.1.
↪tar.gz
```

It's possible that the list could change and the above list becomes outdated. In this case you can run `build_visit` to just build Python and that will end up downloading all the files you need. The following command builds only Python.

```
./build_visit3_0_1 --no-thirdparty --no-visit --python
```

9.2.7 Different versions of `build_visit`

When you use a version of `build_visit` that has a version number in it, for example `build_visit3_0_1` then it builds that tagged version of VisIt. If the version of `build_visit` was from the develop branch of VisIt, then it will grab the latest version of VisIt from the devlop branch. If the version of `build_visit` came from a release candidate branch, for example the v3.0 branch, then it will grab the latest version of VisIt from that branch.

9.2.8 Troubleshooting `build_visit` failures

When `build_visit` runs, it generates a log file with `_log` added to the name of the script. For example, if you are running `build_visit3_0_1` then the log file will be named `build_visit3_0_1_log`. The error that caused the failure should be near the end of the log file. When `build_visit` finishes running, it will leave the directories that it used to build the packages intact. You can go into the directory of the package that failed and correct the issue and finish building and installing the package. You can then execute the `build_visit` command again to have it continue the build.

9.2.9 Why can't I use the Qt, Python, VTK, Mesa/GL, etc. that came on my system?

As much as we might like to believe it, large, complex libraries like Qt, Python and VTK are rarely 100% compatible between newer or older versions. Furthermore, for large libraries like these, there are often many, many different installation options for a given platform. It is highly unlikely that a given installation of VTK for example, is not only of a version compatibile with a given release of VisIt but also configured and installed on your system in exactly the way VisIt needs it. In addition, VisIt gets developed and thoroughly tested on specific versions and configurations of various libraries meaning that when users encounter issues in other configurations, we are not always able to reproduce them. In some cases, VisIt developers have had to work-around a bug in a library or, worse, had to patch the actual library itself to address an issue that might be specific to just one platform. Together, these issues result in a situation where VisIt often must be compiled with precisely the libraries it is released on and rarely, if ever, can take advantage of an installation that came as part of the system VisIt is being built on. Lastly, it becomes almost impossible to duplicate and diagnose issues reported by users when users are running VisIt in configurations substantially different from that which is being developed and routinely tested.

9.3 Common Build Scenarios

Building VisIt is an involved process and even with `build_visit`, just determining the correct selection of options can sometimes be daunting. To help, here are the steps used to build VisIt on a collection of different platforms that may serve as a starting point for your system.

In each of the scenarios below, the result is a distribution file that can be used with `visit-install` to install VisIt. Furthermore, in all these scenarios, `build_visit` was used to build the third party libraries and the initial config site file. VisIt was then manually built as outlined by doing an out of source build. The advantage to building VisIt manually is that you have more control over the build and its easier to troubleshoot failures. The advantage to an out of source build is that you can easily restart the build simply by deleting the build directory.

9.3.1 Kickit, a RedHat Enterprise Linux 7 system

`build_visit` was run to generate the third party libraries. In this case all the required and optional libraries build without problem, so `--required --optional` could be used. Also, in this case there wasn't a system MPI installed so `--mpich` was specified to use MPICH. The `--osmesa` flag was also included so that VisIt could do off screen rendering.

```
./build_visit3_0_1 --required --optional --mpich --osmesa --no-visit \
--thirdparty-path /usr/gapps/visit/thirdparty_shared/3.0.1 --makeflags -j4
```

This built the third party libraries and generated a `kickit.cmake` config site file. The `Setup VISITHOME & VISITARCH variables.` section was changed to

```
##
## Setup VISITHOME & VISITARCH variables.
##
SET(VISITHOME /usr/gapps/visit/thirdparty_shared/3.0.1)
SET(VISITARCH linux-x86_64_gcc-4.8)
VISIT_OPTION_DEFAULT(VISIT_SLIVR TRUE TYPE BOOL)
```

VisIt was then manually built with the following steps.

```
tar zxf visit3.0.1.tar.gz
cp kickit.cmake visit3.0.1/src/config-site
cd visit3.0.1
mkdir build
cd build
/usr/gapps/visit/thirdparty_shared/3.0.1/cmake/3.9.3/linux-x86_64_gcc-4.8/bin/cmake \
../src -DCMAKE_BUILD_TYPE:STRING=Release \
-DVISIT_INSTALL_THIRD_PARTY:BOOL=ON \
-DVISIT_ENABLE_XDB:BOOL=ON -DVISIT_PARADIS:BOOL=ON
make -j 4 package
```

9.3.2 Quartz, a Linux X86_64 TOSS3 cluster

`build_visit` was run to generate the third party libraries. In this case the system MPI was used, so information about the system MPI had to be provided with environment variables and the `--parallel` flag had to be specified. In this case, all the required and optional third party libraries build without problem, so `--required --optional` could be used. Also, the system OpenGL implementation was outdated and `--mesagl` had to be included to provide an OpenGL implementation suitable for VisIt. Lastly, the Uintah library was built to enable building the Uintah reader.

```
env PAR_COMPILER=/usr/tce/packages/mvapich2/mvapich2-2.3-gcc-4.9.3/bin/mpicc \
    PAR_COMPILER_CXX=/usr/tce/packages/mvapich2/mvapich2-2.3-gcc-4.9.3/bin/mpicxx \
    PAR_INCLUDE=-I/usr/tce/packages/mvapich2/mvapich2-2.3-gcc-4.9.3/include \
    PAR_LIBS=-lmpl \
./build_visit3_0_1 --required --optional --mesagl --uintah --parallel \
--no-visit --thirdparty-path /usr/workspace/wsa/visit/visit/thirdparty_shared/3.0.1/
↪toss3 \
--makeflags -j16
```

This built the third party libraries and generated a `quartz386.cmake` config site file. The `Setup VISITHOME & VISITARCH variables.` section was changed to

```
##
## Setup VISITHOME & VISITARCH variables.
##
SET(VISITHOME /usr/gapps/visit/thirdparty_shared/3.0.1)
SET(VISITARCH linux-x86_64_gcc-4.8)
VISIT_OPTION_DEFAULT(VISIT_SLIVR TRUE TYPE BOOL)
```

The `Parallel build Setup.` section was changed to

```
##
## Parallel Build Setup.
##
VISIT_OPTION_DEFAULT(VISIT_PARALLEL ON TYPE BOOL)
VISIT_OPTION_DEFAULT(VISIT_MPI_CXX_FLAGS -I/usr/tce/packages/mvapich2/mvapich2-2.3-
↪gcc-4.9.3/include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_C_FLAGS   -I/usr/tce/packages/mvapich2/mvapich2-2.3-
↪gcc-4.9.3/include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LD_FLAGS  "-L/usr/tce/packages/mvapich2/mvapich2-2.3-
↪gcc-4.9.3/lib -Wl,-rpath=/usr/tce/packages/mvapich2/mvapich2-2.3-gcc-4.9.3/lib"␣
↪TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LIBS     mpich mpl)
VISIT_OPTION_DEFAULT(VISIT_PARALLEL_RPATH  "/usr/tce/packages/mvapich2/mvapich2-2.3-
↪gcc-4.9.3/lib")
```

VisIt was then manually built with the following steps.

```
tar zxf visit3.0.1.tar.gz
cp kickit.cmake visit3.0.1/src/config-site
cd visit3.0.1
mkdir build
cd build
/usr/workspace/wsa/visit/visit/thirdparty_shared/3.0.1/toss3/cmake/3.9.3/linux-x86_64_
↪gcc-4.9/bin/cmake \
../src -DCMAKE_BUILD_TYPE:STRING=Release \
-DVISIT_INSTALL_THIRD_PARTY:BOOL=ON -DVISIT_PARADIS:BOOL=ON
make -j 16 package
```

9.3.3 Lassen, a Linux Power9 BlueOS cluster

`build_visit` was run to generate the third party libraries. In this case the system MPI was used, so information about the system MPI had to be provided with environment variables and the `--parallel` flag had to be specified. In this case, a few of the optional third party libraries do not build on the system so all the desired optional third party libraries had to be explicitly listed. Also, the system OpenGL implementation was outdated and `--mesagl` had to be included to provide an OpenGL implementation suitable for VisIt. Lastly, the Uintah library was built to enable building the Uintah reader.

```
env PAR_COMPILER=/usr/tce/packages/spectrum-mpi/spectrum-mpi-rolling-release-gcc-4.9.
↪3/bin/mpicc \
 PAR_COMPILER_CXX=/usr/tce/packages/spectrum-mpi/spectrum-mpi-rolling-release-gcc-4.9.
↪3/bin/mpicxx \
 PAR_INCLUDE=-I/usr/tce/packages/spectrum-mpi/ibm/spectrum-mpi-rolling-release/
↪include \
 ./build_visit3_0_1 \
 --no-thirdparty --no-visit \
 --cmake --python --vtk --qt --qwt \
 --adios --adios2 --advio --boost --cfitsio --cgns --conduit \
 --gdal --glu --h5part --hdf5 --icet --llvm --mfem \
 --mili --moab --mxml --netcdf \
 --silo --szip --vtkm --vtkh --xdmf --zlib \
 --mesagl --uintah --parallel \
 --thirdparty-path /usr/workspace/wsa/visit/visit/thirdparty_shared/3.0.1/blueos \
 --makeflags -j16
```

This built the third party libraries and generated a `lassen708.cmake` config site file. The `Setup VISITHOME & VISITARCH variables.` section was changed to

```
##
## Setup VISITHOME & VISITARCH variables.
##
SET(VISITHOME /usr/workspace/wsa/visit/visit/thirdparty_shared/3.0.1/blueos)
SET(VISITARCH linux-ppc64le_gcc-4.9)
VISIT_OPTION_DEFAULT(VISIT_SLIVR TRUE TYPE BOOL)
```

The `Parallel build Setup.` section was changed to

```
##
## Parallel Build Setup.
##
VISIT_OPTION_DEFAULT(VISIT_PARALLEL ON TYPE BOOL)
VISIT_OPTION_DEFAULT(VISIT_MPI_CXX_FLAGS -I/usr/tce/packages/spectrum-mpi/ibm/
↪spectrum-mpi-rolling-release/include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_C_FLAGS   -I/usr/tce/packages/spectrum-mpi/ibm/
↪spectrum-mpi-rolling-release/include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LD_FLAGS  "-L/usr/tce/packages/spectrum-mpi/ibm/
↪spectrum-mpi-rolling-release/lib -Wl,-rpath=/usr/tce/packages/spectrum-mpi/ibm/
↪spectrum-mpi-rolling-release/lib" TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LIBS     mpi_ibm)
VISIT_OPTION_DEFAULT(VISIT_PARALLEL_RPATH  "/usr/tce/packages/spectrum-mpi/ibm/
↪spectrum-mpi-rolling-release/lib")
```

VisIt was then manually built with the following steps.

```
tar zxf visit3.0.1.tar.gz
cp lassen708.cmake visit3.0.1/src/config-site
cd visit3.0.1
mkdir build
cd build
/usr/workspace/wsa/visit/visit/thirdparty_shared/3.0.1/blueos/cmake/3.9.3/linux-
↪ppc64le_gcc-4.9/bin/cmake \
../src -DCMAKE_BUILD_TYPE:STRING=Release \
-DVISIT_INSTALL_THIRD_PARTY:BOOL=ON
make -j 16 package
```

9.3.4 Cori, a Cray KNL cluster

The system is set up to support the Intel compiler by default so we need to swap out the Intel environment for the GNU environment.

```
module swap PrgEnv-intel/6.0.4 PrgEnv-gnu/6.0.4
```

The Cray compiler wrappers are set up to do static linking, which causes a problem with building parallel hdf5. The linking can be changed to link dynamically by setting a couple of environment variables.

```
export XTPE_LINK_TYPE=dynamic
export CRAYPE_LINK_TYPE=dynamic
```

The linker has a bug that prevents VTK from building, which is fixed with the linker in binutils 2.32. Binutils was then manually built with the following steps.

```
wget https://mirrors.ocf.berkeley.edu/gnu/binutils/binutils-2.32.tar.gz
mkdir /project/projectdirs/visit/thirdparty_shared/3.0.1/binutils
```

(continues on next page)

(continued from previous page)

```
tar zxf binutils-2.32.tar.gz
cd binutils-2.32
./configure --prefix=/project/projectdirs/visit/thirdparty_shared/3.0.1/binutils
make
make install
```

The following lines in `build_visit`

```
vopts="${vopts} -DCMAKE_C_FLAGS:STRING=\"${C_OPT_FLAGS}\""
vopts="${vopts} -DCMAKE_CXX_FLAGS:STRING=\"${CXX_OPT_FLAGS}\""
```

were changed to

```
vopts="${vopts} -DCMAKE_C_FLAGS:STRING=\"${C_OPT_FLAGS} -B/project/projectdirs/visit/
↪thirdparty_shared/3.0.1/binutils/bin\""
vopts="${vopts} -DCMAKE_CXX_FLAGS:STRING=\"${CXX_OPT_FLAGS} -B/project/projectdirs/
↪visit/thirdparty_shared/3.0.1/binutils/bin\""
```

to build VTK with the linker from binutils 2.32.

`build_visit` was run to generate the third party libraries. In this case the system MPI was used, so information about the system MPI had to be provided with environment variables and the `--parallel` flag had to be specified. In this case, all the required and optional third party libraries built without problem, so `--required --optional` could be used. Also, the system OpenGL implementation was outdated and `--mesagl` had to be included to provide an OpenGL implementation suitable for VisIt. Lastly, the Uintah library was built to enable building the Uintah reader.

```
env PAR_COMPILER=/opt/cray/pe/craype/2.5.15/bin/cc \
    PAR_COMPILER_CXX=/opt/cray/pe/craype/2.5.15/bin/CC \
    PAR_INCLUDE=-I/opt/cray/pe/mpt/7.7.3/gni/mpich-gnu/7.1/include \
    PAR_LIBS="-L/opt/cray/pe/mpt/7.7.3/gni/mpich-gnu/7.1/lib -Wl,-rpath=/opt/cray/pe/
↪mpt/7.7.3/gni/mpich-gnu/7.1/lib -lmpich" \
 ./build_visit3_0_1 --required --optional --mesagl --uintah --parallel \
 --no-visit --thirdparty-path /project/projectdirs/visit/thirdparty_shared/3.0.1 \
 --makeflags -j8
```

This built the third party libraries and generated a `cori08.cmake` config site file. The `Setup VISITHOME & VISITARCH variables.` section was changed to

```
##
## Setup VISITHOME & VISITARCH variables.
##
SET(VISITHOME /project/projectdirs/visit/thirdparty_shared/3.0.1)
SET(VISITARCH linux-x86_64_gcc-7.3)
VISIT_OPTION_DEFAULT(VISIT_SLIVR TRUE TYPE BOOL)
```

The `VISIT_C_FLAGS` and `VISIT_CXX_FLAGS` were changed to

```
VISIT_OPTION_DEFAULT(VISIT_C_FLAGS " -m64 -fPIC -fvisibility=hidden -B/project/
↪projectdirs/visit/thirdparty_shared/3.0.1/binutils/bin" TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_CXX_FLAGS " -m64 -fPIC -fvisibility=hidden -B/project/
↪projectdirs/visit/thirdparty_shared/3.0.1/binutils/bin" TYPE STRING)
```

The `Parallel build Setup.` section was changed to

```
##
## Parallel Build Setup.
```

(continues on next page)

(continued from previous page)

```
##
VISIT_OPTION_DEFAULT(VISIT_PARALLEL ON TYPE BOOL)
VISIT_OPTION_DEFAULT(VISIT_MPI_CXX_FLAGS -I/opt/cray/pe/mpt/7.7.3/gni/mpich-gnu/7.1/
↪include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_C_FLAGS   -I/opt/cray/pe/mpt/7.7.3/gni/mpich-gnu/7.1/
↪include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LD_FLAGS  "-L/opt/cray/pe/mpt/7.7.3/gni/mpich-gnu/7.1/
↪lib -Wl,-rpath=/opt/cray/pe/mpt/7.7.3/gni/mpich-gnu/7.1/lib" TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LIBS     mpich)
VISIT_OPTION_DEFAULT(VISIT_PARALLEL_RPATH  "/opt/cray/pe/mpt/7.7.3/gni/mpich-gnu/7.1/
↪lib")
```

VisIt was then manually built with the following steps.

```
tar zxf visit3.0.1.tar.gz
cp cori08.cmake visit3.0.1/src/config-site
cd visit3.0.1
mkdir build
cd build
/project/projectdirs/visit/thirdparty_shared/3.0.1/cmake/3.9.3/linux-x86_64_gcc-7.3/
↪bin/cmake \
../src -DCMAKE_BUILD_TYPE:STRING=Release \
-DVISIT_INSTALL_THIRD_PARTY:BOOL=ON -DVISIT_PARADIS:BOOL=ON
make -j 8 package
```

9.3.5 Summit, a Linux Power9 BlueOS cluster

The system is set up to support the IBM XL compiler by default so we need to swap out the XL compiler for the GNU compiler.

```
module swap xl/16.1.1-3 gcc/6.4.0
```

There was an error building CMake, so we used the system CMake after module loading CMake 3.9.2.

```
module load cmake/3.9.2
```

`build_visit` was run to generate the third party libraries. In this case the system MPI was used, so information about the system MPI had to be provided with environment variables and the `--parallel` flag had to be specified. In this case, a few of the optional third party libraries do not build on the system so all the desired optional third party libraries had to be explicitly listed. Also, the system OpenGL implementation was outdated and `--mesagl` had to be included to provide an OpenGL implementation suitable for VisIt. Lastly, the Uintah library was built to enable building the Uintah reader.

```
env PAR_COMPILER=/autofs/nccs-svm1_sw/summit/.swci/1-compute/opt/spack/20180914/linux-
↪rhel7-ppc64le/gcc-6.4.0/spectrum-mpi-10.3.0.1-20190611-
↪cyaenjgora6now2nusxzkfli4mzjnudx/bin/mpicc \
    PAR_COMPILER_CXX=/autofs/nccs-svm1_sw/summit/.swci/1-compute/opt/spack/20180914/
↪linux-rhel7-ppc64le/gcc-6.4.0/spectrum-mpi-10.3.0.1-20190611-
↪cyaenjgora6now2nusxzkfli4mzjnudx/bin/mpicxx \
    PAR_INCLUDE=-I/autofs/nccs-svm1_sw/summit/.swci/1-compute/opt/spack/20180914/
↪linux-rhel7-ppc64le/gcc-6.4.0/spectrum-mpi-10.3.0.1-20190611-
↪cyaenjgora6now2nusxzkfli4mzjnudx/include \
    ./build_visit3_0_1 \
    --no-thirdparty --no-visit \
```

(continues on next page)

(continued from previous page)

```
    --system-cmake --python --vtk --qt --qwt \
    --adios --adios2 --advio --boost --cfitsio --cgns --conduit \
    --gdal --glu --h5part --hdf5 --icet --llvm --mfem \
    --mili --moab --mxml --netcdf \
    --silo --szip --xdmf --zlib \
    --mesagl --uintah --parallel \
    --thirdparty-path /autofs/nccs-svm1_home1/brugger1/visit/thirdparty_shared/3.0.1 \
    --makeflags -j8
```

This built the third party libraries and generated a `login1.cmake` config site file. The `Setup VISITHOME & VISITARCH variables.` section was changed to

```
##
## Setup VISITHOME & VISITARCH variables.
##
SET(VISITHOME /autofs/nccs-svm1_home1/brugger1/visit/thirdparty_shared/3.0.1)
SET(VISITARCH linux-ppc64le_gcc-6.4)
VISIT_OPTION_DEFAULT(VISIT_SLIVR TRUE TYPE BOOL)
```

The `Parallel build Setup.` section was changed to

```
##
## Parallel Build Setup.
##
VISIT_OPTION_DEFAULT(VISIT_PARALLEL ON TYPE BOOL)
VISIT_OPTION_DEFAULT(VISIT_MPI_CXX_FLAGS -I/autofs/nccs-svm1_sw/summit/.swci/1-
↪compute/opt/spack/20180914/linux-rhel7-ppc64le/gcc-6.4.0/spectrum-mpi-10.3.0.1-
↪20190611-cyaenjgora6now2nusxzkfli4mzjnudx/include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_C_FLAGS   -I/autofs/nccs-svm1_sw/summit/.swci/1-
↪compute/opt/spack/20180914/linux-rhel7-ppc64le/gcc-6.4.0/spectrum-mpi-10.3.0.1-
↪20190611-cyaenjgora6now2nusxzkfli4mzjnudx/include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LD_FLAGS  "-L/autofs/nccs-svm1_sw/summit/.swci/1-
↪compute/opt/spack/20180914/linux-rhel7-ppc64le/gcc-6.4.0/spectrum-mpi-10.3.0.1-
↪20190611-cyaenjgora6now2nusxzkfli4mzjnudx/lib -Wl,-rpath=/autofs/nccs-svm1_sw/
↪summit/.swci/1-compute/opt/spack/20180914/linux-rhel7-ppc64le/gcc-6.4.0/spectrum-
↪mpi-10.3.0.1-20190611-cyaenjgora6now2nusxzkfli4mzjnudx/lib" TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LIBS     mpi_ibm)
VISIT_OPTION_DEFAULT(VISIT_PARALLEL_RPATH  "/autofs/nccs-svm1_sw/summit/.swci/1-
↪compute/opt/spack/20180914/linux-rhel7-ppc64le/gcc-6.4.0/spectrum-mpi-10.3.0.1-
↪20190611-cyaenjgora6now2nusxzkfli4mzjnudx/lib")
```

The compiler didn't like one of the boost header files, so it was manually patched.

```
vi /autofs/nccs-svm1_home1/brugger1/visit/thirdparty_shared/3.0.1/boost/1_67_0/linux-
↪ppc64le_gcc-6.4/include/boost/numeric/interval/detail/ppc_rounding_control.hpp

line 99:
 namespace detail {

 typedef union {
-    ::boost::long_long_type imode;
+    ::boost::ulong_long_type imode;
    double dmode;
 } rounding_mode_struct;
```

VisIt was then manually built with the following steps.

```
tar zxf visit3.0.1.tar.gz
cp login1.cmake visit3.0.1/src/config-site
cd visit3.0.1
mkdir build
cd build
/autofs/nccs-svm1_sw/summit/.swci/0-core/opt/spack/20171006/linux-rhel7-ppc64le/gcc-4.
↪8.5/cmake-3.9.2-lnpnk356fyio3b6rq5bdhr2djjirtsxk/bin/cmake \
../src -DCMAKE_BUILD_TYPE:STRING=Release \
-DVISIT_INSTALL_THIRD_PARTY:BOOL=ON
make -j 8 package
```

9.3.6 Trinity, a Cray KNL cluster

The system is set up to support the Intel compiler by default so we need to swap out the Intel environment for the GNU environment.

```
module swap PrgEnv-intel/6.0.8 PrgEnv-gnu/6.0.8
```

The Cray compiler wrappers are set up to do static linking, which causes a problem with building parallel hdf5. The linking can be changed to link dynamically by setting a couple of environment variables.

```
export XTPE_LINK_TYPE=dynamic
export CRAYPE_LINK_TYPE=dynamic
```

`build_visit` was run to generate the third party libraries. In this case the system MPI was used, so information about the system MPI had to be provided with environment variables and the `--parallel` flag had to be specified. In this case, all the required and optional third party libraries built except for Sphinx and Mili, so `--no-sphinx --no-mili` in addition to `--required --optional` could be used. Also, the system OpenGL implementation was outdated and `--mesagl` had to be included to provide an OpenGL implementation suitable for VisIt.

```
env PAR_COMPILER=/opt/cray/pe/craype/2.7.0/bin/cc \
    PAR_COMPILER_CXX=/opt/cray/pe/craype/2.7.0/bin/CC \
    PAR_INCLUDE=-I/opt/cray/pe/mpt/7.7.15/gni/mpich-gnu/8.2/include \
    PAR_LIBS="-L/opt/cray/pe/mpt/7.7.15/gni/mpich-gnu/8.2/lib -Wl,-rpath=/opt/cray/pe/
↪mpt/7.7.15/gni/mpich-gnu/8.2/lib -lmpich" \
 ./build_visit3_1_3 --required --optional --no-sphinx --no-mili \
 --mesagl --parallel --no-visit \
 --thirdparty-path /usr/projects/views/visit/thirdparty_shared/3.1.3 \
 --makeflags -j6
```

This built the third party libraries and generated a `tr-fe2.cmake` config site file. The `Setup VISITHOME & VISITARCH variables.` section was changed to

```
##
## Setup VISITHOME & VISITARCH variables.
##
SET(VISITHOME /usr/projects/views/visit/thirdparty_shared/3.1.3)
SET(VISITARCH linux-x86_64_gcc-9.3)
VISIT_OPTION_DEFAULT(VISIT_SLIVR TRUE TYPE BOOL)
```

The `Parallel build Setup.` section was changed to

```
##
## Parallel Build Setup.
##
```

(continues on next page)

(continued from previous page)

```
VISIT_OPTION_DEFAULT(VISIT_PARALLEL ON TYPE BOOL)
VISIT_OPTION_DEFAULT(VISIT_MPI_CXX_FLAGS -I/opt/cray/pe/mpt/7.7.15/gni/mpich-gnu/8.2/
↪include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_C_FLAGS   -I/opt/cray/pe/mpt/7.7.15/gni/mpich-gnu/8.2/
↪include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LD_FLAGS  "-L/opt/cray/pe/mpt/7.7.15/gni/mpich-gnu/8.2/
↪lib -Wl,-rpath=/opt/cray/pe/mpt/7.7.15/gni/mpich-gnu/8.2/lib" TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LIBS     mpich)
VISIT_OPTION_DEFAULT(VISIT_PARALLEL_RPATH  "/opt/cray/pe/mpt/7.7.15/gni/mpich-gnu/8.2/
↪lib")
```

VisIt was then manually built with the following steps.

```
tar zxf visit3.1.3.tar.gz
cp tr-fe2.cmake visit3.1.3/src/config-site
cd visit3.1.3
mkdir build
cd build
/usr/projects/views/visit/thirdparty_shared/3.1.3/cmake/3.9.3/linux-x86_64_gcc-9.3/
↪bin/cmake \
../src -DCMAKE_BUILD_TYPE:STRING=Release \
-DVISIT_INSTALL_THIRD_PARTY:BOOL=ON -DVISIT_PARADIS:BOOL=ON
make -j 8 package
```

9.4 Building on Windows

In this chapter, we will discuss how to build visit on Windows.

9.4.1 Prerequisites

VisIt's Source Code

For a released version

For building a released version of VisIt version 3.2.1 or earlier, you can download a windows installer that contains all that is necessary from the downloads page. Look for the *Win 10/8/7 development* link for the particular version you want. The file will be named *visitdev3.2.1.exe* (or similar, based on version chosen).

For building a released version of VisIt version 3.2.2 or newer, download the *Source* tgz file as well as the *Win 10/8/7 development* file which will be named *visit_windowsdev_3.2.2.zip* (or similar, based on version chosen). The .zip file contains all the pre-built third-party binaries needed for building VisIt on Windows. It is best to extract these files to the same folder so that *src* from the source tarball is peer to *windowsbuild* from the windows-dev zip file.

For the latest development version

If you want to build the latest development version from our repository, you need to obtain source from the visit repo, and the pre-built third party dependencies from the visit-deps repo on GitHub.

Other Software

1. CMake version 3.15 or greater.
    - Don't use the CMake included with cygwin if you plan on using the pre-built thirdparty libraries.
2. Visual Studio 2017 64-bit
    - Needed if you want to use our pre-built thirdparty libraries.
3. NSIS *Optional*
    - For creating an installer for VisIt. NSIS 2 or 3 should work.
4. Microsoft MPI. *Optional*
    - For building VisIt's parallel engine. Redistributable binaries and SDK's are needed, so download and install both msmpisdk.msi and msmpisetup.exe.

9.4.2 Configuring With CMake GUI

Run cmake-gui.exe, which will display this window. Figure 9.1

Locating Source and Build Directories

Fill in the location of VisIt's src directory in the *Where is the source code:* section.

Then tell CMake where you want the build to go by filling in *Where to build the binaries*. It is best to create a new build directory somewhere other than inside the *src* or *windowsbuild* directories. This is called *out-of-source build* and it prevents pollution of your src directory.

The Browse buttons come in handy here.

If you are building from a clone of the github repository, it is recommended to do the build in a directory outside the repo (eg peer to *visit*) to keep your checkout clean. Figure 9.2

Location of windowsbuild Directory

For a released version of VisIt's source code, the *windowsbuild* directory containing the pre-built thirdparty binaries is located peer to *src*. CMake generation should locate this directory automatically. Figure 9.3

In order for CMake to locate the directory automatically for a development build cloned or downloaded from the github repositories, *visit-deps* should be peer to *visit*. Figure 9.4

If neither of the above is true for your situation, use the CMake gui to set *VISIT_WINDOWS_DIR* to the location of the *windowsbuild* directory. Figure 9.5

Limiting Plugins

By default, most of the supported database reader plugins are built, which can slow down loading of the solution in the Visual Studio IDE, and slow down the build. If you want to reduce the number of plugins built, add a CMake var using the **Add Entry** Button. If you are producing a version of VisIt that you plan to distribute, you should skip this step so all database reader plugins are built. Figure 9.6

To limit the database plugins to a specific set of plugins, set the **Name:** to *VISIT_SELECTED_DATABASE_PLUGINS*. The **Type:** should be *STRING*. The **Value:** should be a ';' separated list of database plugins names. Case must match the name of the folder in */src/databases*.

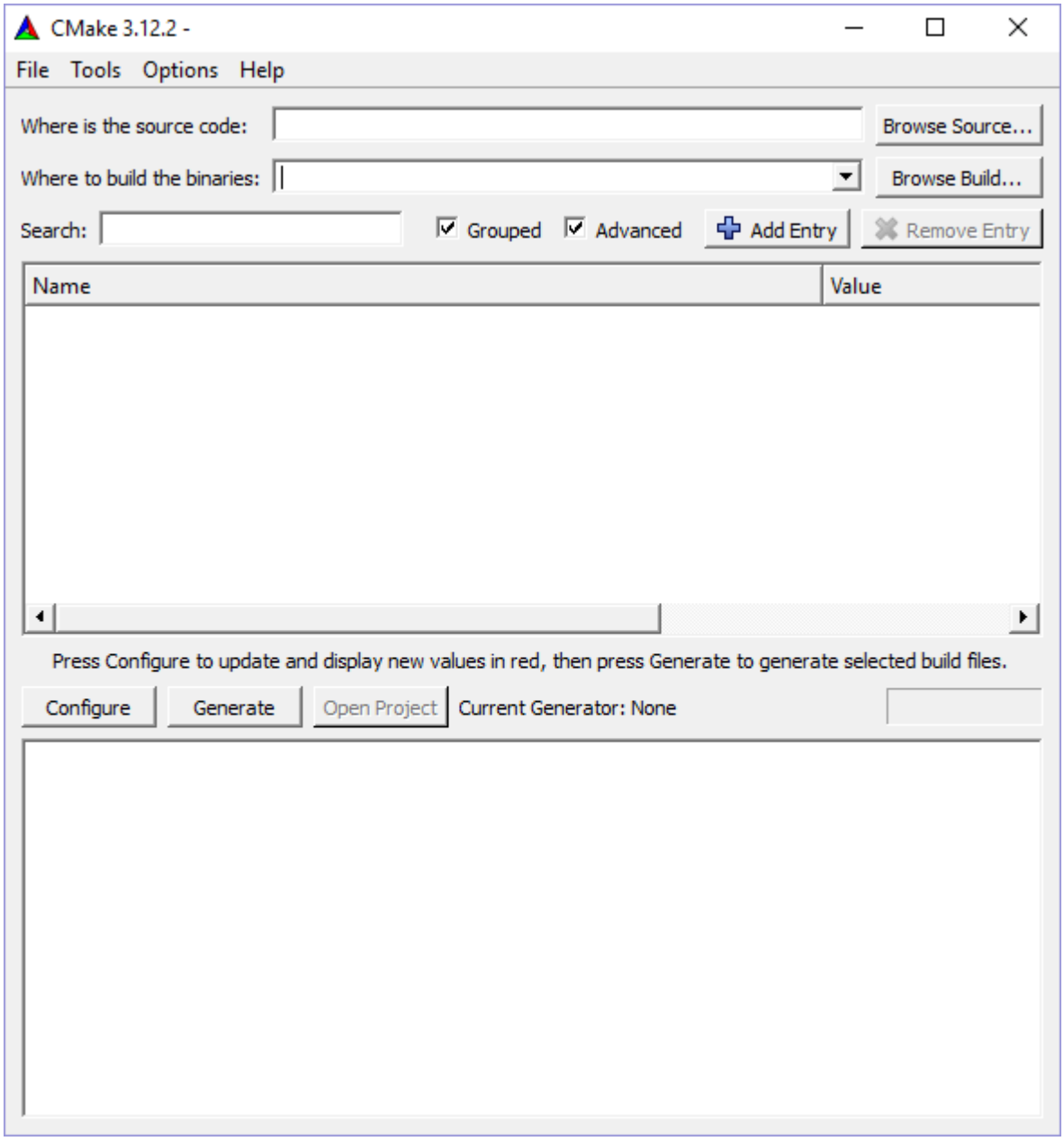


Fig. 9.1: CMake-gui

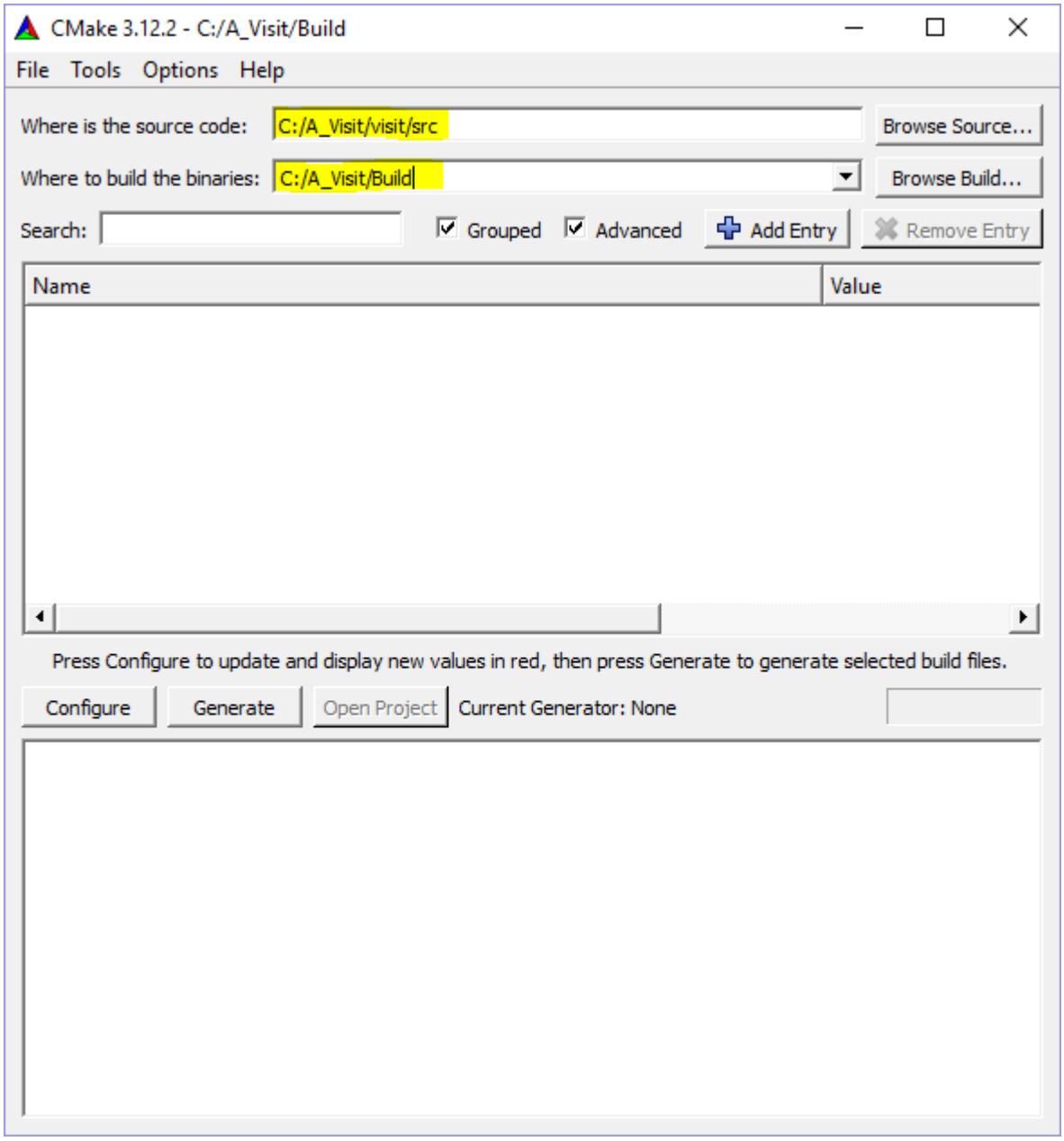


Fig. 9.2: Setting source and build directories

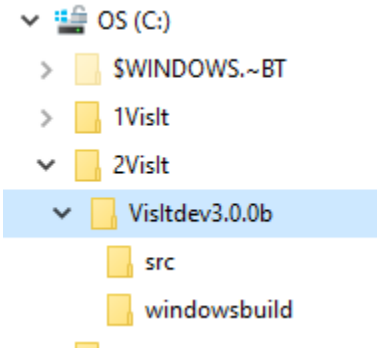


Fig. 9.3: Directory structure with source from a released version

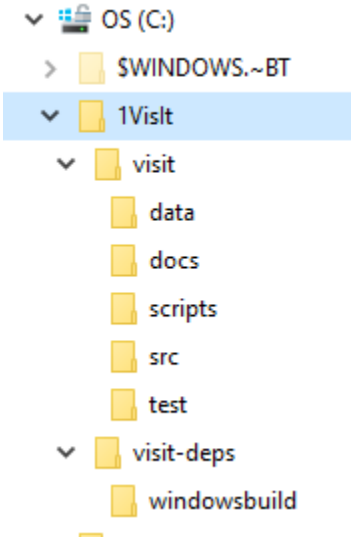


Fig. 9.4: Expected directory structure with source from GitHub repo

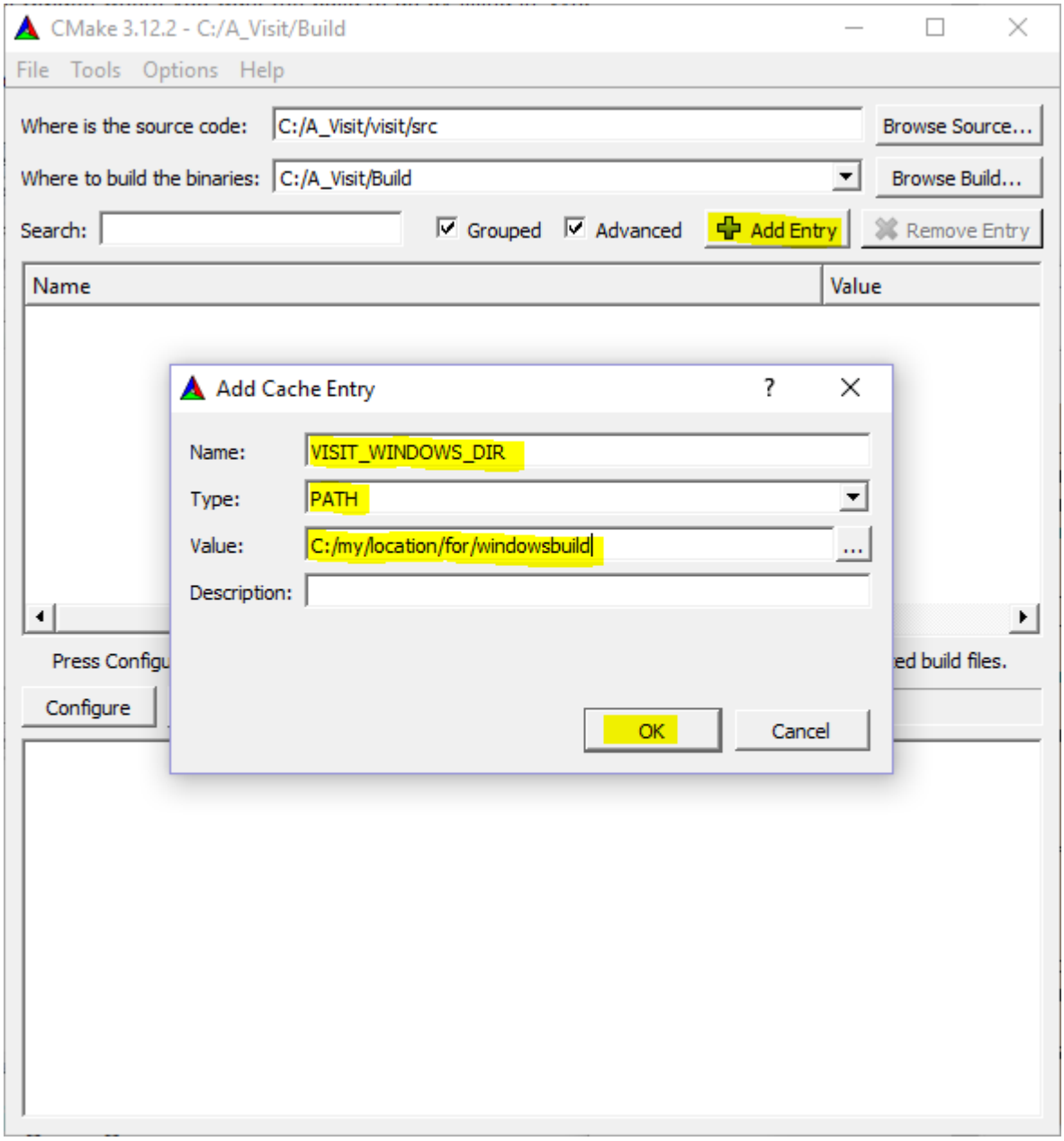


Fig. 9.5: Setting VISIT_WINDOWS_DIR

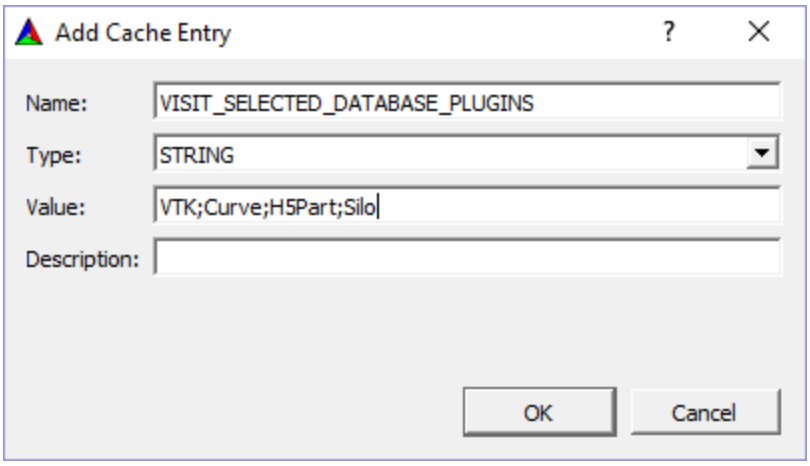


Fig. 9.6: Selecting a limited number of database plugins

The same procedure applies to plots and operators. The VisIt CMake variables to limit plots and operator plugins are *VISIT_SELECTED_PLOT_PLUGINS* and *VISIT_SELECTED_OPERATOR_PLUGINS*, respectively.

Click **OK** when finished.

Configuring

Before configuring, you may want to suppress warnings. From the **Options** menu, choose *Warnings*. Check the *Developer Warnings* and *Deprecated Warnings* in the *Suppress Warnings* section. Click **OK**. Figure 9.7

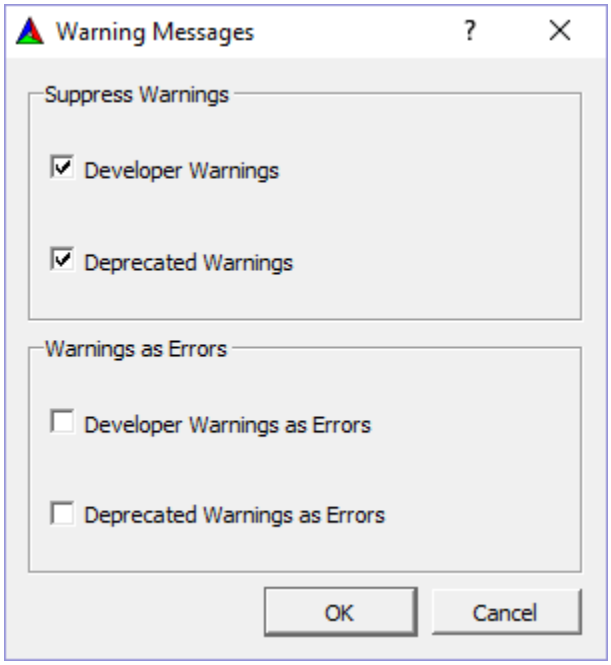


Fig. 9.7: Suppress CMake warnings

In the main CMake Window, click the **Configure** button.

If the build directory does not exist, you will be prompted to allow its creation.

You will also be prompted to choose a *generator*. On Windows, this corresponds to the version of Visual Studio for which you plan to generate a solution and projects.

Currently, only Visual Studio version 2017 64-bit is supported by the prebuilt thirdparty libraries. Choose *Visual Studio 15 2017 Win64* from the generator dropdown. Choose *x64* from the dropdown for the *Optional platform*. Then enter *host=x64* in the *Optional toolset* to use the full 64-bit toolset. Figure 9.8

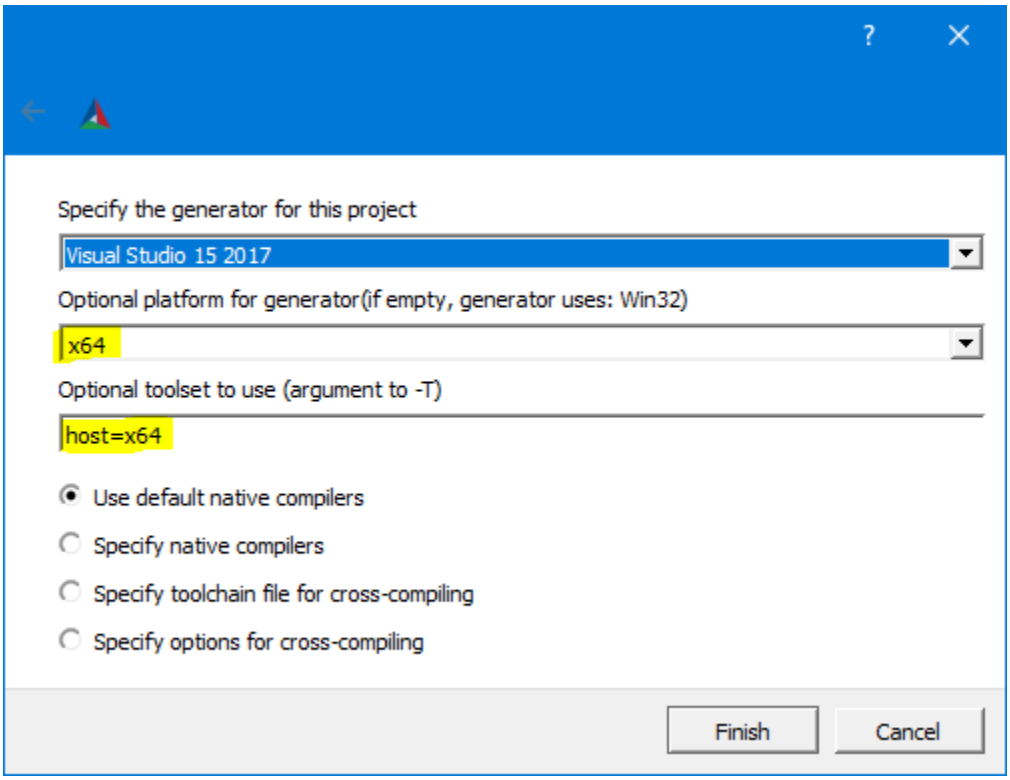


Fig. 9.8: Choosing the generator

CMakeCache entries will be displayed after the initial configure. All entries at this point will be highlighted reddish orange – a signal that you may want to modify some of them. Subsequent clicks of the **Configure** button highlight only entries that contain errors or entries that are new since the last configure.

You can modify how many entries are seen, and how they are viewed by selecting the: **Grouped**, and/or **Advanced** buttons. *Grouped* option groups similarly named items, *Advanced* option shows all the entries. Using both is probably the easiest to navigate for use with VisIt. Mouse-hover over individual entries (not groups) will generate a brief description. Figure 9.9

Most of the default settings should be fine, though you may want to change *CMAKE_INSTALL_PREFIX* from its default location within the Build directory. If you've grouped the entries, click the **+** button next to *CMAKE*, find *CMAKE_INSTALL_PREFIX* and modify it as desired.

See *CMake Variables* for a comprehensive list of settings that can be modified to control aspects of the build.

Parallel

If you have an MPI implementation installed (Microsoft's MPI), you can choose to create a parallel build. Expand the *VISIT* section within the CMake gui, then check the box for *VISIT_PARALLEL*. You will have to scroll to find it.

Click the **Configure** button again to have CMake check the prerequisites for building parallel VisIt. If the prerequisites are met then some new cache entries related to MPI will be created. If not, the MPI entries may have to be modified

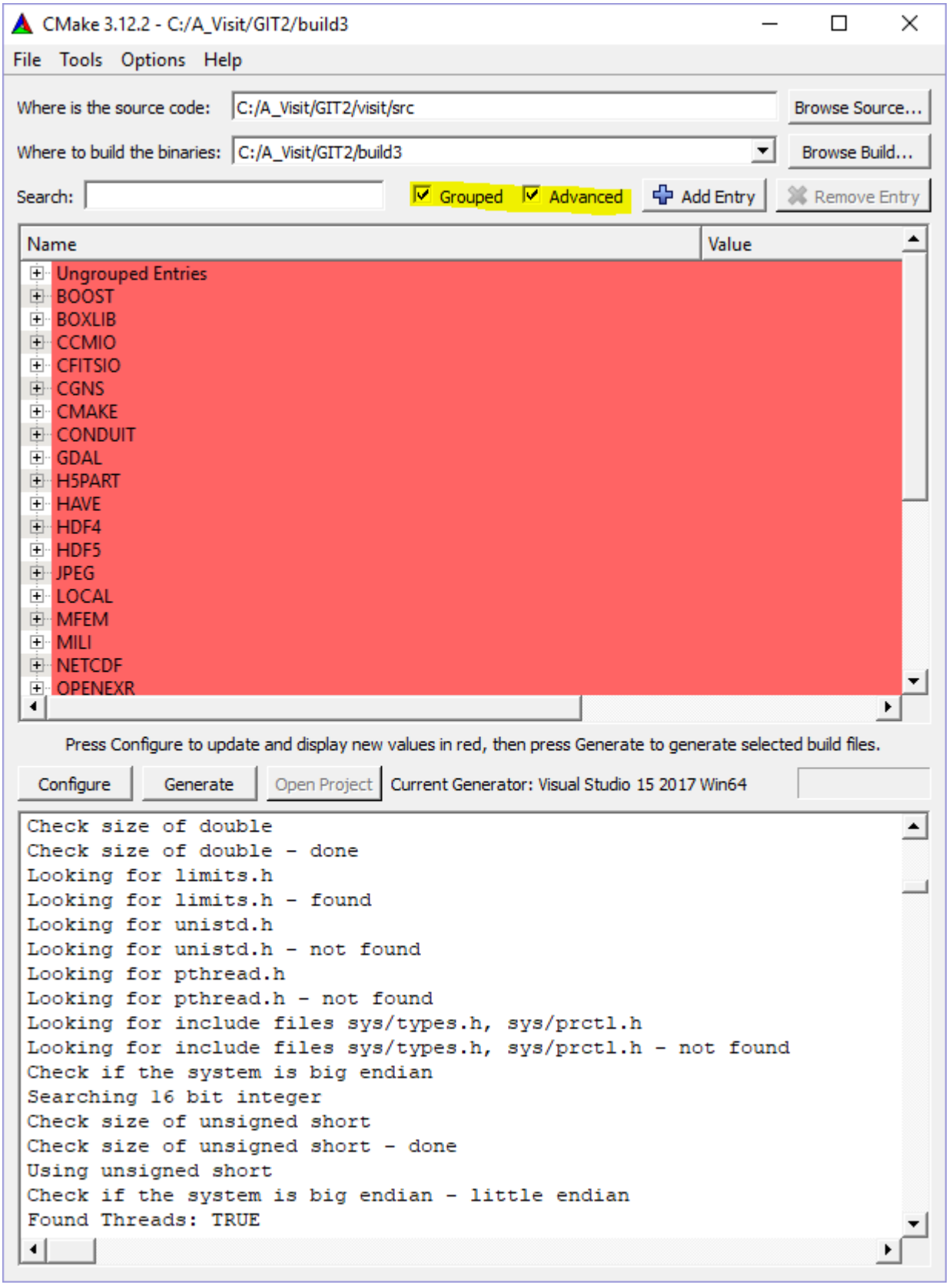


Fig. 9.9: After first configure

by hand.

Suppressing Regeneration

The solution file that CMake creates has a project called *ZERO_CHECK* that is occasionally invoked to regenerate the projects. This can be highly undesirable during development, since it may be triggered during a build and can cause numerous projects to be reloaded into the VS IDE, wasting time unnecessarily. To avoid this behavior, you can create a new CMake cache entry named *CMAKE_SUPPRESS_REGENERATION*, with type *BOOL* and make sure that it is checked. If you made this change click **Configure** again.

You can automate this step in your host.cmake file by adding this line to your host.cmake file:

