VisIt User Manual Documentation

Release 2.11

LLNL

Contents

VisIt GUI User Manual

Contents:

1.1 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 OSX 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://wci.llnl.gov/simulation/computer-codes/visit.

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.

1.1.1 Understanding how VisIt works

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.

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 1.1. 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.
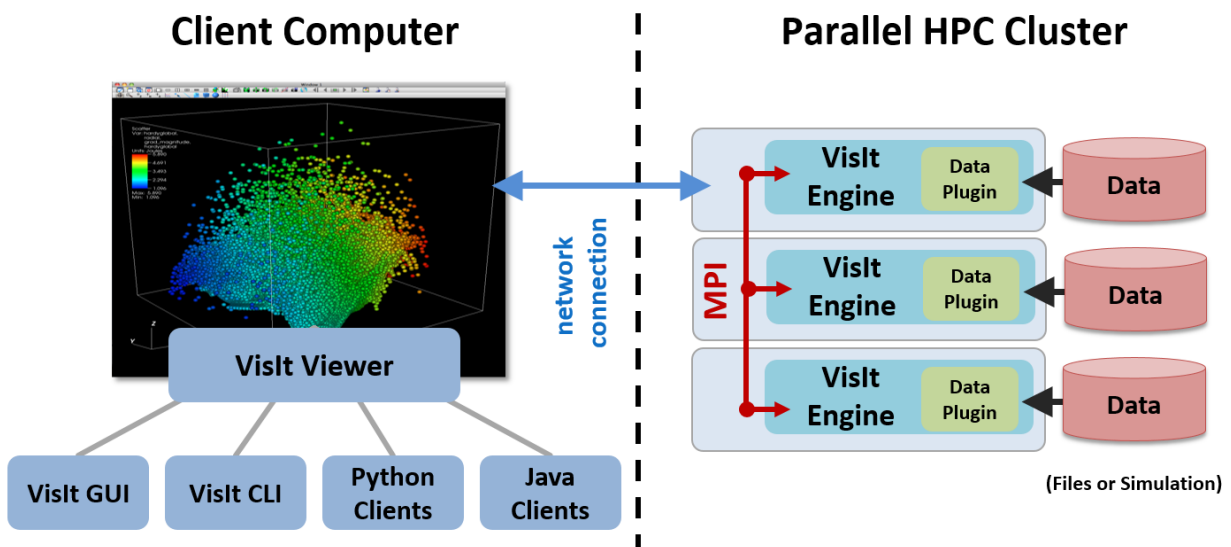

Fig. 1.1: VisIt's architecture

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 1.2. 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.
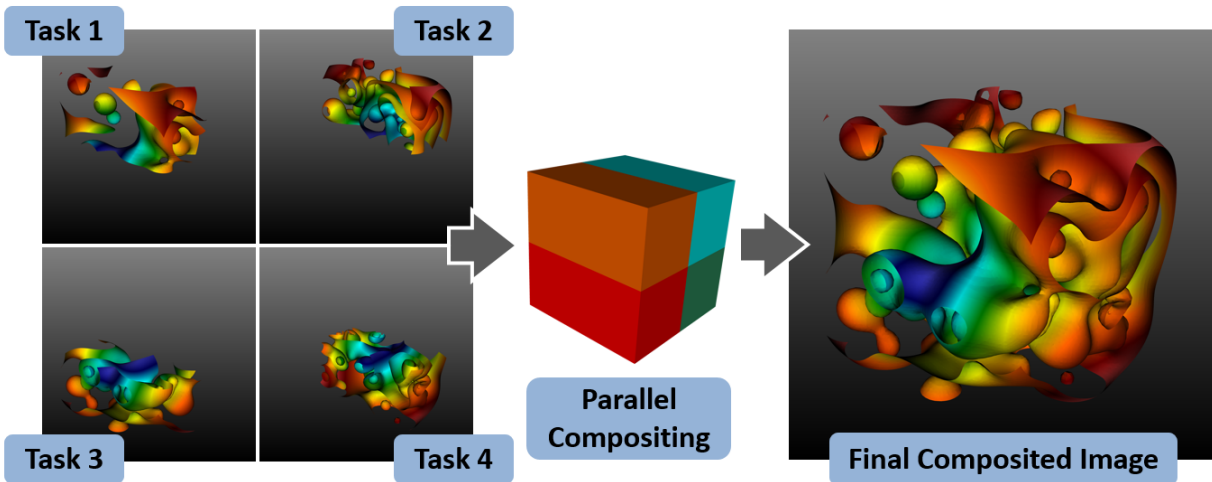
Fig. 1.2: VisIt's scalable rendering

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 1.3.

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.

1.1.2 Installing and Starting VisIt

VisIt runs on the following platforms:

- Linux (including Ubuntu, RedHat, SUSE, TOSS)
- Mac OSX
- Microsoft Windows

A new version of VisIt is usually released every 2-3 months, you can find VisIt release executables at: https://wci.llnl.gov/simulation/computer-codes/visit/executables.