```
set(CMAKE_SUPPRESS_REGENERATION TRUE)
```

Note that setting this flag means that CMake won't automatically reconfigure from within the VS IDE when changes are made to the build scripts (CMakeLists.txt) or Cache entries. You will have to manually reconfigure. Once reconfigured, Visual Studio will notify you the project files have been modified and prompt you to reload.

Generate

The *Generate* step creates the Visual Studio project and solution files. Make sure any changes made to the cache entries have been *Configured* and that no entries remain red, then click the **Generate** button.

Compile

Open the generated *VisIt.sln* file with Visual Studio (it may take awhile to load all the project files). Select the desired Configuration and Build the solution.

Note: if VISIT_CREATE_XMLTOOLS_GEN_TARGETS is ON, choose the *ALL_BUILD* project to build instead of the *Solution* . More information on this variable can found in the *CMake Variables* section.

9.5 Building on macOS with masonry

In this chapter, we will discuss how to build VisIt on macOS using **masonry**. Masonry is a collection of Python scripts and JSON files that use `build_visit`, and other system tools, to create a macOS Disk Image File (DMG).

9.5.1 Setup

Masonry Scripts

The masonry scripts are bundled with VisIt's source code. You will need to download the source code and extract masonry from `visit/src/tools/dev`. There are a few options for downloading the source code. If you want a released version of VisIt then go to the source code downloads page and look for the *VisIt sources* link. The other option is to download from the git repository. Once you have the source code, copy `visit/src/tools/dev/masonry` to a location of your choosing.

Configuration

1. In the `opts` directory copy one of the `*.json` files and rename it as desired. For example: `cp mb-3.1.1-darwin-10.14-x86_64-release.json mb-3.1.2-darwin-10.14-x86_64-release.json`

2. Open the JSON configuration file (see Figure 9.10) created in **step 1** and modify or add the following options as needed:

arch: required The build architecture (e.g., darwin-x86_64).

boost_dir: optional The path to boost if installed on your system. This also triggers the setting of two CMake options (**VISIT_USE_BOOST:BOOL** and **BOOST_ROOT:PATH**).

branch: required The git branch to checkout and build.

build_dir: optional The directory to place all of the files generated from the build process. If this option isn't specified the build directory will default to `build-<json_base>` (e.g., build-mb-3.1.2-darwin-10.14-x86_64-release) in your current working directory.

build_types: required A list of builds for masonry to create.

build_visit: required Allows you to set the `build_visit` options.

`cmake_ver`: **required** - the CMake version to use

`args`: **optional** - arguments for `build_visits`

`libs`: **optional** - third-party libraries to build

`make_flags`: **optional** - Make flags

build_xdb: optional Set the **VISIT_ENABLE_XDB:BOOL** option to *ON* if true.

cert: required for signing/notarization The Developer ID signing certificate **Common Name**.

cmake_extra_args: optional Specify extra arguments for CMake.

config_site: optional Specify a path for the config site file.

cxx_compiler: optional Specify the C++ compiler

c_compiler: optional Specify the C compiler

entitlements: required for notarization Specify the location of VisIt's entitlements file. The one used for VisIt releases is located in the `opts` directory and is named **visit.entitlements**. See Hardened Runtime for more details on entitlements.

force_clean: optional Removes all files and directories from your build folder.

git: required `mode`: **required** - set this option to **ssh** or **https**

`git_uname`: **optional** - github username

`depth`: **optional** - specify an integer value for a shallow clone with a history truncated to the specified number of commits.

make_nthreads: optional The number of parallel threads to use when building the source code.

notarize: required for notarization Specify the options needed for notarization.

`username`: - Apple ID email

`password`: - App-specific password or keychain string containing the App-specific password

`asc_provider`: - Provider short name

`bundle_id`: - VisIt's bundle identifier

platform: optional Specify the platform (**osx** or **linux**)

skip_checkout: optional if you have to restart masonry and already have the source code checked out you can skip that step by setting this option to *yes*.

tarball: optional Specify the path to the source tar file. This option is currently not being used.

`version`: required The version of VisIt you are building.

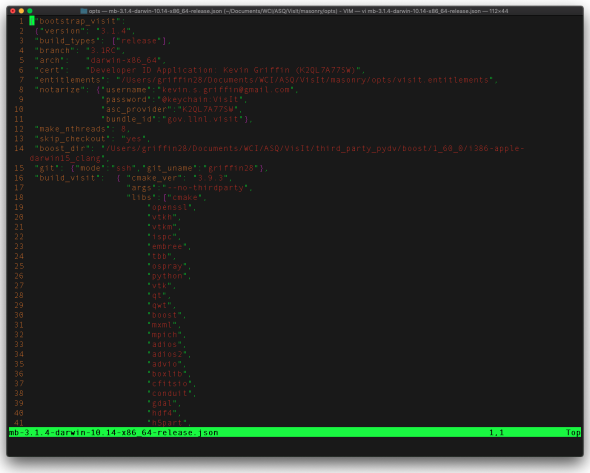


Fig. 9.10: Masonry's JSON config file

You Might Have to Adjust your PATH

There are cases where individual package's configuration processes wind up executing tools in your path (`$PATH`) to help set variables used in compilation. For example, the `MPICH` configuration winds up using `pkg-config`, if it is found in your path, to help set `CPPFLAGS` and `LDFLAGS`. In one case, `libxml2` was installed in both `/opt/local/lib` and in `/usr/lib` but because `/opt/local/bin` was first in `$PATH`, `/opt/local/bin/pkg-config` was used to set paths for `libxml2` to `/opt/local/include` and `/opt/local/lib`. This results in a build of `MPICH` using non-system `libxml2` which will fail the `macOS` (`otool -L`) `@rpath` validation checks on each of the shared libs VisIt is built with later.

The solution is to be sure your `$PATH` variable is set such that nothing *non-standard* can be found in your path prior to running masonry. One way to do this is to `echo $PATH | tr ':' '\n'` to display, line-by-line, each path defined in `$PATH` and then remove from that list all *non-standard* (for example, anything beginning with `/opt`) paths

```
set STD_PATH=`echo $PATH | tr ':' '\n' | grep -v '/opt' | tr '\n' ':'`
env PATH=$STD_PATH python3 src/tools/dev/masonry/bootstrap_visit.py src/tools/dev/
↪masonry/opts/mb-3.3.0-darwin-10.14-x86_64-release.json
```

What Can Go Wrong?

Apart from commonly encountered issues building each third party library built by a `bv_xxx.sh` script, there are many other ways getting the macOS release ready can fail.

- There are at least 3 levels of *certifications* (from your macOS keychain) involved.
  - The Apple *Root* Certificate Authority (CA) certificates (`Apple Root CA` or `Apple Root CA - G2` or `Apple Root CA - G3`
  - Apple Worldwide Developer Relations Certificate Authority certificates (also called the *intermediate* certificate authority)
  - Developer ID Certificate Authority for the organization (Lawrence Livermore) and yourself. These all need to be correctly configured and tied to each other from the lowest level (also called a *leaf* certificate), yourself to the root.

- Sometimes Apple expires its certificates and you may need to go get updated certificates.
- Sometimes your own certificate can expire. Currently, Charles Heizer is the LLNL point of contact for adding developers and updating their expired certificates.
- You might need to *evaluate* the validity of your certificate using Apple KeyChain Certificate Assistant to confirm its all working.
- If you are VPN'd into LLNL, codesigning and notorizing a release may fail.
- If you have MacPorts, Homebrew, Fink or other macOS package managers, python package builds may wind up enabling (and then creating a release that is dependent upon) libraries that are available only to users with similar package managers installed. Worse, you won't have any idea this has happend until you give the release to another developer who has a mac that is not using said package managers and they try to use it and it doesn't work due to missing libraries. You can use `otool` combined with `find` to try to find any cases where the release has such dependences. For example, the command

```
find /Volumes/VisIt-3.3.0/VisIt.app/Contents/Resources/3.3.0/darwin-x86_64 -name
↪'*.so -o -name '*.dylib' -exec otool -L {} \; -print | grep -v rpath
```

  will find cases of non-rpath'd library dependencies build into either shared (`.so`) or dynamic (`.dylib`) libraries. Note, however, that sometimes Python EGG's contained Zip-compressed `.so` files that this check won't find.

  Just removing associated stuff from your `$PATH` will not prevent these build dependencies. Fixing them likely means finding some of the individual packages in `bv_python.sh` and adding `site.cfg` files or otherwise finding build switches that explicitly disable the features creating the need for these dependencies.
- Sometimes, a python package winds up using the python interpreter in `Xcode` instead of the one built for the release of VisIt you are preparing. For example, Sphinx can wind up getting installed with all command-line scripts using a shebang which is an absolute path to `Xcode`'s python interpreter. We've added patching code to `bv_python.sh` to help correct for this.
- You can check whether `VisIt.app` is properly codesigned using this command

```
codesign -v -v /Volumes/VisIt-3.3.0/VisIt.app
```

  which should produce output like so...

```
/Volumes/VisIt-3.3.0/VisIt.app: valid on disk
/Volumes/VisIt-3.3.0/VisIt.app: satisfies its Designated Requirement
```

- You can check whether `VisIt.app` is properly notarized using this command

```
spctl -a -t exec -vv /Volumes/VisIt-3.3.0/VisIt.app
```

  which should produce output like so...

```
/Volumes/VisIt-3.3.0/VisIt.app: accepted
source=Notarized Developer ID
origin=Developer ID Application: Lawrence Livermore National Laboratory␣
↪(A827VH86QR)
```

- You can get more details about why a notorization failed using the command

```
xcrun altool --notarization-info ``uuid`` --username ``username-email`` --password␣
↪@keychain:VisIt
```

  where `uuid` (also called the *request identifier*) is the id you get (in email or printed by masonry to the logs) and `username-email` is your Apple developer ID email address. And, this should produce output like so...

```
No errors getting notarization info.

          Date: 2022-07-18 18:59:44 +0000
          Hash: af5e1231bae06e051e5f1a0cfa37219ec4cb5ec2f76971557d9389f1d8edcadd
    LogFileURL: https://osxapps-ssl.itunes.apple.com/itunes-assets/Enigma122/v4/
↪d6/a7/08/d6a7083b-c6d7-6e8f-8a3c-0db0d406d3da/developer_log.json?
↪accessKey=1658530108_1173406174696701459_H
↪%2BzLM3o4PjKeGN8jgdqPBTVz5wUY5BMY7R7Lvh8UrBmn6kxu%2F4XVdkvSsUE5wrAlbmW%2FCCLSRzU
↪%2FUHBGNhgOa1DZDoXQXtXOAUx3sk1ptivix7dQhzQa11SU9EnbESN6PkZ5je9g4IOrqMdibbB7hhxhTgMvQhiKEO9BXjn
↪%3D
   RequestUUID: 1a8b8756-9c4c-4908-b6db-387a5144ce43
        Status: invalid
   Status Code: 2
Status Message: Package Invalid
```

Cut and paste the `LogFileURL` into a browser to go examine the results. Doing so for the above example yielded a json page like so…


```
{
  "logFormatVersion": 1,
  "jobId": "1a8b8756-9c4c-4908-b6db-387a5144ce43",
  "status": "Invalid",
  "statusSummary": "Archive contains critical validation errors",
  "statusCode": 4000,
  "archiveFilename": "VisIt.dmg",
  "uploadDate": "2022-07-18T18:59:44Z",
  "sha256": "af5e1231bae06e051e5f1a0cfa37219ec4cb5ec2f76971557d9389f1d8edcadd",
  "ticketContents": null,
  "issues": [
    {
      "severity": "error",
      "code": null,
      "path": "VisIt.dmg/VisIt.app/Contents/Resources/3.3.0/darwin-x86_64/lib/
↪python/lib/python3.7/site-packages/Pillow-7.1.2-py3.7-macosx-10.15-x86_64.egg/
↪PIL/_webp.cpython-37m-darwin.so",
      "message": "The binary is not signed.",
      "docUrl": null,
      "architecture": "x86_64"
    },
    {
      "severity": "error",
      "code": null,
      "path": "VisIt.dmg/VisIt.app/Contents/Resources/3.3.0/darwin-x86_64/lib/
↪python/lib/python3.7/site-packages/Pillow-7.1.2-py3.7-macosx-10.15-x86_64.egg/
↪PIL/_webp.cpython-37m-darwin.so",
      "message": "The signature does not include a secure timestamp.",
      "docUrl": null,
      "architecture": "x86_64"
    },
    {
      "severity": "error",
      "code": null,
      "path": "VisIt.dmg/VisIt.app/Contents/Resources/3.3.0/darwin-x86_64/lib/
↪python/lib/python3.7/site-packages/Pillow-7.1.2-py3.7-macosx-10.15-x86_64.egg/
↪PIL/_imagingft.cpython-37m-darwin.so",
      "message": "The binary is not signed.",
      "docUrl": null,
```


(continues on next page)

(continued from previous page)

```
      "architecture": "x86_64"
    },
    {
      "severity": "error",
      "code": null,
      "path": "VisIt.dmg/VisIt.app/Contents/Resources/3.3.0/darwin-x86_64/lib/
↪python/lib/python3.7/site-packages/Pillow-7.1.2-py3.7-macosx-10.15-x86_64.egg/
↪PIL/_imagingft.cpython-37m-darwin.so",
      "message": "The signature does not include a secure timestamp.",
      "docUrl": null,
      "architecture": "x86_64"
    }
  ]
}
```

Signing macOS Builds

To code sign your VisIt build, you must be enrolled in the Apple Developer Program and have a valid Developer ID certificate. Below are simple steps to get started, reference the links for more detailed information.

1. Enroll in the Apple Developer Program, if needed, and create your Developer ID certificates.
2. Install Apple certificates into your keychain
   - From **Xcode** go to the account preferences (`Xcode->Preferences->Account`) and select the **Manage Certificates...** button.
   - Click the **+** to add your certificates (see Figure 9.11).
3. Add the Developer ID signing certificate **Common Name** to the **cert** option in the masonry JSON configuration file.

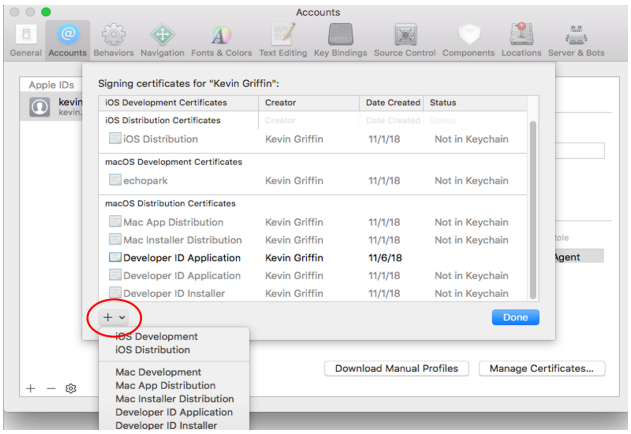


Fig. 9.11: Xcode Manage Certificates Dialog

Warning: Remain disconnected from VPN when building and code signing a VisIt release. The code signing process talks to Apple servers and validates credentials with them. If you are on VPN, the validation may fail with a message similar to:

```
Certificate trust evaluation did not return expected result. (5)  [leaf AnchorApple␣
↪ChainLength CheckIntermediateMarkerOid CheckLeafMarkersProdAndQA]
Certificate trust evaluation for api.apple-cloudkit.com did not return expected␣
↪result. No error..
Certificate trust evaluation did not return expected result. (5)  [leaf AnchorApple␣
↪ChainLength CheckIntermediateMarkerOid CheckLeafMarkersProdAndQA]
Certificate trust evaluation for api.apple-cloudkit.com did not return expected␣
↪result. No error..
Could not establish secure connection to api.apple-cloudkit.com
```

Read more about Apple's Code Signing documentation.

App-Specific Password

To create an app-specific password go to: https://appleid.apple.com/account/manage . Generate the app-specific password by navigating to: *Security->App-Specific Password*.

To avoid having a plain-text password in your config file, you can add the app-specific password to your macOS keychain. To do this, run the following command:

```
security add-generic-password -a "apple-id-email" -w "app-specific password"
-s "notarizing-name"
```

The `-s` parameter is the name that this item will have in your keychain. Apple's documentation on Customizing the Notarization Workflow provides a good overview of the notarization process and a link detailing how to generate and manage app-specific passwords.

9.5.2 Running Masonry Scripts

bootstrap_visit.py

The `bootstrap_visit.py` file contains all of the logic to execute the necessary steps for creating the macOS Disk Image File (DMG). It takes the JSON configuration file as an argument:

```
python3 bootstrap_visit.py opts/<file-name>.json
```

masonry_view_log.py

Once masonry is running, it will produce log files in the `_logs` directory. To view the logs in HTML format (see Figure 9.12), run the `masonry_view_log.py` script. This script takes the log file as an argument:

```
python3 masonry_view_log.py _logs/<log-fle>.json
```

The script will launch a web browser to connect to a local web server. If you already have a web browser running on your system the script will use it. In this situation that web browser may not be able to connect to the local web server. If this happens you should exit your existing web browser and try again.

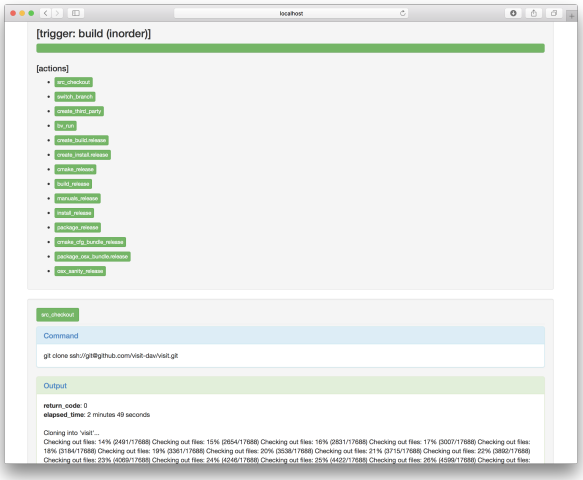


Fig. 9.12: Mansonry Logs in HTML format

9.6 Building with Spack

9.6.1 Overview

Spack is a multi-platform package manager that builds and installs multiple versions and configurations of software. It works on Linux, macOS, and many supercomputers. Spack is non-destructive: installing a new version of a package does not break existing installations, so many configurations of the same package can coexist.

Spack offers a simple "spec" syntax that allows users to specify versions and configuration options. Package files are written in pure Python, and specs allow package authors to write a single script for many different builds of the same package. With Spack, you can build your software all the ways you want to.

The complete documentation can be found here.

9.6.2 The VisIt spack package

The spack package for VisIt currently only builds a subset of the libraries that `build_visit` builds. It builds the core libraries needed by VisIt and the following I/O libraries:

- HDF5
- Silo

The rest of the I/O libraries needed by the VisIt readers are currently not built. The expectation is that the list of I/O libraries built will grow over time.

9.6.3 Building VisIt with spack

The first step is to clone spack from GitHub.

```
# Using https.
git clone https://github.com/spack/spack
```

(continues on next page)

(continued from previous page)

```
# Using ssh.
git clone ssh://git@github.com/spack/spack
```

Now you need to set up your spack environment.

```
cd spack

# bash shell
. share/spack/setup-env.sh

# c shell
source share/spack/setup-env.csh
```

Now you need to have spack find the compilers available on your system.

```
spack compilers
```

This will also create a `.spack` directory in your home directory. The `.spack` directory also contains a `compilers.yaml` file, which contains properties about your compilers. You can modify the `compilers.yaml` file if you need to change some properties of one or more of your compilers, such as additional rpaths. This is also the directory that you would put a `package.yaml` file for specifying the version of mpi to use.

If there is a compiler you want to use and spack didn't find it, you can tell spack about the compiler.

```
spack compiler find path_to_compiler_install_dir
```

If you would like to find a list of all the packages that spack will build you can use the spec command.

```
spack spec visit ^python+shared ^mesa+opengl
```

To build VisIt with the default compiler.

```
spack install visit ^python+shared ^mesa+opengl
```

To build VisIt with a specific compiler.

```
spack install visit%gcc@11.2.0 ^python+shared ^mesa+opengl
```

To build VisIt and specify a specific version of a dependent library.

```
spack install visit%gcc@11.2.0 ^python+shared ^mesa+opengl ^llvm@11.0.1
```

To uninstall a package (the `--dependents` uninstalls all the packages that depend on the uninstalled packages).

```
spack uninstall --dependents visit vtk
```

To uninstall all the packages.

```
spack uninstall --all
```

The package files for a package are located in `var/spack/repos/builtin/package`. Here are some examples.

```
./var/spack/repos/builtin/packages/visit
./var/spack/repos/builtin/packages/vtk
```

The installed packages are stored in `opt/spack/<architecture>/<compiler>`. Here are some examples.

```
opt/spack/cray-sles15-zen2/gcc-11.2.0/hdf5-1.8.22-c3djozhlmrvy7wpu46f36qeakemiactw
opt/spack/cray-sles15-zen2/gcc-11.2.0/cmake-3.14.7-nnahgnkkl2d2ty2us46we75pnjepci35
```

9.6.4 Building VisIt with the development version of spack

The following 3 patches should be applied. These are patches to *package.py* files that cannot be patched as part of the VisIt spack package.

```
diff --git a/var/spack/repos/builtin/packages/libxcb/package.py b/var/spack/repos/
↪builtin/packages/libxcb/package.py
index 1db0f5de5a..f19f1856de 100644
--- a/var/spack/repos/builtin/packages/libxcb/package.py
+++ b/var/spack/repos/builtin/packages/libxcb/package.py
@@ -50,3 +50,4 @@ def configure_args(self):

     def patch(self):
         filter_file("typedef struct xcb_auth_info_t {", "typedef struct {", "src/xcb.
↪h")
+        filter_file("python python2 python3", "python3", "configure")
```

```
diff --git a/var/spack/repos/builtin/packages/visit/package.py b/var/spack/repos/
↪builtin/packages/visit/package.py
index 290280e17d..b042998979 100644
--- a/var/spack/repos/builtin/packages/visit/package.py
+++ b/var/spack/repos/builtin/packages/visit/package.py
@@ -72,7 +72,7 @@ class Visit(CMakePackage):
     version("3.0.1", sha256=
↪"a506d4d83b8973829e68787d8d721199523ce7ec73e7594e93333c214c2c12bd")

     root_cmakelists_dir = "src"
-    generator("ninja")
+    #generator("ninja")

     variant("gui", default=True, description="Enable VisIt's GUI")
     variant("osmesa", default=False, description="Use OSMesa for off-screen CPU␣
↪rendering")
@@ -99,7 +99,7 @@ class Visit(CMakePackage):
     patch("visit32-missing-link-libs.patch", when="@3.2")

     # Exactly one of 'gui' or 'osmesa' has to be enabled
-    conflicts("+gui", when="+osmesa")
+    #conflicts("+gui", when="+osmesa")

     depends_on("cmake@3.14.7:", type="build")

@@ -264,24 +264,24 @@ def cmake_args(self):
         args.extend(
             [
                 self.define("VISIT_USE_X", "glx" in spec),
-                self.define("VISIT_MESAGL_DIR", "IGNORE"),
-                self.define("VISIT_OPENGL_DIR", "IGNORE"),
-                self.define("VISIT_OSMESA_DIR", "IGNORE"),
                 self.define("OpenGL_GL_PREFERENCE", "LEGACY"),
                 self.define("OPENGL_INCLUDE_DIR", spec["gl"].headers.directories[0]),
                 self.define("OPENGL_glu_LIBRARY", spec["glu"].libs[0]),
```

(continues on next page)

(continued from previous page)

```
            ]
        )
        if "+osmesa" in spec:
+           args.append(self.define("VISIT_MESAGL_DIR", spec["mesa"].prefix))
+           if '+llvm' in spec['mesa']:
+               args.append(self.define('VISIT_LLVM_DIR', spec['libllvm'].prefix))
+       else:
            args.extend(
                [
-                   self.define("HAVE_OSMESA", True),
-                   self.define("OSMESA_LIBRARIES", spec["osmesa"].libs[0]),
-                   self.define("OPENGL_gl_LIBRARY", spec["osmesa"].libs[0]),
+                   self.define("VISIT_MESAGL_DIR", "IGNORE"),
+                   self.define("VISIT_OPENGL_DIR", "IGNORE"),
+                   self.define("VISIT_OSMESA_DIR", "IGNORE"),
+                   self.define("OPENGL_gl_LIBRARY", spec["gl"].libs[0])
                ]
            )
-       else:
-           args.append(self.define("OPENGL_gl_LIBRARY", spec["gl"].libs[0]))

        if "+hdf5" in spec:
            args.append(self.define("HDF5_DIR", spec["hdf5"].prefix))
```

```
diff --git a/var/spack/repos/builtin/packages/vtk/package.py b/var/spack/repos/
↪builtin/packages/vtk/package.py
index c7bec82c74..d87f61ea0b 100644
--- a/var/spack/repos/builtin/packages/vtk/package.py
+++ b/var/spack/repos/builtin/packages/vtk/package.py
@@ -61,7 +61,7 @@ class Vtk(CMakePackage):
     patch("xdmf2-hdf51.13.2.patch", when="@9:9.2 +xdmf")

     # We cannot build with both osmesa and qt in spack
-    conflicts("+osmesa", when="+qt")
+    #conflicts("+osmesa", when="+qt")

     with when("+python"):
         # Depend on any Python, add bounds below.
```

Building on Frontier

You will first need to copy the *compilers.yaml* and *packages.yaml* files from *scripts/spack/configs/olcf/frontier/* to your *.spack* directory in your home directory.

In order to have spack install the packages in the User Managed Software space the following patch will need to be applied.

```
diff --git a/etc/spack/defaults/config.yaml b/etc/spack/defaults/config.yaml
index 43f8a98dff..e9560a9304 100644
--- a/etc/spack/defaults/config.yaml
+++ b/etc/spack/defaults/config.yaml
@@ -17,7 +17,7 @@ config:
   # This is the path to the root of the Spack install tree.
   # You can use $spack here to refer to the root of the spack instance.
   install_tree:
```

(continues on next page)

(continued from previous page)

```
-    root: $spack/opt/spack
+    root: /sw/frontier/ums/ums022
     projections:
       all: "{architecture}/{compiler.name}-{compiler.version}/{name}-{version}-{hash}
↪"
     # install_tree can include an optional padded length (int or boolean)
```

The following spack command is used to build VisIt.

```
spack install visit@3.3.3%gcc@11.2.0+mpi+gui+osmesa+vtkm ^python@3.7.7+shared ^
↪mesa@21.2.5+opengl ^vtk@8.1.0+osmesa ^kokkos@3.7.01 ^vtk-m@1.9.0+kokkos+rocm~
↪openmp+fpic amdgpu_target=gfx90a
```

The installation will fail to install some shared libraries in the VisIt lib directory. The following script will copy the necessary libraries.

```
#!/bin/bash
cp /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/libtiff-4.5.0-
↪ir2ffe7vygycyfrjz7efnohvyk7vfxnw/lib64/libtiff.so.6.0.0 /sw/frontier/ums/ums022/
↪linux-sles15-zen3/gcc-11.2.0/visit-3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/
↪linux-x86_64/lib
ln -s libtiff.so.6.0.0 /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/visit-3.3.
↪3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib/libtiff.so.6
cp /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/kokkos-3.7.01-
↪wm7zn4cuywfzttqg4o3xk454zulq6ebp/lib64/libkokkoscontainers.so.3.7.01 /sw/frontier/
↪ums/ums022/linux-sles15-zen3/gcc-11.2.0/visit-3.3.3-
↪zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib
cp /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/kokkos-3.7.01-
↪wm7zn4cuywfzttqg4o3xk454zulq6ebp/lib64/libkokkoscore.so.3.7.01 /sw/frontier/ums/
↪ums022/linux-sles15-zen3/gcc-11.2.0/visit-3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.
↪3.3/linux-x86_64/lib
cp /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/kokkos-3.7.01-
↪wm7zn4cuywfzttqg4o3xk454zulq6ebp/lib64/libkokkossimd.so.3.7.01 /sw/frontier/ums/
↪ums022/linux-sles15-zen3/gcc-11.2.0/visit-3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.
↪3.3/linux-x86_64/lib
ln -s libkokkoscontainers.so.3.7.01 /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.
↪2.0/visit-3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib/
↪libkokkoscontainers.so.3.7
ln -s libkokkoscontainers.so.3.7 /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/
↪visit-3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib/
↪libkokkoscontainers.so
ln -s libkokkoscore.so.3.7.01 /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/
↪visit-3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib/libkokkoscore.
↪so.3.7
ln -s libkokkoscore.so.3.7 /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/visit-
↪3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib/libkokkoscore.so
ln -s libkokkossimd.so.3.7.01 /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/
↪visit-3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib/libkokkossimd.
↪so.3.7
ln -s libkokkossimd.so.3.7 /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/visit-
↪3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib/libkokkossimd.so
cp /opt/rocm-5.2.0/lib/libamdhip64.so.5.2.50200 /sw/frontier/ums/ums022/linux-sles15-
↪zen3/gcc-11.2.0/visit-3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib
ln -s libamdhip64.so.5.2.50200 /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/
↪visit-3.3.3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib/libamdhip64.so.5
ln -s libamdhip64.so.5 /sw/frontier/ums/ums022/linux-sles15-zen3/gcc-11.2.0/visit-3.3.
↪3-zfoh2caq5tbshlvtujditymjizstvewe/3.3.3/linux-x86_64/lib/libamdhip64.so
```

Building on Perlmutter

You will first need to copy the *packages.yaml* files from *scripts/spack/configs/nersc/perlmutter/* to your *.spack* directory in your home directory.

The following spack command is used to build with spack.

```
spack install visit@3.3.3%gcc@11.2.0+mpi+gui+osmesa~vtkm ^python@3.8.17+shared ^
↪mesa@21.2.5+opengl ^llvm@11.0.1 ^vtk@8.1.0+osmesa ^silo@4.11 ^libfabric@1.15.2.0 ^
↪adios2~libcatalyst
```

The installation will fail to install some shared libraries in the VisIt lib directory. The following script will copy the necessary libraries (you will need to modify the paths as appropriate.)

```
#!/bin/bash
cp /global/cfs/cdirs/alpine/brugger/spack/opt/spack/linux-sles15-zen3/gcc-11.2.0/
↪libtiff-4.5.1-vour2lgk4cvegrlxnuhwk3bz3ldfwzb5/lib64/libtiff.so.6.0.1 /global/cfs/
↪cdirs/alpine/brugger/spack/opt/spack/linux-sles15-zen3/gcc-11.2.0/visit-3.3.3-
↪gud54yyp44tv4gomn3i62wewdknpc2at/3.3.3/linux-x86_64/lib
ln -s libtiff.so.6.0.1 /global/cfs/cdirs/alpine/brugger/spack/opt/spack/linux-sles15-
↪zen3/gcc-11.2.0/visit-3.3.3-gud54yyp44tv4gomn3i62wewdknpc2at/3.3.3/linux-x86_64/lib/
↪libtiff.so.6
```

9.6.5 Working around recurring download failures

Depending on context, recurring issues downloading a particular *dependent* package may arise. When this happens, SSL certificate handling may be the cause. A quick work-around is to disable this security checking feature in Spack by adding the `--insecure` command-line option as the second option *just* after `spack`. Alternatively, you may be able to manually download the needed files and place them in a directory for Spack to use as a mirror. For example, starting from the point of having successfully downloaded the `Python-3.7.13.tgz` file somewhere, here are the Spack steps...

```
spack mirror add my_local_mirror file://`pwd`/my_local_mirror
mkdir -p my_local_mirror/python
cp Python-3.7.13.tgz my_local_mirror/python/python-3.7.13.tgz
```

Note that change in case of the file name. Doing this will cause Spack to go get the file you manually downloaded. The first step to add the mirror is only needed once. To add additional files for which recurring download failures are occurring, just copy them into the mirror following the Spack naming conventions for packages.

9.6.6 The spack environment files

Spack uses two files to control the environment on a system. They are the `compilers.yaml` file and the `packages.yaml` file.

The `compilers.yaml` file is used to specify information about compilers on a system. The `spack compilers` command, as mentioned earlier, will create one for you with all the compilers on the system. You can then customize it. The complete documentation on the `compilers.yaml` file can be found here.

The `packages.yaml` file is used to specify information about external packages on a system. By default, spack will want to build everything from scratch for your system. If some of the packages are already installed on the system you can use those by listing them in a `packages.yaml` file. Typically, you will want to use an external MPI library on most HPC systems. The `spack external find` command will create an initial `packages.yaml` file for you. The `spack external find` command is non-destructive and will append to an existing `packages.yaml` file. You can then customize it. The complete documentation on the `packages.yaml` file can be found here.

These files are stored in your ~/.spack directory.

```
.spack
.spack/<platform>
.spack/<platform>/compilers.yaml
.spack/packages.yaml
```

The VisIt repository at GitHub contains compilers.yaml and / or packages.yaml files for popular systems in the directory scripts/spack/configs.

Here is the packages.yaml file for frontier.olcf.ornl.gov for VisIt.

packages.yaml

```
packages:
  autoconf:
    externals:
    - spec: autoconf@2.69
      prefix: /usr
  automake:
    externals:
    - spec: automake@1.15.1
      prefix: /usr
  bison:
    externals:
    - spec: bison@3.0.4
      prefix: /usr
  cmake:
    buildable: false
    externals:
    - spec: cmake@3.23.2
      prefix: /sw/crusher/spack-envs/base/opt/linux-sles15-x86_64/gcc-7.5.0/cmake-3.
↪23.2-4r4mpiba7cwdw2hlakh5i7tchi64s3qd
      modules:
      - cmake/3.23.2
  cpio:
    externals:
    - spec: cpio@2.12
      prefix: /usr
  diffutils:
    externals:
    - spec: diffutils@3.6
      prefix: /usr
  file:
    externals:
    - spec: file@5.32
      prefix: /usr
  findutils:
    externals:
    - spec: findutils@4.6.0
      prefix: /usr
  flex:
    externals:
    - spec: flex@2.6.4+lex
      prefix: /usr
  gawk:
    externals:
    - spec: gawk@4.2.1
      prefix: /usr
```

(continues on next page)

(continued from previous page)

```
  gcc:
    externals:
    - spec: gcc@7.5.0 languages=c,c++,fortran
      prefix: /usr
      extra_attributes:
        compilers:
          c: /usr/bin/gcc-7
          cxx: /usr/bin/g++
          fortran: /usr/bin/gfortran-7
  ghostscript:
    externals:
    - spec: ghostscript@9.52
      prefix: /usr
  git:
    externals:
    - spec: git@2.26.2~tcltk
      prefix: /usr
  gmake:
    externals:
    - spec: gmake@4.2.1
      prefix: /usr
  groff:
    externals:
    - spec: groff@1.22.3
      prefix: /usr
  m4:
    externals:
    - spec: m4@1.4.18
      prefix: /usr
  ncurses:
    externals:
    - spec: ncurses@6.1.20180317+termlib abi=6
      prefix: /usr
  openssh:
    externals:
    - spec: openssh@8.1p1
      prefix: /usr
  openssl:
    externals:
    - spec: openssl@1.1.1d
      prefix: /usr
    buildable: False
  perl:
    externals:
    - spec: perl@5.26.1~cpanm+shared+threads
      prefix: /usr
  pkg-config:
    externals:
    - spec: pkg-config@0.29.2
      prefix: /usr
  rsync:
    externals:
    - spec: rsync@3.1.3
      prefix: /usr
  ruby:
    externals:
    - spec: ruby@2.5.9
```

(continues on next page)

(continued from previous page)

```
    prefix: /usr
sed:
  externals:
  - spec: sed@4.4
    prefix: /usr
tar:
  externals:
  - spec: tar@1.30
    prefix: /usr
texinfo:
  externals:
  - spec: texinfo@6.5
    prefix: /usr
xz:
  externals:
  - spec: xz@5.2.3
    prefix: /usr
all:
  compiler: [gcc, cce]
  providers:
    mpi: [cray-mpich]
rocm:
  buildable: false
  externals:
  - prefix: /opt/rocm-5.2.0
    spec: rocm@5.2.0
    modules:
    - rocm/5.2.0
cray-mpich:
  buildable: false
  externals:
  - prefix: /opt/cray/pe/mpich/8.1.23/ofi/gnu/9.1
    spec: cray-mpich@8.1.23%gcc
    modules:
    - cray-mpich/8.1.23
      cray-pmi/6.1.8
      libfabric/1.15.2.0
  - prefix: /opt/cray/pe/mpich/8.1.23/ofi/cray/10.0
    spec: cray-mpich@8.1.23%cce
    modules:
    - cray-mpich/8.1.23
      cray-pmi/6.1.8
      libfabric/1.15.2.0
hip:
  version: [5.2.0]
  buildable: false
  externals:
  - spec: hip@5.2.0
    prefix: /opt/rocm-5.2.0/hip
llvm-amdgpu:
  version: [5.2.0]
  buildable: false
  externals:
  - spec: llvm-amdgpu@5.2.0
    prefix: /opt/rocm-5.2.0/llvm
hsa-rocr-dev:
  version: [5.2.0]
```

(continues on next page)

(continued from previous page)

```
    buildable: false
    externals:
    - spec: hsa-rocr-dev@5.2.0
      prefix: /opt/rocm-5.2.0/
  rocminfo:
    version: [5.2.0]
    buildable: false
    externals:
    - spec: rocminfo@5.2.0
      prefix: /opt/rocm-5.2.0/
  rocm-device-libs:
    version: [5.2.0]
    buildable: false
    externals:
    - spec: rocm-device-libs@5.2.0
      prefix: /opt/rocm-5.2.0/
  rocprim:
    version: [5.2.0]
    buildable: false
    externals:
    - spec: rocprim@5.2.0
      prefix: /opt/rocm-5.2.0/
```

9.6.7 Debugging a spack package

When doing a spack install and the install fails, it will automatically keep the directory where it did the work, called the *stage* directory, which will allow you debug the failure. If you want to modify an otherwise successful install or explore the state of a successful install you can use the `--keep-stage` flag to the `spack install` command.

```
spack install --keep-stage visit ^python+shared ^mesa+opengl
```

If you are developing a new package from scratch and need to create the stage directory.

```
spack stage visit
```

To go to the stage directory and set up the spack environment.

```
spack cd visit
spack build-env visit bash
```

Note that this will create a new shell so you will want to do an `exit` when you are finished.

Spack will cache various items that will sometimes undermine changes you are making while developing a package. If you believe this is happening then you can clear all the caches.

```
spack clean -a
```

Here are some common locations of stage directories.

```
/tmp/<username>/spack-stage
/var/tmp/<username>/spack-stage
```

9.6.8 E4S Project

The Extreme-scale Scientific Software Stack (E4S) is a community effort to provide open source software packages for developing, deploying and running scientific applications on high-performance computing (HPC) platforms. E4S provides from-source builds and containers of a broad collection of HPC software packages.

E4S exists to accelerate the development, deployment and use of HPC software, lowering the barriers for HPC users. E4S provides containers and turn-key, from-source builds of more than 80 popular HPC products in programming models, such as MPI; development tools such as HPCToolkit, TAU and PAPI; math libraries such as PETSc and Trilinos; and Data and Viz tools such as HDF5 and VisIt.

E4S packages build on most computer systems, from laptops to supercomputers by using spack as the meta-build tool for the packages.

The E4S software distribution is tested regularly on a variety of platforms, from Linux clusters to leadership platforms.

The E4S testsuite

As a member of E4S, VisIt has tests that are part of the E4S-Project testsuite repository at GitHub.

Complete information on the testsuite can be found in the README at the bottom of the testsuite repository located here.

Running the tests

The first step is to clone the testsuite from GitHub.

```
# Using https.
git clone https://github.com/E4S-Project/testsuite

# Using ssh.
git clone ssh://git@github.com/brugger1/testsuite
```

Now you need to set up your spack environment.

```
# bash shell
/path/to/spack/share/spack/setup-env.sh

# c shell
source /path/to/spack/share/spack/setup-env.csh
```

Now you are ready to run the VisIt tests.

```
cd testsuite
./test-all.sh ./validation_tests/visit
```

9.7 Building Directly with CMake

If a *config site file* is available for the platform you wish to build on, VisIt can often be built without the use of the `build_visit` script, with these steps.

```
git clone --recursive git@github.com:visit-dav/visit.git
mkdir visit/build
cd visit/build
```

If `build_visit` was used to build VisIt on the platform in the past, it should have created a cmake file specific to your machine which we call a *config site* file. CMake simply needs to be told where to find it using the `-DVISIT_CONFIG_SITE` option. Examples of *config site* files for a variety of machines VisIt developers directly support can be found in the config-site directory.

```
/path/to/cmake ../src/ -DVISIT_CONFIG_SITE="/path/to/your_computer.cmake"
make -j
```

9.7.1 CMake Variables

The following CMake vars can be modified to suit your build needs.

When specified via a command-line invocation of CMake, they should be specified as: VARNAME:TYPE=value, eg 'VISIT_BUILD_ALL_PLUGINS:BOOL=ON'.

The defaults listed are the settings used if the Variable has not been set in a *config-site* file.

Controlling major components being built

VISIT_DBIO_ONLY [BOOL][OFF] Toggles building of only visitconvert and engine plugins.

VISIT_ENGINE_ONLY [BOOL][OFF] Toggles building of only the compute engine and its plugins.

VISIT_SERVER_COMPONENTS_ONLY [BOOL][OFF] Build only vcl, mdserver, engine and their plugins.

VISIT_ENABLE_LIBSIM [BOOL][ON] Toggles building of libsim.

Controlling plugins being built

VISIT_BUILD_ALL_PLUGINS [BOOL][OFF] Toggles the building of all plugins. When turned on the following optional plugins will be added to the build:

Database: PICS_Tester, Rect

Operator: Context ConnCompReduce, MetricThreshold, RemoveCells, SurfCompPrep

Plot: Topology

Note: the list of optional plugins is subject to change.

VISIT_BUILD_MINIMAL_PLUGINS [BOOL][OFF] Toggles the building of a minimal set of database, operator, and plot plugins. When turned on, only the following plugins will be built:

Database: Curve2D, RAW, VTK, PICS_Tester

Operator: Lineout, Slice, Threshold

Plot: Curve, Mesh, Pseudocolor

Note: the list of minimal plugins is subject to change.

VISIT_SELECTED_DATABASE_PLUGINS [STRING] ';' separated list of database plugins to build, eg: VTK;Silo

If not empty, will supersede the settings of VISIT_BUILD_MINIMAL_PLUGINS and VISIT_BUILD_ALL_PLUGINS for database plugins.

VISIT_SELECTED_OPERATOR_PLUGINS [STRING] ';' separated list of operator plugins to build, eg: Slice;Lineout;Transform

If not empty, will supersede the settings of VISIT_BUILD_MINIMAL_PLUGINS and VISIT_BUILD_ALL_PLUGINS for operator plugins.

VISIT_SELECTED_PLOT_PLUGINS [STRING] ';' separated list of plot plugins to build, eg: Mesh;Pseudocolor

If not empty, will supersede the settings of VISIT_BUILD_MINIMAL_PLUGINS and VISIT_BUILD_ALL_PLUGINS for plot plugins.

Controlling extra tools being built

VISIT_ENABLE_ANNOTATION_TOOLS [BOOL][ON] Toggles the generation of annotation tools: text2polys, time_annotation.

VISIT_ENABLE_DATAGEN [BOOL: ON] Toggles the generation of sample data files.

VISIT_ENABLE_DATA_MANUAL_EXAMPLES: BOOL [OFF] Toggles generation of *Getting Data Into Visit* examples.

VISIT_ENABLE_DIAGNOSTICS [BOOL][ON] Toggles building of diagnostic tools: exceptiontest, mpitest, networktest, osmesatest.

VISIT_ENABLE_MANUALS [BOOL][ON] Toggles building of manuals, requires Sphinx in Python.

VISIT_ENABLE_SILO_TOOLS [BOOL][ON] Toggles building of Silo tools: mrgtree2dot, add_visit_searchpath.

VISIT_ENABLE_UNIT_TESTS [BOOL][ON] Toggles building of unit tests: MRUCache, Namescheme, Utilty, StringHelpers, exprconfig, exprtest.

Useful for developers

VISIT_CREATE_SOCKET_RELAY_EXECUTABLE: BOOL [ON] Toggles creation of separate executable that forwards VisIt's socket connection between engine and component launcher.

VISIT_CREATE_XMLTOOLS_GEN_TARGETS [BOOL][ON] Toggles the creation of build targets to run xmltools code generation. More information can be found in the *XML Tools* section of the *Developer Manual*

Be careful on Windows, *all of the codegen targets will be built unless you tell Visual Studio to build the ALL_BUILD project (instead of the Solution). This will cause a lot of source files to be regenerated and may cause problems with the build.*

VISIT_RPATH_RELATIVE_TO_EXECUTABLE_PATH [BOOL][OFF] Install rpath relative to executable location using $ORIGIN tag.

CMAKE_SUPPRESS_REGENERATION [BOOL][OFF] When on, tells CMake to suppress regeneration of project/make files when CMakeLists.txt or .cmake files have changed.

Miscellany

CMAKE_BUILD_TYPE [STRING][Release] Specifies the build type for single-configuration generators (like Makefiles).

CMAKE_INSTALL_PREFIX [PATH][*default is system dependent*] Specifies the location for files installed with *make install.*

IGNORE_THIRD_PARTY_LIB_PROBLEMS [BOOL][OFF] Ignore problems finding requested third party libraries.

VISIT_CONFIG_SITE [FILEPATH][${VISIT_SOURCE_DIR}/config-site/<localhost>.cmake] Location of a config-site cmake file that has settings to control the build, including locations of third-party libraries. Created automatically by build_visit script.

VISIT_DDT [BOOL][OFF] Toggles support for the DDT debugger.

VISIT_DEFAULT_SILO_DRIVER [STRING][PDB] Designates the default Silo driver to use when generating silo data. Options: PDB, HDF5

VISIT_DISABLE_SELECT [BOOL][OFF] Toggles the disablement for use of the select() function.

VISIT_FORCE_SSH_TUNNELING [BOOL][OFF] Toggles use of SSH tunneling for sockets.

VISIT_FORTRAN [BOOL][OFF] Toggles building of Fortran example programs.

VISIT_INSTALL_THIRD_PARTY [BOOL][OFF] Intall VisIt's 3rd part I/O libraries and includes to permit plugin development.

VISIT_JAVA [BOOL][OFF] Build VisIt's Java client interface.

VISIT_NOLINK_MPI_WITH_LIBRARIES [BOOL][OFF] Do not link MPI with VisIt's parallel shared libraries; just with executables

VISIT_OSPRAY [BOOL][OFF] Build VisIt with support for the OSPRay volume rendering library.

VISIT_PARALLEL [BOOL][ON] Build VisIt's parallel compute engine.

VISIT_PYTHON_SCRIPTING [BOOL][ON] Build VisIt with Python scripting support.

VISIT_PYTHON_FILTERS [BOOL][ON] Build VisIt with Python Engine Filter support.

VISIT_SLIVR [BOOL][ON] Build VisIt with support for the SLIVR volume rendering library.

VISIT_STATIC [BOOL][OFF] Build VisIt statically.

Mac OS only

VISIT_CREATE_APPBUNDLE_PACKAGE [BOOL][OFF] Toggles creation of DMG file with Mac App bundle with make package.

Windows OS only

VISIT_MAKE_NSIS_INSTALLER [BOOL][OFF] Toggles creation of an installer package using NSIS.

The windows.cmake *config-site* file turns this ON.

VISIT_MESA_REPLACE_OPENGL [BOOL][OFF] Toggles use of Mesa as a drop-in replacement for OpenGL when system OpenGL is insufficient.

The windows.cmake *config-site* file turns this ON.

VISIT_WINDOWS_APPLICATION [BOOL][ON] Toggles creation of Windows-style applications with no console.

VISIT_WINDOWS_DIR [PATH :] Specifies the location of the pre-built third party library binaries. See *Location of windowsbuild directory* for default locations.

9.8 Linux X11 Requirements for Qt

On Linux, Qt has specific X11 requirements that must be met for its Platform Plugin to be built successfully. Many developers encounter errors building Qt due to missing support libraries related to X11 and xcb.

Here are a couple of Qt links that prove helpful in knowing which packages need to be installed for particular OS distros.

- For Qt 5
  - The libxcb section of Building Qt5 from Git
  - Qt5's X11 Requirements page.
- For Qt 6 (only on develop branch)
  - Qt6's X11 Requirements page.

VisIt also maintains a few Docker files that may prove useful:

- centos8
- debian9
- debian10
- debian11
- fedora31
- ubuntu18
- ubuntu20
- ubuntu21

CHAPTER

TEN

VISIT DEVELOPER MANUAL

10.1 Project Meetings

Project meetings are held one hour per week currently on Tuesday afternoons. A second one hour *special topics* meeting is scheduled on all contributor's calendars but is used infrequently. Dialog at the end of the Tuesday project meeting determines if the *special topics* meeting time is needed.

10.1.1 Round Robin Dialog

The meeting begins with round-robin remarks from contributors on key items from the preceding week's work. The intention is to keep remarks brief and to highlights mainly to ensure everyone is aware of key and impending changes related to VisIt. Sometimes contributors have nothing relevant to VisIt to mention in which case its perfectly fine for the contributor to *pass* during this round-robin dialog.

10.1.2 New Issue Triage

New issues are triaged. New issues can be found through a browser using GitHub's issue search feature and searching for issues missing the `reviewed` label. During triage, issues are discussed in some detail to help refine meaning and determine `impact` (low, medium or high) and `likelihood` (low, medium or high) as well as other special considerations such as whether the issue is manifesting an outright `crash`, should be given higher `priority` or may represent `low hanging fruit` (a small amount of work that will yield high dividends). Descriptions for the meaning of various labels to be applied to issues are provided on the label description page. Some issue submissions require lengthy discussion and dialog to help refine into one (or more) specific bug fixes or enhancement requests. Such issues are converted to discussions and should be gathered together to be discussed in a *special topics* meeting.

10.1.3 Unanswered Q&A Discussion Triage

Discussion items that have yet to be answered are triaged. Sometimes a new discussion will get converted to an issue. We don't currently use any labels for discussions but we have several discussion *categories* and we may re-categorize discussions to put them in the right category. We close any discussions which have been waiting for user response for more than 21 days.

10.1.4 Special Topics Meeting Topics/Agenda

We consider the need and interest in using the upcoming special topics meeting time. Sometimes, we opt to use this meeting time for further issue triage. Other times, we decide on some topic(s) to discuss and contributors may prepare some informal materials.

10.1.5 Inclusive Moment

To begin the first project meeting of each new month a contributor shares an inclusive moment. An inclusive moment is a factoid, resource, event, experience, etc. related to STEAM with the purpose of raising awareness, seeding deeper thinking and/or informing future action.

Whenever possible, we try to rotate responsibility to a different contributor each time. There is an article that goes into more details about the inclusive minute exercise.

10.2 Developing at GitHub

10.2.1 Overview

The VisIt project has a number of repositories located at the GitHub visit-dav organization.

https://github.com/visit-dav/

The primary repository for doing VisIt development is the visit repository.

https://github.com/visit-dav/visit/

The following top level directories exist in the visit repository.

- data - Data files used by the test suite.
- docs - Legacy documentation including design documents and presentations.
- scripts - Various scripts used for doing VisIt developement including scripts for managing docker containers and doing continuous integration.
- src - The VisIt source code. It includes the Read the Docs documentation and the regression test suite.
- test/baseline - The baseline results for the regression test suite.

10.2.2 Setting Up Git LFS

Git LFS (Large File Storage) is a mechanism to help revison control large files efficiently with git. Instead of storing large files in the repo, LFS provides an extension that stores small text files with meta data in the repo and the actual files on another server. These meta data files are called "pointer" files. We use LFS for binary data including our test data tar files, source code for third party libraries, and regression test baseline images.

Git LFS is not part of the standard git client. See https://git-lfs.github.com/ for how to obtain Git LFS.

When installing, use the following option:

```
git lfs install --force --skip-smudge
```

The "skip smudge" command sets up LFS in a way that skips automatically pulling our large files on clone. We do this to conserve bandwith.

To obtain these files you will need to do some extra incantations followed by an explicit:

```
git lfs pull
```

For more details about using Git LFS, read *our additional notes*.

10.2.3 Accessing GitHub

The following link points to a page for creating a personal access token to use for the password when accessing GitHub through the command line. Use the following scopes for the token:

repo:status

repo_deployment

public_repo

https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/

The following link describes how to add your ssh key to your GitHub account.

https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/

10.2.4 Cloning the Repository and Setting Up Hooks

You can access GitHub either through https or ssh. If you use https you will be prompted for your password whenever you push to GitHub. There are ways you can have your password cached for a period of time to reduce the frequency of entering your password. However, if you have two-factor authentication set up you will need to create a personal access token to use in place of the password. If you use the ssh protocol you can set things up so that you never have to enter a password by adding your ssh key to your GitHub account.

To clone the repository:

```
git clone --recursive https://github.com/visit-dav/visit.git
```

or:

```
git clone  --recursive ssh://git@github.com/visit-dav/visit.git
```

If for some reason the `--recursive` flag was overlooked when the repository was originally cloned, this can be easily remedied by:

```
cd visit
git submodule init
git submodule update
```

To setup our hooks:

```
cd visit
./scripts/git-hooks/install-hooks.sh
```

10.2.5 Creating a Branch

Development for VisIt is done off of two main branches, the `develop` branch and the current release candidate branch, which was `3.2RC` when this content was written. The `develop` branch is used for development that will go into the next major or minor release. Major releases are releases where the first digit of the release number is incremented, Minor releases are releases where the second digit of the release number is incremented. The release candidate branch is used for development that will go into the next patch release. Patch releases are releases where the third digit of the release number is incremented.

There is no convention on the names of a branch. One commonly used convention is `task\Username\YYYY_MM_DD_Description` where `Username` is your GitHub user name, `YYYY` is the current year, `MM` is the current month, `DD` is the current day, and `Description` is a short description of the task

to be performed. Since branches only exist while you are doing the development, the name isn't critical, but it should be sufficiently descriptive so that someone can have some idea what the development on the branch is about.

To create a branch off of the `develop` branch:

```
git checkout develop
git pull
git checkout -b task/user/2021_05_07_bug_fix
```

To create a branch off of the current release candidate:

```
git checkout 3.2RC
git pull
git checkout -b task/user/2021_05_07_bug_fix
```

When you switch branches, you may also need to update submodules so they match your branch:

```
git submodule update
```

10.2.6 Doing Development

Doing development using the Git version control system can be complex and take considerable time and effort to master. The primer below is just meant to get you started in modifying files and then pushing the changes to GitHub so that they can be integrated into VisIt.

To add a new file or modify an existing file, edit the file with your favorite text editor and then use the `add` command so that git knows you want the file to be part of your next commit. To add a file:

```
git add src/myfile
```

To delete an existing file use the `rm` command:

```
git rm src/myfile
```

Once you have modified one or more files you can commit the change to git. You will typically do a commit after having modified one or more files that completes a logical unit of change. To commit the added files with a comment:

```
git commit -m "Description of my change."
```

It is recommended that you make commits frequently so that you can better track individual changes. The commit descriptions are typically brief. The record of the individual commits will not go into the final record of the commit, since we do "Squash and Merge" commits that merge all the commits into a single commit at GitHub. The individual commits will be helpful to you as a developer if you need to go back and understand when making many changes over a period of time. It may also potentially make it easier for reviewers to understand your commits.

Once you have finished all your changes you can push the change to GitHub. To push your changes to GitHub:

```
git push --set-upstream origin task/user/2021_05_07_bug_fix
```

Once you have pushed your changes to GitHub, you can submit a *pull request*.

10.2.7 CMake Build System

VisIt's build system uses BLT CMake helpers. BLT is included in VisIt's git repo as a git submodule. To obtain the submodule, use *git clone –recursive* when cloning, or manually setup the submodule after cloning using:

```
git submodule init
git submodule update
```

When you switch branches, you may also need to update submodules so they match your branch:

```
git submodule update
```

Branch development with git submodules can lead to unintended submodule commits. To avoid this, we have an CI check that ensures the active submodule commits match a version explicitly listed in a *hashes.txt* file at the root of the git repo.

10.3 Coding Style Guide

10.3.1 Naming Conventions

- Class names start with capital letters, (eg Mesh). If a class name has multiple words, capitalize the first letter of each word, (eg RectilinearMesh). Exceptions can be made to help group classes in a package (eg vtkRectilinearMesh).
- Fields and variables have the first letter uncapitalized and the first letter for each subsequent word capitalized, (eg theImportantValue).
- Methods and functions follow the same naming convention as classes. The distinction between methods and classes will be clear because methods will always have attached parentheses. This is not true when a function is passed as a pointer, but that should be clear from context.
- Try to avoid using the following names as they are reserved by ANSI for future expansion:

Names	Description
E[0-9A-Z][0-9A-Za-z]*	Errno values
is[a-z][0-9A-Za-z]*	Character classification
to[a-z][0-9A-Za-z]*	Character manipulation
LC_[0-9A-Za-z_]*	Locale
SIG[_A-Z][0-9A-Za-z_]*	Signals
str[a-z][0-9A-Za-z_]*	String manipulation
mem[a-z][0-9A-Za-z_]*	Memory manipulation
wcs[a-z][0-9A-Za-z_]*	Wide character string manipulation

10.3.2 File Structure

File Names

A file containing the definition for class Foo will be named Foo.h. A file containing the methods for the class Foo will be named Foo.C.

.h File Contents

Each of the .h files will have the following format:

#ifndef/#define
Includes
Class Description
Class Definition
Variable Declarations
Inline Functions
#endif

- The ifndef is used to prevent the class from being defined multiple times.
  - To prevent name collisions, the symbolic name being defined should be the class name in all capital letters, with each word separated by underscores. _H should be appended.
- Forward declare classes rather than including their header file when possible but be sure to use the correct struct or class keyword in the forward declaration.
- Only one class should be defined per file. Exceptions can be made for very closely related classes.
- Inline functions should only be included in the .h file if they are public or protected. Private * inline functions should be placed in the .C file for that class.
  - Note that public and protected inline functions should be used sparingly. All code that includes the header must be recompiled if the function is changed.
- The .h file should be valid as a stand alone file.
  - If other header files are included before this files inclusion, it may be making use of their definitions.
- All variables declared here should be externed. Class-scoped static variables should not be defined here.
- Avoid `using::` directives in header files. They will effect not only the current header file but any files in which the header file is included, directly or indirectly.

.C File Content

Each of the .C files will have the following format:

Includes
Variable Declarations
Static Function Prototypes
Constructors
Destructors
Method Definitions
Friend Functions
Static Functions

The friend operators included in the .C file must be directly related to the class whose methods are defined in that file.

Copyright notice

The copyright notice shall appear at the top of each .C, .h, .java, CMakeLists.txt, and Python sources.

```
// Copyright (c) Lawrence Livermore National Security, LLC and other VisIt
// Project developers.  See the top-level LICENSE file for dates and other
// details.  No copyright assignment is required to contribute to VisIt.
```

Includes

- Include files should use angle brackes. For example: #include <vtkRectilinearGrid.h>
- Class.C should include Class.h first. This is to make sure that Class.h is not using any previously declared headers. Class.C file may use quotes instead of angle brackets to include Class.h.
- Include files should be grouped from wider scope to narrower scope. This leads to grouping the include files in the following order:
  - System include files. Examples are: <math.h>, <stdio.h>.
  - X and Qt include files. Examples are: <Xlib.h>, <qgl.h>.
  - Library include files. Examples are: <dmf.h>, <vtk.h>.
  - Class definition files. Examples are: <Mesh.h>, <Field.h>.
  - Within a group, include files should be listed alphabetically.
- If the include files must be listed in a specific order, which is not alphabetic, then a comment must be added justifying it.
- Some C header files contain C++ keywords that cause compilations to fail. With the exception of header files for the standard C library, X, and Motif, all C header files must be wrapped with an *extern C* directive.

```
extern "C" {
#include <hdf5.h>
}
```

Forbidden Constructs

exit() and abort()

Please do not use *exit()* or *abort()* in your code since we do not want VisIt to fail unexpectedly. Use exceptions instead. VisIt's check-in hooks will not permit unconditionally compiled code calling exit or abort to be checked-in.

using statements

using statements of any kind are not permitted in header files since they can indirectly cause compilation problems for other compilation units that may include your header file either directly or indirectly. In header files, you will have to use the fully qualified class name for any class you need to refer to. (e.g. *std::vector* and not just *vector*). Yes, this does make for somewhat uglier header files but it also prevents a lot of problems. VisIt's check-in hooks will not permit code containing *using* statements in a header file.

In source files, when you need to use *using* statements, we prefer that you narrow the scope of the statement as much as is practical. So, please don't use *using namespace std* to use something like *std::string*. Instead use *using std::string*.

10.3.3 Class Description

Each class must have a brief description, like the below example.

```
// ****************************************************
// Class: Example
//
// Purpose:
```

(continues on next page)

(continued from previous page)

```
//   What this class does.
//
// Notes:     Any special notes for users of the class.
//
// Programmer: Joe Smith
// Creation: August 29, 2007
//
// Modifications:
//   Joe Smith, Fri Oct 15 13:31:51 EST 2007
//   I added a new method to do ...
//
// ****************************************************
```

It is important to use these category names because they will be picked up by doxygen to create our documentation. The asterisks should fill out the line, 76 asterisks in all. The category labels (Class, Purpose, etc) should be indented two spaces past the comment (//). When the text after a category label wraps to the following line, it should be indented fours spaces after the comment.

10.3.4 Class Definition

Class definitions should follow these rules:

- There should be no public fields. They violate the basic object-oriented philosophy of data hiding.
- The sections inside the class should be ordered public, protected, private. This way users of the class can stop reading when they reach protected/private. The fields in each section should be grouped together, as should the methods.
- All inheritance should be public to avoid confusion, bar good reason.
- Every non-trivial field should have a comment preceding it that describes its purpose. This comment will be picked up by doxygen when the documentation is built.
- Friends should be avoided when possible. When it is necessary to grant friend access to a series of derived types, grant it only to their base type and define protected methods for the base type that access the class.
- Define a copy constructor and assignment operator for every class.
- Constructor and destructor method definitions should never appear in the class header file because of compiler bugs on some platforms.
- Note that C++ automatically provides a constructor, a copy constructor, an assignment operator, two address-of operators, and a destructor for you:

```
// You write
class Empty { };
// You get
class Empty
{
public:
    Empty() {};
    ~Empty() {};
    Empty(const Empty &);
    Empty &operator=(const Empty &);
    Empty *operator&();
    const Empty *operator&() const;
};
```

- If you are redefining a pure virtual method that should not be used, declare it private and have it throw an exception.
- The copy constructor and assignment operator provided by the compiler perform blind copies, meaning that pointers will also be copied, potentially introducing many bugs.

This is because many publicly available libraries, such as STL, use these methods.

10.3.5 Method Structure

The structure of a method should follow this format:

Prologue
Declaration
Body

Prologue

Each method must have a prologue with the following format:

```
// ****************************************************
// Method: ClassName::MethodName
//
// Purpose:
//   What this method does.
//
// Arguments:
//   arg1 : What the first argument does...
//   arg2 : What the second argument does...
//   ...
//
// Returns:  <0 on failure, 0 on success.
//
// Note: Assumes coordinates have already been read.
//
// Programmer: Joe VisIt
// Creation: August 29, 2007
//
// Modifications:
//   Joe VisIt, Fri Oct 15 13:31:51 EST 2007
//   Fixed bug with ...
//
// ****************************************************
```

The category label '"Method"' can be replaced '"Function"', '"Operator"', '"Constructor"', or '"Destructor"' and still be accepted by doxygenate.

10.3.6 Definition

The definition should follow this form:

```
Zone *
RectilinearMesh::GetZone(int i, int j, int k)
```

If multiple lines are needed for all of the arguments, each subsequent line should be indented to the opening parenthesis, or if that is too far, 4 spaces.

10.3.7 Body

Size

The body should be small. Try to keep functions under 100 lines. This promotes clarity and correctness. This tradeoff should not be paid for a substantial speed penalty, however.

Arguments

- All input arguments passed by reference should be declared *const*.
- Unused arguments should not be named. Note that this eliminates the need for the lint directive ARGSUSED.

Variables

- Variables should be declared near their first use.
- Variable names may not coincide with any of the class' field names.
- All local pointer variables should be set to NULL or 0 when declared. This helps with later tracking of memory problems when looking at core files.
- Associate * and & with the variable, not with the type.

For example, the following code:

```
int* i, j;
```

misleads the reader into thinking both i and j are pointers to ints, while j is actually only an int.

- Only use variables declared in the initializer list of a for loop inside that for loop.

The code fragments:

```
for (int i = 0 ; i < size; i++)
   { ... }
for (int i = 0 ; i < length ; i++)
   { ... }
```

and

```
for (int i = 0 ; i < size ; i++)
   { ... }
if (i == size)
   ...
```

Comments

- Avoid using C-style comments. This way, when debugging, they can be used to comment out long blocks of code without worrying about nested comments.
- Indent comments to the same level as the statement to which they apply.

- Both block and single line comments are acceptable, but when modifying a pre-existing file, they should follow its convention.
- Comments are highly encouraged!

Control Structures

- Use *for (;;)* instead of *while(1)*. They both result in infinite loops, but *while(1)* is flagged by many compilers as a constant condition. This eliminates the need for the lint directive CONSTCOND in this case.
- Any case of a switch statement that does not end with a break should have a FALLTHRU comment to show that this is intentional.
- When the body of a for or while is empty, place a continue in it to make the intent clear.

The following code:

```
for (int i = 0; p[i] != '\0' ; i++);
```

Is more clearly represented as:

```
for (int i = 0; p[i] != '\0' ; i++)
    continue;
```

Also note that this eliminates the need for the lint directive EMPTY in this case.

Whitespace

- TAB characters are **NOT ALLOWED** in VisIt source code.
- Semicolons should immediately follow the last character. (i.e. there is no space between the last character in a statement and its semicolon).
- Lines should not exceed 79 characters in length. Note that it is not necessary to violate this rule for strings.

```
char *str1 = "Hello world";
char *str2 = "Hello "
             "world";
```

In the code above, str1 is equal to str2.

- All variable declarations should occur on separate lines unless closely related (e.g. int i, j, k;).
- Do not use any tabs in the source. Use \t to simulate a tab in a string.
- The parenthesis of a function should immediately follow the function name. This makes searching easier for functions with common names.
- There should not be any spaces surrounding the . or ->, operators and no spaces preceding a [operator.
- An indentation block is four spaces.
- The labels case, public, protected, and private are indented 0 or two spaces.
- Any time a new block is started, a { should be put on the following line at the same indentation level. The next statement should be indented an additional four spaces.
- Within reason, adding whitespace to line up parentheses or brackets on consecutive lines is encouraged, even when it violates one of the previous rules.

Reformatting

Automatic source code reformatting may be performed using a program called ‘’[http://astyle.sourceforge.net/ artistic style]”. Here is some basic usage that reformats a source file into a form compatible with VisIt coding style:

```
astyle --brackets=break < inputfile > outputfile
```

End of line

The UNIX convention for end of line characters must be followed for VisIt source code.

Preprocessor

- Macros should only be used if the # or ## operators are used.
- Any macros used to define a constant should be declared as a const global variable.
- Parameterized macros used to perform a short routine should be implemented as an inline.
- Macros should only be used if the # or ## operators are used.
- The code inside the #ifdef section should be indented as if the #ifdef were not present.
- Comments should not be added on the same line after preprocessor directives because some compilers do not accept them.
- Preprocessor directives should have the # in column 1.

Pointer vs. References

References are preferred over pointers.

References:

- Always refer to a real object.
- Do not change objects they refer to.

Pointers:

- Can represent no object (NULL).
- Can change the object they refer to.
- Can represent an array.
- Can represent a location (like the end of an array).

10.3.8 Caveats for ensuring that VisIt builds on Windows

The rules that have been covered before in this document apply mainly to source code style and are conventions to simplify maintenance. This section describes some source code constructs that must be avoided at all times in order to ensure compatibility with the Microsoft Windows Visual C++ (MSVC). Windows is an important development platform for VisIt. Adhering to these additional coding rules will reduce the amount of time required to fix minor source code problems that burden Windows developers.

API macros

VisIt's header files have API macros that help the MSVC (all versions) compiler and linker produce dynamic link libraries (DLLs) and their associated import libraries. A DLL is a file containing executable code which is loaded by an application at runtime and all applications that require the code stored in the DLL use the same instance of the DLL in the computer's memory, which saves resources. An import library is a small stub library that contains enough symbolic information to satisfy the linker so that all unresolved symbols are resolved at link time and still allowing the application code to be loaded dynamically at runtime. This link step is mostly avoided on other platforms where VisIt's libraries are linked exclusively at runtime.

Import libraries are difficult to create manually due to the amount of symbols in all of VisIt's libraries so the VisIt source code has been augmented with API macros that allow the compiler to automatically create the import libraries. VisIt's API macros come from an API include file and there is one API include file per VisIt library. The name of the API include file is usually the name of the library appended with the "_exports.h" suffix. The API macro is added to class declarations when the class should be made accessible to other VisIt libraries.

```
#ifndef MY_EXAMPLE_CLASS_H
#define MY_EXAMPLE_CLASS_H
#include <example_exports.h>

class EXAMPLE_API MyExampleClass
{
public:
    MyExampleClass();
    virtual ~MyExampleClass();
};

void EXAMPLE_API example_exported_function();
void this_function_not_exported();
#endif
```

In the above example, the header file that gets included defines the EXAMPLE_API macro, which tells the MSVC compiler to add the flagged symbols to its list of symbols for the import library that goes along with the DLL that contains the class. The EXAMPLE_API macro evaluates to whitespace on other platforms so its inclusion in VisIt's source code is not disruptive. Note that the EXAMPLE_API macro has been applied to a class and to a function to ensure that both the class's methods and the function are both added to the import library. Any class, function, variable, etc that lacks an export macro is not added to the import library and will not be available to other programs or libraries.

Now that the mechanism by which symbols are added to import libaries has been explained, suppose that you move a class from one library to another. What happens? Well, the answer is that the class will be compiled into the new library but it will not be put into the import library because its API macro was not changed. To avoid this problem, it is very important that when you move classes from one library to another library that you change the class so it uses the appropriate API macro for the new host library. This goes especially for VTK classes that have become part of one of VisIt's libraries.

No constructor or destructor definitions in header file

Do not put class constructor or destructor definitions in the class header file. When you put class constructors and destructors in the class header file, MSVC gets confused when you attempt to use the class from another DLL because sometimes the virtual method table is messed up when the constructor and destructor are placed in the header file possibly due to function inlining. When this happens, it is impossible to successfully link against the library that is supposed to contain your class. To be safe, always create a .C file that contains the constructor and destructors for your class.

```
#ifndef MY_CLASS_H
#define MY_CLASS_H
#include <mylib_exports.h>
class MYLIB_API MyClass
{
public:
    // Never do this
    MyClass() { };
    virtual ~MyClass() { };
};
```

Do this instead:

MyClass.h file contents:

```
#ifndef MY_CLASS_H
#define MY_CLASS_H
#include <mylib_exports.h>
class MYLIB_API MyClass
{
public:
    MyClass();
    virtual ~MyClass();
};
```

MyClass.C file contents:

```
#include <MyClass.h>
MyClass::MyClass() { }
MyClass::~MyClass() { }
```

Do not use 'sprintf'

VisIt source code should not use *sprintf* into a static sized buffer due to the possibility of buffer overruns, which introduce memory problems and possible security threats. To combat this, the use of *sprintf* is deprecated and all new code should use *snprintf*, which behaves the same but also takes the size of the buffer as an argument so buffer overruns are not possible.

Do not use variables called near or far

The MSVC compiler reserves the *near* and *far* keywords for backward compatibility with older 16-bit versions of the compiler that used *near* and *far* to determine pointer size. Do not use near or far for variable names because it will cause a strange compiler error.

Do not create a file called parser.h

Windows provides a file called parser.h and if you also provide such a file, you had better change the include directory order or you will run into hundreds of errors when the compiler uses Microsoft's parser.h instead of yours.

Do not create functions or methods called GetMessage

The WIN32 API is used in certain places in VisIt to implement Windows-specific functionality. Occasionally, we have run into problems where VisIt classes have names such as *GetMessage*. The windows.h include file defines a macro

called GetMessage and sets it to GetMessageEx. This caused the preprocessor to replace all GetMessage method calls on a VisIt object with GetMessageEx, which is not a method of the object. Needless to say, this is a confusing compilation problem. Steer clear of defining method names that conflict with WIN32 macro names!

Comparing QString and std::string

Call the .toStdString() method to compare QString to std::string.

Example:

```
QString string1("my q string");
std::string string2("my std string");

// Do this:
if (string1.toStdString() == string2))
```

Do not use unistd.h

Windows does not have the unistd.h header file so do not use functions from it without making conditionally compiled code.

```
#if defined(_WIN32)
    // Windows implementation ...
#else
    #include <unistd.h>
    // Unix implementation ...
#endif
```

Do not use libgen.h

Windows does not have libgen.h, which is sometimes used for functions such as dirname(), basename(). Refrain from using functions from libgen or provide a Windows implementation as well.

Sign of size() method return value

The .size() method for STL containers returns a *size_t*. Be aware if you attempt to do arithmetic on the value returned by .size()

Example:

```
// Consider what happens when the following code is
// executed with myvector being empty (size is zero)

if (val > myvector.size()-1) // if test fails
{
   return;
}
myvector[val] =  ... // SEGV!
```

Allocate dynamic arrays on the heap, not the stack

If the size of an array cannot be determined at compile-time, then it cannot be allocated on the stack, but must be allocated on the heap.

Example:

```
const int nPoints = dataset->GetNumberOfPoints();

// Since value of nPoints can only be determined at run-time,

// this will not compile with Visual Studio
int myarray[nPoints];

// this will compile
int *myarray2 = new int[nPoints];
```

10.3.9 CMake Conventions

Starting with VisIt version 3.4, new more modern CMake conventions will be adopted, and BLT will be used whenever feasible.

Handling subdirectories

Each subdirectory should have its own CMakeLists.txt which either creates a new target or adds sources to a target defined in a parent directory's CMakeLists.txt. If a given target has source files spread out across multiple subdirectories, the *add_library* or *add_executable* calls should be in the CMakeLists.txt of the topmost directory, along with the *add_subdirectory* and any common *target_include_directories* or *target_link_libraries* calls. The subdirectory will add its sources to the parent's target via *target_sources*.

Here's an example from src/avt/DBAtts and src/avt/DBAtts/SIL:

```
add_library(avtdbatts)

add_subdirectory(MetaData)
add_subdirectory(SIL)

target_link_libraries(avtdbatts visitcommon)
target_include_directories(avtdbatts PUBLIC ${VISIT_COMMON_INCLUDES})

VISIT_INSTALL_TARGETS(avtdbatts)
```

```
target_sources(avtdbatts PRIVATE
    avtSIL.C
    avtSILArray.C
    avtSILCollection.C
    avtSILEnumeratedNamespace.C
    avtSILMatrix.C
    avtSILNamespace.C
    avtSILRangeNamespace.C
    avtSILRestriction.C
    avtSILRestrictionTraverser.C
    avtSILSet.C)

target_include_directories(avtdbatts PUBLIC .)
```

10.4 Creating a Pull Request

10.4.1 Overview

Pull Requests (PR (Pull Request)s) allow developers to review work before merging it into the develop branch. PRs are extremely useful for preventing bugs, enforcing coding practices, and ensuring changes are consistent with VisIt's overall architecture. Because PR reviews can take time, we have adopted policies to help tailor the review effort and balance the load among developers. We hope these policies will help ensure PR reviews are completed in a timely manner. The benefits of reviews outweigh the added time.

10.4.2 Forking the repo

Developers who do not have write access to the primary VisIt repo may make contributions by forking the repo and submitting pull requests. GitHub provides excellent informational articles about forking a repo and creating pull requests from a fork.

10.4.3 Working with the Template

PR submissions are populated with a template to help guide the content. Developers do not have to use this template. Keep in mind, however, that reviewers need structured context in order to accurately and quickly review a PR. So, it is best to use the template or something very similar to it. The text sections in the template are designed to be *replaced* by information relevant to the work involved. For example, replace a line that says *Please include a summary of the change* with an actual summary of the change.

In general, if part of the template is not relevant, please delete it before submitting the PR. For example, delete any items in the *checklist* that are not relevant.

If additional structured sections in the PR submission are needed, please use GitHub markdown styling.

In the sections below, we describe each of the sections of the PR template in more detail.

Description

GitHub supports a number of idioms and keywords in PR submissions to help automatically link related items. Please use them.

For example, when typing a hashtag (`#`) followed by a number or text, a search menu will appear providing potential matches based on issue or PR numbers or headlines. Sometimes no matches will be produced even if the number being entered is correct, but the link will still occur when the PR is submitted. By placing the keyword "Resolves" in front of a link to an issue, the issue will automatically close when the PR is merged.

If a PR is unrelated to a ticket, please delete the "Resolves #..." line for clarity.

Type of Change

Bug fixes, features, and documentation improvements are among the most common types of PRs. You may select from the menu by replacing the space between the square brackets (`[ ]`) with an uppercase X, so that it looks exactly like `[X]`. You can also make this selection *after* submitting the PR by checking the box that appears on the submitted PR page.

If "Other" is checked, please describe the type of change in the space below.

Testing

Replace the content of this section with a description of how the change was tested.

The Checklist

The Checklist serves as a list of suggested tasks to be performed before submitting the PR. Those that have been completed should be checked off. Any items that do not relate to the PR should be deleted. For example, if the PR is not for a bugfix or feature, adding a test may not be required and this checklist item *should* be deleted.

10.4.4 Reviewers

GitHub will not allow non-owners to merge PRs into develop without a reviewer's approval. Non-owners will need at least one reviewer. Owners may merge a PR into develop without review. But, that does not necessarily mean they should. Follow the guidelines below to determine the need for and number of reviewers. Note, these guidelines serve as a "lower bound"; you may always add more reviewers to your PR if you feel that is necessary.

No Reviewers (owners only)

If your changes are localized, you have satisfied all the testing requirements and you are confident in the correctness of your changes (where correctness is measured by both the correctness of your code for accomplishing the desired task and the correctness of *how* you implemented the code according to VisIt's standard practices) then you may merge the PR without a reviewer *after* the CI tests pass.

One reviewer

If the changes have a broader impact or involve an unfamiliar area of VisIt or existing behavior is being changed, then a reviewer should be added.

Non-owners must always have at least one reviewer even if you satisfy all other guidelines for the *No Reviewers* case.

Two or more reviewers

If your changes substantially modify existing behavior or you are updating significant amounts of the code or you are designing new architectures or interfaces, then you should have at least two reviewers.

Choosing Reviewers

GitHub automatically suggests reviewers based on the blame data for the files you have modified. You should choose the GitHub suggested reviewer unless you have a specific need for a specific reviewer.

10.4.5 Iteration Process

Review processes are iterative by nature, and PR reviews are no exception. A typical review process looks like this:

1. The developer submits a PR and selects a reviewer.
2. The reviewer reviews the PR and writes comments, suggestions, and tasks.
3. The developer gets clarification for anything that us unclear and updates the PR according to the suggestions.

4. Repeat steps 2 and 3 until the reviewer is satisfied with the PR.
5. The reviewer approves the PR.

The actual amount of time it takes to perform a review or update the PR is relatively small compared to the amount of time the PR *waits* for the next step in the iteration. The wait time can be exacerbated in two ways: (1) The reviewer or developer is unaware that the PR is ready for the next step in the iteration process, and (2) the reviewer or developer is too busy with other work. To help alleviate the situation, we recommend the following guidelines for the developer (guidelines for the reviewer can be found here).

- Make sure the code is clear and well commented and that the PR is descriptive. This helps the reviewers quickly familiarize themselves with the context of the changes. If the code is unclear, the reviewers may spend a lot of time trying to grasp the purpose and effects of the PR.
- Immediately answer any questions the reviewers ask about the PR. Enabling notifications will help speed this along.
- When the reviewers have finished reviewing (step 2), quickly update the PR according to the requested changes. Use the `@username` idiom to notify the reviewers for any clarification
- When you have finished updating your PR (step 3), write a comment on the PR using `@username` to let the reviewers know that the PR is ready to be looked at again.

10.5 Reviewing a Pull Request

10.5.1 Overview

Pull Requests (PRs) allow developers to review work before merging it into the develop branch. PRs are extremely useful for preventing bugs, enforcing coding practices, and ensuring changes are consistent with VisIt's overall architecture. Because PR reviews can take time, we have adopted policies to help tailor the review effort and balance the load among developers. We hope these policies will help ensure PR reviews are completed in a timely manner. The benefits of reviews outweigh the added time.

10.5.2 Checklist

In the course of reviewing a PR, the reviewer should use the following as a checklist. The reviewer should verify that any deleted items are rightfully so.

- The developer followed Visit's style guidelines
- The developer commented the code, particularly in hard-to-understand areas
- The developer updated the release notes
- The developer made corresponding changes to the documentation
- The developer added debugging support
- The developer added tests that prove the fix is effective or that the feature works
- The developer has confirmed new and existing unit tests pass
- The developer has NOT changed any *protocol* or public *interfaces* on an RC branch
- If necessary, the developer added any new baselines to the repository

These reminders will appear as checklist items in the PR template. However, not all items *apply* in all PRs. For the items that do apply be sure you have done the associated work and then check off the items by replacing the space in `[ ]` with an `x` (or if you prefer you can submit the PR and then check the boxes with the mouse). For items that do

not apply, be sure to change these lines to strikeout style by adding ~~ just before the check box [ ] (but after the bullet −) and also at the end of the line like so:

```
- [ ] This item is unchecked.
- [x] This item is checked.
- ~~[ ] This item has been striken out.~~
```

10.5.3 Comments and Tasks

GitHub provides two ways to add comments to the PR.

Generic Comments

The first type of comment is a generic PR comment for communicating about general things related to the changes or the PR process. This comment box is found at the bottom of the "Conversation" tab, which is the main tab on the PR page. The reviewer should use this when pinging the developer to update changes (see *Iteration Process* below).

Code Related Comments

The "Files changed" tab in the PR will show a diff of all the changes. Hover the mouse over the white space to the right of the line number and a blue plus sign will appear. Click this and a comment box will pop up. Type any comments and click either "Add single comment" or "Start a review" (see *Review Changes* for more information). This type of comment can be used to ask specific questions or suggest specific changes to the PR.

10.5.4 Review Changes

In addition to comments, the reviewer should also explicitly mark the state of the PR. There are two ways to do this.

Upon writing a code related comment, select the "Start a review" button. This will initiate a review. Click "Add review comment" for each new comment. When you are done, navigate to the top-right of the page and click "Finish your review".

Alternately, the reviewer can first write all the comments and then submit a review. Use the "Add single comment" button for each code related comment. Then, once you have finished commenting, navigate to the top-right of the page and click "Finish your review".

Upon clicking the green "Finish your review", GitHub will present the ability to add additional generic comments and to update the state of the PR. If you left comments via the "Add single comment" button, then you *must* add an additional comment here to be able to submit a review. These are the three options for updating the PR:

1. Comment - Submit general feedback without explicit approval. This is ambiguous and should not be used because the developer does not always know if the reviewer think changes should be made. It does not update the state of the PR.
2. Approve - Submit feedback and approve merging these changes. Use this when the PR is ready to be merged into develop.
3. Request changes - Submit feedback that must be addressed before merging. Use this when the developer should make additional changes to the PR.

10.5.5 Iteration Process

Review processes are iterative by nature, and PR reviews are no exception. A typical review process looks like this:

1. The developer submits a pull request and selects a reviewer.
2. The reviewer writes comments and submit a "Request change" review or an "Approve" review.
3. The developer updates the PR according to the suggestions.
4. Repeat steps 2 and 3 until the PR is ready.
5. The reviewer approves the PR.

The actual amount of time it takes to perform a review or update the PR is relatively small compared to the amount of time the PR *waits* for the next step in the iteration. The wait time can be exacerbated in two ways: (1) The reviewer or developer is unaware that the PR is ready for the next step in the iteration process, and (2) the reviewer or developer is too busy with other work. To help alleviate the situation, we recommend the following guidelines for the reviewer (guidelines for the developer can be found here)

- Immediately address the PR. Enabling notifications will help speed this along.
- If anything in the PR is unclear, ask specific questions using generic or code related comments. Make use of the `@username` idiom to directly ping the developer.
- Clearly mark the review as "Approved" or "Request changes".
- Notify the developer with the `@username` idiom that the PR is ready for updates.
- When the developer has updated the PR, make it a top priority to review it again.
- When the PR is ready to be merged into develop, approve the PR and squash-merge the PR into develop with a succinct description of the changes.

If you are chosen as a reviewer and you know that you will not be able to review the PR in a timely manner, please let the developer know and provide suggestions for who to choose instead. Once you start a PR review, you should make it a priority and stick with it until the end.

10.6 Release Candidate (RC) Development

10.6.1 Overview

VisIt supports three types of releases, major, minor and patch. VisIt's version number is of the form `major.minor.patch`. Patch releases are the most common type of release and typically occur three to four times a year. Minor releases are the next most common type of release and may occur once or twice a year. Major releases happen infrequently, with perhaps several years passing between major releases.

For example, the release sequence goes something like patch (2.12.0), patch (2.12.1), minor (2.13.0), patch (2.13.1), patch (2.13.2), patch (2.13.3), minor (2.14.0), patch (2.14.1), minor (2.15.0), patch (2.15.1), patch (2.15.2), major (3.0).

The VisIt project normally maintains just two stable branches of development. These are the *current* release candidate (RC) branch named something like `3.3RC` and the main *development* branch named `develop`.

All patch releases of VisIt are made from the *current* RC branch. For example, `3.3.0`, `3.3.1`, `3.3.2` and `3.3.3` are all *patch* releases made from the `3.3RC` branch.

When the next *minor* release is to be made, say `3.4.0`, a new RC branch, named `3.4RC`, is created from the current `develop` branch. The `3.4RC` branch becomes the *current* RC branch and all work on the `3.3RC` branch ceases. However, the `3.3RC` branch will remain forever available.

Most work on VisIt is performed first on the *current* RC branch and then the same changes are applied also to the `develop` branch. In this way, `develop` will have all changes that were made on the *current* and all previous RC branches.

Sometimes, short term fixes are performed on the RC branch *only* because they are never intended to become permanent. Likewise, sometimes long term enhancements are performed *only* on the `develop` branch because they introduce significant changes in interfaces and/or dependencies.

Tip: Unless you have been instructed otherwise, plan to do your work *first* on a branch *from* the current RC branch in one pull request and then apply those same changes on a branch *from* `develop` in a second pull request.

Tip: Use VisIt's GitHub milestones page or or reach out to the VisIt team on our GitHub discussions page if you need help identifying the current RC branch or deciding upon which branch you should start your work.

When doing work on the release candidate the normal sequence of operations is as follows:

- A branch is created off the current release candidate.
- Changes are made on the branch.
- A pull request is generated to merge the changes to the current release candidate.
- The changes are then merged into the release candidate.
- A branch is created off of develop.
- The changes from the branch off the release candidate are applied to the branch.
- A pull request is generated to merge the changes to develop.
- The changes are then merged into develop.

In some instances the changes made to the release candidate are not applied to develop, in many instances the exact same changes can be applied to both the release candidate and develop, and in some instannces the changes applied to the two branches are slightly or significantly different.

Changes to files impacting communication protocols or public APIs are not permitted on a release candidate (RC) branch unless explicitly agreed to by the team. Communication protocol files are any XML files and their associated auto-generated header files for *state* objects (any class derived from `AttributeSubject`) passed between VisIt components (e.g. `viewer` and `engine_par`) such as all XML and header files in `src/common/state` and `src/avt/DBAtts/MetaData`. Files impacting public APIs include any XML or header files used by database, plot or operator plugins as well as `src/avt/Database/Database` and `src/avt/Database/Formats`.

The rest of the section will go through the steps of the most common case of making the exact same changes to both branches using an example of updating the 3.0.2 release notes on the 3.0RC and develop.

10.6.2 Creating the RC branch

First you checkout the 3.0RC and then create your branch.