Download a binary release compatible with the machine you want to install VisIt on. If you are installing VisIt on Linux, also download the `visit-install` script.
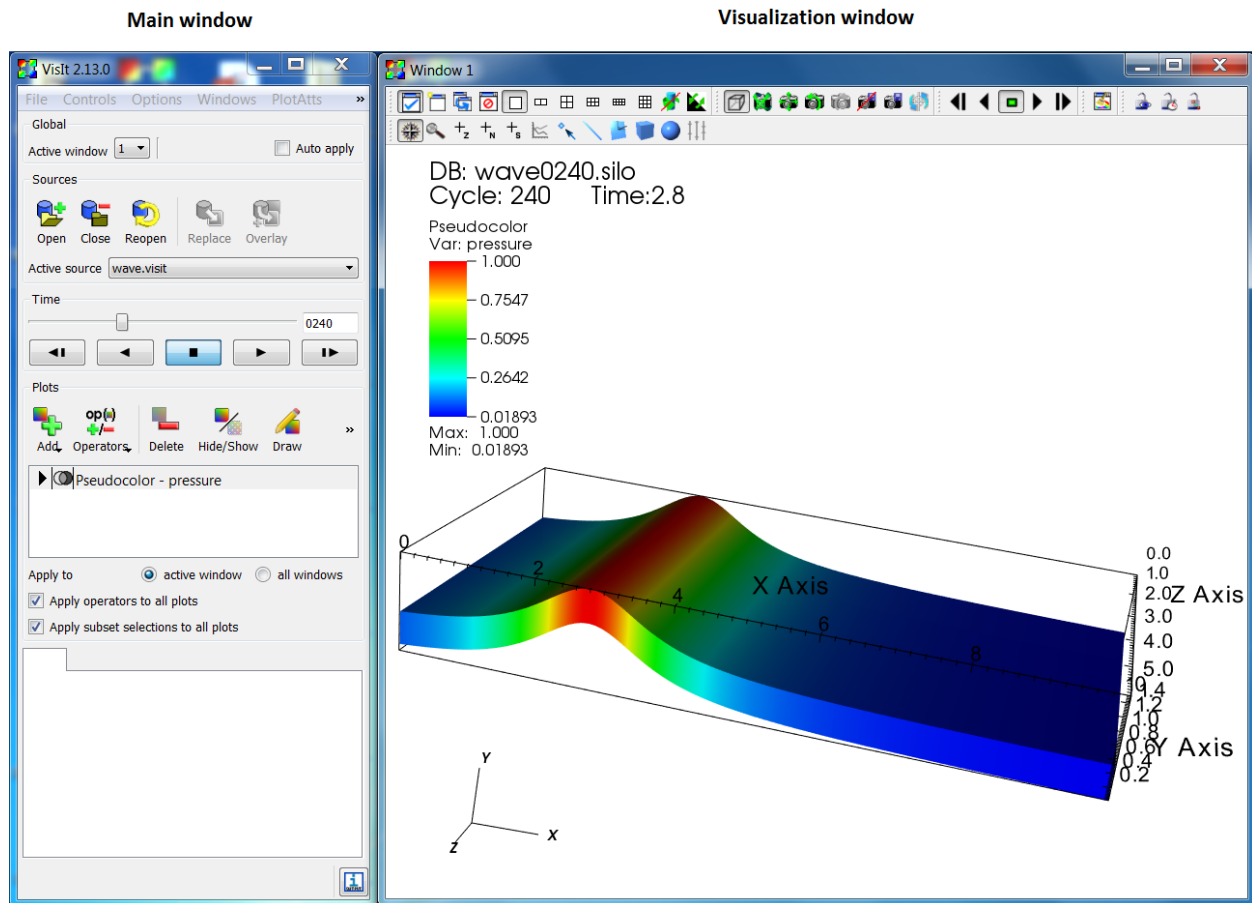
Main window

Visualization window

Fig. 1.3: VisIt's graphical user interface

Installing on Mac OSX

VisIt releases include an app-bundle for Mac OSX 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. To run VisIt double click on the VisIt app-bundle. The `visit-install` script can also be used to install tarball packaged OSX binaries. For this case follow the Linux installation instructions.

Installing on Linux

Installing VisIt on Linux (and optionally on Mac OSX) 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 3.0.0, use:

```
visit-install 3.0.0 linux-x86_64 /usr/local/visit
```

This command will install the 3.0.0 version of VisIt into the `/usr/local/visit` directory. Note that when you enter the above command, the file `visit3_0_0.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.

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.

Startup Options

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

1.1.3 The Main Window

VisIt's **Main** window, shown in Figure 1.4, 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.
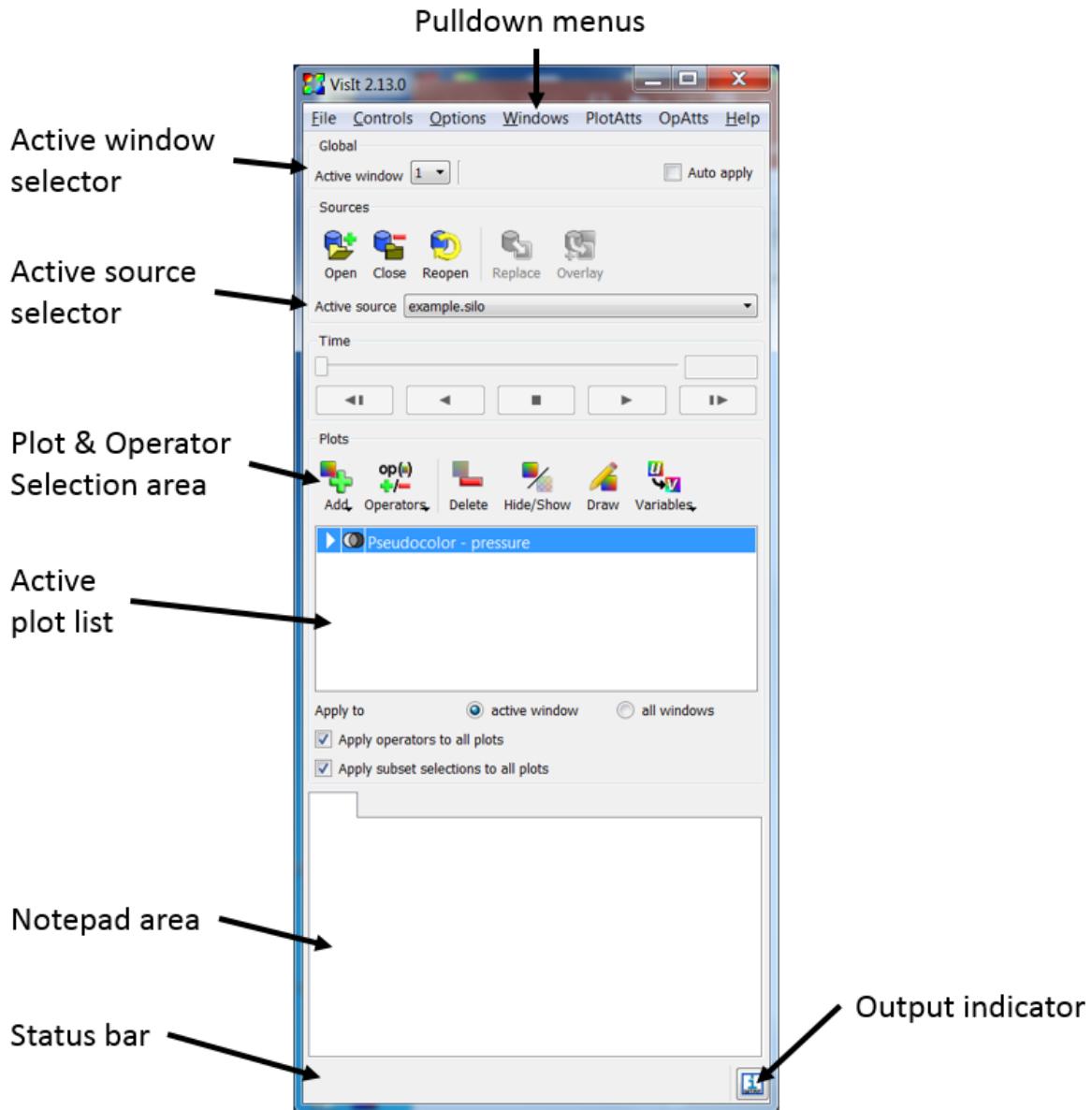