```
git checkout 3.0RC
git checkout -b task/brugger1/2019_09_05_update_release_notes
```

10.6.3 Making the changes

At this point you would modify your branch as you normally do, modifying, adding or deleting files, and then commiting the changes to the branch.

10.6.4 Creating the pull request on the release candidate

Once you have committed all your changes to the branch you are ready to create the pull request. You will start out by pushing your changes to GitHub as normal.

```
git push --set-upstream origin task/brugger1/2019_09_05_update_release_notes
```

Now you go over to GitHub and create your pull request. When creating your pull request, make sure that you are merging it into the release candidate.

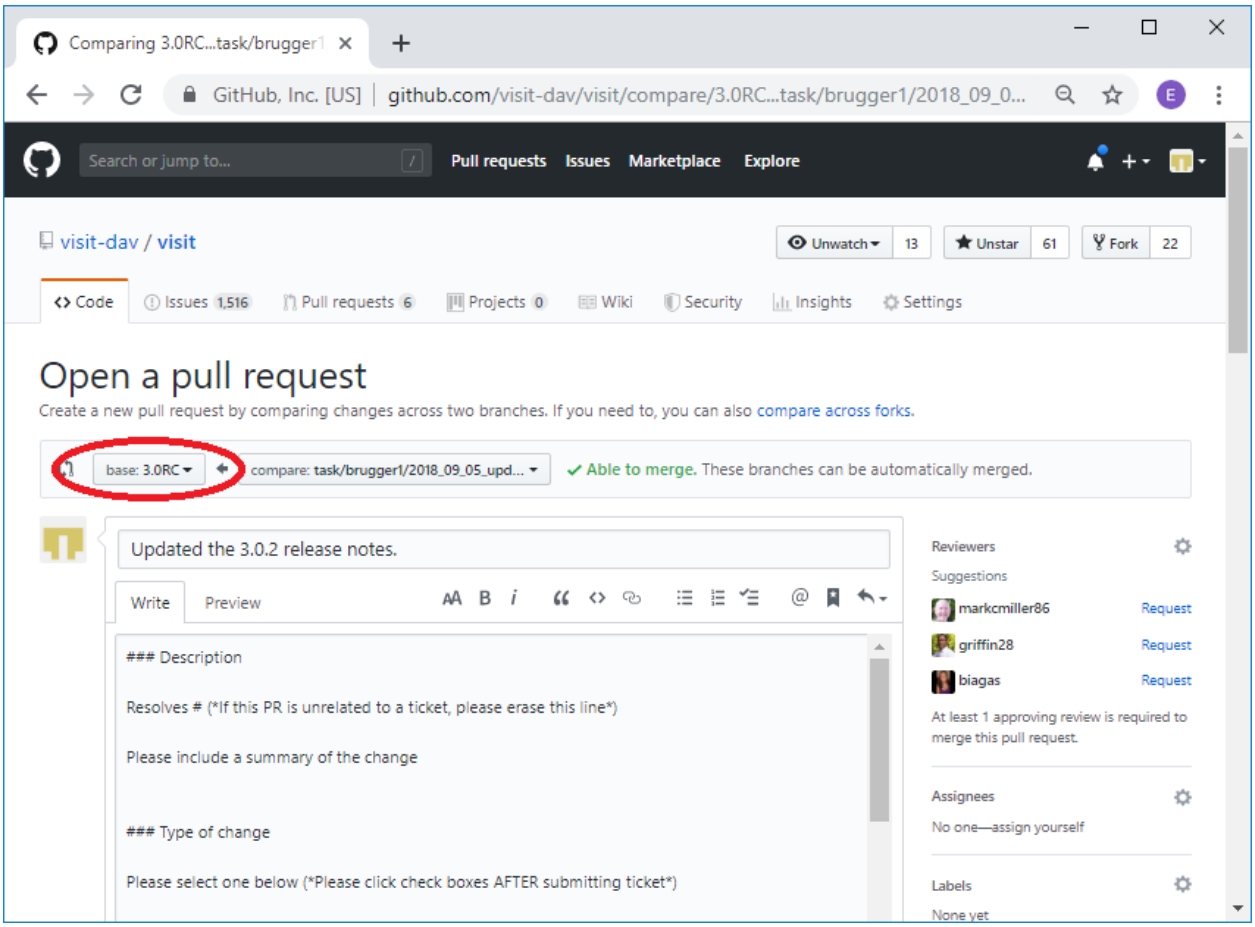


Fig. 10.1: Merging into the release candidate.

Now you go through the normal pull request process. Once you have merged your changes into the release candidate you can delete the branch at GitHub and locally.

```
git remote prune origin
git remote update
git checkout 3.0RC
```

(continues on next page)

(continued from previous page)

```
git pull
git branch -D task/brugger1/2019_09_05_update_release_notes
```

10.6.5 Apply the same changes to develop

Typically, the same changes applied to the release candidate also need to be applied to `develop`. This is not always the case however. Some changes are made only for the release candidate and should never get applied to `develop`. Our practice is to require the *last* comment in every pull request to the release candidate to include a remark indicating either that the PR was not applied to `develop` or that the PR was applied to `develop` along with the commit in which it was applied to `develop`. Typically, the PR for the release candidate has already been closed when this comment needs to be added. This is fine. Developers can still add this comment to a PR when it is in a closed state.

You will apply your changes from the 3.0RC to develop by creating a patch of your changes to the 3.0RC and applying them to a branch created off of develop. The easiest way to create the patch is immediately after you have merged your changes into the release candidate before anyone else makes any changes. In this case you can get the last set of changes from the head. If someone else has made changes in the mean time you will need to use the SHA of your merge to the release candidate. When we create the branch to make the changes on develop, you can use the same name as you used on the release candidate branch and add `_develop`. Normally, you can omit the first two steps below since you presumably just did that a moment ago.

```
git checkout 3.0RC
git pull
rm -f patch.txt
git format-patch -1 HEAD --stdout > patch.txt
git checkout develop
git pull
git checkout -b task/brugger1/2019_09_05_update_release_notes_develop
git am -3 < patch.txt
```

In the case where you need to use the SHA to create the patch, you can get it from the code tab at GitHub for the release candidate branch.

The command to create the patch would then look like:

```
git format-patch -1 69b0561 --stdout > patch.txt
```

Sometimes conflicts occur when applying the patch. This may happen with frequently updated files such as the release notes. If that happens you will get a message similar to the one below indicating which files had conflicts.

```
Applying: Updated the 3.0.2 release notes. (#3867)
Using index info to reconstruct a base tree...
M       src/resources/help/en_US/relnotes3.0.2.html
Falling back to patching base and 3-way merge...
Auto-merging src/resources/help/en_US/relnotes3.0.2.html
CONFLICT (content): Merge conflict in src/resources/help/en_US/relnotes3.0.2.html
error: Failed to merge in the changes.
Patch failed at 0001 Updated the 3.0.2 release notes. (#3867)
The copy of the patch that failed is found in: .git/rebase-apply/patch
When you have resolved this problem, run "git am --continue".
If you prefer to skip this patch, run "git am --skip" instead.
To restore the original branch and stop patching, run "git am --abort".
```

In our case it was the release notes. The file will be modified with the conflicts highlighted in the normal >>>>>>>>>, =========, and <<<<<<<<< notation. You can go in and edit the files and then do a `git add` for each file that was in conflict. After that point you can do a `git am --continue`.

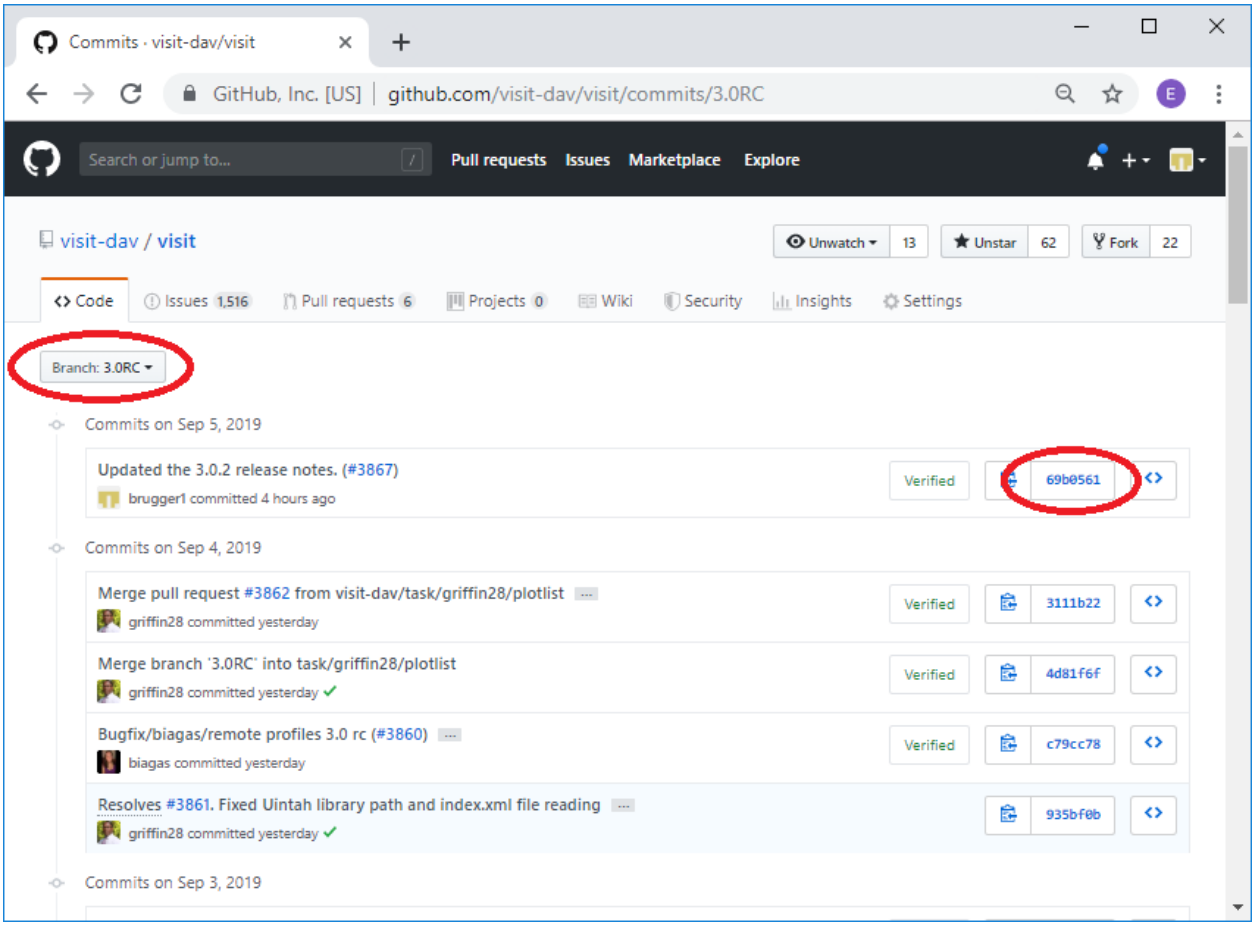


Fig. 10.2: Getting the SHA for the merge into the release candidate.

```
vi src/resources/help/en_US/relnotes3.0.2.html
git add src/resources/help/en_US/relnotes3.0.2.html
git am --continue
```

Now you changes will have been commited to the branch with the appropriate commit message. You are now ready to push the change to GitHub and create a new pull request.

10.6.6 Creating the pull request for develop

You first push your changes to GitHub as normal.

```
git push --set-upstream origin task/brugger1/2019_09_05_update_release_notes_develop
```

Now you go over to GitHub and create your pull request. When creating your pull request, make sure that you are merging it into develop.

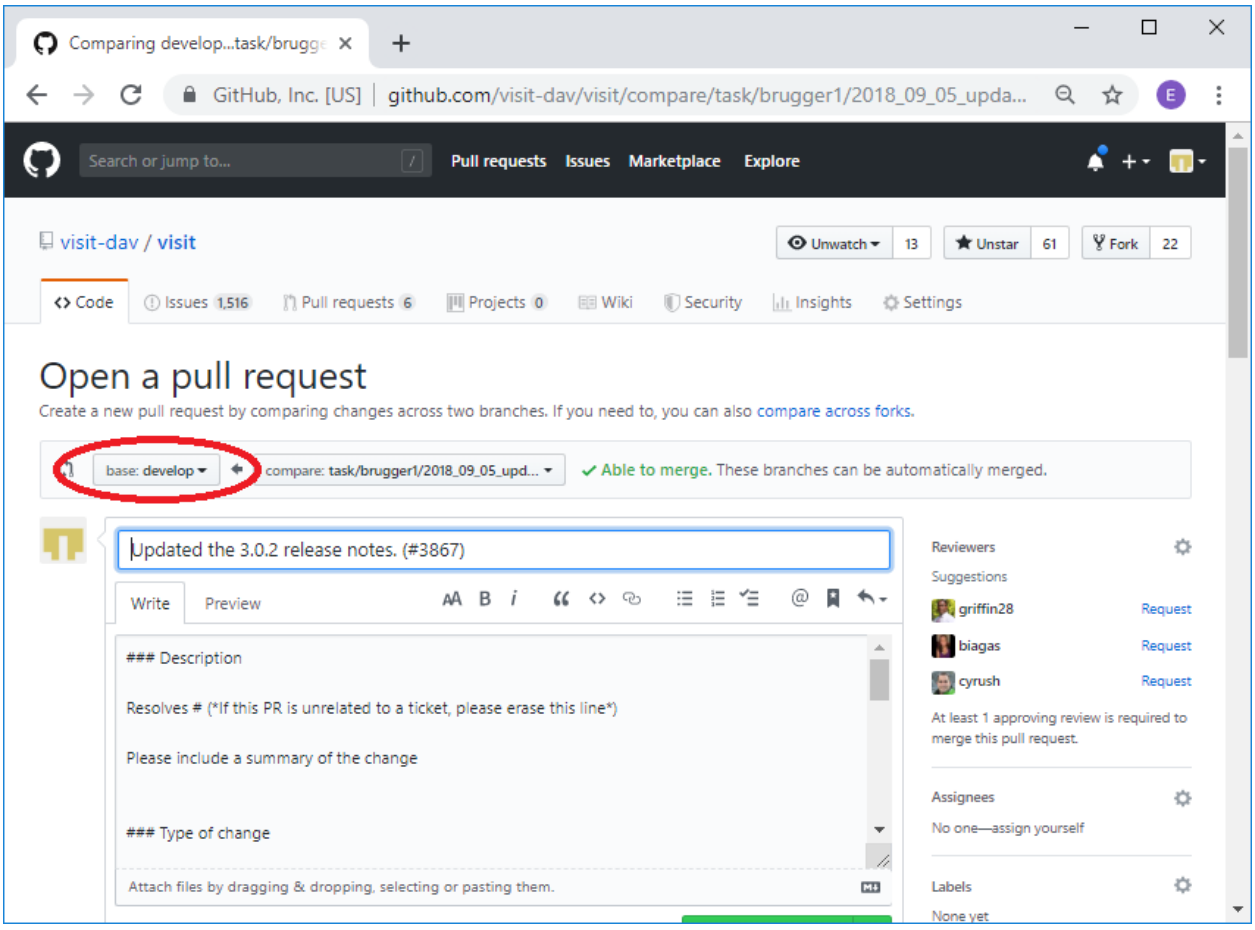


Fig. 10.3: Merging into develop.

In the description you can simply say that you are merging from the release candidate into develop rather than providing all the normal pull request information. If you are resolving an issue, you will want to mention that, since the automatic closing of issues only happens when you merge into `develop`.

Once you have merged your changes into develop you can delete the branch at GitHub and locally.

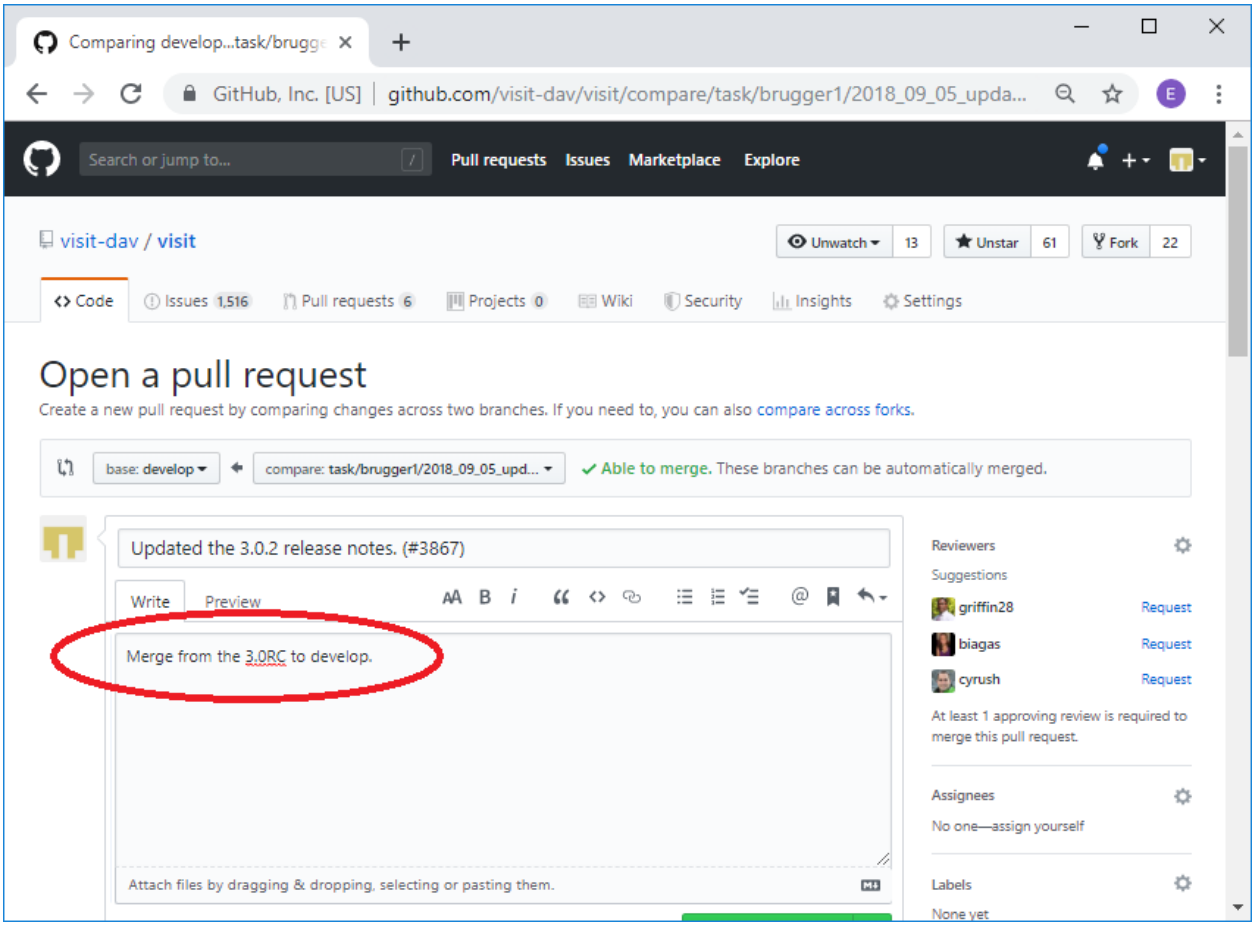


Fig. 10.4: The pull request with the abbreviated description.

```
git remote prune origin
git remote update
git checkout 3.0RC
git pull
git branch -D task/brugger1/2019_09_05_update_release_notes_develop
```

That's it. You have now made the exact same change to both the 3.0RC and develop.

Once the PR to `develop` is merged, go back to the PR for the release candidate (it will probably be in a closed state but that is fine) and add a comment there indicating that the PR was also applied to develop and include the commit, from above, where it happened.

Lastly, sometimes changes worth including in the release candidate nonetheless get done *first* on `develop`. When this happens, we need to backport the changes to the release candidate. A procedure similar to what is described above can be followed except the roles of `develop` and release candidate branches are reversed. In addition, once the changes are backported to the release candidate, go back to the PR for `develop` (it will probably be in a closed state but that is fine) and add a comment there indicating that the changes were also *backported* to the release candidate and include the commit.

10.6.7 Re-review of PRs for merging already reviewed and merged work to a different branch

As described above, there are typically two *active* branches where work may be going on in VisIt; the currently active release candidate branch and `develop`. The common case is for developers to do work on the release candidate and then apply the same work to `develop` using the format-patch or cherry-pick workflows. As noted in the section just above, sometimes the reverse happens and the work is originally done on `develop` and then *backported* to the release candidate.

In either case, the question arises, is a second review of a pull-request of the same work to another branch required? The short answer is no. Work that was done and originally reviewed as a pull request to the release candidate does not require a second review in the pull request and merge to `develop`. This is true even when backporting from `develop` to the release candidate.

However, there are cases where the release candidate and `develop` branches have diverged significantly enough that re-review of the work might be needed. A good indicator of this need is if *conflicts* are encountered when using the format-patch or cherry-pick workflows to merge the changes to a different branch. When that happens, the developer should give some thought as to whether the changes necessary to resolve the conflicts are significant enough that re-review may be required. This is entirely up to the developer doing the work though other developers who may be watching are also free to make a request to re-review the pull request to the different branch.

10.7 Regression Testing

10.7.1 Overview

VisIt has a large and growing test suite. VisIt's test suite involves a combination of python scripts in `src/test`, raw input data files in archives in the top-level `data` directory and data generation tools in `src/tools/data/datagen`. Regression tests are run nightly and results are posted to VisIt's test dashboard. Testing exercises VisIt's viewer, mdserver, engine and cli. The GUI, however, is not exercised during regression testing and is instead tested manually.

10.7.2 Running regression tests

Where nightly regression tests are run

The regression suite is run on LLNL's Pascal Cluster. Pascal runs the TOSS operating system, which is a flavor of Linux. If you are going to run the regression suite yourself you should run on a similar system or there will be differences due to numeric precision issues. If you do have to run the test suite on a different system there are options for doing *fuzzy matching*.

The regression suite is run on Pascal using a cron job that checks out VisIt source code, builds it, and then runs the tests.

A note about git lfs

The regression suite relies on having a working VisIt build and test data available on your local computer. Our test data and baselines are stored using git lfs. Git lfs is an extension to git to *effectively* support large, binary files. To run VisIt's regression suite, git lfs needs to be installed and the command `git lfs pull` needs be run. In addition, `git lfs pull` will likely need to be periodically rerun as various git operations update files.

Files in git lfs exist as either a *pointer/proxy* file or as the *actual/real* file. A git lfs file's content *mutates* between these two state as various git (and git lfs) operations are performed. Only the pointer/proxy file, which is a small (< 150 bytes) text file, is managed in git. The actual/real file, which can be very large, is managed in git lfs. When a file is in its *pointer/proxy* state, its contents look something like

```
version https://git-lfs.github.com/spec/v1
oid sha256:4d7a214614ab2935c943f9e0ff69d22eadbb8f32b1258daaa5e2ca24d17e2393
size 12345
```

The command `git lfs pull` *dereferences* pointer/proxy files causing *all* pointer/proxy files in the currently checked out branch to be replaced with their actual/real contents.

Warning: If there are many pointer/proxy files in the current branch and/or the actual/real files to which the pointer files refer are very large, a `git lfs pull` operation can take a long time (many minutes or more). The operation can be restricted to specific files using the `--include` and `--exclude` options to `git lfs`.

Other git operations can wind up updating the contents of an actual/real lfs file in the local checkout and replacing the file with updated pointer/proxy contents.

Being aware of these two states of a git lfs file is important because when such files are in their pointer/proxy state, various other kinds of VisIt development activities can fail indicating a problem with the file's *format*. For example, expanding a data archive can fail

```
% make ANAME=zipwrapper_test_data.tar.xz expand
[100%] Generating _archive_expand
CMake Error: Problem with archive_read_open_file(): Unrecognized archive format
CMake Error: Problem extracting tar: /Users/miller86/visit/visit/data/zipwrapper_test_
↪data.tar.xzor as another example using ImageMagick's ``display`` command on an lfs'd␣
↪``.png`` file still in its *pointer* state ::
```

Or, trying to display a baseline image can fail

```
% display ../test/baseline//databases/silo/silo_curvilinear_3d_surface_6.png
display: improper image header '../test/baseline//databases/silo/silo_curvilinear_3d_
↪surface_6.png'
```

When this happens, its likely because a `git lfs pull` operation is again needed.

There are other tell tale signs to help recognize whether an lfs'd file is in its pointer/proxy state or actua/real state. In the examples below, `xolotl_test_data.tar.xz` and `xyz_test_data.tar.xz` are in their actual/real state while `zipwrapper_test_data.tar.xz` is in its pointer/proxy state. First, in their pointer/proxy state, the files are very small text files, usually less than 150 bytes

```
% wc -c xolotl_test_data.tar.xz xyz_test_data.tar.xz zipwrapper_test_data.tar.xz
1294672 xolotl_test_data.tar.xz
 348584 xyz_test_data.tar.xz
    132 zipwrapper_test_data.tar.xz
```

The `file` command will show `ASCII text`

```
% file xolotl_test_data.tar.xz xyz_test_data.tar.xz zipwrapper_test_data.tar.xz
xolotl_test_data.tar.xz:     XZ compressed data
xyz_test_data.tar.xz:        XZ compressed data
zipwrapper_test_data.tar.xz: ASCII text
```

The file's contents will show the lfs pointer/proxy data

```
% cat zipwrapper_test_data.tar.xz
version https://git-lfs.github.com/spec/v1
oid sha256:0de21481f2a2e1ddd0eb8e5bcf44e12980285455ce4724557d146c1fa884eb1e
size 6696960
```

A `git lfs pull` operation will mutate all lfs'd pointer/proxy files in the current branch to their actual/real contents. Or, in cases where that operation might take too long, restrict it to the needed files as in

```
git lfs pull --include zipwrapper_test_data.tar.xz
git lfs pull --include ../test/baseline/databases/silo/silo_curvilinear_3d_surface_6.
↪png
git lfs pull --include "*.silo"
```

How to run the regression tests manually

The test suite is written in python and the source is in `src/test`. The main driver to run the whole test suite is `src/test/visit_test_main.py`. Individual test `.py` files are in `src/test/tests/<category>/*.py`. When you configure VisIt, a bash script is generated in the build directory that you can use to run the test suite out of source with all the proper data and baseline directory arguments.

```
cd visit-build/test/
./run_visit_test_suite.sh
```

Here is an example of the contents of the generated `run_visit_test_suite.sh` script

```
/Users/harrison37/Work/github/visit-dav/visit/build-mb-develop-darwin-10.13-x86_64/
↪thirdparty_shared/third_party/python/2.7.14/darwin-x86_64/bin/python2.7
/Users/harrison37/Work/github/visit-dav/visit/src/test/visit_test_suite.py \
   -d /Users/harrison37/Work/github/visit-dav/visit/build-mb-develop-darwin-10.13-x86_
↪64/build-debug/testdata/  \
   -b /Users/harrison37/Work/github/visit-dav/visit/src/test/../../test/baseline/   \
   -o output \
   -e /Users/harrison37/Work/github/visit-dav/visit/build-mb-develop-darwin-10.13-x86_
↪64/build-debug/bin/visit "$@"
```

Once the test suite has run, the results can be found in the `output/html` directory. There, you will find an `index.html` file entry that you can use to browse all the results.

If you want to restrict the amount of parallelism used in running the test suite you can do so with the `-n` command line option. By default, the test suite will be run using all the cores on your system. We have found that on some systems, running more than one test at a time may result in failures. To work around this issue you can run one test at a time.

```
./run_visit_test_suite.sh -n 1
```

If you want to run a single test or just a few tests from the test suite you can list them on the command line. The list of tests must be the last entries on the command line.

```
./run_visit_test_suite.sh -n 1 tests/databases/silo.py tests/databases/xdmf.py
```

There are a number of additional command-line options to the test suite. `./run_visit_test_suite.sh -help` will give you details about these options.

Accessing nightly regression test results

The nightly test suite results are posted to GitHub.

In the event of failure on the nightly run

If any tests fail, **all** developers who updated the code from the last time all tests successfully passed will receive an email indicating *something* failed. In addition, failed results should be available on the web.

10.7.3 How regression testing works

The workhorse script that manages the testing is `visit_test_suite.py` in `src/test`. Tests can be run in a variety of ways called *modes*. For example, VisIt's nightly testing is run in `serial`, `parallel` and `scalable,parallel,icet` modes. Each of these modes represents a fundamental and relatively global change in the way VisIt is doing business under the covers during its testing. For example, the difference between `parallel` and `scalable,parallel,icet` modes is whether the scalable renderer is being used to render images. In the `parallel` mode, rendering is done in the viewer. In `scalable,parallel,icet` mode, it is done, in parallel, on the engine and images from each processor are composited with IceT. Typically, the entire test suite is run in each mode specified by the regression test policy.

The mode is specified with the `-m` command line option. For example, to run in `scalable,parallel,icet` mode use:

```
./run_visit_test_suite.sh -n 1 -m "scalable,parallel,icet"
```

For simplicity, we maintain baselines only for one *blessed* platform which is conveniently accessible to the *core* development team. Running the test suite anywhere else requires the use of *fuzzy matching* to ignore minor differences. Use of these options on platforms other than the currently adopted testing platform will facilitate filtering big differences (and probably real bugs that have been introduced) from differences due to platform or configuration.

There are a number of different categories of tests. The test categories are the names of all the directories under `src/test/tests`. The `.py` files in this directory tree are all the actual test driver files that drive VisIt's CLI and generate images and text to compare with baselines. In addition, the `src/test/visit_test_main.py` file defines a number of helper Python functions that facilitate testing including two key functions; `Test()` for testing image outputs and `TestText()` for testing text outputs. Of course, all the `.py` files in `src/test/tests` subtree are excellent examples of test scripts.

When the test suite finishes, it will have created a web-browseable HTML tree in the html directory. The actual image and text raw results will be in the current directory and difference images will be in the diff directory. The difference

images are essentially binary bitmaps of the pixels that are different and not the actual pixel differences themselves. This is to facilitate identifying the location and cause of the differences.

Adding a test often involves:

a) adding a `.py` file to the appropriate test *category* subdirectory in `src/test/tests`,

b) optionally adding the expected baseline files to `test/baselines` and, depending on the test,

c) optionally adding any necessary input data files to the top-level `data` directory.

Warning: Steps b) and c) can almost never be avoided for tests involving new database plugins. However, in almost all other cases, steps b) and c) can and probably should be avoided. Instead, developers are encouraged to adopt new practices and use new testing features where tests *and* their expected outcomes are programmatically included in *just* the `.py`, so there is no need for separate *baseline* files and/or new data files.

The test suite will find your added `.py` files the next time it runs. So, you don't have to do anything special other than adding the .py file.

One subtlety about the current test modality is what we call *mode specific baselines*. In theory, it should not matter what mode VisIt uses to produce an image or numerical/textual output. The results should be identical across modes. In practice there is a long list of things that can contribute to subtle pixel differences images and small numerical differences in text. This has lead to mode specific baselines. In the baseline directory, there are subdirectories with names corresponding to modes we currently run. When it becomes necessary to add a mode specific baseline, the baseline file should be added to the appropriate baseline subdirectory.

In some cases, we skip a test in one mode but not in others. Or, we temporarily disable a test by skipping it until a given problem in the code is resolved. This is handled by the `--skiplist` argument to the test suite. We maintain a list of the tests we currently skip and update it as necessary. The default skip list file is `src/test/skip.json`.

Types of Test Results

VisIt's testing system, `visit_test_main.py`, uses three different methods to process and check results.

- `Test()` and `TestAutoName()` which processes `.png` image files
- `TestText()` and `TestTextAutoName()` which process `.txt` text files.
- `TestValueXX()` (where `XX``==>``EQ`, `LT`, `LE`, etc.) which processes no files and simply checks *actual* and *expected* values passed as arguments.
- `TestPOA()` and `TestFOA()` which integrate directly with python if-then-else and try-except logic.

The `Test()` and `TestText()` methods both take the name of a file. To process a test result, these methods output a file produced by the *current* test run and then compare it to a blessed *baseline* file stored in test/baseline.

The `TestAutoName()` and `TestTextAutoName()` methods are preferred and perform the equivalent work of `Test()` and `TestText()` but generate the names of the baseline files automatically. The auto-naming algorithm requires that the `.py` file be structured such that calls to `TestAutoName()` and/or `TestTextAutoName()` are made from within only top-level functions in the `.py` file. Auto naming does not work if these methods are called from either the top/main of the `.py` file or from functions two or more levels deep. Auto naming catenates the `.py` file's name with the name of the top-level function from which the call was made and adds an index/count.

Below, we outline the preferred structure for a VisIt test `.py` file. The file is divided into top-level functions and calls to the various `TestXXX()` methods are issued from within one of these top-level functions. Each top-level function performs one or more related tests involving common or highly similar setup. Each top-level function is then invoked from the `.py` file's main body. Each top-level function should return to main leaving the VisIt session in largely the same state as before the top-level function was invoked. This includes deleting all associated plots, closing all

associated databases, and possibly resetting any other relevant global state such as the view, lights, color table, SIL selection, etc.

Given a python file named `gorfo.py` structured as below, the resulting auto generated names (and section names) are indicated in the associated comments.

```
def histogram():
    ...
    TestAutoName() # Uses baseline file named 'gorfo_histogram_0' and calls␣
↪TestSection('histogram')
    ...
    TestAutoName() # Uses baseline file named 'gorfo_histogram_1'
    ...
    TestValueEQ(name,bval,cval) # Compares baseline value, bval, to current value,␣
↪cval

def curve():
    ...
    TestAutoName() # Uses baseline file named 'gorfo_curve_0' and calls TestSection(
↪'curve')
    ...
    TestAutoName() # Uses baseline file named 'gorfo_curve_1'

#
# Main code
#

# Run the Histogram tests
histogram()

# Run the curve tests
curve()
```

The one down side to using the auto-naming methods is that later restructuring of the python code can lead to changes in names of the baseline files. Existing, top-level functions can be moved relative to each other without issue. New tests can be added without issue. But, removing *earlier* tests from a function or moving tests relative to each other *within* a function leads to baseline file name changes.

When they can be used, the `TestValueXX()` are a little more convenient because they do not involve storing data in files and having to maintain separate baseline files. Instead the `TestValueXX()` methods take both an *actual* (current) and *expected* (baseline) result as arguments directly coded in the calling `.py` file. A good example of using `TestValueXX()` can be found in `src/test/tests/database/boxlib.py`.

```
#
# Test double precision is working by reading a known double precision
# database and ensuring we get expected min/max values within 15 digits
# of accuracy.
#
DeleteAllPlots()
CloseDatabase(data_path("boxlib_test_data/3D/plt00000.cartgrid.body.small/Header"))
OpenDatabase(data_path("boxlib_test_data/2D/plt0000000/Header"))
AddPlot("Pseudocolor", "temperature1")
DrawPlots()
SetQueryOutputToValue()
TestValueEQ("temperature1 min", Query("Min"), 295.409999999999968, 15)
TestValueEQ("temperature1 max", Query("Max"), 295.410000000000082, 15)
DeleteAllPlots()
CloseDatabase(data_path("boxlib_test_data/2D/plt0000000/Header"))
```

Likewise, the `TestPOA()` (pass on arrival) and `TestFOA()` (fail on arrival) methods are convenient ways to implement a test based primarily upon python logic itself with if-then-else or try-except blocks. These methods are useful for cases where the majority of logic for determining a passed or failed test exists primarily as the python code itself being executed. A good example is the `src/test/tests/quickrecipes/working_with_annotations.py` tests

```
try:
  # using gradient background colors {
  # Set a blue/black, radial, gradient background.
  a = AnnotationAttributes()
  a.backgroundMode = a.Gradient
  a.gradientBackgroundStyle = a.Radial
  a.gradientColor1 = (0,0,255,255) # Blue
  a.gradientColor2 = (0,0,0,255) # Black
  SetAnnotationAttributes(a)
  # using gradient background colors }
  TestValueEQ('using gradient background colors error message',GetLastError(),'')
  TestPOA('using gradient background colors exceptions')
except Exception as inst:
  TestFOA('using gradient background colors exception "%s"'%str(inst), LINE())
  pass
```

While there may be many instances of `TestFOA()` (many ways a given bit of logic can fail) with the same `name` argument in a given sequence of logic for a single test outcome, they can be differentiated by a unique *tag* (typically the `LINE()` method identifing the line number. However, there should be only a single `TestPOA()` (the one way a given bit of logic can succeed) instance with the same name for the associated test outcome.

As VisIt testing has evolved, understanding and improving productivity related to test design has not been a priority. As a result, there are likely far more image test results than are truly needed to fully vet all of VisIt's plotting features. Or, image tests are used unecessarily to confirm non-visual behavior like that a given database reader is working. Some text tests are better handled as `TestValueXX()` tests and other text tests often contain 90% *noise* text unrelated to the functionality being tested. This has made maintaining and ensuring portability of the test suite more laborious.

Because image tests tend to be the most difficult to make portable, a better design would minimize image tests to only those needed to validate visual behaviors, text tests would involve only the *essenteial* text of the test and a majority of tests would involve *value* type tests.

The above explanation is offered as a rational to justify that whenever possible adding *new* tests to the test suite should use the `TestValueXX()` or `TestPOA()`/`TestFOA()` approach as much as practical.

More About TestValueXX and TestPOA/FOA Type Tests

The `TestValueXX()` methods are similar in spirit to `Test()` and `TestText()` except operates on Python *values* passed as args both for the *current* (actual) and the *baseline* (expected) results. The values can be any Python object. When they are floats or ints or strings of floats or ints or lists/tuples of the same, these methods will round the arguments to the desired precision and do the comparisons numerically. Otherwise they will compare them as strings.

`TestValueEQ(case_name, actual, expected, prec=5)`: Passes if `actual == expected` within specific precision otherwise fails.

`TestValueNE(case_name, actual, expected, prec=5)`: Passes if `actual != expected` within specific precision otherwise fails.

`TestValueLT(case_name, actual, expected, prec=5)`: Passes if `actual < expected` within specific precision otherwise fails.

TestValueLE(case_name, actual, expected, prec=5) : Passes if `actual <= expected` within specific precision otherwise fails.

TestValueGT(case_name, actual, expected, prec=5) : Passes if `actual > expected` within specific precision otherwise fails.

TestValueGE(case_name, actual, expected, prec=5) : Passes if `actual >= expected` within specific precision otherwise fails.

TestValueIN(case_name, bucket, expected, eqoper=operator.eq, prec=5) : Passes if bucket *contains* expected according to `eqoper` equality operator. Fails otherwise.

TestFOA(name, tag='unk') Fail on arrival with test case outcome name the concatenation of `name` and `tag`. Whenever python execution arrives at a line with `TestFOA()`, the test is considered a failure. Typically, `tag` is `LINE()` to indicate the python line number where failure occured. A given bit of test logic (e.g. a test *case*) can have *many* `TestFOA()` calls of the same `name` but with different `tag`.

TestPOA(name) Pass on arrival with test case outcome name just `name`. Whenever python execution arrives at a line with `TestPOA()`, the test is considered a pass. A given bit of test logic (e.g. a test *case*) should have *only one* `TestPOA()` call.

For some examples, see test_values_simple.py and atts_assign.py.

Filtering Image Differences

There are many alternative ways for both compiling and even running VisIt to produce any given image or textual output. Nonetheless, we expect results to be nearly if not perfectly identical. For example, we expect VisIt running on two different implementations of the GL library to produce by and large the same images. We expect VisIt running in serial or parallel to produce the same images. We expect VisIt running on Ubuntu Linux to produce the same images as it would running on Mac macOS. We expect VisIt running in client-server mode to produce the same images as VisIt running entirely remotely.

In many cases, we expect outputs produced by these alternative approaches to be nearly the same but not always bit-for-bit identical. Minor variations such as single pixel shifts in position or slight variations in color are inevitable and ultimately unremarkable.

When testing, it would be nice to be able to ignore variations in results attributable to these causes. On the other hand, we would like to be alerted to variations in results attributable to changes made to the source code.

To satisfy both of these goals, we use bit-for-bit identical matching to track the impact of changes to source code but *fuzzy* matching for anything else. We maintain a set of several thousand version-controlled, baseline results computed for a specific, fixed *configuration and test mode* of VisIt. Nightly testing of key branches of development reveals any results that are not bit-for-bit identical to their baseline.

These *failures* are then corrected in one of two ways. Either the new result is wrong and additional source code changes are required to ensure VisIt continues to produce the original baseline. Or, the original baseline is wrong and it must be updated to the new result. In this latter situation, it is also prudent to justify the new result with a plausible explanation as to why it is expected, better or acceptable as well as to include such explanation in the commit comments.

Mode specific baselines

VisIt testing can be run in a variety of modes; serial, parallel, scalable-parallel, scalable-parallel-icet, client-server, etc. For a fixed configuration, in most cases baseline results computed in one mode agree bit-for-bit identically with the other modes. However, this is not always true. About 2% of results vary with the execution mode. To handle these cases, we also maintain *mode-specific* baseline results as the need arises.

The need for a mode-specific baseline is discovered as new tests are added. When testing reveals that VisIt computes slightly different results in different modes, a single mode-agnostic baseline will fail to match in all test modes. At that time, mode-specific baselines are added.

Changing Baseline Configuration

One weakness with this approach to testing is revealed when it becomes necessary to change the configuration used to compute the baselines. For example, moving VisIt's testing system to a different hardware platform or updating to a newer compiler or third-party library such as VTK, may result in a slew of minor variations in the results. Under these circumstances, we are confronted with having to individually assess possibly thousands of *minor* image differences to rigorously determine whether the new result is in fact *good* or whether some kind of issue or bug is being revealed.

In practice, we use fuzzy matching (see below) to filter out *minor* variations from *major* ones and then focus our efforts only on fully understanding the *major* cases. We summarily *accept* all minor variations as the *new* baselines.

Promise of Machine Learning

In theory, we should be able to develop a machine-learning approach to filtering VisIt's test results that enable us to more effectly attribute variations in results to various causes. A challenge here is in developing a sufficiently large and fully labeled set of example results to prime the machine learning. This would make for a great summer project.

Fuzzy Matching Metrics

Image difference metrics are reported on terminal output and in HTML reports.

Total Pixels (`#pix`) : Count of all pixels in the test image

Non-Background (`#nonbg`) : Count of all pixels which are not background either by comparison to constant background color or if a non-constant color background is used to same pixel in background image produced by drawing with all plots hidden. Note that if a plot produces a pixel which coincidentally winds up being the same color as the background, our accounting logic would count it as *background*. We think this situation is rare enough as to not cause serious issues.

Different (`#diff`) : Count of all pixels that are different from the current baseline image.

% Diff. Pixels (`~%diff`) : The *precentage* of different pixels computed as `100.0*#diff/#nonbg`

Avg. Diff (`avgdiff`) : The average *luminance* (gray-scale, obtained by weighting RGB channels by 1/3rd and summing) difference. This is the sum of all pixel luminance differences divided by `#diff`.

Fuzzy Matching Thresholds

There are some command-line arguments to run tests that control *fuzzy* matching. When computed results match bit-for-bit with the baseline, a **PASS** is reported and it is colored green in the HTML reports. When a computed result fails the bit-for-bit match but passes the fuzzy match, a **PASS** is reported on the terminal and it is colored yellow in the HTML reports.

Pixel Difference Threshold (`--pixdiff`) : Specifies the acceptable threshold for the `#diff` metric as a *percent*. Default is zero which implies bit-for-bit identical results.

Average Difference Threshold (`--avgdiff`) : Specifies the acceptable threshold for the `avgdiff` metric. Note that this threshold applies *only* if the `--pixdiff` threshold is non-zero. If a test is above the `pixdiff` threshold but below the `avgdiff` threshold, it is considered a **PASS**. The `avgdiff` option allows one to specify a second tolerance for the case when the `pixdiff` tolerance is exceeded.

Numerical (textual) Difference Threshold (`--numdiff`): Specifies the acceptable *relative* numerical difference threshold in computed, non-zero numerical results. The relative difference is computed as the ratio of the magnitude of the difference between the current and baseline results and the minimum magnitude value of the two results.

The command-line with `--pixdiff=0.5 --avgdiff=0.1` means that any result with *fewer* than 0.5% of pixels that are different is a **PASS** and anything with more than 0.5% of pixels different but where the average pixel gray-scale difference is less than .1 is still a **PASS**.

Testing on Non-Baseline Configurations

When running the test suite on platforms other than the currently adopted baseline platform or when running tests in modes other than the standard modes, the `--pixdiff` and `--avgdiff` command-line options will be very useful.

For numerical textual results, there is also a `--numdiff` command-line option that specifies a *relative* numerical difference tolerance in numerical textual results. The command-line option `--numdiff=0.01` means that if a numerical result is different but the magnitude of the difference divided by the magnitude of the expected value is less than `0.01` it is considered a **Pass**.

When specified on the command-line to a test suite run, the above tolerances wind up being applied to *all* test results computed during a test suite run. It is also possible to specify these tolerances in specific tests by passing them as arguments, for example `Test(pixdiff=4.5)` and `TestText(numdiff=0.01)`, in the methods used to check test outputs.

Finally, it may make sense for developers to generate (though not ever commit) a complete and validated set of baselines on their target development platform and then use those (uncommitted) baselines to enable them to run tests and track code changes using an exact match methodology.

Tips on writing regression tests

- Whenever possible, add only new `TestValueXX()` type tests.
- Test images in which plots occupy a small portion of the total image are fraught with peril and should be avoided. Images with poor coverage are more likely to produce false positives (e.g. passes that should have failed) or to exhibit somewhat random differences as test scenario is varied.
- Except in cases where annotations are being specifically tested, remember to call TurnOffAllAnnotations() as one of the first actions in your test script. Otherwise, you can wind up producing images containing machine-specific annotations which will produce differences on other platforms.
- When setting plot and operator options, take care to decide whether you need to work from *default* or *current* attributes. Methods to obtain plot and operator attributes optionally take an additional `1` argument to indicate that *current*, rather that *default* attributes are desired. For example `CurveAttributes()` returns *default* **Curve** plot attributes wherease `CurveAttributes(1)` returns *current* **Curve** plot attributes which will be the currently active plot, if it is a **Curve** plot or the first **Curve** plot in the plot list of the currently active window whether it is active or hidden. If there is no **Curve** plot available, it will return the *default* attributes.
- When writing tests involving text differences and file pathnames, be sure that all pathnames in the text strings passed to `TestText()` are absolute. Internally, VisIt testing system will filter these out and replace the machine-specific part of the path with `VISIT_TOP_DIR` to facilitate comparison with baseline text. In fact, the .txt files that get generated in the *current* dir will have been filtered and all pathnames modified to have `VISIT_TOP_DIR` in them.
- Here is a table of python tests scripts which serve as examples of some interesting and lesser known VisIt/Python scripting practices:

Script	What it demonstrates
tests/faulttolerant/savewindow.py	• uses python exceptions
tests/databases/itaps.py	• uses OpenDatabase with specific plugin • uses SIL restriction via names of sets
tests/databases/silo.py	• uses OpenDatabase with virtual database and a specific timestep
tests/rendering/scalable.py	• uses OpenComputeEngine to launch a parallel engine
tests/rendering/offscreensave.py	• uses Test() with alternate save window options
tests/databases/xform_precision.py	• uses test-specific enviornment variable settings

Rebaselining Test Results

A python script, `rebase.py`, in the `test/baseline` dir can be used to rebaseline large numbers of results. In particular, this script enables a developer to rebase test results without requiring access to the test platform where testing is performed. This is becase the PNG files uploaded (e.g. posted) to VisIt's test results dashboard are suitable for using as baseline results. To use this script, run `./rebase.py --help`.

Here is an example workflow to rebaseline a set of results that were originally committed from macOS and are subtley different on the tier 1 testing platform we use for nightly testing...

1. First, go to the test dashboard and browse for any failed results. Ensure you are browsing the *current* results from the previous evening. Failing results will appear something like what is shown below...

   Be sure to scroll through the *entire* table of results to find all failures.

2. To learn more about which specific tests are failing, click into them and they will appear something like what is shown below...

3. To learn even more specific details about each failing case, click into them to find details which will appear something like what is shown below...

4. Take note of some of the components of the URL of these cases. This information is needed if the results need to be rebaselined.

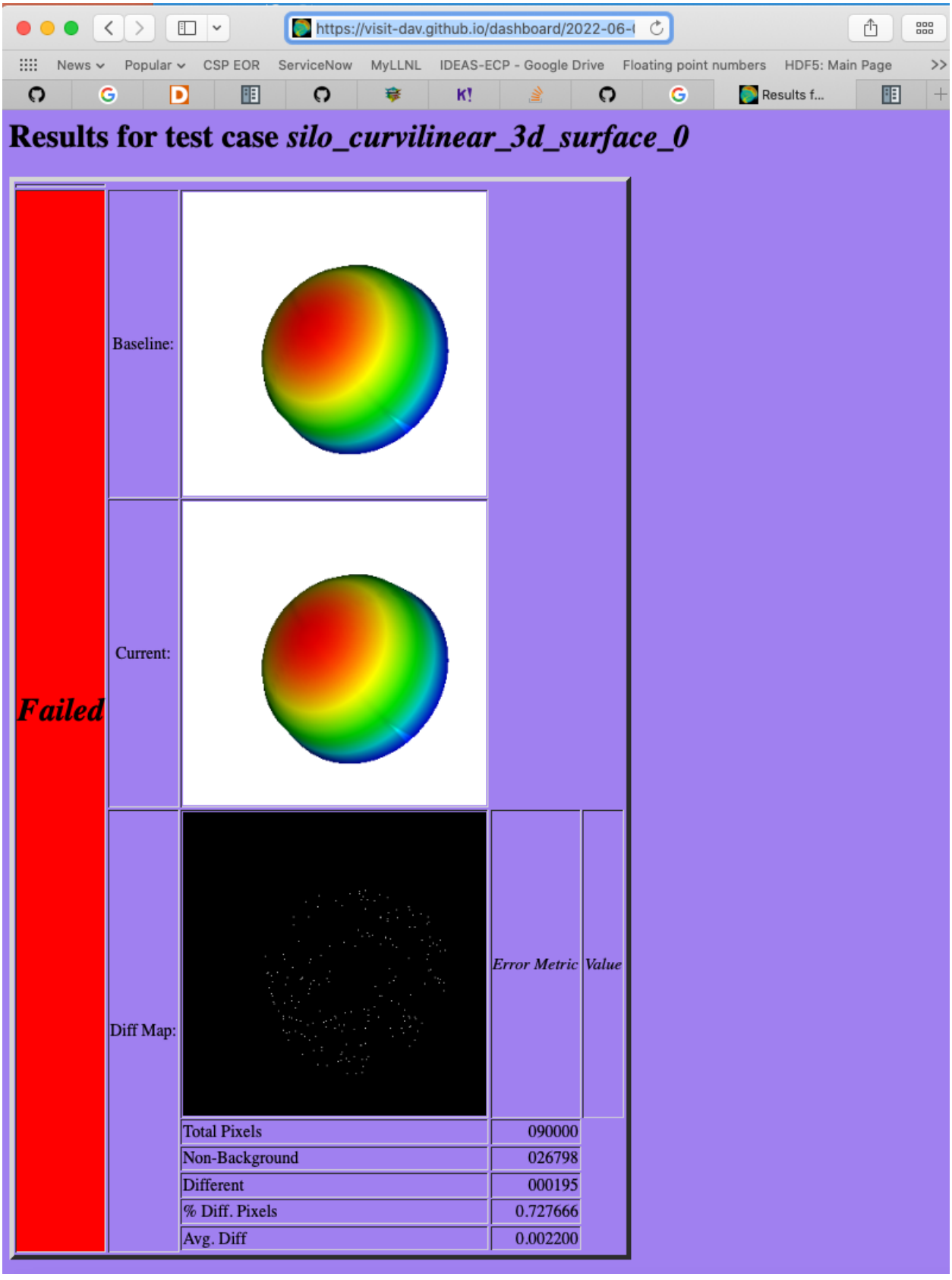
If after examining the results, the new results are deemed the *correct* ones, the baselines need to be updated. Use `rebase.py` for that. That python script is designed to be launched as a standalone application. So, the invocation looks something like...

```
% ./rebase.py -c databases -p silo -m serial -d '2022-06-02-22:00' "silo_curvilinear_
↪3d_surface_*"
Copying file "silo_curvilinear_3d_surface_4.png"
Warning: dramatic change in size of file (old=129/new=5939)"databases/silo/silo_
↪curvilinear_3d_surface_4.png"!
Copying file "silo_curvilinear_3d_surface_5.png"
Warning: dramatic change in size of file (old=129/new=3988)"databases/silo/silo_
↪curvilinear_3d_surface_5.png"!
```

(continues on next page)

63	databases	pixie.py	Succeeded	10.0
64	databases	plaintext.py	Succeeded	8.0
65	databases	rect.py	Succeeded	8.0
66	databases	reopen.py	Succeeded	13.0
67	databases	sami.py	Succeeded	9.0
68	databases	samrai.py	Succeeded	18.0
69	databases	scale_mesh.py	Succeeded	7.0
70	databases	shapefile.py	Succeeded	19.0
71	databases	silo.py	Unacceptable	50.0
72	databases	silo_altdriver.py	Unacceptable	47.0
73	databases	silo_datatypes.py	Succeeded	25.0
74	databases	singlemulti.py	Succeeded	7.0
75	databases	sw4.py	Succeeded	10.0
76	databases	tecplot.py	Succeeded	15.0
77	databases	timesliders.py	Succeeded	11.0
78	databases	uintah.py	Succeeded	8.0
79	databases	unv.py	Succeeded	9.0
80	databases	vtk.py	Succeeded	25.0
81	databases	wave_tv.py	Succeeded	7.0
82	databases	xdmf.py	Succeeded	15.0

Curvilinear (quad) surfaces in 3D					
silo_curvilinear_3d_surface_0	0.73	0.00			
silo_curvilinear_3d_surface_1	0.73	0.00			
silo_curvilinear_3d_surface_2	0.00	0.00			


https://visit-dav.github.io/dashboard/2022-06-
News
Popular
CSP EOR
ServiceNow
MyLLNL
IDEAS-ECP - Google Drive
Floating point numbers
HDF5: Main Page
Results f...
Results for test case silo_curvilinear_3d_surface_0
Failed
Baseline:
Current:
Diff Map:
Error Metric
Value
Total Pixels 090000
Non-Background 026798
Different 000195
% Diff. Pixels 0.727666
Avg. Diff 0.002200

(continued from previous page)

```
Copying file "silo_curvilinear_3d_surface_1.png"
Warning: dramatic change in size of file (old=130/new=24466)"databases/silo/silo_
↪curvilinear_3d_surface_1.png"!
Copying file "silo_curvilinear_3d_surface_0.png"
Warning: dramatic change in size of file (old=130/new=24467)"databases/silo/silo_
↪curvilinear_3d_surface_0.png"!
Copying file "silo_curvilinear_3d_surface_2.png"
Warning: dramatic change in size of file (old=130/new=11474)"databases/silo/silo_
↪curvilinear_3d_surface_2.png"!
Copying file "silo_curvilinear_3d_surface_3.png"
Warning: dramatic change in size of file (old=129/new=2842)"databases/silo/silo_
↪curvilinear_3d_surface_3.png"!
```

The reason for the warnings, above, is that the local files are the LFS *pointer* files. If a `git lfs pull` had been done ahead of time (which is not necessary), then the local files would have been the actual `.png` image files and not the LFS'd pointer files.

Once `rebase.py` is used, don't forget to push the changes in a new PR back to the repository.

10.7.4 Test data archives

Testing VisIt requires input data sets. Because of the wide variety of data formats and readers VisIt supports, we have a wide variety of test data archives. A tar-compatible archive format using the *highest* and *commonly* available compression are the two basic requirements for data archives in our development workflow.

Our practice is to store test data archives as maximally xz compressed, tar-compatible archives. We use xz (e.g. lzma2) compression instead of the more familiar gzip compression because `xz` is known to compress 2-3x smaller and because in most circumstances only VisIt developers (not users) are burdened with having to manage any additional tooling if needed. Any data archives for users, we make available in a choice of compressed formats which include the more familiar gzip compression.

The `CMakeLists.txt` file in the top-level `data` directory is designed to be useable independently of the rest of the VisIt source code tree. After running `cmake` there, the command `make help-archive` explains how to use some convenient `make` targets for managing data archives. We define four convenient `make` targets for creating, expanding and listing data archives. The `archive` target uses python's tarfile module to create a *maximally* xz compressed archive. On some platforms, that operation may fail. If it does, an error message is reported informing the user to use the `fbarchive` target instead.

The `fbarchive` target is a fall-back if the `archive` target fails. It uses CMake's run a command-line tool feature to run `cmake -E tar cvfJ` but may not compress the resultant archive as well. Users are not *required* to use these targets but they are highly recommended to ensure optimal compression and portability of the resulting data archives.

Sometimes, bulk operations on all the test data archives may take a while and developers may desire better or faster tooling. In this case, developers may wish to manipulate the archive and compression tooling directly. For example, this command pipe on linux...

```
tar cvf - my_test_data | xz -9e -T0 - > my_test_data.tar.xz
```

...will create a *maximally* compressed (`-9e`) archive of `my_test_data` using multi-threaded xz compression where the number of threads will be chosen (`-T0`) equal to match the number of hardware cores. For more information about advanced archive and compression operations, readers are encouraged to have a look at the tar and xz man pages.

If users do use tar and compression tools directly to *create* data archives instead of through the convenient make targets, users are required to at least confirm that *expanding* the archives with the `expand` target does work. Doing so will ensure it will work for everyone everywhere.

Adding test data

Sometimes new data files need to be added to support the new tests. This involves adding either an entirely new data archive or adding a new file to an existing data archive. With names like `hdf5_test_data.tar.xz`, all the data archives are named more or less for the data format(s) in which the data files they contain are stored.

Adding new tests

- Add code to an existing `.py` file or create a new `.py` file copying the basic format of an existing one including boilerplat calls to functions like `TurnOffAllAnnotations()`, using `data_path()` when opening a database file and `Exit()` when terminating a test.
- If adding a new `.py` file, be careful to use the correct *category* directory. For example, when writing tests for a new database format, add the `.py` file to the *databases* directory or when adding a new `.py` file to test a new plot, add it to the *plots* directory. To see existing categories, have a look at the directory/folder names in the tests directory. If an entirely new kind of category needs to be introduced, be sure to discuss this with other developers first.
- From within a `.py` file, image results are generated with the `Test()` function and textual results with the `TestText()` function. But, see *above* for why `TestValueXX()` is preferred over image or text results.

Once logic to produce new test results via `Test()`, `TestText()` or `TestValueXX()` are added to a `.py` file, the new tests can be run for the *first* time.

`Test()` and `TestText()` type tests will of course *fail* the first time because there are no associated baseline results defined for them. However, *current* results from `Test()` and `TestText()` type tests will be written to a directory name of the form `output/current/<category>/<.py-file-name>/`. The new results should be inspected for correctness. If they are as expected, to create the baseline results simply copy the new `.png` or `.txt` file(s) to their respective place(s) in the `test/baseline` directory tree being careful to follow the same *category* and *pyfile* name as was introduced above. Of course, don't forget to `git add` them for eventual commit.

Rebaselining for different configurations

Note that if you work on a machine or software configuration different from how VisIt's nightly testing is run, there is a chance the baseline results you create won't match, bit-for-bit, with those same results from nightly testing. Often there can be single-pixel shifts in position or rgb color values can be off by one or two values. Typically the differences are imperceptible except by direct, numerical comparison. Because only developers with access to LLNL CZ systems can *generate* baselines *guaranteed* to match nightly results there, our practice is to permit developers to commit potentially non-matching baselines and allow the nightly tests to run and maybe fail. Then, any developer can use the `rebase.py` tool in `test/baseline` (also see the *above paragraph about using rebase.py*) to update the baselines to whatever nightly testing produced to create perfect matches.

To make debugging a new test case easier, add the `-v` (-verbose flag) or `-v --vargs "-debug 5"` to the `run_visit_test_suite.sh` command, above.

Finally, make sure to tag the test in a comment block with a space separated list of CLASSES and MODES the test supports.

10.7.5 Using VisIt's test routines in other applications

VisIt's testing infrastructure can also be used from any VisIt installation by other applications that want to write their own Visit-based tests. For more details about this, see: Leveraging VisIt in Sim Code RegressionTesting.

10.7.6 Diagnosing pluginVsInstall failures

pluginsVsInstall test output is generated in the `current/plugins` subdirectory of the test results location. There will be a further subdirectory for each type of plugin: databasesVsInstall, operatorsVsInstall and plotsVsInstall. The output consists of text files containing the name of each plugin tested and either `success` or one of the following errors:

- `No installed package.` Indicates a failure in install of VisIt.
- `cmake configure failed` Failure with cmake to configure the plugin for build.
- `make failed` Failure with the build of the plugin.
- `cmake executable could not be found` (rare, just for completeness)
- `make executable could not be found` (rare, just for completeness)

When a failure occurs, another output file is generated in `logs/plugins` subdirectory in the form `<PluginName>_build_res.txt` which should contain sufficient information for fixing the error.

The most likely culprit for errors is missing information in one of the following files:

- `src/include/visit-cmake.h.in` – Holds all the #defines needed for a build (HAVE_LIBXXX, etc).
- `src/CMake/PluginVsInstall.cmake.in` – Ensures third-party include/library locations are correct for an install.
- `src/CMake/FilterDependnecies.cmake.in` – Filters library dependency paths to account for differences between locations of third-party libraries used in a build vs. where they are located within an installed version of VisIt.

10.7.7 Regression testing on Windows

Running the regression suite manually on Windows is a good way to detect Windows-specific run-time errors that may have been inadverently introduced.

A dos-batch script (`run_visit_test_suite.bat`) is generated in the `<build>/test` directory, and is similar to the shell script created on Linux. The generated script turns on `--lessverbose` mode so that output can be viewed while the test is running. Output can be redirected using this syntax:

```
run_visit_test_suite.bat > test_results.txt and 2> test_general_output.txt
```

Windows-specific baselines are stored in the **testing_baselines** subdirectory in the visit-deps repo, and were generated from a Windows 10 system with NVIDIA Quadro P1000 graphics card. Most likely, running from a different system will yield a large number of failures due to minor pixel diffs. The use of *fuzzy matching* to ignore minor differences might be helpful here.

When first running the test suite after new tests have been added, it is generally best to copy the baselines from `visit/test/baselines` to `visit-deps/testing_baselines` to have a good starting point for comparison.

10.8 XML Tools

10.8.1 Overview

VisIt developers use several xml-based code generation tools to implement VisIt's features. The source core for these tools is kept in `src/tools/dev/xml/` and `src/tools/dev/xmledit/`.

10.8.2 Types Managed by XML Tools

These XML tools are designed to be aware of a number of basic data types. These include...

- `Bool`
- `Int`
- `Float`
- `Double`
- `UChar` (aka Unsigned Char)
- `String`

In addition to these *basic* types as *scalars*, we also support `Arrays` of fixed length and `Vectors` of arbitrary length of these basic types. There is also support for a number of types that require special handling. These are...

- types defined in the `avtTypes.h` header file

  Show/Hide Code for `avtTypes.h`

```
#define AVT_TYPES_H

#include <dbatts_exports.h>

#include <vector>
#include <string>

enum avtVarType
{
    AVT_MESH                      = 0,
    AVT_SCALAR_VAR,              /* 1 */
    AVT_VECTOR_VAR,              /* 2 */
    AVT_TENSOR_VAR,              /* 3 */
    AVT_SYMMETRIC_TENSOR_VAR,    /* 4 */
    AVT_ARRAY_VAR,               /* 5 */
    AVT_LABEL_VAR,               /* 6 */
    AVT_MATERIAL,                /* 7 */
    AVT_MATSPECIES,              /* 8 */
    AVT_CURVE,                   /* 9 */
    AVT_UNKNOWN_TYPE             /* 10 */
};


enum avtSubsetType
{
    AVT_DOMAIN_SUBSET       = 0,
    AVT_GROUP_SUBSET,      /* 1 */
    AVT_MATERIAL_SUBSET,   /* 2 */
```

(continues on next page)

(continued from previous page)

```
    AVT_ENUMSCALAR_SUBSET,/* 3 */
    AVT_UNKNOWN_SUBSET    /* 4 */
};


enum avtCentering
{
    AVT_NODECENT          = 0,
    AVT_ZONECENT,        /* 1 */
    AVT_NO_VARIABLE,     /* 2 */
    AVT_UNKNOWN_CENT     /* 3 */
};


enum avtExtentType
{
    AVT_ORIGINAL_EXTENTS      = 0,
    AVT_ACTUAL_EXTENTS,      /* 1 */
    AVT_SPECIFIED_EXTENTS,   /* 2 */
    AVT_UNKNOWN_EXTENT_TYPE  /* 3 */
};


enum avtMeshType
{
    AVT_RECTILINEAR_MESH      = 0,
    AVT_CURVILINEAR_MESH,    /* 1 */
    AVT_UNSTRUCTURED_MESH,   /* 2 */
    AVT_POINT_MESH,          /* 3 */
    AVT_SURFACE_MESH,        /* 4 */
    AVT_CSG_MESH,            /* 5 */
    AVT_AMR_MESH,            /* 6 */
    AVT_UNKNOWN_MESH         /* 7 */
};


enum avtGhostType
{
    AVT_NO_GHOSTS         = 0,
    AVT_HAS_GHOSTS,      /* 1 */
    AVT_CREATED_GHOSTS,  /* 2 */
    AVT_MAYBE_GHOSTS     /* 3 */
};

//
// Note:
// These are used in a bit mask.
// If you need to extend the available types
// make sure to shift new enum values properly.
//
enum avtGhostsZonesPresent
{
    AVT_NO_GHOST_ZONES          = 0,
    AVT_BOUNDARY_GHOST_ZONES,   /* 1 */
    AVT_NESTING_GHOST_ZONES     /* 2 */
};
```

(continues on next page)

(continued from previous page)

```
enum avtMeshCoordType
{
    AVT_XY       = 0,
    AVT_RZ,     /* 1 */
    AVT_ZR      /* 2 */
};

enum avtPrecisionType
{
    AVT_PRECISION_FLOAT     = 0,
    AVT_PRECISION_NATIVE,   /* 1 */
    AVT_PRECISION_DOUBLE    /* 2 */
};

enum avtBackendType
{
    AVT_BACKEND_VTK      = 0,
    AVT_BACKEND_VTKM    /* 1 */
};

enum SetState
{
    NoneUsed             = 0,
    SomeUsed,           /* 1 */
    AllUsed,            /* 2 */
    SomeUsedOtherProc,  /* 3 */
    AllUsedOtherProc    /* 4 */
};


enum LoadBalanceScheme
{
    LOAD_BALANCE_UNKNOWN                          =-1,
    LOAD_BALANCE_CONTIGUOUS_BLOCKS_TOGETHER       = 0,
    LOAD_BALANCE_STRIDE_ACROSS_BLOCKS,          /* 1 */
    LOAD_BALANCE_RANDOM_ASSIGNMENT,             /* 2 */
    LOAD_BALANCE_DBPLUGIN_DYNAMIC,              /* 3 */
    LOAD_BALANCE_RESTRICTED,                    /* 4 */
    LOAD_BALANCE_ABSOLUTE,                      /* 5 */
    LOAD_BALANCE_STREAM                         /* 6 */
};

typedef std::vector<std::string> MaterialList;
```

- types representing some high-level knowledge such as a `LineWidth`, a database `VariableName`, an `Opacity` (slider) value, etc.

The implementation of these types in C++, Java, Python and Qt is handled by various of the `GenerateXXX.h` header files.

For more details, please see this issue.

10.8.3 CMake Integration

We rely on xml code generation to keep our State object, Attribute, and Plugin APIs up-to-date. To automate the process we provide CMake targets that call our xml code generation tools for each object or plugin registered. Individual

code gen targets are all wired into top level targets that allow you to apply the code gen tools to categories of code gen tasks. These targets replace older tools such as regen-ajp and various regenerateatts.py scripts. Keep in mind however, that these targets are only created for plugins that are enabled for building. Any use of the plugin-reducing CMake vars (*VISIT_BUILD_MINIMAL_PLUGINS* and any of the *VISIT_SELECTED_XXX_PLUGINS*) will limit the created code gen targets to those plugins being built.

Top Level CMake Code Gen Targets

CMake Target	Target Action
gen_cpp_all	Run xml2atts on all identified objects
gen_python_all	Run xml2python on all identified objects
gen_java_all	Run xml2java on all identified objects
gen_info_all	Run xml2info on all plugins
gen_cmake_all	Run xml2cmake on all plugins
gen_plugin_all	Run all applicable xml tools for all plugins

CMake Code Gen Functions

These are the helper functions we use to create targets that call xml tools in our CMake build system.

XML Tools Helper functions in `src/CMake/VisItMacros.cmake`:

CMake Function	Target Action
ADD_CPP_GEN_TARGET	Calls xml2atts
ADD_PYTHON_GEN_TARGET	Calls xml2python
ADD_JAVA_GEN_TARGET	Calls xml2java
ADD_INFO_GEN_TARGET	Calls xml2info
ADD_CMAKE_GEN_TARGET	Calls xml2cmake

The xml2plugin and xml2avt tools are only called when you first create a new plugin or object, they are not exposed here.

Plugin Tools Helper functions in `src/CMake/PluginMacros.cmake`:

- **ADD_PLUGIN_CODE_GEN_TARGETS** - wires up:
- **ADD_CPP_GEN_TARGET**
- **ADD_PYTHON_GEN_TARGET**
- **ADD_JAVA_GEN_TARGET**
- **ADD_INFO_GEN_TARGET**
- **ADD_CMAKE_GEN_TARGET**
- **ADD_DATABASE_CODE_GEN_TARGETS** - wires up:
- **ADD_INFO_GEN_TARGET**
- **ADD_CMAKE_GEN_TARGET**
- **ADD_OPERATOR_CODE_GEN_TARGETS** - alias for ADD_PLUGIN_CODE_GEN_TARGETS
- **ADD_PLOT_CODE_GEN_TARGETS** - alias for ADD_PLUGIN_CODE_GEN_TARGETS

10.9 Preparing for a Release

10.9.1 Updating copyright notice dates

At the beginning of every calendar year, the copyright notice needs to be updated. There are only a handful of files that still contain copyright dates, including "/src/LICENSE". There is a script called "update_copyright" in "/src/tools/dev/scripts" that can be used but may need updating if copyright dates were added to any new file types. This should be kept to a minimum.

10.9.2 Preparing for a Patch Release

Preparing for a minor release is pretty straightforward and consists of updating a few files. These consist of

```
VERSION
INSTALL_NOTES
gui/Splashscreen.C
```

A ticket should be created and assigned so that the release can be tested for any obsolete code that should be removed. Testing for obsolete code involves configuring with the CMake var **VISIT_REMOVE_OBSOLETE_CODE** turned on, then compiling and looking for compile errors of the form: *This code is obsolete in this version. Please remove it.*

10.9.3 Preparing for a Minor Release

Preparing for a minor release consists of performing all the steps involved in preparing for a patch release, along with some additional ones, such as creating the release candidate branch and updating the splash screen.

Creating the Release Candidate Branch

Creating a release candidate branch is just like creating a normal branch. Here are the steps used to create the 3.1RC.

```
git checkout develop
git pull
git checkout -b 3.1RC
git push --set-upstream origin 3.1RC
```

Updating the Splashscreen

The splashscreen is the first thing the user sees when running VisIt so the version number included in the splashscreen image should be up to date. Updating the splashscreen usually means just updating the version number in the current splashscreen images but in the event of a major or minor release (when the first or second digit in the version changes), the splashscreen images should be redesigned to showcase new features.

There are two image files associated with the splashscreen, both of which are XCF. XCF files are the native image format of the GIMP image-editing program. One is for the splashscreen and the second is for the icon on MacOS X. They are both used as the first step in the process to create the splashscreen and icon.

The rest of this section will be focused on updating the version number. Changing the images would be the same in terms of the mechanics involved except that it would involve more editing of the image files.

Changing the version on the splashscreen

Follow these steps to update the version on the splashscreen.

1. Go to the `src/common/icons` directory.
2. The splashscreen image's XCF files are named `VisIt3.0.xcf`, `VisIt3.1.xcf`, etc.
3. Copy the file from the last version to the new name for the current version.
4. Open the file in GIMP.

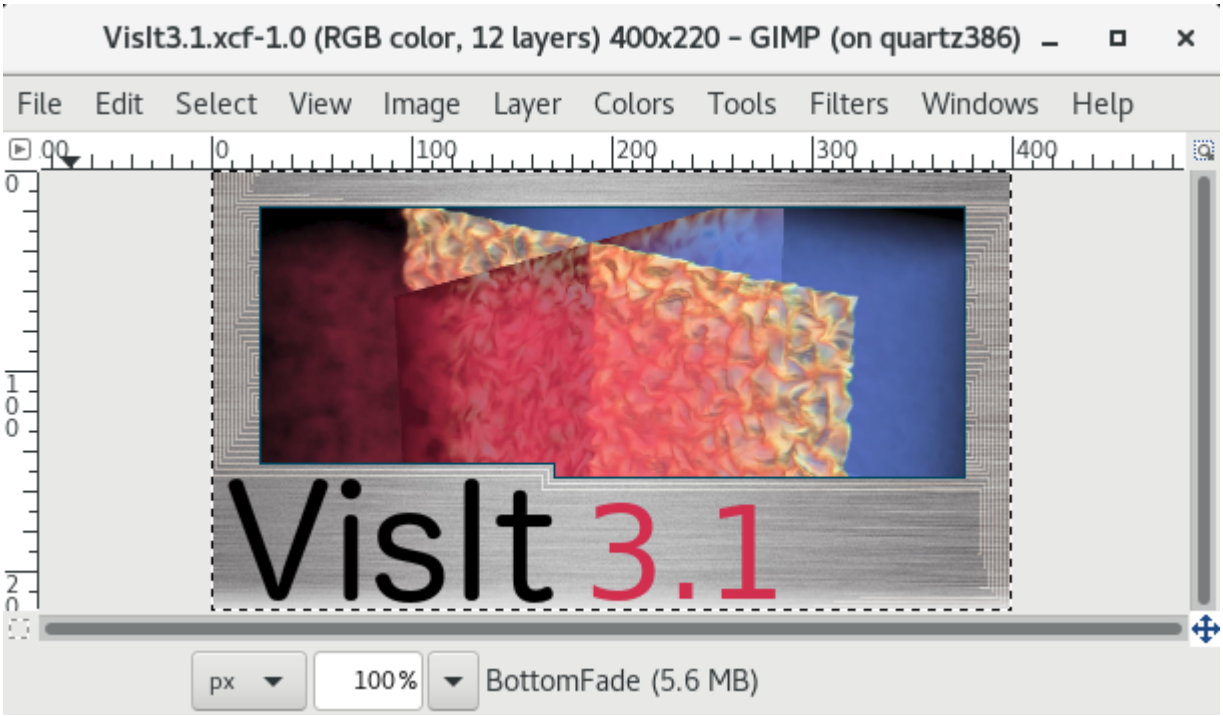


Fig. 10.5: The splashscreen in GIMP.

You'll see that the file has several layers to it. There are four layers for each of the four splash screen images that get randomly choosen from when starting VisIt or are cycled through when you select *About* in the *Help* menu.

4. Select the text layer containing the version number and change it.
5. Save the file.

Now you are ready to create the png images that are actually read in by Qt. When you open the XCF file all the layers corresponding to the four different splashscreen images will be enabled. When you save the first image you will have them all shown. To save the second image you will hide the layer corresponding to the first splashscreen image. You will successively hide one additional layer until you have saved all four of the png images.

6. Go to *File->Export As* and change *Name* to `VisIt1.png`.
7. Click on *Export*.
8. Click on *Export* on the window that pops up to allow you to set the save options.
9. Hide `Background1`.
10. Repeat steps 6 - 9, saving images `VisIt2.png`, `VisIt3.png` and `VisIt4.png`.

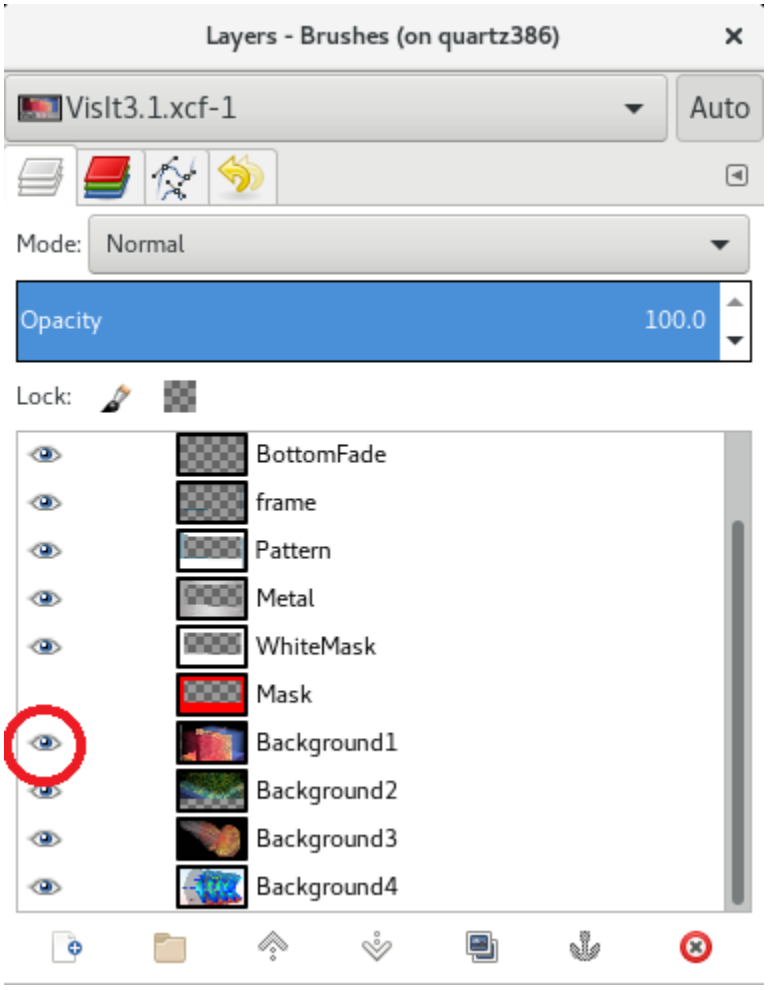


Fig. 10.6: Hiding the Background1 layer in GIMP.

The images saved by GIMP result in warning messages when read by Qt. To modify the images so that the warning message disappears do the following.

```
convert VisIt1.png VisIt1a.png
convert VisIt2.png VisIt2a.png
convert VisIt3.png VisIt3a.png
convert VisIt4.png VisIt4a.png
mv VisIt1a.png VisIt1.png
mv VisIt2a.png VisIt2.png
mv VisIt3a.png VisIt3.png
mv VisIt4a.png VisIt4.png
```

11. Copy the files to `src/resources/images`.

Changing the version on the MacOS X icon

When VisIt starts on MacOS X systems, it adds an icon into the Mac application dock. The icon that we use is based on the splashscreen but is stored in MacOS X icon format.

Follow these steps to update the version on the MacOS X icon.

1. Go to the `src/common/icons` directory.
2. Create the directory `VisItIcon.iconset`.
3. Open the file `VisIt3.x-square.xcf` in GIMP.
4. Select the text layer containing the version number and change it.
5. Go to *Image->Scale Image*.
6. Change the *Image Size Width* and *Height* to `1024`.
7. Click on *Scale*.
8. Go to *File->Export As* and change *Name* to `VisItIcon.iconset/icon_512x512@2x.png`.
9. Click on *Export*.
10. Click on *Export* on the window that pops up to allow you to set the save options.

Now you need to create several sizes of the file. You will use ImageMagick for this.

```
cd VisItIcon.iconset
convert -geometry 512x512 icon_512x512@2x.png icon_512x512.png
convert -geometry 512x512 icon_512x512@2x.png icon_256x256@2x.png
convert -geometry 256x256 icon_512x512@2x.png icon_256x256.png
convert -geometry 256x256 icon_512x512@2x.png icon_128x128@2x.png
convert -geometry 128x128 icon_512x512@2x.png icon_128x128.png
convert -geometry 64x64 icon_512x512@2x.png icon_32x32@2x.png
convert -geometry 32x32 icon_512x512@2x.png icon_32x32.png
convert -geometry 32x32 icon_512x512@2x.png icon_16x16@2x.png
convert -geometry 16x16 icon_512x512@2x.png icon_16x16.png
```

Now you will use iconutil to create the icns file. Note that iconutil is only available on the Mac.

```
cd ..
iconutil --convert icns VisItIcon.iconset
```

Creating a new release notes file

A final step in making a release is to create the release notes file for the *next* release. To do this, you must be reasonably certain what the next release's version number will be. Typically, we do 3-4 patch releases for each minor release. So, if the release you are *just now* making is version `3.1.2`, then the *next* release is likely to be `3.1.3`. However, if the current release is `3.1.3`, the next release might be `3.1.4` or it might be `3.2`.

In any event, to make the release notes file for the *next* release, you need to create an new, empty release notes file by going to `src/resources/help/en_US` and copying either the *minor* release notes template, `relnotes_minor_templ.html`, or the *major* release notes template, `relnotes_major_templ.html` to a file name of the form `relnotesA.B.C.html` where `A.B.C` is the version number for the *next* release. The `.C` part of the file name is missing for *minor* releases.

Patch release notes should go on the RC branch (e.g. `3.1RC`) and minor release notes should go on `develop`. *Always* assume there will be another patch release and just create the next patch release file. If there isn't another patch release, the notes from the patch release can be incorporated into the minor release notes file. When finishing a minor release, create the files for the next minor release *and* the next patch release.

10.9.4 Manual Smoke Check Testing Check List

The following is a list of manual tests to perform once a release has been packaged.

GUI Checks

1. Plot Pseudocolor and Mesh plots for nodal data from curv2D.silo.
2. Plot Pseudocolor and Mesh plots for zonal data from multi_ucd3d.silo.
3. Test Navigation mode (rotate, pan, zoom).
4. Test rubberband zoom.
5. Execute a Node and Zone Pick.
6. Execute a Pick Query.
7. Execute a Lineout.
8. Check for Release Notes and Help.
9. Check VisIt manual was populated in Help.
10. Test "Make Movie" with dba00.pdb.

CLI Checks

1. Start VisIt with CLI and check that *import numpy* works.
2. Test *import visitmodule*.

Additional macOS Checks

1. Check install names for `@rpath`.
2. Test Parallel Launch by plotting procid expr on multi_ucd3d.silo.
3. Make sure to test both the DMG / App Bundle package and the `tar.gz` package.

4. Under Options->Appearance, make sure the GUI style has the macintosh option.
5. Verify OSpray is installed (look at the 'Advanced' tab under Options->Rendering...).
6. Verify that the xmledit tool works from the bundle (/Application/VisIt.app/Contents/Resources/bin/xmledit).
7. Verify that the DMG has been signed with a Developer ID and works properly.
8. Try descending into `Downloads`, `Documents` and `Desktop` from an instance launched by double-clicking the icon and from an instance launched from the Terminal command line.

10.9.5 Preparing for a Major Release

Preparing for a major release is the same as preparing for a minor release with the addition of putting VisIt through the Information Management software release process.

10.10 Creating a Release

10.10.1 Overview

When we put out a new release we create two initial assets and tag the release candidate branch.

1) A source code tar file that includes a `build_visit` script that references the tagged release.
2) A unified `build_visit` script that references the tagged release and includes the checksum for the source code tar file.

10.10.2 Creating the release

We will describe creating a release by way of example using the steps used to create the 3.3.3 release.

Tagging the release

Commit a change that references the tagged release, which in our case is 3.3.3.

```
git checkout 3.3RC
git pull
git checkout -b task/brugger1/2023_03_30_build_visit
vi src/tools/dev/scripts/build_visit
git add src/tools/dev/scripts/build_visit
git commit -m "Temporarily set release in build_visit to v3.3.3."
git push --set-upstream origin task/brugger1/2023_03_30_build_visit
git branch -D task/brugger1/2023_03_30_build_visit
```

At this point you are ready to tag the release.

```
git checkout 3.3RC
git pull
git tag v3.3.3
git push origin v3.3.3
```

Creating first two release assets: source tarball and build_visit script

Now you are ready to create the distribution tar file.

```
src/tools/dev/scripts/visit-dist visit3.3.3
```

Now you are ready to create the unified `build_visit` script.

```
src/tools/dev/scripts/build_visit --write-unified-file build_visit3_3_3
```

Now we revert the `build_visit` script on the 3.3RC branch to point at the 3.3RC.