Fig. 1.4: VisIt's Main window

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.
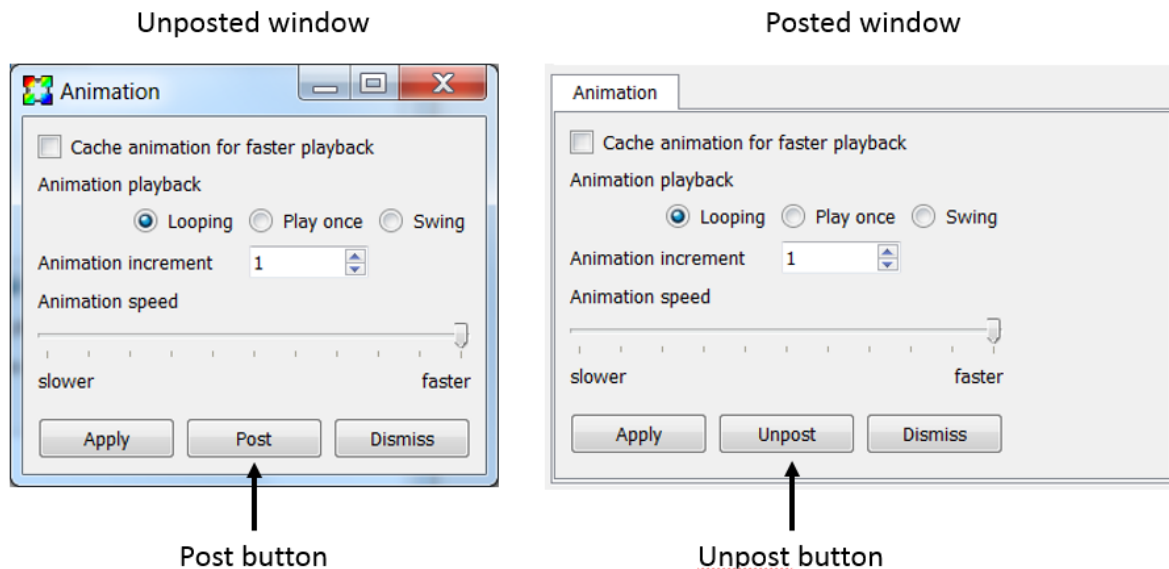

Fig. 1.5: An unposted and posted window

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 1.5 shows an example of a window with a **Post** button and also shows the same window when it is posted to the notepad area.

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 1.6 and will be described in detail later in this manual.
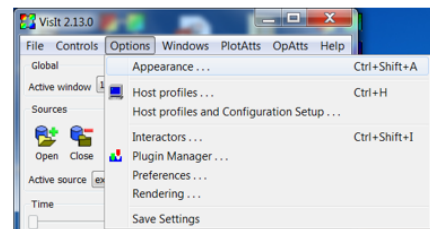
The **Main** menu and the **Plots** and **Operators** menus are merged in the OSX version of VisIt because OSX applications always have all menus in the system menu along the top of the display.
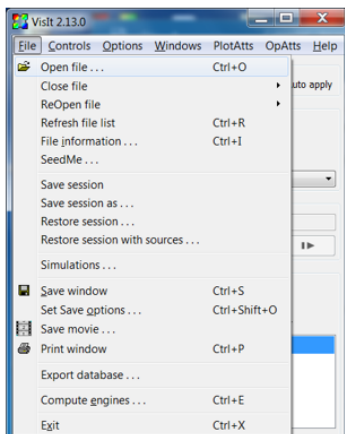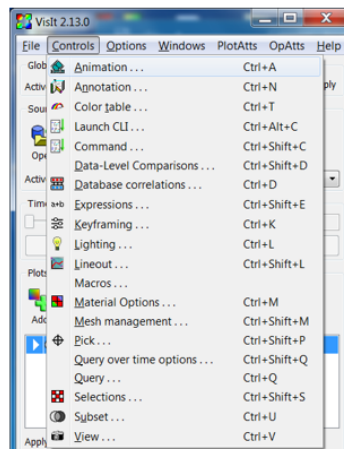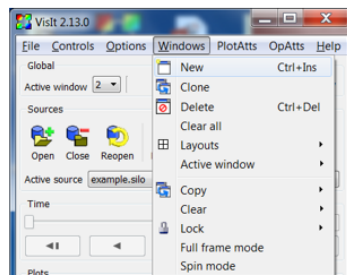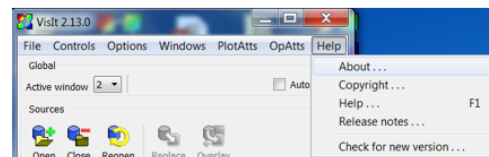
Fig. 1.6: VisIt's main menus

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 1.7. 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. 1.7: The output window and output indicator

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.
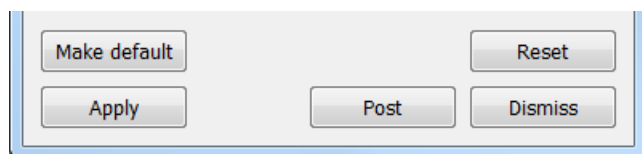
1.1.4 Getting Started

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
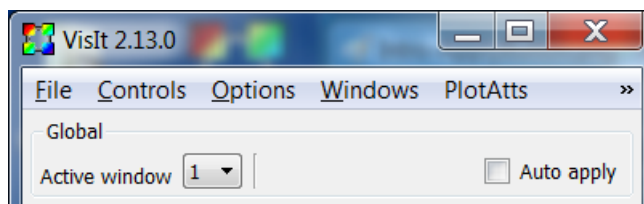
Fig. 1.8: The Apply button and Auto apply check box

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.