```
git checkout -b task/brugger1/2023_03_30_build_visit_v2
vi src/tools/dev/scripts/build_visit
git add src/tools/dev/scripts/build_visit
git commit -m "Restore the release in build_visit to 3.3RC."
git push --set-upstream origin task/brugger1/2023_03_30_build_visit_v2
git checkout 3.3RC
git branch -D task/brugger1/2023_03_30_build_visit_v2
```

10.10.3 Creating the release assets page at GitHub

Now we are ready to create a new release assets page at GitHub. If you go to GitHub and go to the *Releases* tab you can create the new release. Click on *Draft a new release* to bring up the form to create a new release.

Now you can choose the tag for the release. Click on *Choose a tag* and select the `v3.3.3` tag.

Now you can describe the release. Occasionally, we update the description of a release after the release is completed to deal with minor changes post-release. Enter `v3.3.3` for the title and add the description as shown below.

At this point you can go to the bottom of the window and click on *Publish release*.

Your newly created release will now appear.

Now you can edit the release and add the unified `build_visit` script and the source code tar file. Do not *Publish* the release as you add assets. You should just *Save* the release. If you publish the release, a notification will go out to everyone watching the VisIt repository and it will show up as the latest release on the releases tab. You should wait until you have most, if not all, of the assets before publishing the release.

10.10.4 Updating the VisIt website

Once you have created the release you will need to update the VisIt website. It is in the repository `visit-dav/visit-website`. Changes are typically made to the VisIt website repository directly on the master branch.

Creating the release notes

The website contains a copy of the release notes. The release notes in the VisIt repository is written in raw html and the website is written in Markdown. The release notes are located in `pages/releases`. Copy one of the release notes files that is closest to the type of release you are making, either a patch release or a minor release, as a starting point. Remove or update any version specific content from the new release notes file. Copy the raw html from the VisIt repository release notes into the Markdown release notes file and convert the raw html into Markdown. Commit the changes when you are finished.

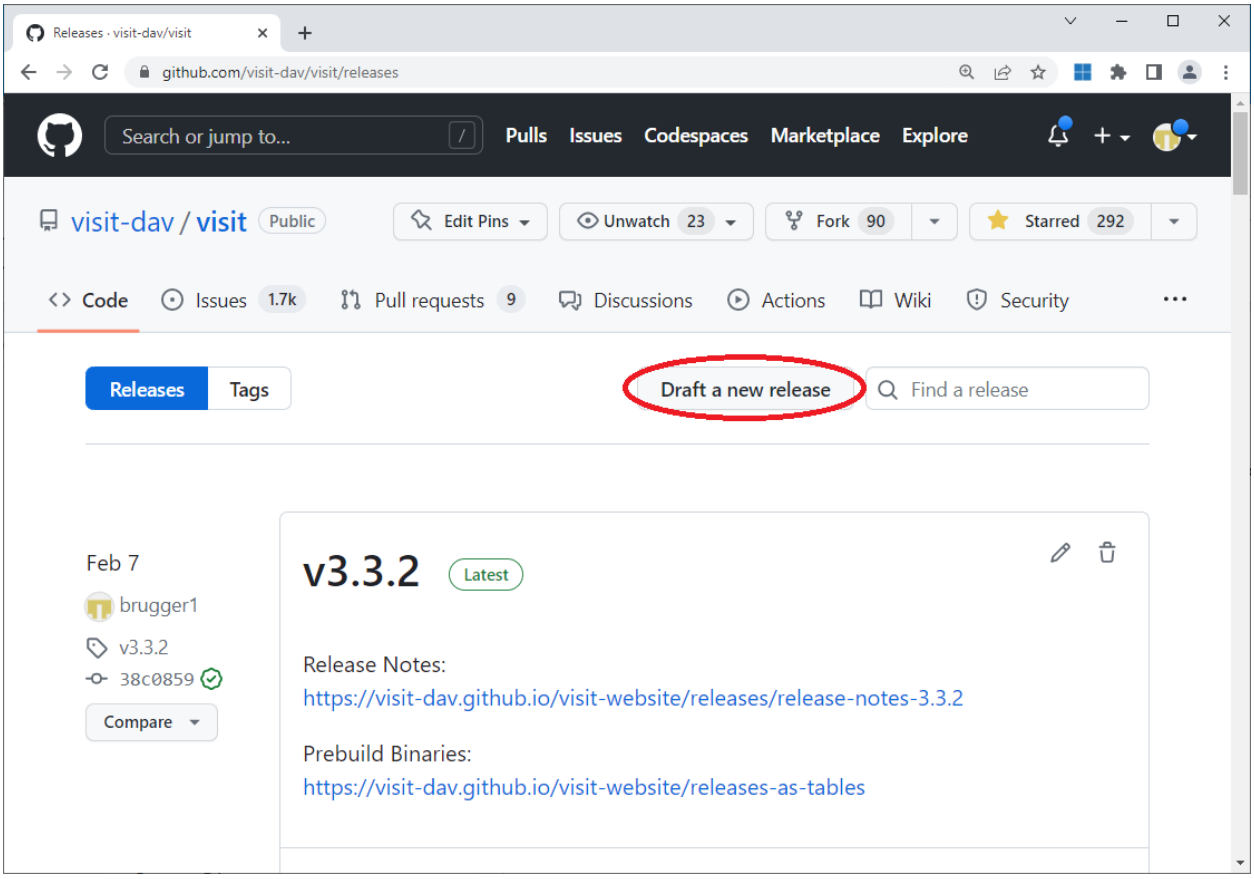


Fig. 10.7: Drafting the new release.

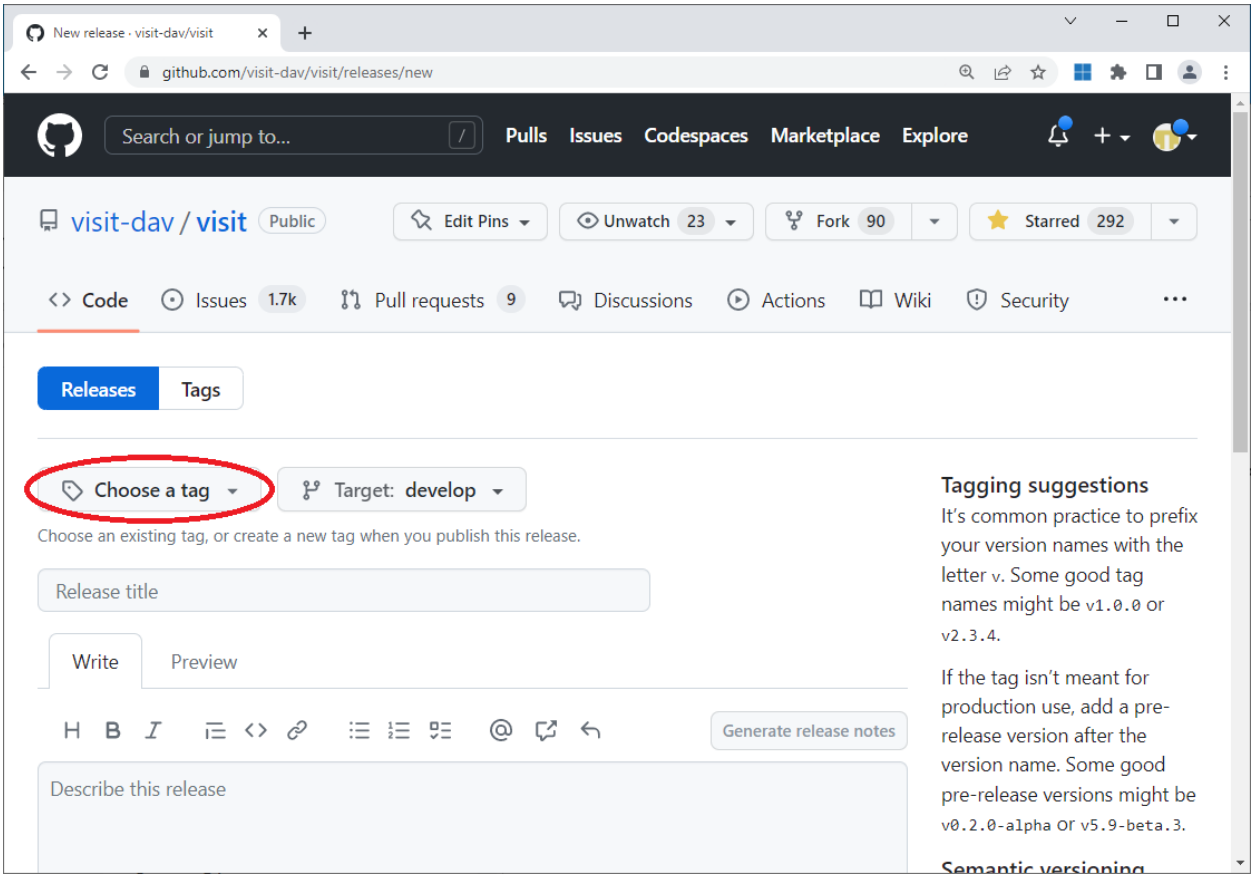


Fig. 10.8: Choosing the tag.

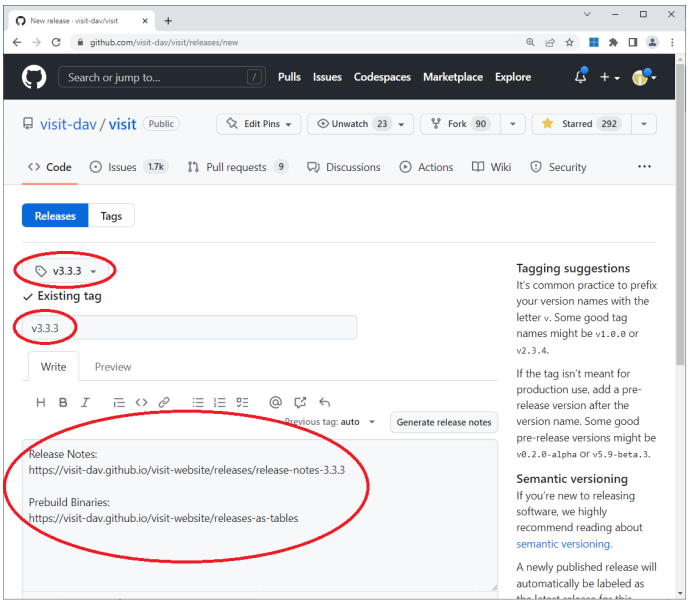


Fig. 10.9: Describing the release.

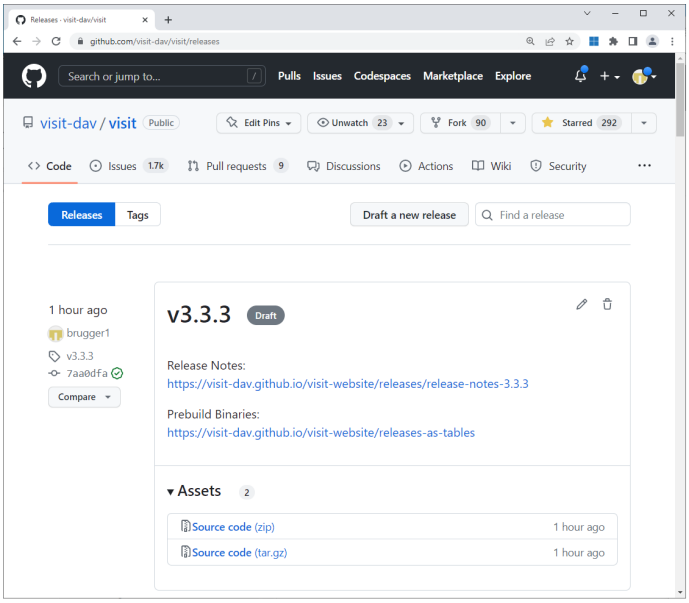


Fig. 10.10: The newly created release.

Updating the release table

The website contains a page with a series of tables, one for each minor release series with links to the assets for the releases. The tables are located in `pages/releases-as-tables.md`. If you are releasing a new minor release series you will need to add a new table. If you are releasing a new patch release you will need to add a column on the left side of the table. The most recent release is added as the first column in the table. In each case you should use an existing table as an example for adding the new release. Each release also has a series of shortcuts for each of the release assets. You can copy an existing series of shortcuts and update them for the current release. If you don't have all of the release assets added to the release you should use *Coming soon!* as a substitute for the link. If you don't do this, people will follow the broken links and report it to you. You should only commit the changes once the release has been published so that the links actually point to something.

Creating a blog entry for the new release

We create a new blog entry for each release. The blog entries are located in `_posts`. Copy one of the existing blog posts for a release that is closest to the type of release you are making, either a patch release or a minor release, as a starting point. Remove or update any version specific content from the new blog post. Patch releases list the number of bug fixes and enhancements along with a teaser of an interesting enhancement. Minor releases also contain a teaser followed by the two or three major enhancements in the release. Three major enhancements is prefereable to two and sometimes you will need to aggregate multiple enhancements into a major enhancement. Use the existing posts as examples. You should only commit the new blog post once the release has been published.

10.10.5 Updating the Spack `package.py` file

Once a new VisIt release is actually available *as a release*, the Spack package.py file for building VisIt with Spack should be reviewed for any changes needed to build this release. Generally, this work should be put in a pull request to Spack's `develop` branch. We think Spack is being released often enough that changes pushed to their `develop` will make it into a public release less than a few months later. If earlier public availability of this release of VisIt with Spack is needed, then have a look at Spack's project boards to find a suitable upcomming minor release and consider

pushing it there. Be aware, however, that if any of the changes made result in changes to how VisIt conrcretizes in Spack, it may be required to be delayed to a major release of Spack.

10.10.6 Deleting a release

If you mess up the tag or the release you can delete the tag using git commands.

```
git tag -d v3.0.1
git push origin :refs/tags/v3.0.1
```

You can then remove the release at GitHub. The release will change to a draft release because the tag no longer exists. Go ahead and click on the release to bring up the draft release.

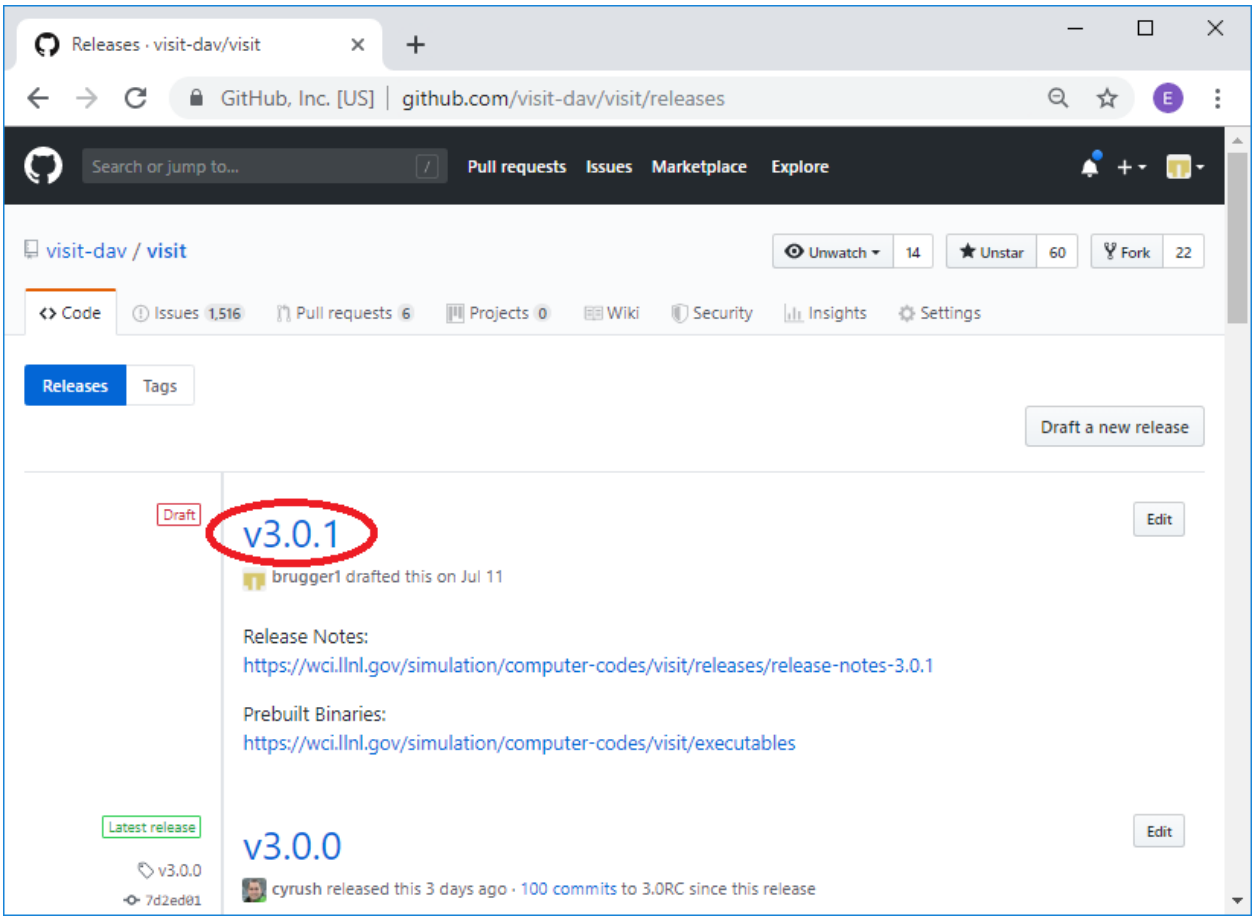


Fig. 10.11: Selecting the draft release corresponding to the deleted tag.

Click on *Delete* to delete the release.

10.11 Contributing to the Documentation

This is a short contributing guide on the VisIt project's use of Sphinx for documentation. All of VisIt's documentation is found in `<root>/src/doc` where `<root>` is the top of the git repository. There are sub-directories there for the major sub-sections of the manual including the GUI, the Python CLI, the developer's manual and more. If your

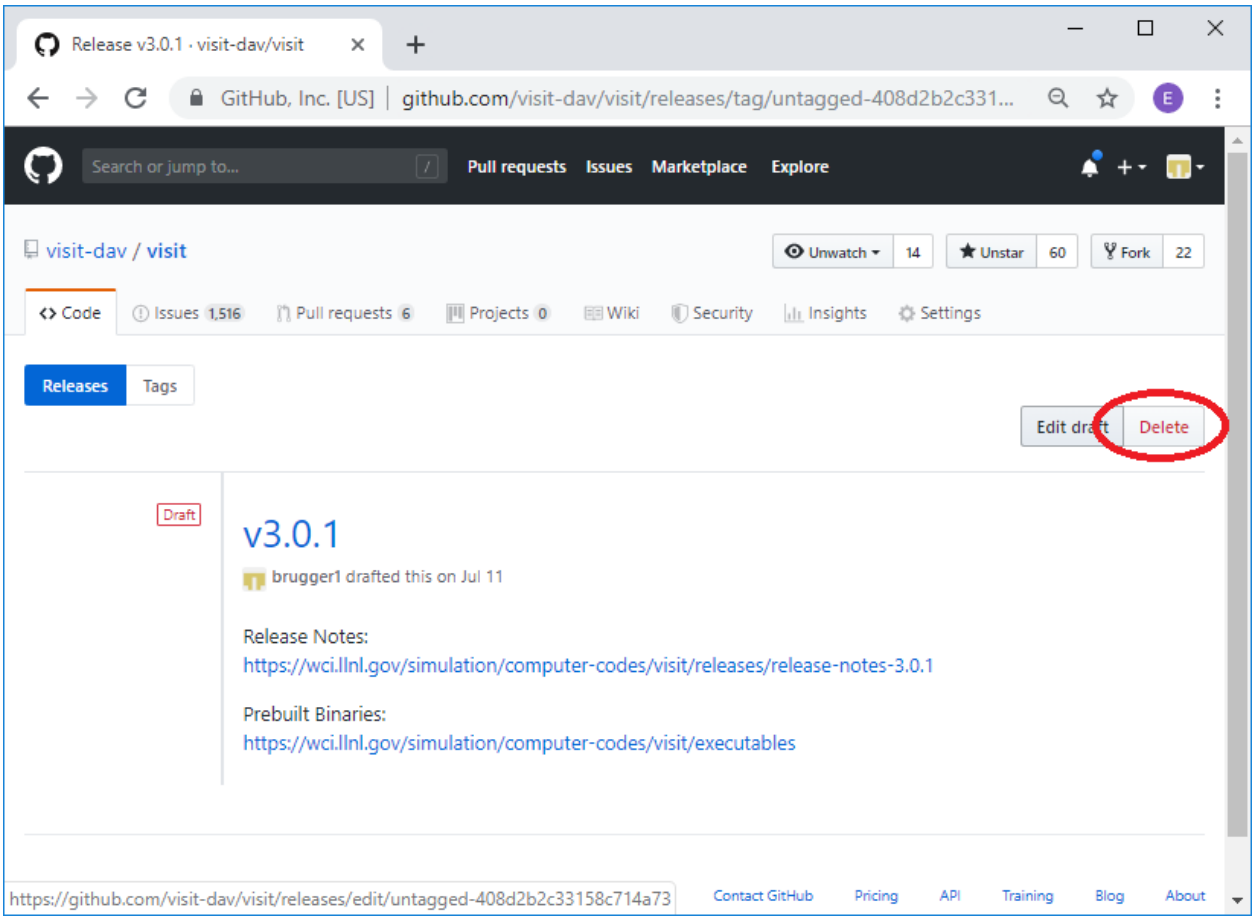


Fig. 10.12: Deleting the draft release corresponding to the deleted tag.

build is configured to build the manual, you can use the command `make manuals` to build the manual locally in the `build` directory. Otherwise, if you have Sphinx, you can try manually building locally in the `src` directory using the command:

```
cd src/doc
sphinx-build -b html . _build -a
```

The `-a` forces a re-build of everything. Remove it when you are constantly revising and rebuilding. You can then browse the root of the manual by pointing your browser to `./_build/index.html`.

Changes to any `.rst` files in `<root>/src/doc` will go live here soon after they are merged to the `develop` branch. If ReadTheDocs (RTD) resources are busy, a rebuild of the docs may take as long as 15 minutes. If want to see your branch's changes on ReadTheDocs before it is merged, the branch must be *activated* on ReadTheDocs. If you yourself do not have access to the ReadTheDocs account, you may ask another developer who does to *activate* your branch there. If the branch is *activated* on ReadTheDocs, once it is merged, it should be *deactivated*.

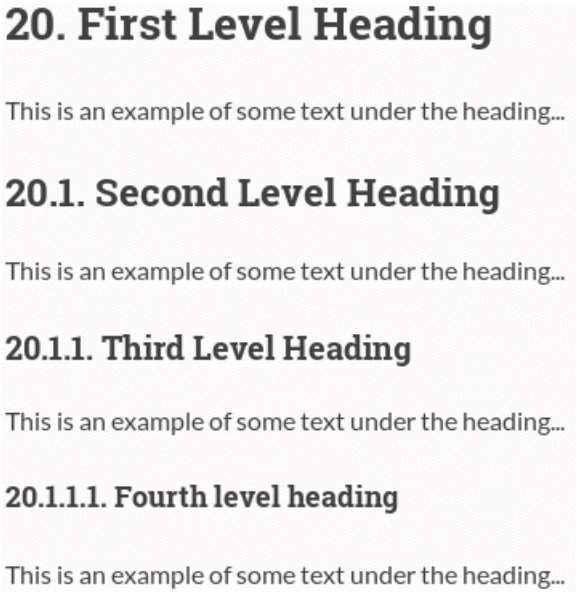
10.11.1 Quick Reference

Note that the original source of most of the content here is the OpenOffice document produced with heroic effort by Brad Whitlock. A conversion tool was used to move most of the content there to Sphinx. As such, most of the Sphinx usage conventions adopted here were driven by whatever the conversion tool produced. There are numerous opportunities for adjusting this to make better use of Sphinx as we move forward. These are discussed at the *end* of this section.

- A few documents about reStructuredText and Sphinx are useful:
  - reStructuredText Primer
  - Sphinx Documentation
  - reStructuredText Markup Specification
  - reStructuredText Reference Documentation
- Sphinx uses blank lines as block separators and 2 or 4 spaces of indentation to guide parsing and interpretation of content. So, be sure to pay careful attention to blank lines and indentation. They are not there merely for style. They *need* to be there for Sphinx to parse and interpret the content correctly.
- Line breaks *within* reStructuredText inline markup constructs often cause build errors.
- Create headings by a sequence of *separator characters* immediately underneath and the same length as the heading. Different types of separator characters define different levels of headings

```
First Level Heading
===================
This is an example of some text under the heading...

Second Level Heading
--------------------
This is an example of some text under the heading...

Third Level Heading
~~~~~~~~~~~~~~~~~~~
This is an example of some text under the heading...

Fourth level heading
""""""""""""""""""""
This is an example of some text under the heading...
```

yields these headings...



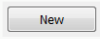
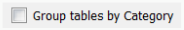
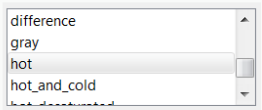
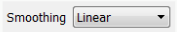
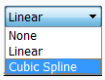
- If you want to divide sections and subsections across multiple `.rst` files, you can link them together using the `.. toctree::` directive as is done for example in the section on VisIt **Plots**

```
Plots
=====

This chapter explains the concept of a plot and goes into detail
about each of VisIt's different plot types.

.. toctree::
    :maxdepth: 1

    Working_with_Plots
    PlotTypes/index
```

  Note that the files listed in the `.. toctree::` block do not include their `.rst` extensions.

- Do not break full sentences by *wrapping* them on arbitrary column boundaries. Instead, keep each full sentence to its own *single line*, regardless of line length.
- Avoid contractions such as `isn't`, `can't` and `you've`.
- Avoid hyphenation of words.
- Use `VisIt_` or `VisIt_'s` when referring to VisIt by name.
- Use upper case for all letters in acronyms (IDE (Integrated Development Environment), GUI)
- Use case conventions of product names (MPI, VTK, QuickTime, TotalView, Valgrind).
- Bracket word(s) with one star (`*word*`) for *italics*.
- Bracket word(s) with two stars (`**some words**`) for **bold**.
- Bracket word(s) with two backticks (``` ``some words`` ```) for `literal`.
- Use `:vundl:`words to underline`` to underline words.
- Bracketed word(s) should not span line breaks.

- Use `literals` for code, commands, arguments, file names, etc.
- Use **bold** to refer to VisIt **Widget**, **Operator** or **Plot** names and other named objects part of VisIt's interface(s).
- Avoid use of **bold** for other purposes. Instead use *italics*.
- Use the following terminology when referring to widget names.

button		Usage: The button is referred to by the text on the button. E.g. To create a new color table click on the **New** button.
check box		Usage: The check box is referred to by the text to the right of the check box. E.g. To group tables by category check the **Group tables by Category** check box.
label		Usage: The label is referred to by the text in the label. E.g. Click on the color table menu next to the **Continuous** label.
list		Usage: Lists don't typically have labels associated with them, so the name is chosen to be descriptive of the list. E.g. The **Color table** list.
menu		Usage: The menu is referred to by the text to the left of the menu. E.g. To enable linear smoothing select **Linear** from the **Smoothing** menu.
option		Usage: The option is referred to by the name in the menu and the word "option" isn't included. E.g. To enable cubic smoothing select the **Cubic Spline** option from the **Smoothing** menu.

- Use `:term:`glossary term`` at least for the *first* use of a glossary term in a section.
- Use `:abbr:`ABR (Long Form)`` at least for the *first* use of an acronym or abbreviation in a section.
- Subscripting, H_2O, and superscripting, $E = mc^2$, are supported:

```
Subscripting, H\ :sub:`2`\ O, and superscripting, E = mc\ :sup:`2`, are supported
```

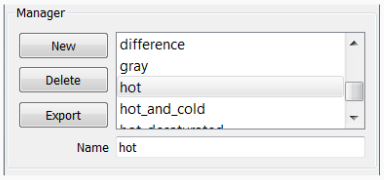
  Note the use of backslashed spaces so Sphinx treats it all as one word.

- Use `.. figure::` and not `.. image::`, include captions with figures and use `:scale: P %` to adjust image size where needed (*see more below*).
- LaTeX style equations can be included too (*see below*).
- Spell checking is supported too (*see below*) but you need to have PyEnchant and sphinx-contrib.spelling installed.
- Link checking is also supported (*see link checking*).
- Begin a line with `..` followed by space for single line comments:

```
.. this is a single line comment

..
    This is a multi-line
    comment
```

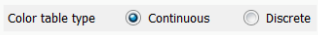
- Define anchors ahead of sections or paragraphs you want to cross reference:

panel



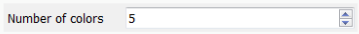
Usage: The panel is referred to by the text in the box surrounding the panel. E.g. The **Manager** panel has controls for managing color tables.

radio box



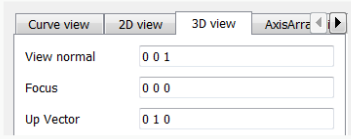
Usage: The radio box is referred to by the text to the left of the radio buttons. E.g. Select the **Continuous** radio button from the **Color table type** radio box.

spin box



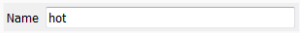
Usage: The spin box is referred to by the text to the left of the spin box. E.g. Set the number of colors by changing the value in the **Number of colors** spin box.

tab



Usage: The tab is referred to by the text at the top of the tab. E.g. The controls for setting the 3D view are located in the **3D view** tab.

text field



Usage: The text field is referred to by the text to the left of the text field. E.g. Enter the name of the new color table in the **Name** text field.

```
.. _my_anchor:

Section Heading
---------------
```

Note that the leading underscore is **not** part of the anchor name.

- Make anchor names unique over all pages of documentation by using the convention of prepending heading and subheading names.
- Link to anchors *within* this documentation like *this one*:

```
Link to anchors *within* this documentation like :ref:`this one <my_anchor>`
```

- Link to other documents elsewhere online like visitusers.org:

```
Link to other documents elsewhere online like
`visitusers.org <https://www.visitusers.org/>`_
```

- Link to *numbered* figures or tables *within* this documentation like Fig. 10.14:

```
Link to *numbered* figures or tables *within* this documentation like
:numref:`Fig. %s <my_figure2>`
```

- Link to a downloadable file *within* this documentation like `this one`:

```
Link to a downloadable file *within* this documentation like
:download:`this one <../../using_visit/Quantitative/VerdictManual-revA.pdf>`
```

- Link to different URLs with same *link text* in same `.rst` file. Sometimes you might wind up using the same phrase in a `.rst` file that is linked to different URLs. When you do, you will get a warning such as `WARNING: Duplicate explicit target name...`. For example if you have one example and another example. To correct this, you need to add an extra underscore to the end of the link as in:

```
For example, if you have one `example <http://www.llnl.gov>`__ and
another `example <http://www.llnl.gov>`__.
```

- If you are having trouble getting the formatting for a section worked out and the time involved to re-gen the documentation is too much, you could try temporarily editing a new GitHub Wiki Page with format set to reStructuredText to quickly try different things and hit the **Preview** button there to see how they work.

10.11.2 About Line Length

When we originally converted from OpenOffice to Sphinx, we decided to restrict line lengths to 80 columns. More recently, we've decided that we should not impose any absolute character count on line length and instead adopt the practice of a single sentence per line. Some lines will be very short. Other lines can be very long, especially if they include long URLs like this one. There are many advantages to using a single sentence per line mostly having to do with the way diffing tools compute and display diffs.

This practice, of course, does not apply to source code. It applies only to ascii files that are intended to represent, more or less, human readable prose. Going forward, we will not reformat existing documentation to a sentence per line en masse. However, when updating any *existing* paragraph or adding new paragraphs, we will encourage developers to follow this practice for the *whole* paragraph and request changes in PRs when it is not followed.

10.11.3 More on Images

Try to use PNG formatted images. We plan to use the Sphinx generated documentation both for online HTML and for printed PDF. So, images sizes cannot be too big or they will slow HTML loads but not so small they are unusable in PDF.

Some image formats wind up enforcing **physical** dimensions instead of just pixel dimensions. This can have the effect of causing a nicely sized image (from pixel dimensions perspective anyways), to either be unusually large or unusually small in HTML or PDF output. In these cases, you can use the Sphinx `:scale:` and `:width:` or `:height:` options for a `.. figure::` block. Also, be sure to use a `.. figure::` directive instead of an `.. image::` directive for embedding images. This is because the `.. figure::` directive also supports anchoring for cross referencing.

Although all images get copied into a common directory during generation, Sphinx takes care of remapping names so there is no need to worry about collisions in image file names potentially used in different subdirectories within the source tree.

An ordinary image...

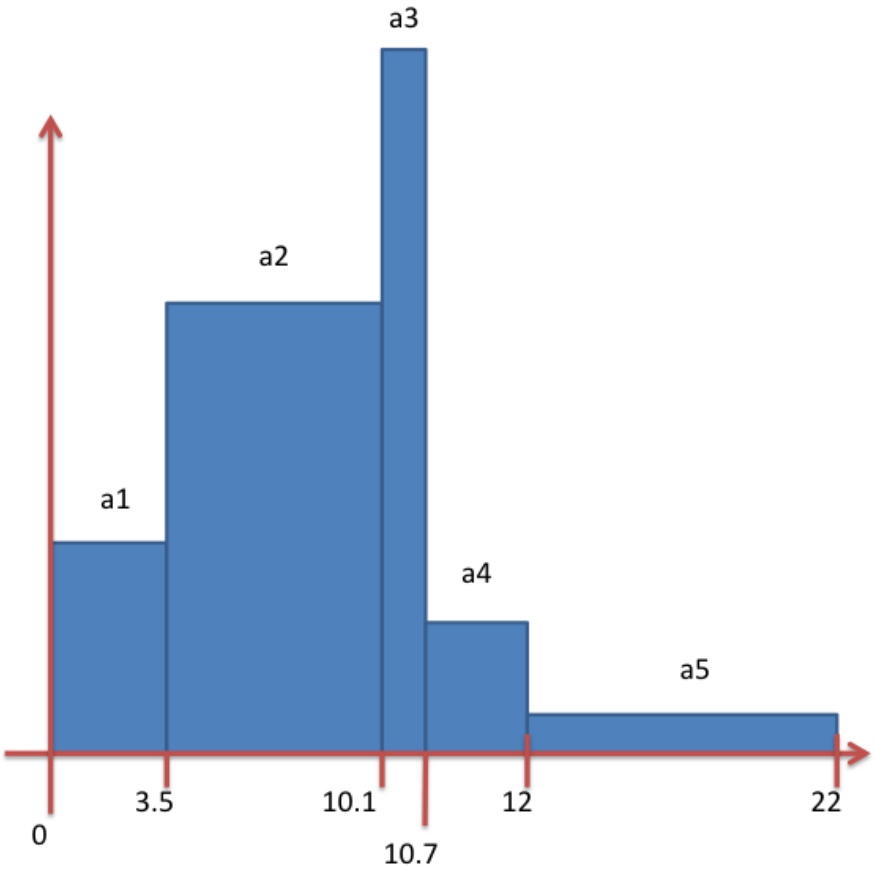
```
.. figure:: images/array_compose_with_bins.png
```

Same image with `:scale: 50%` option

```
.. figure:: images/array_compose_with_bins.png
   :scale: 50%
```

Same image with an anchor for cross referencing...

```
.. _my_figure:

.. figure:: images/array_compose_with_bins.png
   :scale: 50%

A caption
```


a3
a2
a1
a4
a5
0
3.5
10.1
10.7
12
22

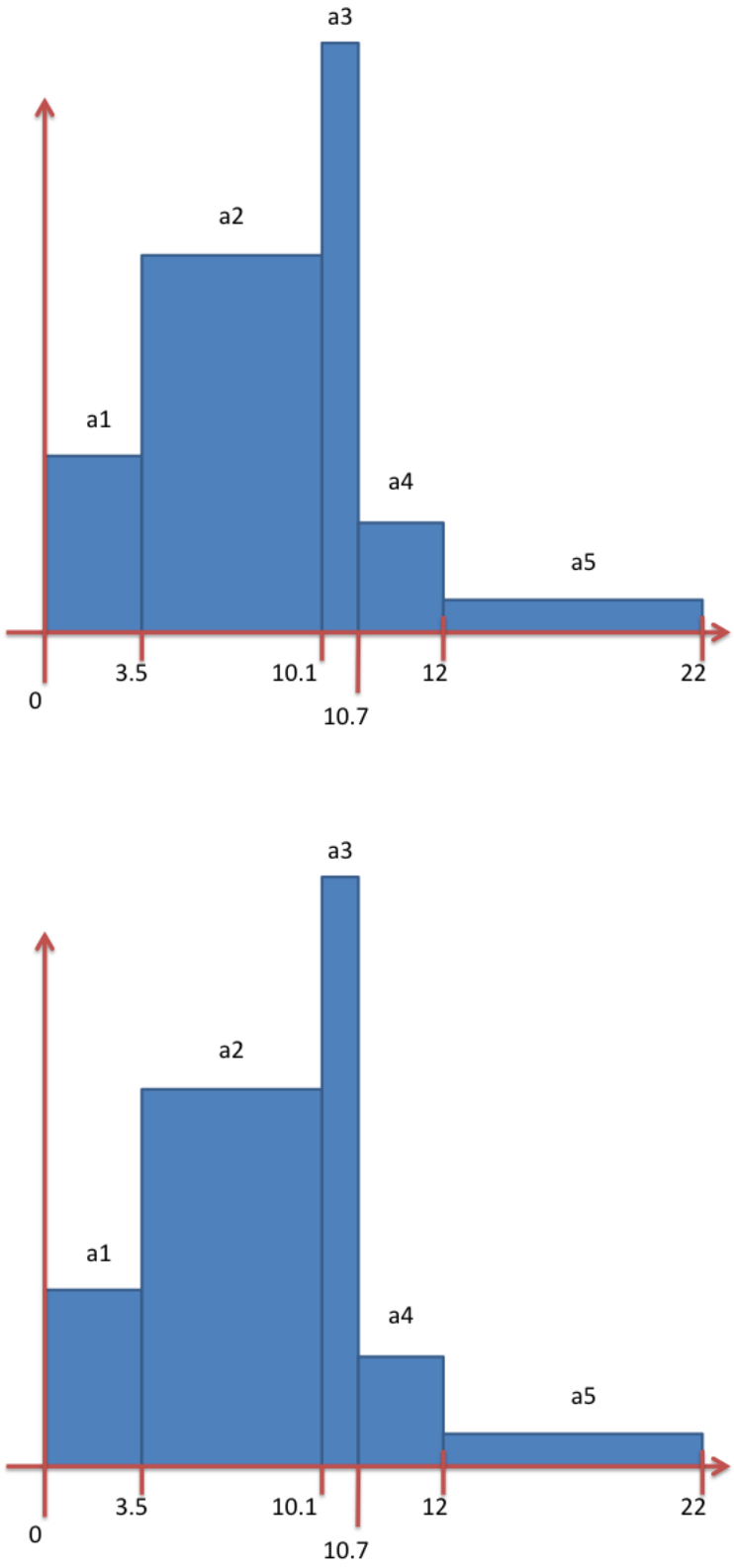


Fig. 10.13: A caption

which can now be cross referenced using an inline Fig. 10.13 like so...

```
Which can now be cross referenced using an inline :numref:`Fig. %s <my_figure>`
like so...
```

Note the anchor has a leading underscore which the reference does not include.

Same image (different anchor though because anchors need to be unique) with a caption.

```
.. _my_figure2:

.. figure:: images/array_compose_with_bins.png
   :scale: 50%

   Here is a caption for the figure.
```

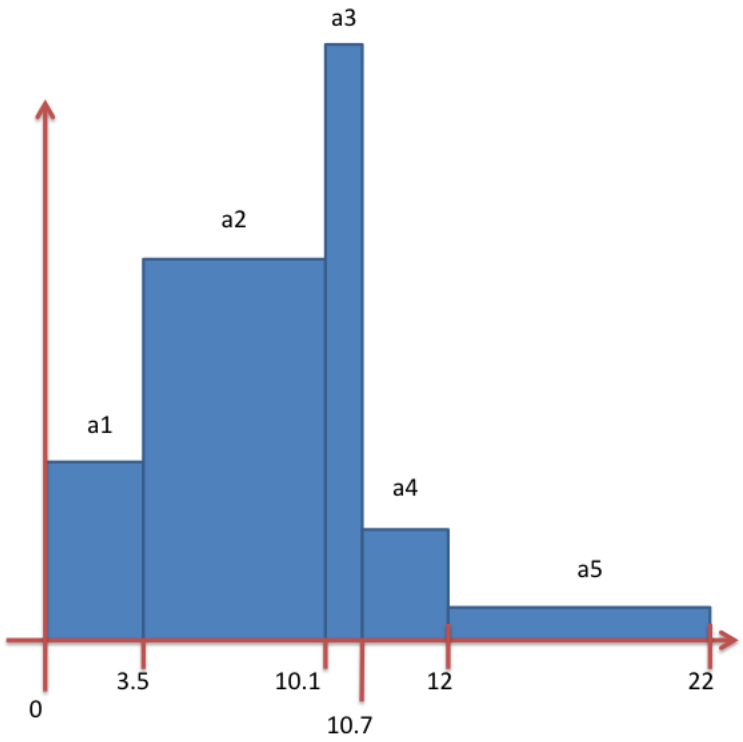


Fig. 10.14: Here is a caption for the figure.

Note that the figure label (e.g. Fig 20.2) will not appear if there is no caption.

10.11.4 Tables

Sphinx supports a variety of mechanisms for defining tables. The conversion tool used to convert this documentation from its original OpenOffice format converted all tables to the *grid* style of table which is kinda sorta like ascii art. Large tables can result in individual lines that span many widths of the editor window. It is cumbersome to deal with but rich in capabilities. Often, the best answer is to *NOT* use tables and instead use definition lists as is used in the documentation on *expressions*.

10.11.5 Collapsible content

Extra details and code samples are a couple of reasons for wanting collapsible content.

Click me to see how its done.

```
.. container:: collapsible

  .. container:: header

    Click me to see how its done.

  Put any content here.
  Just prose or

  .. code-bock:: c

    std::out << "Hello World" << std::endl;
```

10.11.6 Tabbed content

It may be useful at times to create tabbed content, such as wanting to display multiple-language code examples without taking up too much space. The sphinx-tabs extension is useful for this. Another example would be instructions for how to do something on different platforms. See the above referenced documentation for more information.

10.11.7 Math

We add the Sphinx builtin extension `sphinx.ext.mathjax` to the `extensions` variable in `conf.py`. This allows Sphinx to use mathjax to do LaTeX like math equations in our documentation. For example, this LaTeX code

```
:math:`x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}`
```

produces...

$x = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$

You can find a few examples in *Expressions*. Search there for *:math:*. Also, this LaTeX Wiki page has a lot of useful information on various math symbols available in LaTeX and this wiki book has a lot of guidance on constructing math equations with LaTeX.

10.11.8 Spell Checking Using Aspell

You can do a pretty good job of spell checking using the Unix/Linux `aspell` command.

1. Run `aspell` looking for candidate miss-spelled words.

   ```
   find . -name '*.rst' -exec cat {} \; | \
   grep -v '^ *.. image:\|figure:\|code:\|_' | \
   tr '`' '@' | sed -e 's/\(@.*@\)//' | \
   aspell -p ./aspell.en.pws list | \
   sort | uniq > maybe_bad.out
   ```

   The `find` command will find all `.rst` files. Succeeding `grep`, `tr` and `sed` pipes filter some of the `.rst` syntax away. The final pipe through `aspell` uses the personal word list (also called the personal dictionary) option, `-p ./aspell.en.pws` (**note:** the `./` is critical so don't ignore it), to specify a file containing a list

of words we allow that `aspell` would otherwise flag as incorrect. The `sort` and `uniq` pipes ensure the result doesn't contain duplicates. But, be aware that a given miss-spelling can have multiple occurrences. The whole process produces a list of candidate miss-spelled words in `maybe_bad.out`.

2. Examine `maybe_bad.out` for words that you think are correctly spelled. If you find any, remove them from `maybe_bad.out` and add them to the end of `aspell.en.pws` being careful to update the total word count in the first line of file where, for example `572` is the word count shown in that line, `personal_ws-1.1 en 572` when this was written.

3. To find instances of remaining (miss-spelled words), use the following command.

```
find . -name '*.rst' -exec grep -wnHFf maybe_bad.out {} \;
```

4. It may be necessary to iterate through these steps a few times to find and correct all the miss-spellings.

It would be nice to create a `make spellcheck` target that does much of the above automatically. However, that involves implementing the above steps as a `cmake` program and involves more effort than available when this was implemented.

10.11.9 Link checking using Sphinx linkcheck builder

You can run checks on links in all files using Sphinx *builtin* linkcheck builder by running the command:

```
sphinx-build -b linkcheck . _links -a
```

This will produce a file, `output.txt`, in the `_links` output directory. There will be a lot of output regarding various links and the results of checking those links. You want to find those cases where a link's status is reported as `broken` and then try to correct them.

For some reason, Sphinx' linkcheck builder winds up actually downloading links to *.tar.gz* and *.zip* files. This causes the linkcheck to take much more time to run than it ordinarily would. We have filed an issue ticket about this and for the time being are using the `linkcheck_ignore` *option* in `conf.py` to temporarily skip links to data files.

In addition depending on *where* you run the linkcheck (e.g. behind a firewall or other cyber-security apparatus), you may get different results due to any cyber-security IP filtering.

All of the above is automated with the `linkcheck` make target also.

10.11.10 Things To Consider Going Forward

- Decide what to do about compound words such as *timestep*, *time step* or *time-step*. There are many instances to consider such as *keyframe*, *checkbox*, *pulldown*, *submenu*, *sublauncher*, etc.
- Need to populate glossary with more VisIt specific terms such as...
- Mixed materials, Species, OnionPeel, Mesh, Viewer, cycle, timestep Client-server, CMFE, Zone-centering, Node-centering, etc.
- Decide upon and then make consistent the usage of terms like *zone/cell/element* and *node/point/vertex*
- We will need to support *versions* of the manual with each release. RTD can do that. We just need to implement it.
  - If we have tagged content, then those would also represent different *versions* of the manual.
- All VisIt manuals should probably be hosted at a URL like `visit.readthedocs.io` and from there users can find manuals for GUI, CLI Getting Data Into VisIt, etc.
- Additional features of Sphinx to consider adopting...

  - `:guilable:` role for referring to GUI widgets.
  - `:command:` role for OS level commands.
  - `:file:` role for referring to file names.
  - `:menuselection:` role for referring to widget paths in GUI menus. Example: *Controls* → *View* → *Advanced*.
  - `:kbd:` role for specifying a sequence of key strokes.
  - `.. deprecated::` directive for deprecated functionality
  - `.. versionadded::` directive for new functionality
  - `.. versionchanged::` directive for when functionality changed
  - `.. note::`, `.. warning::` and/or `.. danger::` directives to call attention to the reader.
  - `.. only::` directives for audience specific (e.g. tagged) content
    * Could use to also include developer related content but have it not appear in the user manual output
  - `.. seealso::` directive for references
  - named hyper link references for names of products and projects we refer to frequently such as VTK or VisIt. In this document, we explicitly define the following named hyper link references:

    ```
    .. _VTK: https://www.vtk.org
    ```

    In addition, we use the `conf.py` variable, `rst_epilog` to define:

    ```
    .. _VisIt: https://visit.llnl.gov
    ```

    So that this named hyperlink reference definition is available in all `*.rst` files. Finally, be aware that reStructuredText supports a wide variety of approaches for hyper-links.

- Substitutions for frequently used text such as **Viewer Window**:

```
Substitutions for frequently used text such as |viswin|.
```

  with the following substitution defined:

```
.. |viswin| replace:: **Viewer Window**
```

- Possible method for embedding python code to generate and capture images (both of the GUI and visualization images produced by VisIt) automatically
  - With the following pieces....
    * VisIt python CLI
    * pyscreenshot
    * A minor adjustment to VisIt GUI to allow a python CLI instance which used `OpenGUI(args...)` to inform the GUI that widgets are to be raised/mapped on state changes.
  - We can include python code directly in these `.rst` documents (prefaced by `.. only::` directives to ensure the code does not actually appear in the generated manual) that does the work and just slurp this code out of these documents to actually run for automatic image generation.
    * Generate and save VisIt visualization images.
    * Use diffs on screen captured images to grab and even annotate images of GUI widgets.

```
import pyscreenshot
import PIL

# The arg (not yet implemented) sets flag in GUI to map windows
# on state changes
OpenGUI(MapWidgetsOnStateChanges=True)
base_gui_image = pyscreenshot.grab()

OpenDatabase('visit_data_path()/silo_hdf5_test_data/globe.silo')
AddPlot("Pseudocolor","dx")
DrawPlots()

# Save VisIt rendered image for manual
SaveWindow('Plots/PlotTypes/Pseudocolor/images/figure15.png')
ClearPlots()

# Change something in PC atts to force it to map
pcatts = PseudocolorAttributes()
pcatts.colorTableName = 'Blue'
SetPlotOptions(pcatts) # Causes widget to map due to state change
pcatts.colorTableName = 'hot'
SetPlotOptions(pcatts) # Causes widget to map due to state change
gui_image = pyscreenshot.grab()

# Save image of VisIt PC Attr window
#   - computes diff between gui_image and base_gui_image, bounding box
#   - around it and then saves that bounding box from gui_image
diff_bbox = BBoxedDiffImage(gui_image, gui_image_base)
SaveBBoxedImage(gui_image, diff_bbox, 'Plots/PlotTypes/Pseudocolor/images/pcatts_
↪window.png')

# Make a change to another PC att, capture and save it
pcatts.limitsMode = pcatts.ActualData
SetPlotOptions(pcatts) # Causes widget to map due to state change
gui_image = pyscreenshot.grab()
SaveBBoxedImage(gui_image, diff_bbox, 'Plots/PlotTypes/Pseudocolor/images/pcatts_
↪limit_mode_window.png')
```

10.12 Updating the Python Doc Strings

Note: We are still refining this procedure!

The Python doc strings for most functions in VisIt's scripting interface are generated from the examples embedded in the `python_scripting/functions.rst` file. This allows us to have a single source for both our scripting interface sphinx docs and the doc strings embedded in VisIt's compiled Python module. The `functions_to_method_doc.py` helper script generates `MethodDoc.C` from the examples embedded in the rst source.

The Python doc strings for Attribute objects and Events are extracted from the scripting interface for use in the Python scripting sphinx docs. The `sphinx_cli_extractor.py` runs VisIt to generate `python_scripting/attributes.rst` and `python_scripting/events.rst`.

10.12.1 Steps to update the Python scripting manual

1. Modify `python_scripting/functions.rst`.
2. Run `functions_to_plain_py.py` to generate `PY_RST_FUNCTIONS_TO_PYTHON.py`.
3. Run `2to3 -p PY_RST_FUNCTIONS_TO_PYTHON.py` to check for Python syntax errors and Python 3 compatibly.
   - **NOTE:** `PY_RST_FUNCTIONS_TO_PYTHON.py` is just a temporary file to test steps 2 and 3 here. It could be named anything and is not part of the repository.
   - **NOTE:** `2to3` will run to completion and issue a number of messages. A zero return code indicates all is well.
4. Run `functions_to_method_doc.py` to regenerate `MethodDoc.C`.
5. Build and run the VisIt scripting interface and assure yourself `help(<your-new-func-doc>)` produces the desired output.
6. Run the `sphinx_cli_extractor.py` tool producing new `attributes.rst` and `events.rst` files. To do so, you may need to use a combination of the `PATH` and `PYTHONPATH` environment variables to tell the `sphinx_cli_extractor.py` script where to find the VisIt module, `visit` in VisIt's `site-packages` and where to find the Python installation that that module is expecting to run with. In addition, you may need to use the `PYTHONHOME` environment variable to tell VisIt's `visit` module where to find standard Python libraries. For example, to use an installed version of VisIt on my macOS machine, the command would look like...