1.2 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 *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*.

1.2.1 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 send an e-mail inquiry to visit-users@elist.ornl.gov.

File extensions

VisIt uses file extensions to decide which database reader plugin should be used to open a particular file format. 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 suitable plugin, before trying to open the file with the default database reader plugin. If your files do not have file extensions then VisIt will attempt to use the default database reader plugin. You can provide the -default_format command line option with the name of the database reader plugin to use if you want to specify which reader VisIt should use when first trying to open a file. For example, if you want to load a PDB/Flash file, which usually has no file extension, you could provide: -default_format PDB on the command line.

1.2.2 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 1.9), 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 1.10.
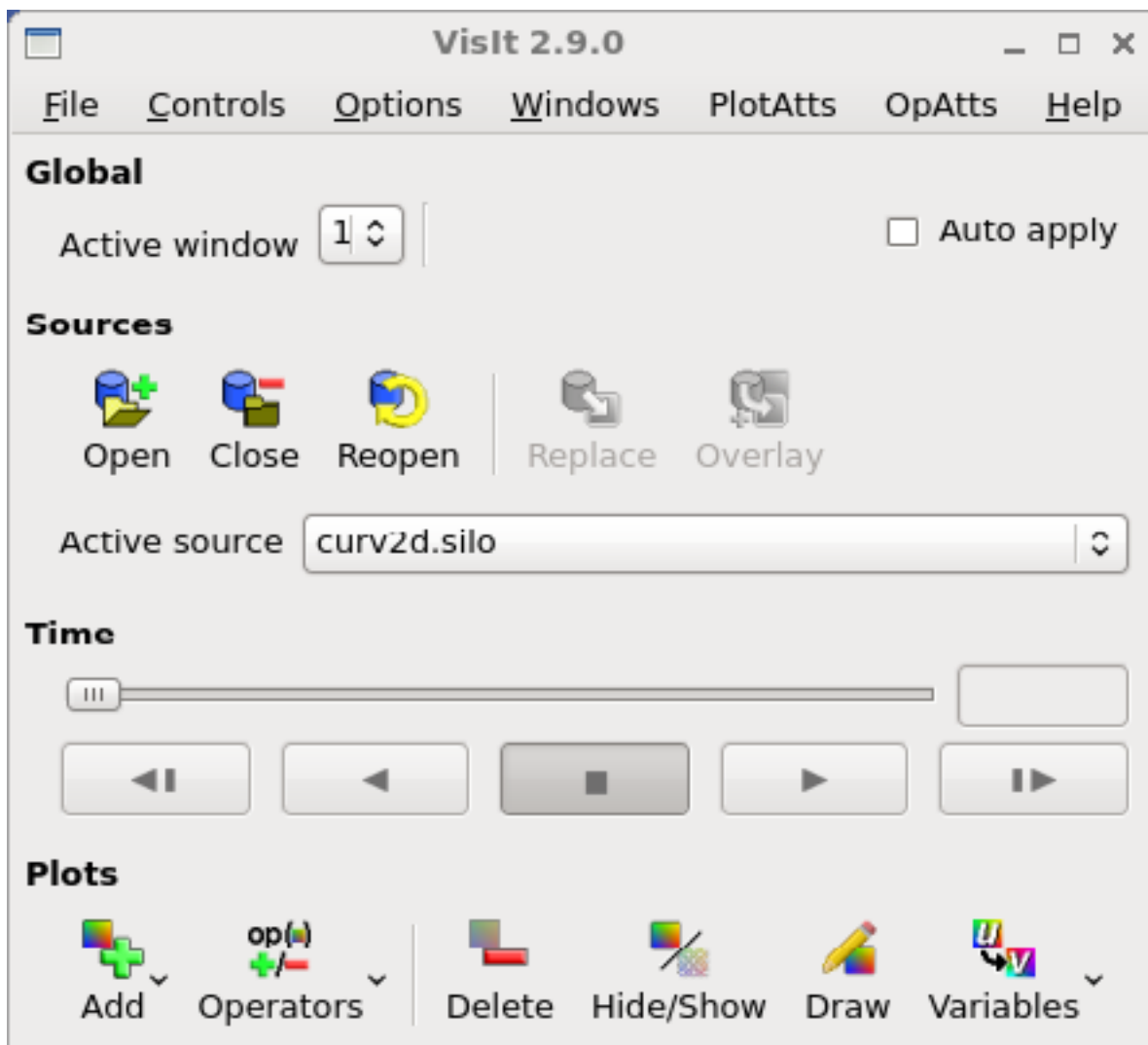
Fig. 1.9: Main gui panel showing Sources section

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.
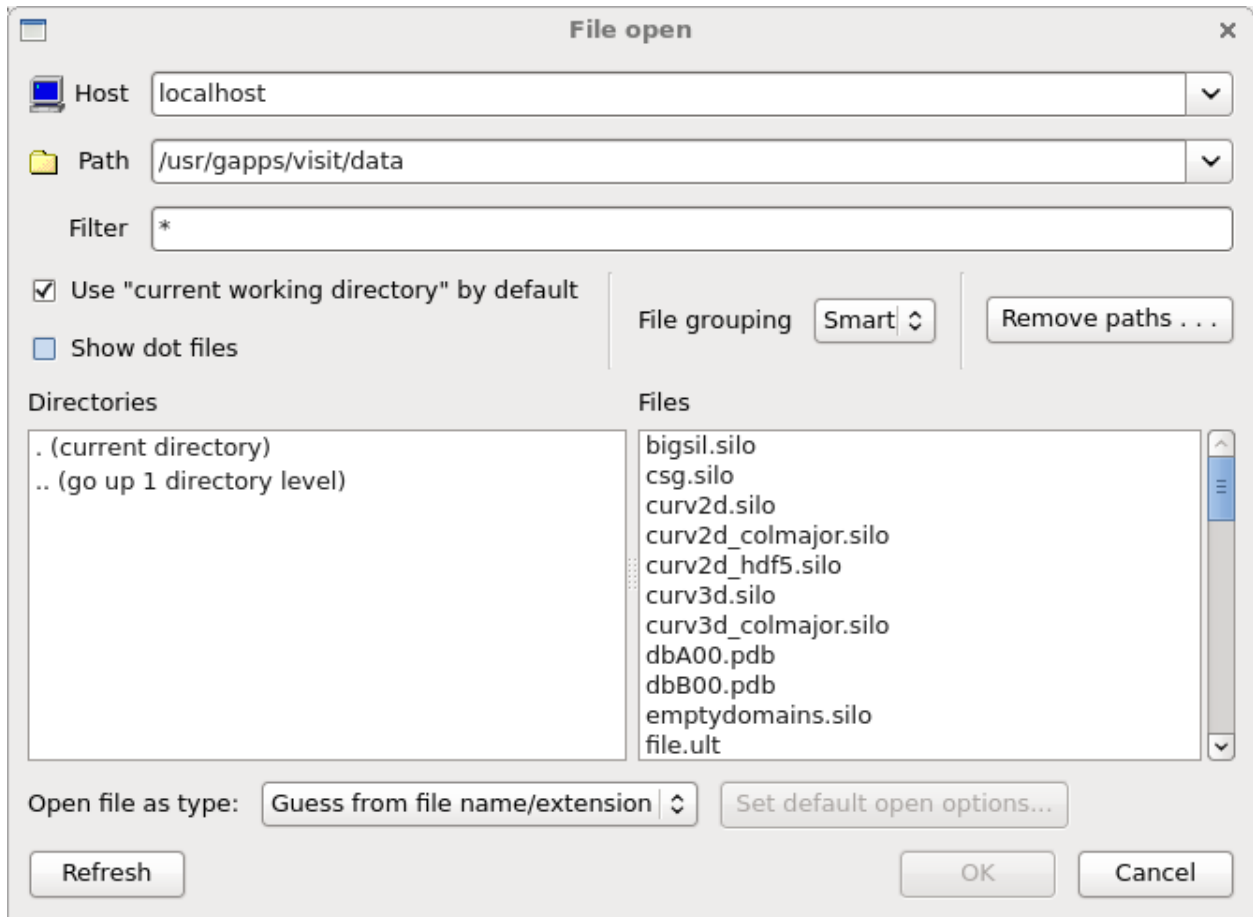
Fig. 1.10: File Open Window

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 1.11) 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 1.12.
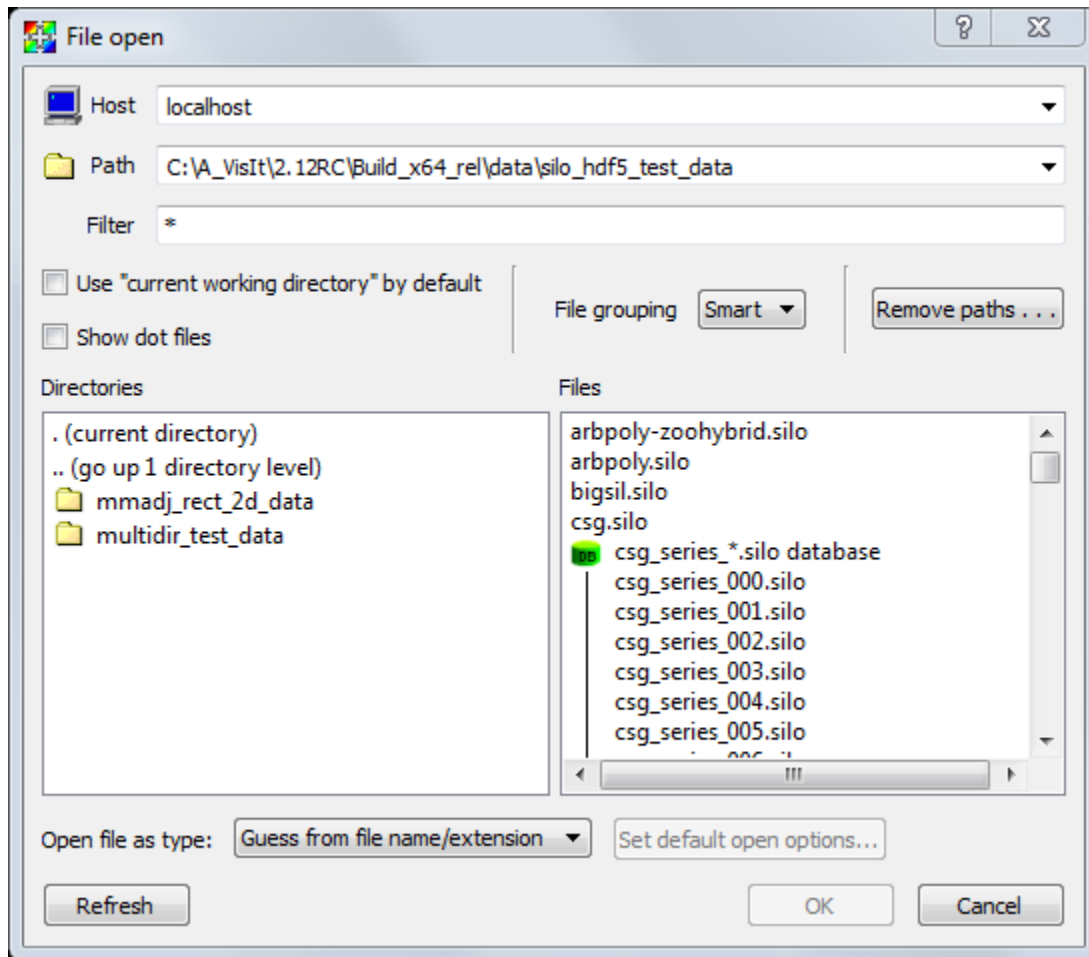
Fig. 1.11: File grouping turned on (Smart setting)

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.