   ```
   env PATH=/Applications/VisIt.app/Contents/Resources/2.13.3/darwin-x86_64/bin:/
   ↪Applications/VisIt.app/Contents/Resources/bin:$PATH \
   PYTHONHOME=/Applications/VisIt.app//Contents/Resources/2.13.3/darwin-x86_64/lib/
   ↪python \
   PYTHONPATH=/Applications/VisIt.app/Contents/Resources/2.13.3/darwin-x86_64/lib/
   ↪site-packages \
   ./sphinx_cli_extractor.py
   ```

   Note that the above command would produce CLI documentation for version 2.13.3 of VisIt. Or, to use a current build of VisIt on which you are working on documentation related to changes you have made to VisIt, the command would look something like...

   ```
   env PATH=/Users/miller86/visit/third_party/3.2.0/python/3.7.7/i386-apple-darwin18_
   ↪clang/bin:/Users/miller86/visit/visit/build/bin:$PATH \
   PYTHONPATH=../../build/lib/site-packages python3 ./sphinx_cli_extractor.py
   ```

   The whole process only takes a few seconds.
7. Assuming you successfully ran the above command, producing new `attributes.rst` and `events.rst` files, then do a local build of the documentation here and confirm there are no errors in the build.

   ```
   sphinx-build -b html . _build -a
   ```
8. Then open the file, `_build/index.html`, in your favorite browser to view.
9. Add all the changed files to a commit and push to GitHub.
10. The GitHub integration with ReadTheDocs should result in your documentation updates going live a short while (<15 mins) after it has been merged to develop.

10.13 Finding Memory Leaks

10.13.1 Overview

We support several mechanisms to find memory leaks. The two best mechanisms are using Valgrind and vtkDebugLeaks. Valgrind is used to detect memory leaks in a specific component. You run VisIt with the appropriate options and then when VisIt is finished running you will get logs with a report of memory leaks and usage. The log files are quite large. The top of the log file contains stack traces of where the memory was allocated for each chunk of memory allocated. It is sorted from the smallest leaks to the largest leaks. At the bottom of the log is a summary of memory leaks and usage. vtkDebugLeaks is specifically used to find VTK memory leaks. It provides a list of all the VTK objects that are still in use when VisIt terminates to the terminal. There will be one list for the viewer and one for the engine (multiple lists if running in parallel).

10.13.2 Building VisIt for Valgrind and vtkDebugLeaks

The following steps were from building and running VisIt 3.1 on Quartz, a Linux cluster.

Building the Third party Libraries

In order to use vtkDebugLeaks you will need to enable it when you build VTK. Edit the build_visit script and find the line:

```
vtk_debug_leaks="false"
```

and change it to:

```
vtk_debug_leaks="true"
```

In order for the stack traces from Valgrid to be the most useful, you should build the third party libraries with debug support. In our case we are going to do a minimal build with just the Silo and HDF5 I/O libraries.

```
./build_visit3_1_0 --required --mesagl --llvm --silo --hdf5 --debug --no-visit \
--thirdparty-path /usr/workspace/wsa/visit/visit/thirdparty_shared/3.1.0/toss3_debug \
--makeflags -j16
```

Building VisIt

Just like the third party libraries, VisIt needs to be built with debug support in order for Valgrind to produce useful stack traces. Furthermore, VisIt contains conditional code that does additional cleanup at exit to eliminate spurious memory leaks. The additional cleanup is enabled with DEBUG_MEMORY_LEAKS. The following steps were used to build VisIt as described.

```
cd visit3.1.0
mkdir build
cd build
/usr/workspace/wsa/visit/visit/thirdparty_shared/3.1.0/toss3_debug/cmake/3.9.3/linux-
↪x86_64_gcc-4.9/bin/cmake -DCMAKE_BUILD_TYPE=Debug -DVISIT_CONFIG_SITE=/usr/
↪workspace/wsa/brugger/visit_memory/quartz2498.cmake -DCMAKE_CXX_FLAGS:STRING="-
↪DDEBUG_MEMORY_LEAKS" ../src
make -j 36
```

You are now ready to start looking for memory leaks.

10.13.3 Running VisIt with Valgrind

Follow these steps to run Valgrind on the viewer to find memory leaks from a basic use case.

1. Run VisIt with Valgrind on the viewer in nowin mode.

```
cd ..
mkdir run1
cd run1
../visit3.1.0/build/bin/visit -valgrind viewer -nowin
```

2. Open `wave.visit`.
3. Create a Pseudocolor plot of `pressure`.
4. Save the window.
5. Delete the plot.
6. Close the database.
7. Exit

There are several things to note. We ran in nowin mode to eliminate leaks from Qt, which are difficult to address. Since we ran in nowin mode we had to save an image in order to do some rendering. After we saved the image we deleted the plot and closed the database to clean up as much memory as possible.

After VisIt exits, you will get vtkDebugLeaks output sent to the terminal as well as log files created by Valgrind. Let's look at the Valgrind output first.

Looking at the Valgrind Output

Valgrind creates several log files with the output. You are interested in the largest one. Here are the files generated from the run described above.

```
ls -l
total 48792
-rw------- 1 brugger brugger 49715025 Dec  4 11:44 vg_viewer_8870.log
-rw------- 1 brugger brugger      449 Dec  4 11:43 vg_viewer_8915.log
-rw------- 1 brugger brugger      449 Dec  4 11:44 vg_viewer_9281.log
-rw------- 1 brugger brugger    32373 Dec  4 11:44 visit0000.png
```

Here is the output from the end of the largest log file.

```
tail -11 vg_viewer_8870.log
==8870== LEAK SUMMARY:
==8870==    definitely lost: 0 bytes in 0 blocks
==8870==    indirectly lost: 0 bytes in 0 blocks
==8870==      possibly lost: 33,097 bytes in 245 blocks
==8870==    still reachable: 30,386,572 bytes in 24,057 blocks
==8870==                       of which reachable via heuristic:
==8870==                         stdstring          : 25,296 bytes in 649 blocks
==8870==         suppressed: 0 bytes in 0 blocks
==8870==
==8870== For counts of detected and suppressed errors, rerun with: -v
==8870== ERROR SUMMARY: 202 errors from 202 contexts (suppressed: 0 from 0)
```

This is actually pretty good. There is still more work to be done to address the possibly lost memory and then there may be issues with the still reachable. This is probably primarily from a lack of cleanup before exiting.

Looking at the vtkDebugLeaks Output

You will get vtkDebugLeaks output from both the viewer and engine since both have VTK code and both were linked against VTK built with vtkDebugLeaks. Here is the engine output, which came out first.

```
vtkDebugLeaks has detected LEAKS!
Class "9vtkBufferIxE" has 1 instance still around.
Class "vtkDataSetAttributes" has 2 instances still around.
Class "vtkGraphInternals" has 1 instance still around.
Class "vtkOutputWindow" has 1 instance still around.
Class "vtkInformation" has 1 instance still around.
Class "vtkInformationIntegerValue" has 4 instances still around.
Class "vtkGraphEdge" has 1 instance still around.
Class "vtkIdTypeArray" has 1 instance still around.
Class "vtkTypeUInt32Array" has 1 instance still around.
Class "vtkFieldData" has 1 instance still around.
Class "vtkMergeTree" has 1 instance still around.
Class "vtkCommand or subclass" has 1 instance still around.
Class "9vtkBufferIjE" has 1 instance still around.
```

As you can see, it had relatively few leaks associated with VTK. These may not even be leaks, they are probably from a lack of cleaning up before exiting. More work needs to be done here.

Here is the viewer output.

```
vtkDebugLeaks has detected LEAKS!
Class "vtkOpenGLRenderTimerLog" has 1 instance still around.
Class "vtkOpenGLTextActor" has 33 instances still around.
Class "vtkOpenGLTextMapper" has 1 instance still around.
Class "vtkTextureObject" has 5 instances still around.
Class "9vtkBufferIxE" has 146 instances still around.
Class "vtkTexturedActor2D" has 1 instance still around.
Class "vtkBackgroundActor" has 1 instance still around.
Class "vtkCellData" has 195 instances still around.
Class "vtkVisItTextActor" has 1 instance still around.
Class "vtkInformationIntegerVectorValue" has 17 instances still around.
Class "vtkInformationVector" has 1454 instances still around.
Class "vtkVisItCubeAxesActor" has 1 instance still around.
Class "vtkPerspectiveTransform" has 6 instances still around.
Class "vtkPointData" has 195 instances still around.
Class "vtkProperty2D" has 16 instances still around.
Class "vtkCompositeDataPipeline" has 290 instances still around.
Class "vtkMatrix3x3" has 236 instances still around.
Class "vtkTrivialProducer" has 100 instances still around.
Class "vtkAxesActor2D" has 1 instance still around.
Class "vtkOpenGLIndexBufferObject" has 1006 instances still around.
Class "vtkMatrix4x4" has 1331 instances still around.
Class "vtkPickingManager" has 1 instance still around.
Class "QVTKInteractor" has 1 instance still around.
Class "vtkCoordinate" has 160 instances still around.
Class "vtkSimpleTransform" has 35 instances still around.
Class "vtkFollower" has 88 instances still around.
Class "vtkOutputWindow" has 1 instance still around.
Class "vtkPoints" has 192 instances still around.
Class "vtkInformation" has 3503 instances still around.
Class "vtkActorCollection" has 3 instances still around.
Class "vtkLine" has 5 instances still around.
Class "vtkGenericOpenGLRenderWindow" has 1 instance still around.
```

(continues on next page)

(continued from previous page)

```
Class "vtkVolumeCollection" has 3 instances still around.
Class "vtkPropCollection" has 5 instances still around.
Class "vtkInformationIntegerPointerValue" has 36 instances still around.
Class "vtkTriad2D" has 1 instance still around.
Class "vtkPolyData" has 159 instances still around.
Class "vtkLookupTable" has 34 instances still around.
Class "vtkPixel" has 5 instances still around.
Class "vtkAppendPolyData" has 1 instance still around.
Class "vtkOpenGLImageMapper" has 1 instance still around.
Class "vtkPropPicker" has 1 instance still around.
Class "vtkActor2D" has 11 instances still around.
Class "vtkOpenGLCamera" has 3 instances still around.
Class "vtkOpenGLVertexArrayObject" has 1006 instances still around.
Class "vtkOpenGLActor" has 115 instances still around.
Class "vtkOpenGLPolyDataMapper" has 118 instances still around.
Class "vtkIdList" has 10 instances still around.
Class "vtkWorldPointPicker" has 1 instance still around.
Class "vtkDoubleArray" has 10 instances still around.
Class "vtkMatrixToLinearTransform" has 26 instances still around.
Class "vtkAlgorithmOutput" has 194 instances still around.
Class "vtkCullerCollection" has 3 instances still around.
Class "vtkOpenGLRenderer" has 3 instances still around.
Class "vtkPolyDataAlgorithm" has 89 instances still around.
Class "vtkDepthSortPolyData2" has 1 instance still around.
Class "vtkInformationIntegerValue" has 3756 instances still around.
Class "vtkOpenGLLight" has 10 instances still around.
Class "vtkOpenGLPolyDataMapper2D" has 45 instances still around.
Class "vtkTextProperty" has 93 instances still around.
Class "vtkCellArray" has 146 instances still around.
Class "vtkRendererCollection" has 1 instance still around.
Class "vtkShaderProgram" has 6 instances still around.
Class "vtkVisItAxisActor2D" has 9 instances still around.
Class "vtkOpenGLShaderCache" has 1 instance still around.
Class "vtkTDxInteractorStyleCamera" has 3 instances still around.
Class "vtkImageData" has 36 instances still around.
Class "vtkFloatArray" has 222 instances still around.
Class "vtkInformationStringValue" has 108 instances still around.
Class "vtkInformationExecutivePortVectorValue" has 194 instances still around.
Class "vtkOpenGLVertexBufferObject" has 9 instances still around.
Class "vtkIdTypeArray" has 146 instances still around.
Class "vtkTransform" has 541 instances still around.
Class "vtkOutlineSource" has 5 instances still around.
Class "vtkOpenGLVertexBufferObjectGroup" has 163 instances still around.
Class "vtkFieldData" has 195 instances still around.
Class "vtkVisItAxisActor" has 12 instances still around.
Class "vtkOpenGLProperty" has 40 instances still around.
Class "vtkOpenGLTexture" has 36 instances still around.
Class "vtkLineSource" has 1 instance still around.
Class "vtkInformationDoubleVectorValue" has 90 instances still around.
Class "vtkLightCollection" has 3 instances still around.
Class "vtkUnsignedCharArray" has 41 instances still around.
Class "vtkShader" has 18 instances still around.
Class "vtkTDxInteractorStyleSettings" has 3 instances still around.
Class "vtkStreamingDemandDrivenPipeline" has 100 instances still around.
Class "vtkTextureUnitManager" has 1 instance still around.
Class "vtkOpenGLVertexBufferObjectCache" has 1 instance still around.
Class "vtkActor2DCollection" has 3 instances still around.
```

(continues on next page)

(continued from previous page)

```
Class "vtkTimerLog" has 166 instances still around.
Class "9vtkBufferIfE" has 222 instances still around.
Class "9vtkBufferIdE" has 10 instances still around.
Class "vtkCommand or subclass" has 208 instances still around.
Class "9vtkBufferIhE" has 41 instances still around.
Class "vtkInformationExecutivePortValue" has 237 instances still around.
Class "vtkFXAAOptions" has 3 instances still around.
```

As you can see, the viewer has considerably more leaks associated with VTK. Again, these may not be leaks but merely a lack of cleanup before exiting. More work needs to be done here as well.

That's it. Happy hunting!

10.14 Using Docker

10.14.1 Overview

Docker is a platform for building containers. Containers can run either Windows or Linux operating systems. Docker is available on the Mac, Windows and Linux. The rest of this tutorial will primarily be focused on running Docker on Windows. The content on installing and setting up Docker is Windows specific but the remainder of the content on creating and using containers is operating system independent.

10.14.2 Installing Docker on Windows

Install Docker on your system. It is free to download and install. You will need to be running Windows Professional and you will need administrator priviledges. The following link will get you started.

https://docs.docker.com/docker-for-windows/

You will need to enable experimental features to be able to use the `--squash` option when building your container. You can enable experimental features with the Settings window. Go to the *Daemon* tab and check the *Experimental features* checkbox and press *Apply*. Note that this will restart the Docker daemon, which will kill container builds or running containers.

If you run into problems running out of disk space, you can increase the amount of disk space allocated to Docker with the Settings window. Go to the *Advanced* tab and move the *Disk image max size* to the right to increase the amount of disk space and press *Apply*. Note that this will restart the Docker daemon, which will kill container builds or running containers.

10.14.3 Creating a Docker Container

First you will want to bring up a Command window and use that to run Docker commands. Next we'll create a folder to hold all our Docker files. We are assuming that you are at the root of the C: drive.

```
C:\>cd \Users\brugger1
C:\Users\brugger1>mkdir docker
C:cd docker
```

Now you need to copy all the relevant files to your docker folder. You must have the following files in your folder.

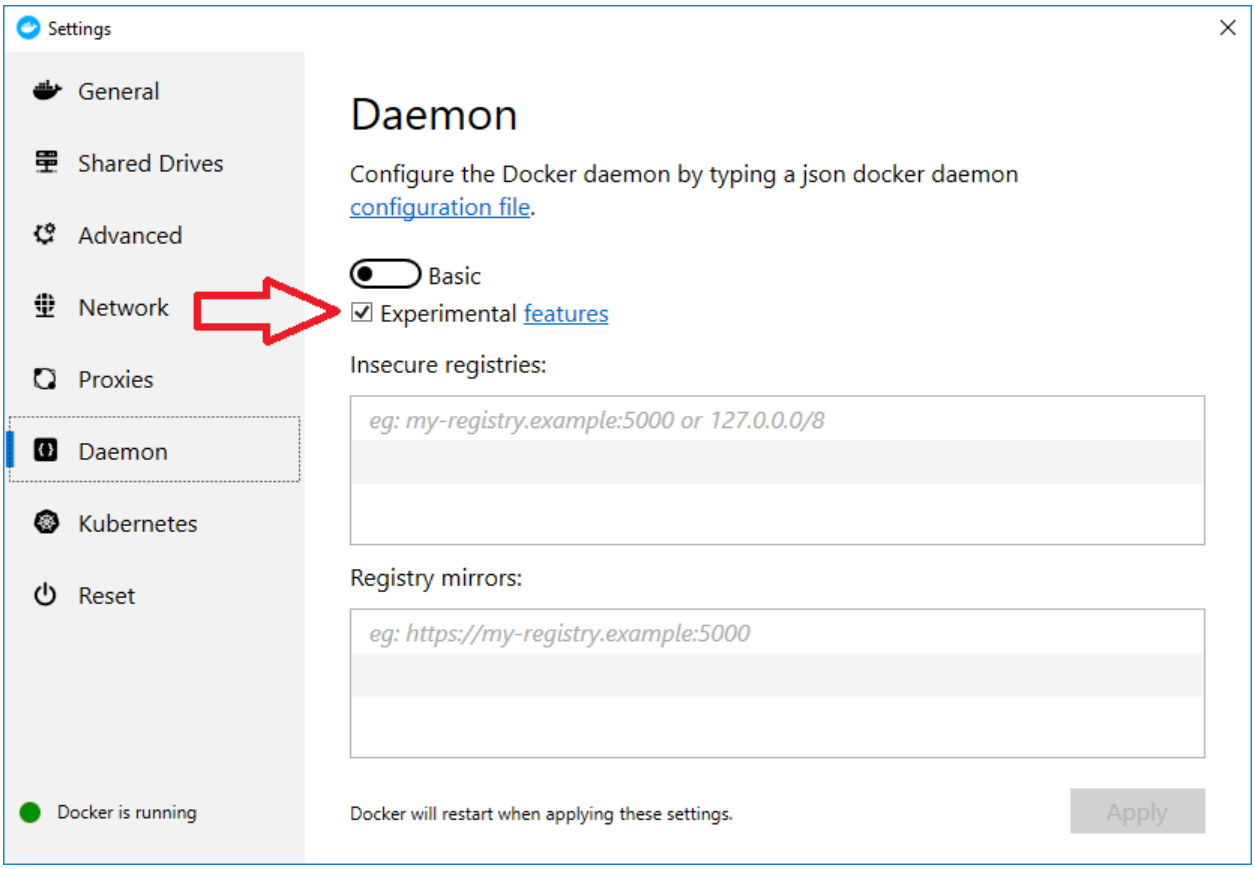


Fig. 10.15: Enabling experimental features.

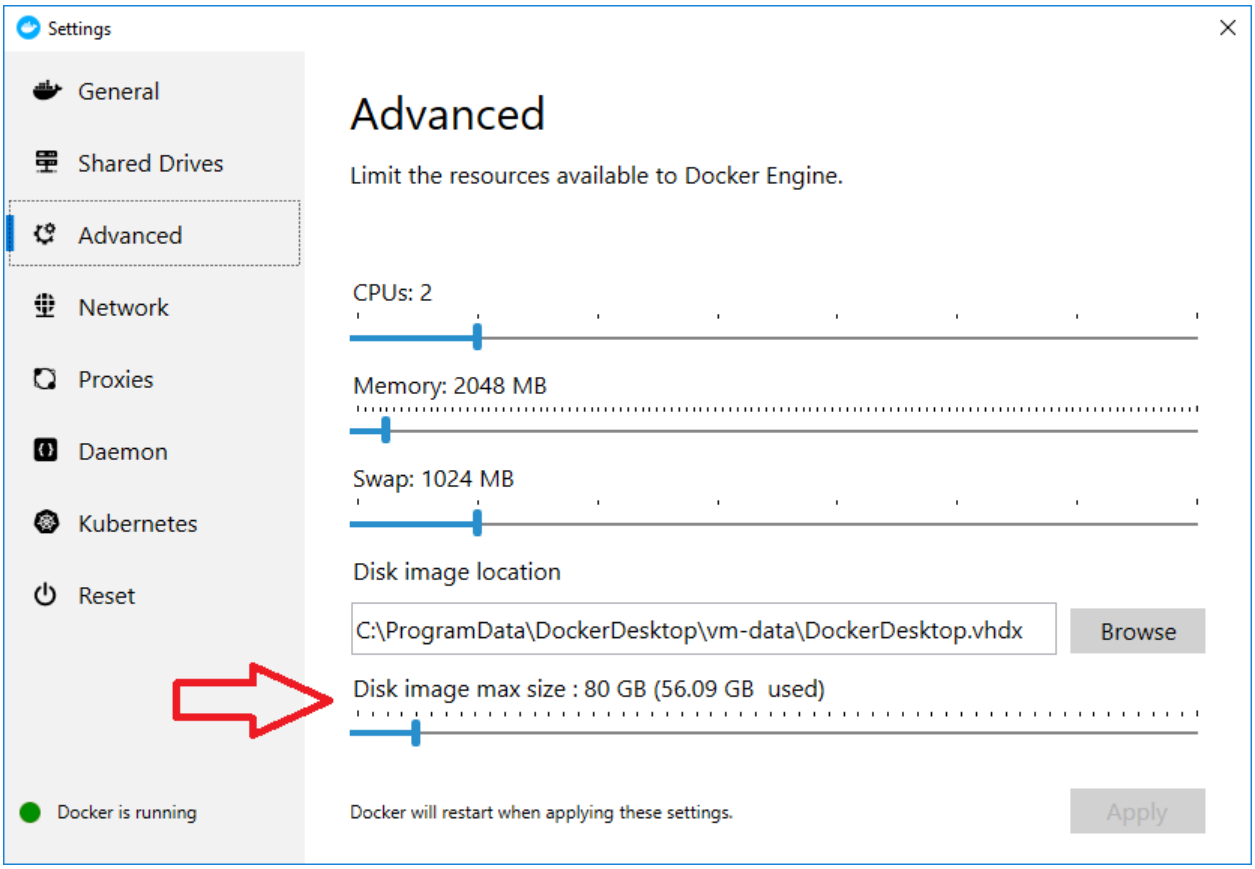


Fig. 10.16: Increasing the disk space allocated to Docker.

```
C:\Users\brugger1\docker>dir
 Volume in drive C is Windows
 Volume Serial Number is A8F6-9F9C

 Directory of C:\Users\brugger1\docker

09/13/2019  02:46 PM    <DIR>          .
09/13/2019  02:46 PM    <DIR>          ..
09/13/2019  02:39 PM           737,636 build_visit3_0_2
08/28/2019  01:45 PM             1,173 build_visit_docker_cleanup.py
09/12/2019  12:24 PM             1,322 Dockerfile-debian9
09/13/2019  07:27 AM             1,176 Dockerfile-fedora27
09/12/2019  03:49 PM             1,337 Dockerfile-ubuntu16
09/12/2019  12:36 PM             1,321 Dockerfile-ubuntu18
09/12/2019  12:23 PM               216 run_build_visit.sh
09/13/2019  02:39 PM       121,776,180 visit3.0.2.tar.gz
               8 File(s)    122,520,361 bytes
               2 Dir(s)  814,047,002,624 bytes free
```

These files can be found in the VisIt repository at GitHub in the following location.

https://github.com/visit-dav/visit/tree/develop/scripts/docker

The Dockerfile determines the type of operating system you will build your container with. The first line in the Dockerfile contains information about the operating system. Here is a link to a reference on Dockerfile.

https://docs.docker.com/engine/reference/builder/

The Dockerfile will need to be specific to the operating system since the way you install packages and do other administrative tasks will vary among different Linux operating systems, although there are only a few unique varients that the rest are built on. You can go to the Docker Hub to find Linux distributions to start with.

https://hub.docker.com/_/centos

https://hub.docker.com/_/debian

https://hub.docker.com/_/fedora

https://hub.docker.com/_/ubuntu

In this example the Dockerfile is set up to use Ubuntu 16. The Dockerfile installs all the packages needed to build VisIt and then uses build_visit to create all the third party libraries as well as the config site file. The build will take several hours. Sometimes I have had it stop sending text to the Command window, so if it looks like it is hung, it may actually be happily progressing along.

```
C:\Users\brugger1\docker>docker build -f Dockerfile-ubuntu16 -t visitdev:3.0.2-
↪ubuntu16 . --squash
```

Start up the container and run it interactively.

```
C:\Users\brugger\docker>docker run -t -i visitdev:3.0.2-ubuntu16 /bin/bash
visit@bea87fee3276:~$
```

Now the container is ready for you to build VisIt. First, you need to copy the tar file with the source code. To do this, you will need to go to another Command window and use the container id shown in the prompt.

```
C:\Users\brugger\docker>docker cp visit3.0.2.tar.gz bea87fee3276:/home/visit
```

Now go back to the first Command window and create your distribution.

```
visit@bea87fee3276:~$ tar zxf visit3.0.2.tar.gz
visit@bea87fee3276:~$ cd visit3.0.2
visit@bea87fee3276:~/visit3.0.2$ mkdir build
visit@bea87fee3276:~/visit3.0.2$ cd build
visit@bea87fee3276:~/visit3.0.2/build$ /home/visit/third-party/cmake/3.9.3/linux-x86_
↪64_gcc-5.4/bin/cmake \
   -DCMAKE_BUILD_TYPE:STRING=Release -DVISIT_INSTALL_THIRD_PARTY:BOOL=ON -DVISIT_
↪ENABLE_XDB:BOOL=ON \
   -DVISIT_PARADIS:BOOL=ON -DVISIT_CONFIG_SITE="/home/visit/visit-config.cmake" ../src
visit@bea87fee3276:~/visit3.0.2/build$ make -j 4 package
visit@bea87fee3276:~/visit3.0.2/build$ mv visit3_0_2.linux-x86_64.tar.gz ../..
```

Now let's test it to make sure we can create an image.

```
visit@bea87fee3276:~/visit3.0.2/build$ cd ../..
visit@bea87fee3276:~$ cp visit3.0.2/src/tools/dev/scripts/visit-install .
visit@bea87fee3276:~$ ./visit-install 3.0.2 linux-x86_64 visit
visit@bea87fee3276:~$ visit/bin/visit -cli -nowin
>>> OpenDatabase("visit/data/curv2d.silo")
>>> AddPlot("Pseudocolor", "d")
>>> DrawPlots()
>>> SaveWindow()
>>> quit()
visit@:~$
```

Now let's go back to the second Command window and copy the binary distribution back out of the container and the image we created.

```
C:\Users\brugger\docker>docker cp bea87fee3276:/home/visit/visit3_0_2.linux-x86_64.
↪tar.gz .
C:\Users\brugger\docker>docker cp bea87fee3276:/home/visit/visit0000.png .
```

At this point you can exit your container.

```
visit@bea87fee3276:~$ exit
C:\Users\brugger\docker>
```

You should view the image to verify that it was produced correctly. You now have the binary distribution for VisIt 3.0.2 for Ubuntu 16.

10.14.4 Creating a Dockerfile From Scratch

To create a Dockerfile from scratch it is best to do so running interactively as root with the base operating system image. You can start by installing packages that your are certain you will need. At that point you can run build_visit until it fails, determining what missing package caused the failure, installing the missing package and repeating until you have gotten build_visit to complete with the third party libraries you want to build. From that experience you can create your Dockerfile.

10.14.5 Useful Docker Commands

Here are some useful Docker commands to manage images and containers.

```
docker image ls
docker container ls --all
```

(continues on next page)

(continued from previous page)

```
docker image rm <image id>
docker container rm <container id>
```

Docker will create a "checkpoint" after each command it executes. Everytime you partially create an image or execute a container it is saving those checkpoints. This can quickly start to consume a lot of disk space, so you should frequently list your images and containers and remove those that you no longer need.

If building an image fails and you want to take a look at what happened, you can convert the container into an image and then launch a bash shell in the image.

```
docker commit <container id> <image name>
docker run  -t -i <image name> /bin/bash
```

10.15 Site Reliability Engineering (SRE)

In the VisIt project, members of the development team are frequently called upon to respond to a variety of inquiries often originating directly from users. Some of these may relate to the *use* of VisIt such as

- How do I do a surface plot?
- How do I compute the mass in a given material?
- How do I get client/server to TACC working?

and some may relate to an *operational* aspect of either the VisIt software itself such as

- A botched *managed* VisIt installation.
- An update to host profiles to address site access changes.
- A missing database reader plugin.

or, the underlying computing infrastructure upon which VisIt depends such as

- An incompatible graphics driver.
- A downed file system or network.
- A trip in the security environment.

Typically, such inquiries originate from users in the midst of using VisIt and are encountering some kind of difficulty. In highly effective software projects, the work involved in handling such inquiries does not end with fixing this one user's problem and sending them on their way. When one user encounters a problem, there are probably others who have encountered the same problem. Furthermore, often the problems users encounter are suggestive of minor, easily fixed deficiencies in either the software itself or its associated processes and artifacts.

The continuous investment of effort to craft and carry out *small corrective actions* in response to such inquiries is a *best practice*. It represents a fusion of aspects of Google's Site Reliability Engineering (SRE) process (sometimes also called Systems Reliability Engineering or Services Reliability Engineering) and aspects of User Experience Regression Testing and/or User Experience Driven Development (UXDD).

For mature DOE software projects with wide reach and many users, SRE activity represents a brand of effort wholly different from conventional software product development, planning and execution. Like most DOE software projects, VisIt has no dedicated SRE resources. Instead, developers themselves must also support SRE work. Nonetheless, managing SRE work effectively and efficiently is an essential part of maintaining the overall productivity and sustainability of the software as well as the productivity of both users and developers of the software alike.

10.15.1 Goals

This document describes how the VisIt project manages its SRE activities. Some of the goals of this process are...

- To maintain a reputation for timely and quality response to customer inquiries.
- To develop a practice of routine *housekeeping* quality improvements to the VisIt software and associated processes and artifacts impacting user and/or developer productivity.
- To *load balance* SRE work in an equitable way across the development team.
- To reduce SRE interruptions for the team as a whole.
- To log, track and evolve a database of SRE activity and effort to help inform ongoing development plans and resource allocation.
- To identify and document escalation paths for major incidents.
- To aim for a four hour response time.

While many aspects of SRE are under the direct control of VisIt developers, some are not and involve collaboration with other teams in resolving. In most cases the extent of the VisIt team's involvement in the *operations* is confined primarly to the VisIt software itself; its development, testing, release and deployment which includes installations the VisIt team directly manages, hosted binary downloads for common platforms and the tools and resources to build from sources. Operational issues impacting VisIt but outside of this scope are typically delegated to other teams who are responsible for the associated processes and resources.

Business Hours

In the IT world where companies like Google, Apple and Amazon have whole teams dedicated to SRE activity, coverage is 24/7 and response time is measured in *minutes*. For the VisIt project where the majority of funded development takes place at Lawrence Livermore National Lab, coverage is during normal West Coast *business* hours, 8am-12pm and 1-5pm (GMT-8, San Francisco time zone), Monday through Friday excluding LLNL holidays and response time may be as much as four hours due to team members having to multi-task among many responsibilities.

Developer Away Notifications

As an aside but nonetheless related to the SRE response time goal of four hours, developers agree to notify the team if, during normal business hours, they will unexpectedly be away from their desk for periods longer than four hours. This is true whether working on or off site. Being away for periods shorter than four hours does not require such notification.

10.15.2 The Basic Process

SRE work is allocated and rotated among developers in one-week *shifts*. During a shift, one developer's *role* is to serve as the **Primary** SRE contact and a second developer's *role* is to serve as a **Backup**. Except for *escalations*, all other developers are free of SRE responsibilities for that week.

The *role* of the **Primary** is to *respond* within the response time goal, to each inquiry. Ideally, all SRE activity during the week is handled and *resolved* solely by the **Primary**. However, *escalations*, which we hope are rare, will wind up engaging the **Backup** and may even engage other developers. In addition, any *active SRE dicussions* that remain unresolved at the end of the week are formally *handed off* to the next **Primary**.

Active SRE discussions will be logged and tracked in a separate GitHub, issues-only repository within the visit-dav GitHub organization. Upon resolution of *serious incidents*, the **Primary** will prepare a brief *postmortem* to inform a discussion at the next project meeting of possible changes in practices to avoid repeating such major incidents.

Because SRE work tends to be *interrupt driven*, there is always the chance that the **Primary** will have no *active* discussions. At these *idle* times, the **Primary** shall use their time to address general *housekeeping* or other *low-hanging fruit* type work. In particular, there shall be no expectation that a developer serving as **Primary** can get any other work done beyond their active or idle SRE obligations. In slow weeks, its conceivable they can. But, there can be no implied assumption or expectation that this will be the case.

A *schedule* of the **Primary** and **Backup** assignments going out several months is periodically negotiated by the team and posted in the form of a shared calendar. **Primary** and **Backup** responsibilities are rotated so as to balance the load among team members.

The preceding paragraphs describe VisIt's SRE processes at a basic level and in the ideal. Nonetheless, several terms here (those that are links or in *italics* in the paragraphs above) require elaboration. In addition, there are also many practical matters which can serve to complicate the basic process. These details are addressed in the remaining sections.

10.15.3 Roles

The **Primary**'s role is to respond, within the response time goal, to each inquiry that occurs during that week including those that come in during the preceding weekend/holiday. The **Primary**'s goal is to *resolve* all inquiries by the end of their week.

The **Primary** has the sole responsibility for responding to inquiries and opening and resolving *SRE issue tickets*. When the **Primary** needs help to *resolve an SRE issue*, s/he should first enlist the **Backup**. This is an *escalation*. Nonetheless, the **Backup** (or other developers for that matter) are called into action only by explicit request of the **Primary**. Note that enlisting additional resources for help is part of *escalation* and is not the same as a *handoff*.

If the **Primary**'s schedule changes such that the response time goal may not be met, the **Primary** may temporarily *delegate* his/her role and responsibilities to the **Backup**. To the extent possible, such temporary delegation from **Primary** to **Backup** should be handled formally and by mutual agreement. Temporary delegation of the **Primary**'s role is also not the same as a *handoff*.

Ideally, the **Primary** is able to handle all SRE activity and no other developers are engaged. Thus, other developers are free to ignore customer inquiries as well as redirect customers who may contact them directly via email, phone or walk-in. It is a best practice to handle such redirections with a formal, three-way *handoff* confirming that the customer indeed makes contact with the **Primary**.

10.15.4 SRE vs. Product Development

Part of the reason for formalizing this process is the recognition of a different category of work, Site Reliability Engineering (SRE), that is essential part of maintaining the overall quality of a software product as well as the productivity of both developers and users of the software alike. Nonetheless, SRE work is very different from conventional *product development* type work where bug fixes, technology refreshes and feature enhancements are estimated and prioritized, methodically planned and resources are assigned to hit target release dates.

Issues that impact one user's productivity often impact others. Likewise for developers. When such issues come to our attention, whenever possible it is often helpful to identify *two* kinds of actions; a short-term *constructive* correction and a longer-term *comprehensive* solution.

Constructive Correction	Comprehensive Solution
Short term	Longer term
Faster response	Slower response
Low cost/benefit	Higher cost/benefit
Low risk	Higher risk
Unplanned	Planned
Mitigation	Resolution

A constructive correction has value only when it represents a step towards the comprehensive solution, can sufficiently reduce the impact of the issue and can be rolled out to users significantly sooner and with lower cost than the comprehensive solution. Ordinarily, a constructive correction is something the **Primary** handles as part of their SRE activity. The comprehensive solution, which often involves more planning and resource allocation, is handled as part of normal product development activities.

Constructive corrections can wind up falling through the cracks of traditional software project management and planning processes. However, such work also often represents low cost high benefit improvements in quality of either the software itself or the development or deployment processes supporting it. We refer to issues of this nature as general *low-hanging fruit* type issues.

Apart from acknowledging their existence, a key part of this process is the allocation of a small fraction of our resources for the sole purpose of supporting SRE activities and developing a practice of continuously crafting constructive corrective actions arising from SRE inquiries.

Consequently, another key role of the **Primary** is to use any time not working active SRE issues to fix other *low-hanging fruit* issues from the *product development* backlog. As a rule of thumb, low-hanging fruit is considered to be anything that the team believes is fixable within a half-day's (4 hours) worth of effort. When there are many such tasks in the system to work on, the **Primary** is free to use his/her judgment to decide which s/he can most productively address.

Part of the acknowledgment of this new category of work is the new issue tracker for tracking it. *New* SRE activity will start with an issue being added there. As an SRE incident unfolds it may result in either the same issue being moved to the product development issue tracker and/or new issue(s) being added to the product development tracker. Any new *product development* issues should be linked back to the original SRE issue that spawned them.

10.15.5 Active SRE Discussions

Active SRE issues will be logged and tracked as discussions in our GitHub Discussions within the visit-dav GitHub organization. For each new inquiry, a discussion will be created either by the primary or by the customer who started the discussion.

The primary will endeavor to capture all relevant information and communications in this issue. The use of GitHub Discussions for this purpose has a number of advantages over other options such as email including better search/browse as well as support for attachments. For the remainder of this document we simply use the term *conversation* to refer to the communication involved in an active SRE issue.

Upon receiving a *new* inquiry, if the inquiry did not start as a discussion (e.g. maybe it was a telephone hotline call, or a walk-in, the procedure is for the **Primary** to include the initial information (with the exception of inquries involving classified information) in a new GitHub discussion, attach the SRE label and from then on handle all communication through the *conversation* associated with that discussion.

For any work the **Primary** performs, even if it is a rather trivial amount of work to resolve, there should be an associated discussion for tracking that work. Tracking even the trivial tasks will help to build a database of activity we may be able to later mine to identify patterns and further process improvements.

An SRE discussion is *answered* when the associated inquiry is *resolved*. Or, it is *answered* and labeled `wontfix` if 21 days pass since the user last engaged in any conversation with VisIt developers to reach a resolution. Because of the manner in which the interface to GitHub discussions behaves, the natural chain of communication may not always lend itself well to using GitHub's *check this reply as the answer* feature to indicate an SRE discussion is resolved. In such circumstances, the adopted approach is to add a *new*, top-level comment with a link to whichever previous, embedded comment best answers the issue and then tag that new, top-level comment as the discussion's answer.

10.15.6 Supported Methods of Contact

An SRE inquiry with the VisIt team begins with a *first contact* and may optionally be followed by *ongoing* conversation. These two kinds of communication have different requirements and can involve different processes. This is due to the fact that we need to balance two priorities; *accessibility* for users and *productivity* for developers.

To maximize accessibility for users, we should support a wide variety of methods of first contact. However, to maximize productivity for developers, we should restrict methods of ongoing conversations.

A key benefit of having the VisIt team *co-located* with our user community is that users can spontaneously make a first contact with any one of us by an office drop-in or a tackle in the hallway or parking lot. This can even occur on social media platforms such as Confluence, Jabber, MS Teams, etc. where users can wind up engaging specific VisIt developers that happen, by nothing more than coincidence, to also be using those platforms.

A challenge with these spontaneous methods of first contact is that they inadvertently single out a specific developer who is then expected to at least *respond* and possibly even to also *resolve* the issue. But, these actions and the effort they involve are the responsibility of the primary SRE. Consequently, spontaneous methods of first contact can wind up jeopardizing the goals of our SRE process by making it difficult to track, allocate and manage SRE effort.

Therefore, the methods of first contact we officially support are those which engage the *whole team* instead of singling out a specific member. This includes...

- Creation of a GitHub discussion.
- Creation of a GitHub issue.
- Telephone call to the VisIt hotline.

Whenever users attempt a first contact through something other than the supported methods listed immediately above, the receiving developer should make an effort to *handoff* the inquiry to the primary SRE as quickly and politely as practical.

What does it mean for a method of first contact to be *supported*? It means there is an assurance that the particular platform is being monitored by VisIt team members during normal business hours such that the response time goal can be maintained. In addition, supported methods are encouraged and promoted in any documentation where VisIt support processes are discussed.

Balancing the priorites of user accessibility with developer productivity involves a compromise on the number of platforms we make an assurance to monitor. Currently, this is limited to those listed above. However, the selected methods should be periodically reevaluated. If there is some platform which seems to be gaining popularity among users, it could either be added to the list of supported platforms or perhaps it could be integrated with email in the same way GitHub issue conversations have been.

10.15.7 Response Time and Response vs. Resolution

The response time goal of four hours was chosen to reflect the worst case practicalities of team members' schedules and responsibilities. For example, if the **Primary** has meetings just before and just after the lunch hour break, there can easily be a four hour period of time where inquiries go unattended. Typically, we anticipate response times to be far less than four hours and certainly, when able, the **Primary** should respond as quickly as practical and not use the four hour goal as an excuse to delay a prompt response.

Since a majority of funding for VisIt is from LLNL and since VisIt developers are co-located with many of its LLNL users, certainly these users as well as their direct collaborators are accustomed to response times of less than four hours. For example, the VisIt project operates a telephone hotline and also frequently handles walk-ins. As an aside, after a recent small test effort to maintain a rapid response time, a noticeable up-tick in user email inquiries was observed suggesting that rapid response times have the effect of encouraging more user interactions.

It is also important to distinguish between *response* and *resolution* here. A key goal in this process is to ensure that customer inquiries do not go unanswered for a long time. However, *responding* to a customer inquiry does not

necessarily mean *resolving* it. Sometimes, the only response possible is to acknowledge the customer's inquiry and let them know that the resources to address it will be allocated as soon as practical. In many cases, an *immediate* response to acknowledge even just the receipt of a customer's inquiry with no progress towards actual resolution goes a long way towards creating the goodwill necessary to negotiate a day or more of time to respond more fully and maybe even resolve.

Resolution of an SRE issue often involves one or more of the following activities...

- Answering a question or referring a user to documentation.
- Diagnosing the issue.
- Developing a work-around for users.
- Developing a reproducer for developers.
  - This may include any relevant user data files as well as approval, where appropriate for world read access to such data as part of attaching to a GitHub issue.
- Identifying any *low-hanging fruit* type work that would address, even if only in part, the original SRE inquiry and then engaging in the *housekeeping* work to resolve it.
- Determining if the user's issue is known (e.g. an issue ticket already exists).
- Updating a known issue with new information from this user, perhaps adjusting labels on the issue or putting the issue back into the un-reviewed state for further discussion at a VisIt project meeting.
- Identifying and filing a new *product development* type issue ticket.

To emphasize the last bullet, *resolution* does not always mean a customer's issue can be addressed to *satisfaction* within the constraints of the SRE process as it is defined here. Sometimes, the most that can be achieved is filing a highly informative issue ticket to be prioritized, scheduled and ultimately resolved as part of normal VisIt product development activities. The SRE issue gets *promoted* to a product development issue. It is closed in the SRE issue tracker and new issue is opened in the product development issue tracker including a reference to the original SRE issue. Doing so does serve to *resolve* the original SRE issue that initiated the work.

10.15.8 Serious Incidents and Postmortems

Serious incidents are those that have significant productivity consequences for multiple users and/or require an inordinate amount of resources (either time or people or both) to diagnose, work-around and/or ultimately properly correct.

When such incidents occur, it is a best practice to spend some time considering adjustments in processes that can help to avoid repeating similar issues in the future.

When such incidents reach SRE resolution, the **Primary** will prepare a brief *postmortem* (often just a set of bullet points) explaining what happened and why, estimating the amount of resources that were needed to resolve the incident, describing key milestones in the work to resolve the incident and suggesting recommendations for changes in processes to prevent such incidents from being repeated. This *postmortem* will be used to guide team discussion during a subsequent weekly project meeting.

10.15.9 Handoffs

Our SRE processes involve two kinds of *handoffs*. One is the redirection of a customer who makes contact with a developer not serving as the **Primary**. The other is the handoff of unresolved SRE issues from one week's **Primary** to the next.

To handle customer redirection handoffs, it is a best practice to use a three-way handoff giving the customer some assurance that their initial contact with someone is successfully handed off to the **Primary**. For example, for a call-in, it is a best practice to try a three-way call transfer. For some developers, the prospect of redirecting friends and colleagues with whom they may have long standing relationships may be initially uncomfortable. But it is important to recognize that this an essential part of achieving one the goals of this process, to reduce SRE interruptions for the team as a whole.

If an active SRE issue cannot be resolved within the week of a **Primary**'s assignment, it gets handed off to the next week's **Primary**. Such handoffs shall be managed formally with a comment (or email) to the customer(s) and the next week's **Primary** and **Backup** in the associated GitHub issue. The associated issue(s) in the SRE issues repository shall be re-assigned by the previous week's **Primary** upon ending their shift. However, a preceding week's **Primary** may be near enough to *resolving* an SRE issue that it makes more sense for him/her to carry it completion in the following week. In this case, s/he will leave such issues assigned to themselves.

10.15.10 Escalation

SRE inquiries may escalate for a variety of reasons. The technical expertise or authority required may be beyond the **Primary**'s abilities or other difficulties may arise. For issues that the **Primary** does not quickly see a path to resolution, the **Backup** should be enlisted first. When developer expertise other than **Backup** is needed, the **Primary** should try to engage other developers using the `@` mention feature in the associated GitHub issue. However, where a **Primary** is responsible for maintaining the response time goal, other developers so enlisted are free to either delay or even decline to respond (but nonetheless inform the **Primary** of this need) if their schedule does not permit timely response. Such a situation could mean that the only remaining course of action for the **Primary** to *resolve* the issue is to file a product development issue as discussed at the end of a preceding section.

If after investigation and diagnosis the work required to resolve an SRE incident remains highly uncertain or is not believed to be a *low-hanging-fruit* type task, the **Primary** should search the *product development* issues to see if this is a known issue and, if so, add additional information to that known issue about this new SRE incident (and perhaps remove the *reviewed* tag from the issue to cause the issue to be re-reviewed at the next VisIt project meeting) or submit a *new* issue to the product development issue tracker. Such action then *resolves* the original SRE issue.

Special Considerations for Classified Computing

Occasionally, incidents arise that may be specific to a classified computing environment. This is not too common but does happen and it presents problems for a geographically distributed team. In many ways, handling such an incident is just a different form of *escalation*.

On the one hand, customers working in a classified computing environment are accustomed to longer response times. On the other hand, such work is often a high priority and requires rapid response from a developer that is on site with classified computing access.