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.
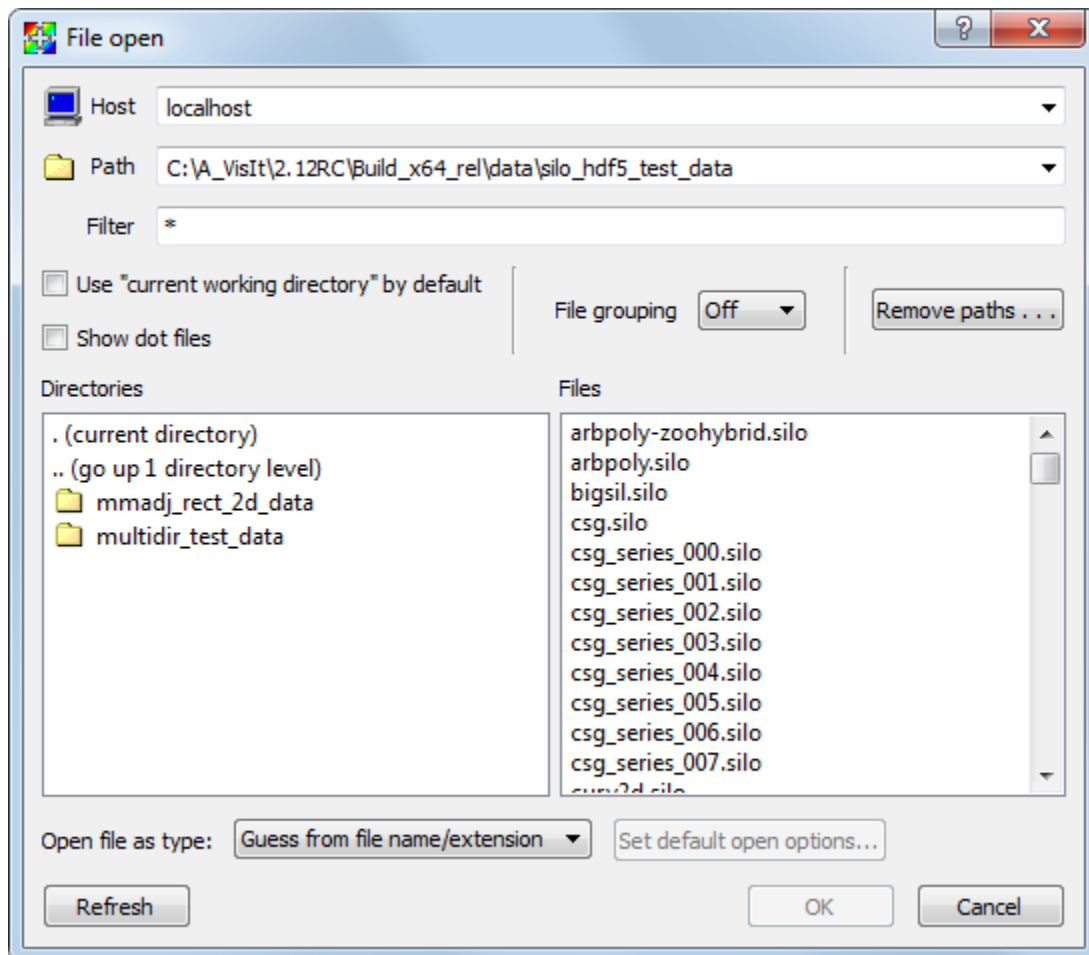
Fig. 1.12: File grouping turned off

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 1.13.
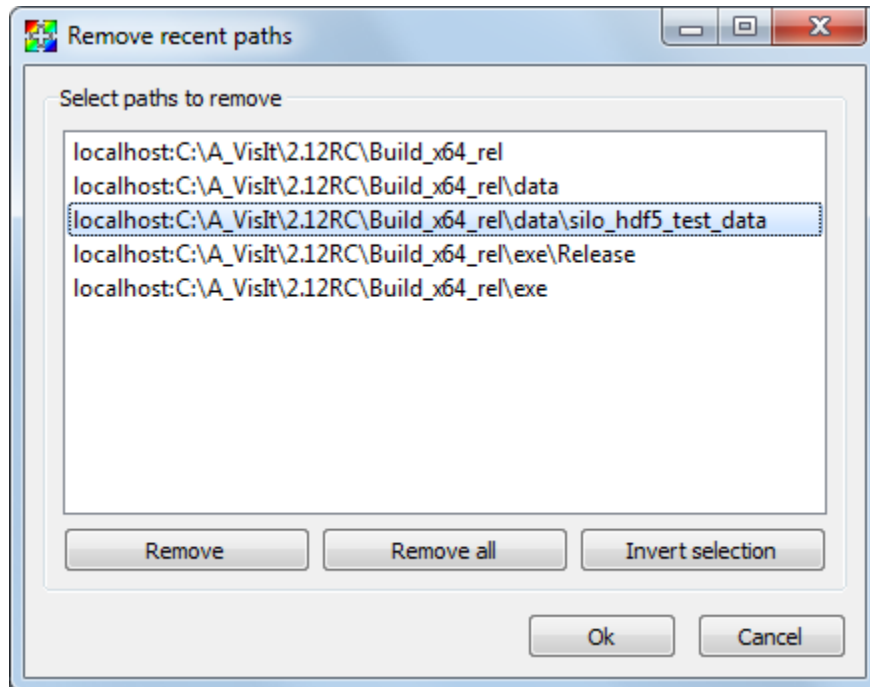


Fig. 1.13: Remove recent paths window

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 1.14). Once that is done, connecting to a simulation is the same as opening any other disk file.
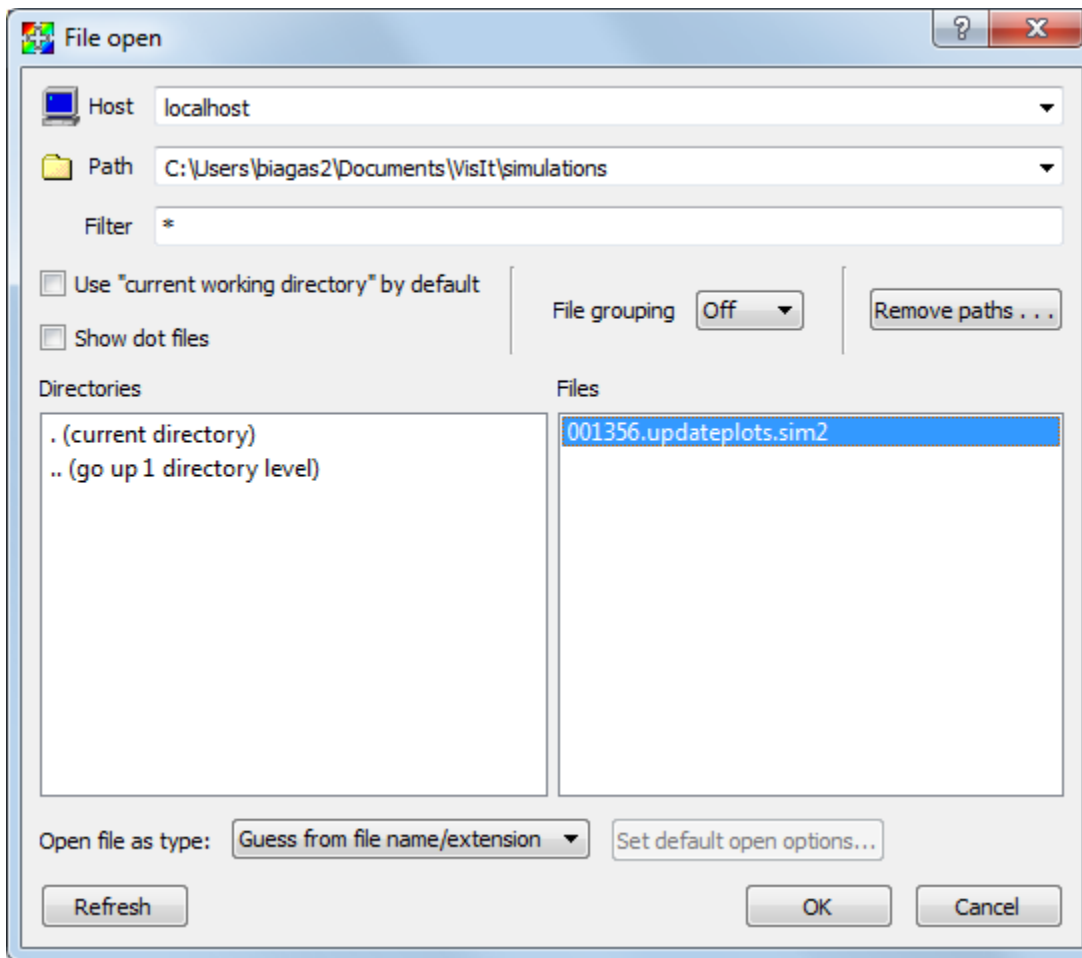
Fig. 1.14: Accessing a simulation using the File Open Window

1.2.3 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.
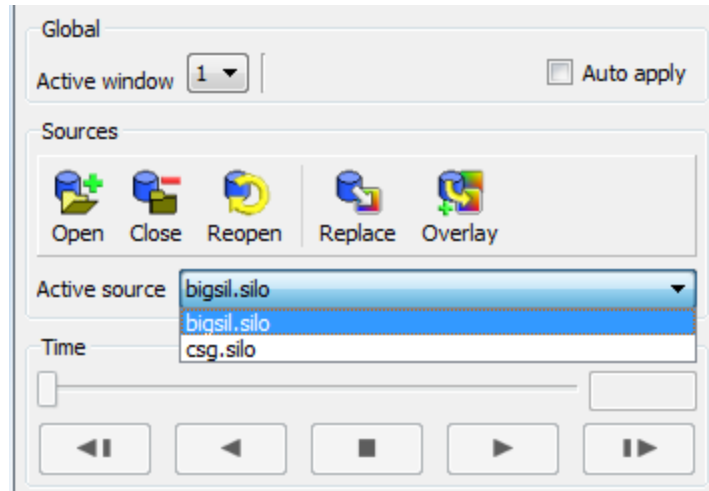
Fig. 1.15: Controls for setting the active source

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, 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.

1.2.4 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.
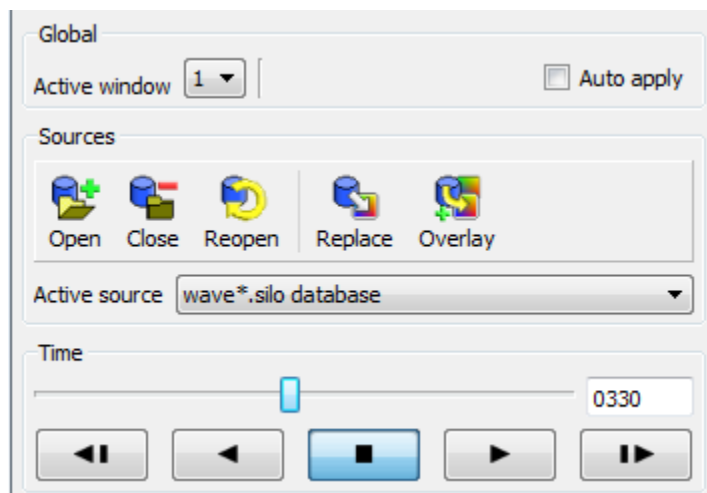
Fig. 1.16: Controls for setting the active time step

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.