Our current plan is to handle this on a case-by-case basis. If neither the **Primary** nor **Backup** are able to handle a customer response incident requiring classified computing, the **Primary** should

- First determine the customer's required response time. It may be hours or it may be days. If it is days. Its conceivable the issue could be handled in the following week by a new **Primary/Backup** pair.
- If customer indicates immediate response is required, the **Primary** should query the whole team to arrange another developer who can handle it.

10.15.11 Scheduling and Load Balancing

To balance the work load of SRE, the responsibilities of the **Primary** and **Backup** are rotated, round-robin among team members. For example, on a team of eight developers, each would serve as **Primary** only one week in eight or 12.5%

of their time. However, a number of factors complicate this simple approach including percent-time assignments of team members, alternate work schedules, working remotely, travel, vacations, trainings, meetings, etc.

Round-robin assignment may lead to a fair load by head-count but isn't weighted by percent-time assignments. From a percent-time assignment perspective, it might be more appropriate for a developer that is only 50% time on VisIt to serve as the **Primary** only half as often as a 100% time developer.

Since a majority of VisIt developers divide their time across multiple projects, we use 50% as the *nominal* developer assignment. Because of all the factors that can effect scheduling, the VisIt project has opted to manage scheduling by periodically negotiating assignments 1-3 months into the future and recording the assignments on a shared calendar. The aim is an approximately round-robin load balancing where contributors who are more than 50% time on VisIt are occasionally assigned an extra week. Either **Primary** or **Backup** can make last minute changes to the schedule by finding a willing replacement, updating the shared calendar and informing the rest of the team of the change.

Whenever possible, an experienced **Backup** will be scheduled with a less experienced **Primary**.

10.15.12 A Common Misconception: SRE is an Interruption to Programmatic Work

When faced with a long backlog of development tasks, team members can all too easily perceive SRE work as an *interruption* to those tasks. This is a common misconception. SRE is an important aspect to a successful product and project on par with any other major development work. It is part of what is involved in keeping the software working and a useful tool in our customer's workflows not only here at LLNL, likely VisIt's biggest customer, but wherever in DOE/DOD and elsewhere in the world VisIt is used.

Indeed, there are several advantages in having developers involved with SRE activities. These include..

- Learning what problems users are using the tool to solve.
- Learning how users use the tool.
- Learning what users find easy and what users find hard about the tool.
- Learning where documentation needs improvement.
- Learning where the user interface needs improvement.
- Learning operational aspects of user's work that the tool can impact.
- Building collaborative relationships with other members of the organization.
- Learning how users operate in performing their programmatic work for the organization which helps to inform planning for future needs.

In short, the work involved in Software Reliability Engineering (SRE) and ensuring productivity of both users and developers of VisIt *is* programmatic work. The practice of having software development staff *integrated* with *operations* is more commonly referred to as *DevOps*. There is a pretty good video that introduces these concepts.

10.16 OpenGL in VisIt

VisIt requires an OpenGL 3.2 context to work properly. Mesa provides a 3.3 context. Most desktop computers or laptops with graphics cards provide an OpenGL 4.6 or 4.7 context. For some unknown reason most (if not all) Linux HPC systems only provide a 3.0 context.

When using the QVTKOpenGLWidget with Qt, the following code snippet needs to be executed before creating the QApplication to tell Qt that it needs an OpenGL 3.2 context.

```
//
// Setting default QSurfaceFormat required with QVTKOpenGLwidget.
//
auto surfaceFormat = QVTKOpenGLWidget::defaultFormat();
surfaceFormat.setSamples(0);
surfaceFormat.setAlphaBufferSize(0);
QSurfaceFormat::setDefaultFormat(surfaceFormat);
```

10.16.1 OpenGL in Qt

The sections of Qt that deal with OpenGL are

```
qtbase/src/opengl
qtbase/src/openglextensions

qtbase/src/plugins/platforms/xcb/gl_integrations/xcb_glx

qtbase/src/platformsupport/glxconvenience
```

The context creation is performed in

```
qtbase/src/plugins/platforms/xcb/gl_integrations/xcb_glx/qglxintegration.cpp

void QGLXContext::init(QXcbScreen *screen, QPlatformOpenGLContext *share)
```

10.16.2 OpenGL in VTK

The sections of VTK that deal with OpenGL are

```
GUISupport/Qt
Rendering/OpenGL2
```

The context creation is performed in

```
GUISupport/Qt/QVTKOpenGLWidget.cxx
```

Other stuff is done in

```
Rendering/OpenGL2/vtkOpenGLRenderWindow.cxx
```

10.16.3 OpenGL documentation

GLX is the OpenGL extension to the X Window System. In the X Window System, OpenGL rendering is made available as an extension to X in the formal X sense: connection and authentication are accomplished with the normal X mechanisms. As with other X extensions, there is a defined network protocol for the OpenGL rendering commands encapsulated within the X byte stream.

Since performance is critical in 3D rendering, there is a way for OpenGL rendering to bypass the data encoding step, the data copying, and interpretation of that data by the X server. This direct rendering is possible only when a process has direct access to the graphics pipeline.

- Documentation on GLX. * GLX functions all start with “glX” and GLX constants all start with “GLX”.

- Documentation on creating an OpenGL 3.0 context. * It is the source of the test in build_visit to determine if the OpenGL on a system supports creating a 3.2 context.
- Documentation on the history of the changes to OpenGL.

10.17 Docker Containers For CI Testing

We use Azure Pipelines for CI testing VisIt's Pull Requests, located at VisIt Azure DevOps Space.

To speed up our CI testing we use Docker containers with pre-built third party libraries (TPLs). These containers leverage our `build_visit` third party build process. The Docker files and build scripts used to create these containers are in `scripts/ci/docker`. The process to create the container varies somewhat if you have Docker installed on the same system as the git checkout of your branch.

10.17.1 Creating the container with Docker on the same system as the git checkout

Create the container using `build_docker_visit_ci.py`.

```
cd scripts/ci/docker
python build_docker_visit_ci.py
```

This creates the container with a tag that will include today's date and a short substring of the current git hash.

Example Tag: `visitdav/visit-ci-develop:2020-11-11-sha433ef0`

This will typically take several hours to complete.

10.17.2 Creating the container without Docker on the same system as the git checkout

Create two tar files and the Docker command to create the container using `build_docker_visit_ci.py`.

```
cd scripts/ci/docker
python build_docker_visit_ci.py
```

This will create the files

```
visit.build_visit.docker.src.tar
visit.masonry.docker.src.tar
```

The command will also fail with output similar to this

```
[exe: git rev-parse HEAD]
[exe: docker build -t visitdav/visit-ci-develop:2020-11-11-sha433ef0 . --squash]
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker␣
↪daemon running?
```

You will use the `docker build ...` command to build the container on the system you have Docker installed.

You can now move over to the system where you have Docker installed. Bring up up a shell window and create a new directory or folder to contain the files necessary to create the container.

```
mkdir docker_ci
cd docker_ci
```

Now copy all the files in `scripts/ci/docker` to your new folder. Now you can run the `docker build ...` command to create the container. For example:

```
docker build -t visitdav/visit-ci-develop:2020-11-11-sha433ef0 . --squash
```

This will typically take several hours to complete.

10.17.3 Push the container to Dockerhub

Now that you have created your Docker container image, you are ready to push it to VisIt's DockerHub Registry using `docker push <container-name>`.

If you do not already have a DockerHub account, go here and sign up for one. Then contact another member of visitdav and ask to be added to the organization.

You will need to be logged into DockerHub to successfully push. Here is an example push command:

```
docker login docker.io
docker push visitdav/visit-ci-develop:2020-11-11-sha433ef0
```

10.17.4 Update VisIt to use the new Docker image

To change which Docker Image is used by Azure, edit `azure-pipelines.yml` and change the *container_tag* variable.

```
#####
# TO USE A NEW CONTAINER, UPDATE TAG NAME HERE AS PART OF YOUR PR!
#####
variables:
  container_tag: visitdav/visit-ci-develop:2020-12-09-shaf6ef22
```

If you change the operating system, you will need to update the *vmImage* variable. It is specified in two locations.

```
pool:
  vmImage: 'ubuntu-18.04'
```

When the PR is merged, the Azure changes will be merged and PRs to develop will now use the new container.

10.18 Contributing Host Profiles

Host profiles live in /src/resources/hosts/<compute center name>.

There is a script and a couple of helper files in the 'hosts' directory that allow the centers and profiles to be added during the installation of VisIt, or imported later via the gui.

- If you add or remove directories from 'hosts', please edit **networks.dat** and **tools/dev/scripts/visit-install** accordingly.
- If you add or remove directories or files, please regenerate **networks.json** by running **dump_dir_to_networks_json.py**.
- Note that if run on Windows, the generated **networks.json** file will have dos-style line endings. Convert it to Unix style before committing.
- If you want to specify a default host configuration for a center, edit **default_configs.dat**.

Steps for creating host profiles can be found in *Host Profiles*.

Steps for creating a Pull Request to contribute host profiles to VisIt's repo can be found in *Creating a Pull Request*

10.19 Process Launching in VisIt

This section describes the process launching of the various processes that run as part of VisIt. It does so using an example where the user is running client / server.

When running in client / server mode, a *VCL* is lauched on the login node on the server system using `ssh`.

When ssh tunneling is used with client / server, ssh port forwarding is used to forward ports from the login node on the server system to the client system. Ssh will only forward ports that originate on the server login node, so *Engines* that are launched in batch will connect to a bridge port from the *VCL*.

When a process is launched, the process is launched and then the launching process listens on a port waiting for the launched process to connect in order to set up a communication socket.

10.19.1 The Sequence of Actions that Occur when Creating the Processes Shown Above

1. The user launches the *GUI*.
2. The *GUI* then launches the *Viewer* with the following command.

```
visit
-v
3.2
-viewer
-forceversion
3.2.1
-debug
5
-geometry
1560x1014+360+28
-borders
26,4,4,4
-shift
0,0
-preshift
4,26
-defer
-host
quartz1148.llnl.gov
-port
5600
-key
b5fb29c487dd7811e14f
```

This launches the *Viewer* process and causes it to connect back to the *GUI* on port 5600.

3. The the *Viewer* launches the *Meta Data Server* with the following command.

```
visit
-v
3.2
```

(continues on next page)

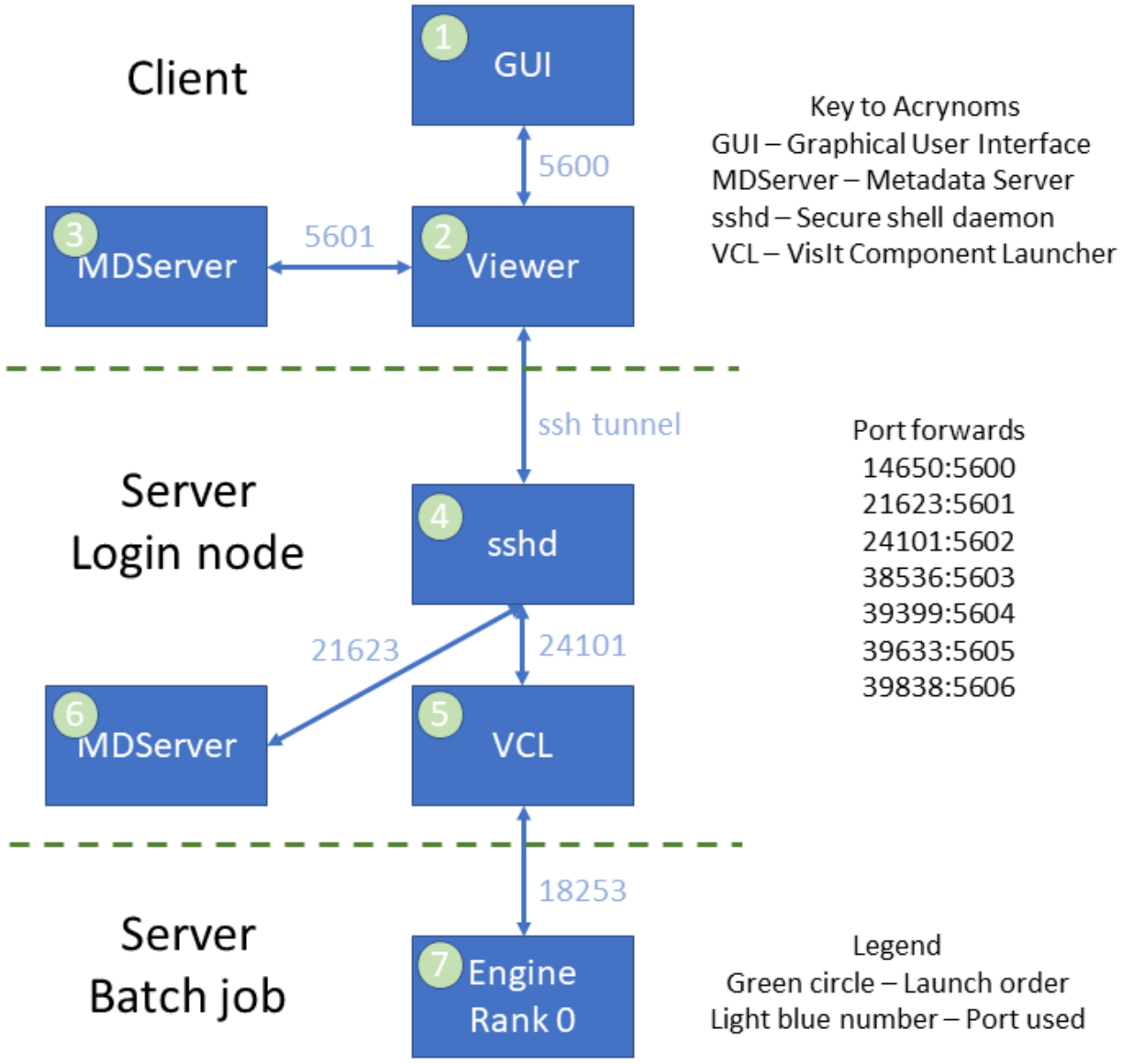


Fig. 10.17: Block diagram of the VisIt processes in the Client / Server example

(continued from previous page)

```
-mdserver
-debug
5
-forceversion
3.2.1
-host
quartz1148.llnl.gov
-port
5601
-key
b5fb29c487dd7811e14f
```

This launches the *Meta Data Server* and causes it to connect back to the *Viewer* on port 5601.

4. The user now goes to the *File open* window and specifies a remote host.
5. The *Viewer* then launches a *Meta Data Server* on the server login node with the following command.

```
/usr/gapps/visit/bin/visit
-v
3.2
-mdserver
-debug
5
-forceversion
3.2.1
-dir
/usr/gapps/visit
-idle-timeout
480
-noloopback
-guesshost
-port
5601
-key
2f2468602a0e1ff62c07
```

6. Since the command is to be run on a remote system and a *VCL* isn't yet running on the remote system, the *Viewer* launches a *VCL* using `ssh` with the following command.

```
ssh
-R
14650:127.0.0.1:5600
-R
21623:127.0.0.1:5601
-R
24101:127.0.0.1:5602
-R
38536:127.0.0.1:5603
-R
39399:127.0.0.1:5604
-R
39633:127.0.0.1:5605
-R
39838:127.0.0.1:5606
pascal.llnl.gov
/usr/gapps/visit/bin/visit
```

(continues on next page)

(continued from previous page)

```
-v
3.2
-vcl
-debug
5
-forceversion
3.2.1
-dir
/usr/gapps/visit
-idle-timeout
480
-noloopback
-sshtunneling
-host
localhost
-port
24101
-key
2f2468602a0e1ff62c07
```

The `-R` options to `ssh` set up port forwards from the server login node to the client system.

7. Once the *VCL* is launched the *Viewer* then tells the *VCL* to launch the *Meta Data Server* on the server login node.

The *VCL* translates the `5601` to `21623`, which is the port that `ssh` forwards to `5601` on the client system.

8. The user opens a database on the server system.
9. The *Viewer* tells the *VCL* to launch the *Engine* with the following command.

```
/usr/gapps/visit/bin/visit
-v
3.2
-engine
-dir
/usr/gapps/visit
-noloopback
-np
36
-nn
1
-p
pvis
-b
wbronze
-t
30:00
-l
msub/srun
-forcestatic
-idle-timeout
480
-debug
5
-forceversion
3.2.1
-noloopback
```

(continues on next page)

(continued from previous page)

```
-guesshost
-port
5600
-key
8e602a31f092894eda54
```

The *VCL* sets up the bridge at port `INADDR_ANY/18253` to the tunneled port `localhost/14650`. Then an `msub` batch job is submitted with the following command.

```
msub -v HOME=/g/g17/brugger,LIBPATH=/usr/gapps/visit/3.2.1/linux-x86_64/lib,
LD_LIBRARY_PATH=/usr/gapps/visit/3.2.1/linux-x86_64/lib/osmesa:
/usr/gapps/visit/3.2.1/linux-x86_64/lib/mesagl:
/usr/gapps/visit/3.2.1/linux-x86_64/lib:
/usr/gapps/visit/bin/../3.2.1/linux-x86_64/lib:
/usr/tce/packages/mvapich2/mvapich2-2.3-intel-19.0.4/lib:
/usr/tce/packages/intel/intel-19.0.4/lib/intel64,VISITHOME=/usr/gapps/visit/3.2.1,
VISITARCHHOME=/usr/gapps/visit/3.2.1/linux-x86_64,
VISITPLUGINDIR=/g/g17/brugger/.visit/3.2.1/linux-x86_64/plugins:
/usr/gapps/visit/3.2.1/linux-x86_64/plugins -l nodes=1 -l walltime=30:00 -q
pvis -A wbronze /tmp/visit.brugger.Mon-Nov-22-11:03:01-2021
```

The file `/tmp/visit.brugger.Mon-Nov-22-11:03:01-2021` contains.

```
#!/bin/sh
cd /g/g17/brugger
ulimit -c 0
srun -n 36 --nodes=1 --tasks-per-node=36 /usr/gapps/visit/3.2.1/linux-x86_64/bin/
↪engine_par
-forceversion 3.2.1 -dir /usr/gapps/visit -forcestatic -idle-timeout 480 -debug 5
-noloopback -sshtunneling -host pascal83 -port 18253 -key 8e602a31f092894eda54
```

Note that the *Engine* is being told to connect to port 18253 on pascal, which is the bridging port set up in the *VCL* to the tunneled port 14650. When the *Engine* is eventually launched, the rank 0 MPI process will connect back to the *Viewer* using port 18253.

10.20 Updating Plugins

When modifying existing plugins, keep in mind that there are certain files that have an automated re-generation process. These include *CMakeLists.txt*, *PluginInfo*, and (for Plots and Operators) *Attributes (cpp, python, java)*. Being able to regenerate these files aids in future maintenance when the API needs to be changed for all plugins.

10.20.1 CMakeLists.txt

Regeneration of CMakeLists.txt is handled by the *xml2cmake XML Tool*. Most changes to CMakeLists.txt can be handled by modifying/adding the necessary tags to the plugin's .xml file. Some tags have an optional *components* attribute that specifies for which component the tag applies. Multiple components can be specified in a single tag, and must be comma separated. If no *components* attribute is specified, the tag will be applied to all components.

Component	Short name used in xml file
Engine(both Serial and Parallel)	**E**
Serial Engine only	**ESer**
Parallel Engine only	**EPar**
MDServer	**M**
GUI	**G**
Python scripting	**S**
Viewer	**V**
Widgets	**W**
Widgets for viewer	**VW**

Here is a list of most often updated tags:

Tag	Purpose	Supported components
CXXFLAGS	Include directories	M,ESer,EPar
DEFINES	Compile-time/preprocessor definitions	M,ESer,EPar
LDFLAGS	Link flags, link directories	M,ESer,EPar
LIBS	Link libraries	G,M,ESer,EPar,V
Files	Files to compile	All
WIN32DEFINES	Windows specific definitions	None, always applies to all

xml2cmake also supports the *Conditional* addition of an include directory, definition, link library or source file. The condition is a CMake variable that describes something related to the build: e.g. an OS-specification (*WIN32, LINUX, MACOS*), the availability of a third-party library (*HAVE_CONDUIT*) or a build option that can be toggled on/off (*VISIT_OSPRAY*). Conditionals must be specified in the *.code* file with *Target* specified as *xml2cmake*, as seen in the code file for the Volume plot:

```
Condition: VISIT_SLIVR
Definitions: -DVISIT_SLIVR

Target: xml2cmake
Condition:       VISIT_OSPRAY
Includes:        ${OSPRAY_INCLUDE_DIR}
VLinkLibraries:  ${OSPRAY_LIBRARIES}
ELinkLibraries:  ${OSPRAY_LIBRARIES}
```

These conditionals create these lines in the CMakeLists.txt:

```
LINK_DIRECTORIES(${VISIT_LIBRARY_DIR} )
```

```
if(VISIT_SLIVR)
    add_definitions(-DVISIT_SLIVR)
```

```
    SET(INSTALLTARGETS ${INSTALLTARGETS} GVolumePlot VVolumePlot)
```

```
ADD_TARGET_DEFINITIONS(EVolumePlot_ser ENGINE)
SET(INSTALLTARGETS ${INSTALLTARGETS} EVolumePlot_ser)
```

```
    ADD_TARGET_DEFINITIONS(EVolumePlot_par ENGINE)
    SET(INSTALLTARGETS ${INSTALLTARGETS} EVolumePlot_par)
```

10.20.2 Info files

Regeneration of Info files is handled by the *xml2info XML Tool*.

Changes to Info files should be placed in the plugin's *.code* file with the *Target* specified as *xml2info*, and a corresponding *Function* tag placed in the *.xml* file.

For instance, if an operator's Filter becomes useful elsewhere in VisIt, it may be moved out of the operator's directory and into src/avt/Filters. In which case, a custom method for allcoating the filter must be created, as was done for the *Displace* operator. Here is the *Displace* operator's code file:

```
Target: xml2info
Function: DisplaceEnginePluginInfo::AllocAvtPluginFilter
Declaration: virtual avtPluginFilter *AllocAvtPluginFilter();
Definition:
// ****************************************************************************
//  Method: DisplaceEnginePluginInfo::AllocAvtPluginFilter
//
//  Purpose:
//    Return a pointer to a newly allocated avtPluginFilter.
//
//  Returns:    A pointer to the newly allocated avtPluginFilter.
//
//  Programmer: childs -- generated by xml2info
//  Creation:   Fri May 18 16:05:37 PST 2007
//
//  Modifications:
//
//    Hank Childs, Fri May 18 16:01:06 PDT 2007
//    Forced to hand-edit return of avtDisplacePluginFilter ... not
//    avtDisplaceFilter.
//
// ****************************************************************************
#include <avtDisplacePluginFilter.h>
avtPluginFilter *
DisplaceEnginePluginInfo::AllocAvtPluginFilter()
{
    return new avtDisplacePluginFilter;
}
```

and the corresponding entry in the .xml file

```
      <Function name="DisplaceEnginePluginInfo::AllocAvtPluginFilter" user="false"␣
↪member="true">
      </Function>
```

10.20.3 Attributes

Regeneration of Attributes is handled by three *XML Tools*: one for cpp files (`xml2atts`, one for python (`xml2python` and one for java (`xml2java`.

When Attributes objects are changed (e.g. members removed or renamed), our policy is to maintain backward compatibility with older versions for at least 2 point releases. VisIt can encounter older Attributes objects in either Python CLI code or from previous saves of XML content (e.g. settings). The logic to support backward compatibility is handled in custom `ProcessOldVersions` (cpp) and `getattr/setattr` (python) methods.

The `VISIT_OBSOLETE_AT_VERSION` macro must be used to specify a version number where support for the old fields will be completely removed from VisIt, at least 2 point releases or later from the version where the removal/renaming occurs. A deprecation message should be issued from both the cpp and python code. Cpp code has a default `DeprecationMessage` method. For a removed field, it takes as arguments: the name of the removed field, the version where it will be completely unsupported. The arguments for a renamed field: the old field name, the new field name, the version where the old name will be completely unsupported.

Here is an example from the WellBore plot, which had a field removed in version 3.0.0

Here's code file for the cpp change:

```
Target: xml2atts
Function: ProcessOldVersions
Declaration: virtual void ProcessOldVersions(DataNode *parentNode, const char␣
↪*configVersion);
Definition:
// ****************************************************************************
// Method: WellBoreAttributes::ProcessOldVersions
//
// Purpose:
//   This method allows handling of older config/session files that may
//   contain fields that are no longer present or have been modified/renamed.
//
// Programmer: Kathleen Biagas
// Creation:   April 4, 2018
//
// Modifications:
//
// ****************************************************************************
#include <visit-config.h>
#ifdef VIEWER
#include <avtCallback.h>
#endif

void
WellBoreAttributes::ProcessOldVersions(DataNode *parentNode,
                                       const char *configVersion)
{
#if VISIT_OBSOLETE_AT_VERSION(3,3,2)
#error This code is obsolete in this version. Please remove it.
#else
    if(parentNode == 0)
    return;

    DataNode *searchNode = parentNode->GetNode("WellBoreAttributes");
    if(searchNode == 0)
        return;
```

(continues on next page)

(continued from previous page)

```
    if (VersionLessThan(configVersion, "3.0.0"))
    {
        if (searchNode->GetNode("wellLineStyle") != 0)
        {
#ifdef VIEWER
            avtCallback::IssueWarning(DeprecationMessage("wellLineStyle", "3.3.2"));
#endif
            searchNode->RemoveNode("wellLineStyle");
        }
    }
#endif
}
```

Now for the python getattr and setattr methods:

```
Target: xml2python
Code: PyWellBoreAttributes_getattr
Prefix:
Postfix:
#if VISIT_OBSOLETE_AT_VERSION(3,3,2)
#error This code is obsolete in this version. Please remove it.
#else
    // Try and handle legacy fields

    //
    //  Removed in 3.0.0
    //
    // wellLineStyle and it's possible enumerations
    bool wellLineStyleFound = false;
    if (strcmp(name, "wellLineStyle") == 0)
    {
        wellLineStyleFound = true;
    }
    else if (strcmp(name, "SOLID") == 0)
    {
        wellLineStyleFound = true;
    }
    else if (strcmp(name, "DASH") == 0)
    {
        wellLineStyleFound = true;
    }
    else if (strcmp(name, "DOT") == 0)
    {
        wellLineStyleFound = true;
    }
    else if (strcmp(name, "DOTDASH") == 0)
    {
        wellLineStyleFound = true;
    }

    if (wellLineStyleFound)
    {
        PyErr_WarnEx(NULL,
            "wellLineStyle is no longer a valid WellBore "
            "attribute.\nIt's value is being ignored, please remove "
            "it from your script.\n", 3);
        return PyInt_FromLong(0L);
```

(continues on next page)

(continued from previous page)

```
    }
#endif


Code: PyWellBoreAttributes_setattr
Prefix:
Postfix:
#if VISIT_OBSOLETE_AT_VERSION(3,3,2)
#error This code is obsolete in this version. Please remove it.
#else
    // Try and handle legacy fields
    if(obj == &NULL_PY_OBJ)
    {
        //
        //  Removed in 3.0.0
        //
        if(strcmp(name, "wellLineStyle") == 0)
        {
            PyErr_WarnEx(NULL, "'wellLineStyle' is obsolete. It is being ignored.",␣
↪3);
            Py_INCREF(Py_None);
            obj = Py_None;
        }
    }
#endif
```

Important: Changes to fields in the Attributes are not allowed in patch releases, as it may cause incompatibility between client and server running different patches of the same major.minor release.

10.21 Debugging Tips

This section describes various method for debugging VisIt when you run into problems. Whether it is a crash with a new filter you've designed, or badly formed data from a new database reader, one of these options should help.

10.21.1 Debug logs

The first method for debugging in VisIt is by using VisIt's debug logs. When you run `visit` on the command line, you can optionally add the `-debug 5` arguments to make VisIt write out debugging logs. The number of debugging logs can be 1, 2, 3, 4, or 5, with debugging log 5 being the most detailed. When VisIt's components are told to run with debugging logs turned on, each component writes a set of debugging logs. For example, the database server component will write A.mdserver.1.vlog, A.mdserver.2.vlog,. . . ,A.mdserver.5.vlog if you pass `-debug 5` on the VisIt command line. Subsequent runs of VisIt will rename existing logs with the initial letter advanced to the next letter in the alphabet. For example A.mdserver.5.vlog will be renamed to B.mdserver.5.vlog. At most five sets of debugging logs will be kept. The logs from the most current run will always begin with *A*. If you don't want that behavior, you may add `-clobber_vlogs` to VisIt's command line arguments. The A.mdserver*.vlog and A.engine*.vlog files are useful when debugging a database reader plug-in. The A.viewer*.vlog and A.engine*.vlog files are usefule when debugging a plot plugin.

The debugging logs will contain information written to them by the debugging statements in VisIt's source code. If you want to add debugging statements to your AVT code then you can use the *debug1*, *debug2*, *debug3*, *debug4*, or *debug5* streams as shown in the following code listing. Keep in mind that level 1 debug log files are less populated than level 5, and may be most useful when making temporary code modifications and debugging specific problems and not just providing general information.

Example for using debug streams

```
// NOTE - This code is incomplete and is for example purposes only.
// Include this header for debug streams.
#include <DebugStream.h>

vtkDataSet *
avtXXXXFileFormat::GetMesh(const char *meshname)
{
    // Write messages to different levels of the debug logs.
    debug1 << "Hi from avtXXXXFileFormat::GetMesh" << endl;
    debug4 << "Many database plug-ins prefer debug4" << endl;
    debug5 << "Lots of detail from avtXXXXFileFormat::GetMesh" << endl;
    return 0;
}
```

Other forms of the debug argument

To create debug logs for only specific components, use `-debug_<compname> <level>`. For example `-debug_mdserver 4` will run the mdserver with level 4 debugging. Multiple `-debug_<compname> <level>` args are allowed.

For parallel engine logs, `-debug_engine_rank <r>` can be used to restrict debug output to the specified rank. You can even have only every Nth processor output debug logs by using `-debug-process-stride N`.

10.21.2 Dumping VTK objects and pipeline information to disk

In addition to the `-debug` argument, VisIt also supports a `-dump` argument. The `-dump` argument tells VisIt's compute engine to write intermediate results from AVT filters, scalable rendered images, and html pages. The `-dump` option takes an optional argument that specifies the directory for -dump output files.

The intermediate results are VTK files containing the data for every stage of the pipeline execution so you can view the changes to the data made by each AVT filter. Each VTK filename begins with a number indicating the order of the filter in the pipeline that saved the data, followed by an indication of whether it is an `input` or `output` for the filter, and finally the filter name. For example, the input to the project filter could be `0006.input.avtProjectFilter.vtk`.

The html files contain information about input to and output from each filter, including spatial and data extents, pipeline flags, and number of data files input and output.

While the VTK files dumped by this option are more useful when debugging plots and operators, they can still be useful for debugging database plugins, as data sent from the plugin can be examined. The files generated at this first stage have `gdb` as the beginning of the filename, such as `gdb.0003.output.GetOutput.dom0000.vtk`

When you run VisIt with the `-dump` argument, many files will be created since the data is saved at every stage in the execution of VisIt's data processing pipeline. It is a good idea to keep this in mind and to remove those files from time to time.

Dumping only pipeline html files

To get only the html pages, and no VTK objects, use `-info-dump` instead. It also takes an optional argument specifying the directory for output files.

Comparing –dump outputs

Sometimes the output from `-dump` looks correct and doesn't immediately reveal why things are broken. In those instances it may be helpful to compare a `-dump` run from a version of VisIt that isn't broken to the `-dump` run from the version that is broken. Doing such a comparison might more quickly reveal which filter in the pipeline has changed it's output VTK object or even pipeline information.

10.21.3 Attaching a debugger

VisIt has various options for attaching debuggers on Linux machines, including `gdb`, `totalview`, and `valgrind` to name a few. The full list is available in the *Startup options* section under `Debugging options`.

10.21.4 WaitUntilFile function

VisIt has a utility function called *WaitUntilFile* that will halt process execution until the file passed into the function has been created. It takes one argument, a full-path filename referencing a file that does not yet exist. The function will enter a loop, alternating between short sleeps and checking if the given filename exists. Once it determines the filename exists, the function will exit and normal program flow will continue. This allows time for you to attach a debugger to the running process and set breakpoints before creating the filename that signals the function to exit.

While this function can be used anywhere in VisIt's pipeline, it is especially useful for debugging problems with a component's startup process, where it may be harder to attach a debugger in time.

WaitUntilFile is declared in VisIt's *Utility.h* header.

To use *WaitUntilFile* to debug a component's startup process, simply modify the *main* program of the component, adding a call to the *WaitUntilFile* at the very beginning of the method. Then rebuild and run VisIt. Once the desired component is in the *wait* state, attach the debugger, and set a breakpoint. Then create the file that was passed as the argument to *WaitUntilFile*.

Don't forget the wait file will need to be deleted in between subsequent debugging sessions.

See the table below for components, the files containing their *main* method, and the name of *main* method.

component	file containing main	main method name
gui	src/gui/main.C	GUIMain
viewer	src/viewer/main/viewer.C	ViewerMain
engine	src/engine/main/main.C	EngineMain
cli	src/visitpy/cli/cli.C	main

An example of modifying GUIMain with WaitUntilFile

```
 // Example only, the code block is incomplete.
#include <Utility.h>

int
GUIMain(int argc, char **argv)
{
    WaitUntilFile("~/guiwait.txt");

    int retval = 0;

    TRY
    {
```

(continues on next page)

(continued from previous page)

```
        // Initialize error logging.
        VisItInit::SetComponentName("gui");
```

10.21.5 Debugging a regression failure outside of the test suite

Sometimes the testing harness infrastructure gets in the way of debugging a failing regression test, and you just want to run the testing script or a portion of the script directly with VisIt's cli. Here's a quick way to do just that.

First, you need a script that mimics some of the testing harness functions, so you don't need to modify the actual testing script as much. Here's an example of what is needed:

TestingStuff.py

```
# script to aid in debugging regression tests outside of the testing harness
# it mimics some of the testing methods so that actual test scripts don't
# need to be modified so much

# use this script by adding 'Source("TestingStuff.py")' to the top of a
# regression test.  Use full path if the regression test doesn't live at
# the same location as this script.



# mimic testing 'data_path' by specifying a location where the testdata
# can be found.  It is best if this points to an actual build/testdata dir
# so that you are using the same data as the regression tests
def data_path(fname):
    return "/my/path/to/VisIts/testdata/%s"%fname

def silo_data_path(fname):
    return data_path("silo_hdf5_test_data/%s"%fname)

def TurnOnAllAnnotations(givenAtts=0):
    """
    Turns on all annotations.

    Either from the default instance of AnnotationAttributes,
    or using 'givenAtts'.
    """
    if (givenAtts == 0):
        a = AnnotationAttributes()
    else:
        a = givenAtts
    a.axes2D.visible = 1
    a.axes3D.visible = 1
    a.axes3D.triadFlag = 1
    a.axes3D.bboxFlag = 1
    a.userInfoFlag = 0
    a.databaseInfoFlag = 1
    a.legendInfoFlag = 1
    SetAnnotationAttributes(a)

def TurnOffAllAnnotations(givenAtts=0):
    """
    Turns off all annotations.
```

(continues on next page)

(continued from previous page)

```
    Either from the default instance of AnnotationAttributes,
    or using 'givenAtts'.
    """
    if (givenAtts == 0):
        a = AnnotationAttributes()
    else:
        a = givenAtts
    a.axes2D.visible = 0
    a.axes3D.visible = 0
    a.axes3D.triadFlag = 0
    a.axes3D.bboxFlag = 0
    a.userInfoFlag = 0
    a.databaseInfoFlag = 0
    a.legendInfoFlag = 0
    SetAnnotationAttributes(a)

def Test(fname):
    swa = SaveWindowAttributes()
    swa.family = 0
    swa.fileName = fname
    swa.screenCapture = 0
    SetSaveWindowAttributes(swa)
    SaveWindow()

def Test(fname, swa = 0, alreadySaved=0):
    if (swa != 0):
        sa = swa
    else:
        sa = SaveWindowAttributes()
    sa.screenCapture = 1
    sa.family = 0
    sa.fileName = fname
    SetSaveWindowAttributes(sa)
    SaveWindow()

def TestText(name, results):
    print("%s: %s"%(name, results))

def TestSection(stuff):
    print(stuff)

def Exit():
    exit()
```

Now, you can copy a regression test to the same directory as this script, add `Source("TestingStuff.py")` to the top of the regression test, and run `visit -cli -s testname.py`, along with any debugging options you desire.

10.22 Creating a build_visit Module

To create the skeleton for a build_visit module, run `construct_build_visit_module.sh <module>`, where `<module>` is the name of the library you wish to build. The script is located in `src/tools/dev/scripts/bv_support`. For purposes of the rest of this section, `<module>` will be named `foo`.

Running `construct_build_visit_module.sh foo` will create the file: `bv_foo.sh` containing most of the code needed, and only a few functions will require modification.

There are three functions in the file that absolutely need to be fleshed out further: `bv_foo_info`, `bv_foo_host_profile`, and `build_foo`. Other build_visit modules may be helpful as examples for filling these out. If the package being built has a decent CMake build system, please use that in the *build_foo* function.

Here is a list of all the functions that will be defined (listed in order of appearance in the file):

bv_foo_initialize Initialize any variables controlled by command line options.

bv_foo_enable Enables the module (sets *DO_FOO* to *yes*).

bv_foo_disable Disables the module (sets *DO_FOO* to *no*).

bv_foo_depends_on What other modules does *foo* depend on. For example, the *osmesa* module returns *llvm*.

bv_foo_info Name of the *foo* tarball, the version, etc.

bv_foo_print Prints info about the *foo* module.

bv_foo_print_usage Prints how to enable the module and any other relevant command line args.

bv_foo_host_profile Adds the necessary information about the *foo* module to config-site host profile cmake file. Most often this is just the insall location, but could also be version information.

bv_foo_initialize_vars Set vars possibly needed by other modules, such as install location for the module (can use EXPORT). This is an optional function and can be removed if not needed.

bv_foo_ensure Ensure the module has been downloaded and extracted properly.

build_foo Where all the steps for building and installing the module reside.

bv_foo_is_enabled Returns true if the module is enabled, and false otherwise

bv_foo_is_installed Returns true if the module is installed, and false otherwise

bv_foo_build Checks if *foo* is already installed, and calls *build_foo* if not.

Most of the above referenced functions will suit fine as originally written by the construction script.

If your module has dependencies on other build_visit modules, then also modify `bv_foo_depends_on`.

If you want to allow use of a system version of your module, then `bv_foo_initialize` needs work to ensure extra command line arguments are added. See `bv_qt.sh` as an example of allowing system or also an alternate (already installed but not system) qt. `bv_qt_system_qt` and `bv_alt_qt_dir` were added and other functions were modified to support this for qt.

Once `bv_foo.sh` has been updated appropriately, add the module name to `bv_support/modules.xml` under the appropriate categories. Then run the build_visit script to ensure that your module builds and installs correctly, and that the host profile entry is correct.

Order of execution of the functions:

1. bv_foo_initialize
2. bv_foo_info
3. bv_foo_[enable|disable]
4. bv_foo_is_enabled
5. bv_foo_initialize_vars (if defined)
6. bv_foo_ensure
7. bv_foo_is_enabled

8. bv_foo_is_installed
9. bv_foo_depends_on
10. bv_foo_build
11. bv_foo_host_profile

10.23 Adding a Find Module for Third-Party Libraries

VisIt uses custom *Find* modules for most of its dependent third-party libraries. The *Find* modules live in `src/CMake` and most of them utilize special functions that live in `src/CMake/SetUpThirdParty.cmake`. The most important of these is `SET_UP_THIRD_PARTY`, as it does the bulk of the work in determining platform-specific extensions, handles `.so` versioning and Windows dll's and import libraries.

The first argument to the function is the name of the package (eg `FOO`), and it expects a capitalized package name that corresponds to a `VISIT_<PKG>_DIR` entry in VisIt's config-site files. This function also uses these keyword arguments:

LIBS: required The name(s) of the library(ies) without platform-specific prefixes, suffixes or extensions. For example: `SETUP_THIRD_PARTY(FOO LIBS foo1 foo2)`, not `SETUP_THIRD_PARTY(FOO LIBS libfoo1.so libfoo2.so)`.

LIBDIR: optional The paths beyond `VISIT_<PKG>_DIR` where the libraries may be found. `VISIT_<PKG>_DIR/lib` or `VISIT_<PKG>_DIR>/lib64` will be assumed if this is not provided.

INCDIR: optional The path beyond `VISIT_<PKG>_DIR` where the headers may be found. `VISIT_<PKG>_DIR/include` will be assumed if this is not provided.

There may be situations where this simple solution is not sufficient: a package may have already created its own *Find* module and duplication of effort isn't desired; or the package may be complex with many components all of which may not necessarily be needed by VisIt (e.g. VTK). For situations where this function is not sufficient, standard CMake find calls (*find_path*, *find_library*, *find_package*) may be used along with the necessary special `INSTALL` commands for ensuring the package's libraries (and possibly headers) are installed alongside VisIt.

The `INSTALL` commands that VisIt uses are custom functions:

THIRD_PARTY_INSTALL_LIBRARY(LIBFILE): Installs the library file.

LIBFILE is the full path to a library file.

THIRD_PARTY_INSTALL_INCLUDE(pkg incdir) Installs pkg's headers.

pkg is the name of the package.

incdir1 is the full path to the headers (root of include tree if it involves multiple subdirectories).

CHAPTER

ELEVEN

SEARCH SYNTAX

Searching here uses simple query string syntax which supports the following operators:

- `+` signifies AND operation
- `|` signifies OR operation
- `-` negates a single token
- `"` wraps a number of tokens to signify a phrase for searching
- `*` at the end of a term signifies a prefix query
- `(` and `)` signify precedence
- `~N` after a word signifies edit distance (fuzziness)
- `~N` after a phrase signifies slop amount
- To use any of the above characters literally, escape it with a preceding backslash (`\`).
- A space between search terms implies the *default* operator of OR.
- When upper case is used, the search is case-sensitive. Otherwise it is case-insenstive.

11.1 Examples

Searching [`annot*`] returns pages with `Annotation`, `Annotations`, `annotate`, `annotated`, etc.

Searching [`Annot*`] is case-sensitive and returns pages with `Annotations`, `Annotation` but not `annotate`.

Searching [`annot* +object`] returns pages with `Annotation`, `Annotations`, `Annotated` AND `object`.

Searching [`getannotationobject\(\)`] returns pages with `GetAnnotationObject()`

Searching [`annot* | object`] returns pages with `Annotation`, `Annotations`, `Annotated` and also returns pages with `object`.

Searching [`load~4`] returns pages including `load`, `lead`, `head`, `goal`

Searching [`"load balance"`] returns pages with the whole quoted phrase as opposed to pages that contain both `load` AND `balance` somewhere on the page.

Searching [`load -balance`] returns pages with contain `load` AND **not** do not also contain `balance`.

Searching [`foo bar -baz`] returns pages containing `foo` or `bar` as well as any pages that do not contain `baz` (which probably *expands* the results well beyond those containing just `foo` or `bar`). This is probably not the intention. This is because the default operator (implied by spaces) is OR. To return documents that contain `foo` or `bar` but do not contain `baz`, the search string would be `foo bar +-baz` or `(foo bar) +-baz`.

CHAPTER

TWELVE

ACKNOWLEDGMENTS

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

CHAPTER

THIRTEEN

GLOSSARY

AAN

Always, Auto, Never Various features in VisIt support an **Always, Auto, Never** choice. A setting of **Never** means to never enable the the feature and a setting of **Always** means to always enable the feature. A setting of **Auto**, which is typically the default, means the allow VisIt to decide when it thinks it is best to enable or disable the feature.

Integral Curve An integral curve is a curve that begins at a seed location and is tangent at every point in a vector field. It is computed by numerical integration of the seed location through the vector field.

Node

Point

Vertex These terms refer to the *corners* or *ends* of mesh elements.

Pathlines A path rendered by an integrator that uses the vector field that is *in-step* with the integrator, so that as the integrator steps through time, it uses data from the vector field at each new time step.

Node-centered

Point-centered These terms refer to a piecewise-linear (one degree of freedom at each of mesh element *corner*) interpolation scheme used to define a variable on a mesh. VTK tends to use the *point* terminology whereas VisIt tends to use the *node* terminology.

Parallel task Although developers are working to enhance VisIt to support a variety of fine-grained parallelism methods (e.g. MC or GPU) and although some portions of VisIt have supported multi-threaded processing for several years, in the currently available implementations, a parallel task is an MPI (Message Passing Interface) rank.

Streamlines A path rendered by an integrator that uses the *same* vector field for the entire integration.

SIL

Subset Inclusion Lattice A **Subset Inclusion Lattice** or **SIL** is a term used to describe the often complex, graph like relationships among a variety of subsets defined for a mesh. A **SIL** describes which subsets and categories of subsets are contained within other subsets and subset categories. The **Subset Window** is the part of VisIt GUI that displays the contents of a **SIL** and allows the user to browse subsets and subset categories and turn subsets (and trees of subsets) on and off in visualizations.

SR

SR mode SR is an abbreviation for **Scalable Rendering**. This is a mode of operation where the VisIt **engine** performs scalable, parallel rendering and ships the final rendered image (e.g. pixels) to the **viewer**. This is in contrast to *standard* mode where the **engine** ships polygons to the **viewer** to be rendered there.

Zone

Cell These terms refer to the the individual computational elements comprising a mesh.

Zone-centered

Cell-centered These terms refer to a piecewise-constant (single degree of freedom for an entire zone) interpolation scheme used to define a field variable on a mesh. VTK tends to use the *cell* terminology whereas VisIt tends to use the *zone* terminology.

INDEX

A

AAN, **1395**
Always, Auto, Never, **1395**

C

Cell, **1395**
Cell-centered, **1396**

I

Integral Curve, **1395**

N

Node, **1395**
Node-centered, **1395**

P

Parallel task, **1395**
Pathlines, **1395**
Point, **1395**
Point-centered, **1395**

S

SIL, **1395**
SR, **1395**
SR mode, **1395**
Streamlines, **1395**
Subset Inclusion Lattice, **1395**

V

Vertex, **1395**
visit_utils.encoding.encode() (*built-in function*), 838
visit_utils.encoding.encoders() (*built-in function*), 839
visit_utils.encoding.extract() (*built-in function*), 839
visit_utils.engine.close() (*built-in function*), 840
visit_utils.engine.open() (*built-in function*), 839
visit_utils.engine.supported_hosts() (*built-in function*), 840

Z

Zone, **1395**
Zone-centered, **1396**