1.2.5 File Information Window

This **File Information Window**, shown in Figure 1.17, 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**.
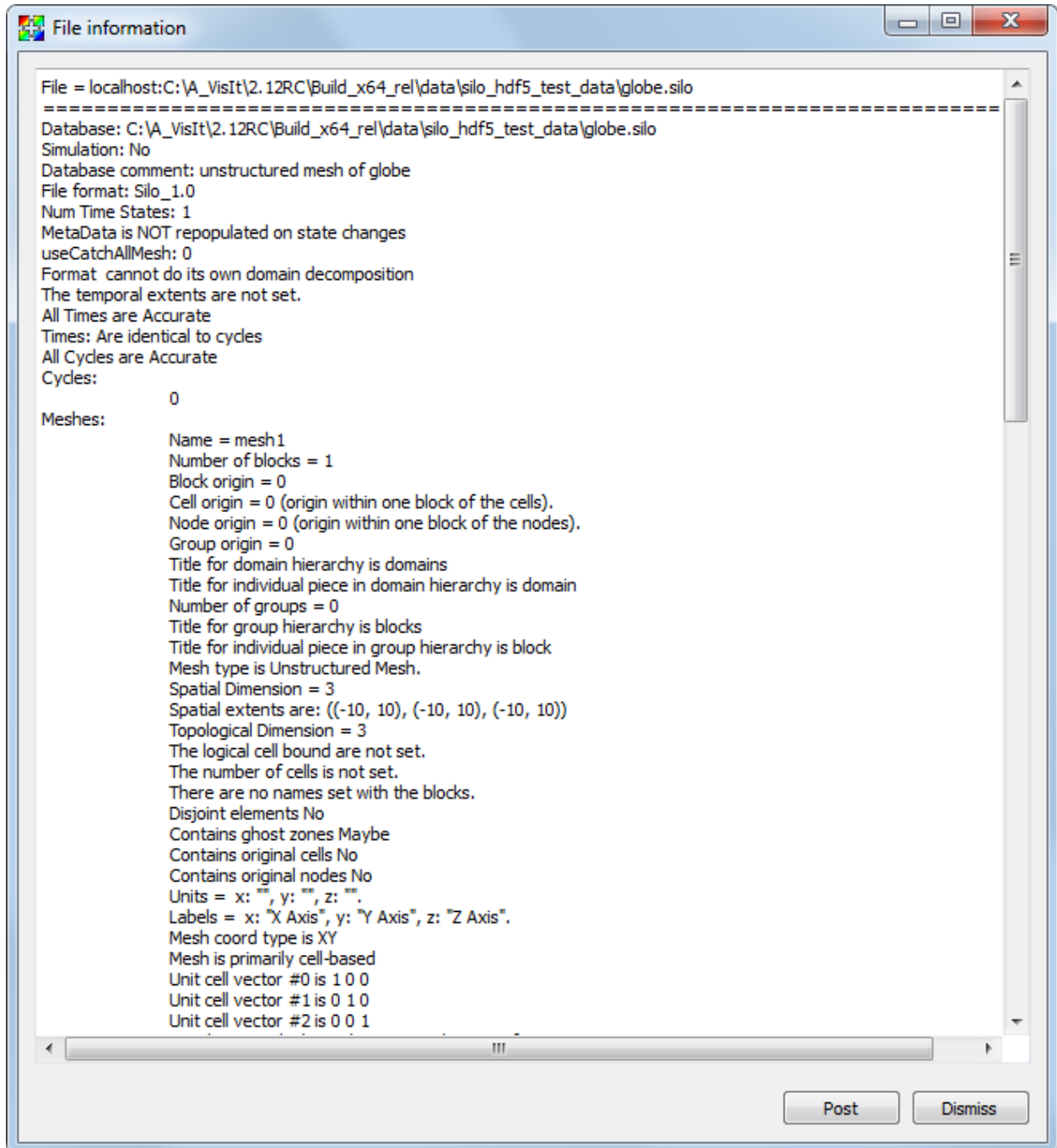
Fig. 1.17: File Information Window

1.3 Plots

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

1.3.1 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. All of VisIt's plots are plugins so you can add new plot types by writing your own plot plugins. See the wiki at visitusers.org for more details on creating new plot plugins or send an e-mail inquiry to visit-users@elist.ornl.gov.

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 1.18, contains the controls for managing plots.
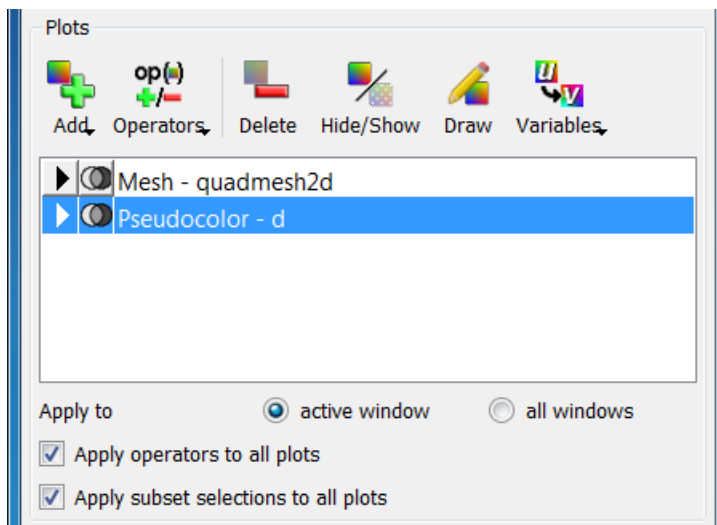
Fig. 1.18: The active plots area

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.
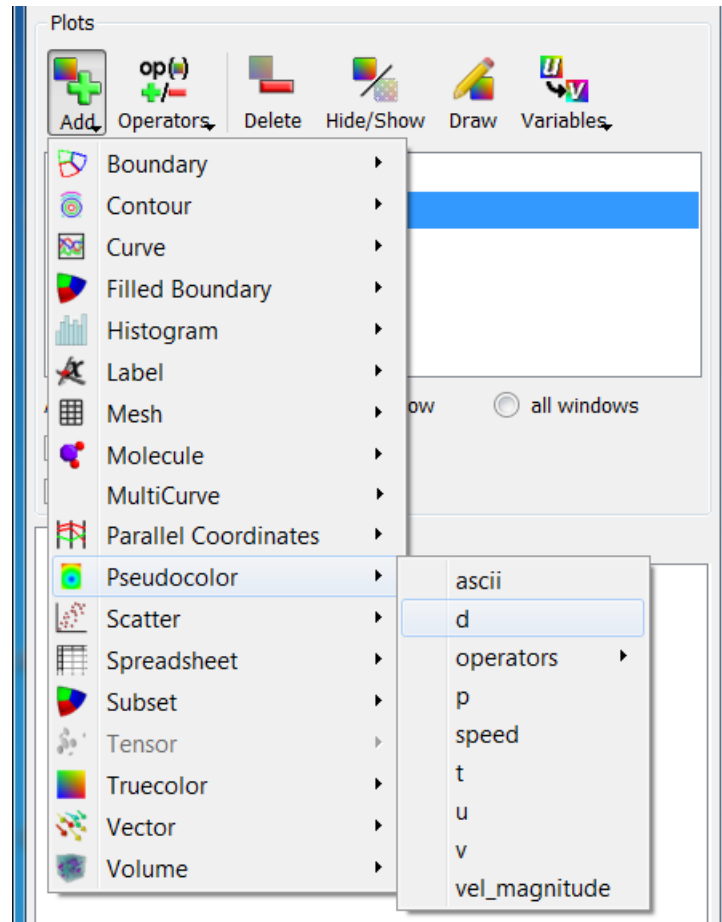
Fig. 1.19: The Add menu

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.

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 1.20,

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.

1.3.2 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.

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 style** and **Point type**, **Log** or **Linear** scaling, the **Legend** checkbox and others. These common plot attributes are described here first using the **Pseudocolor plot** as an example.

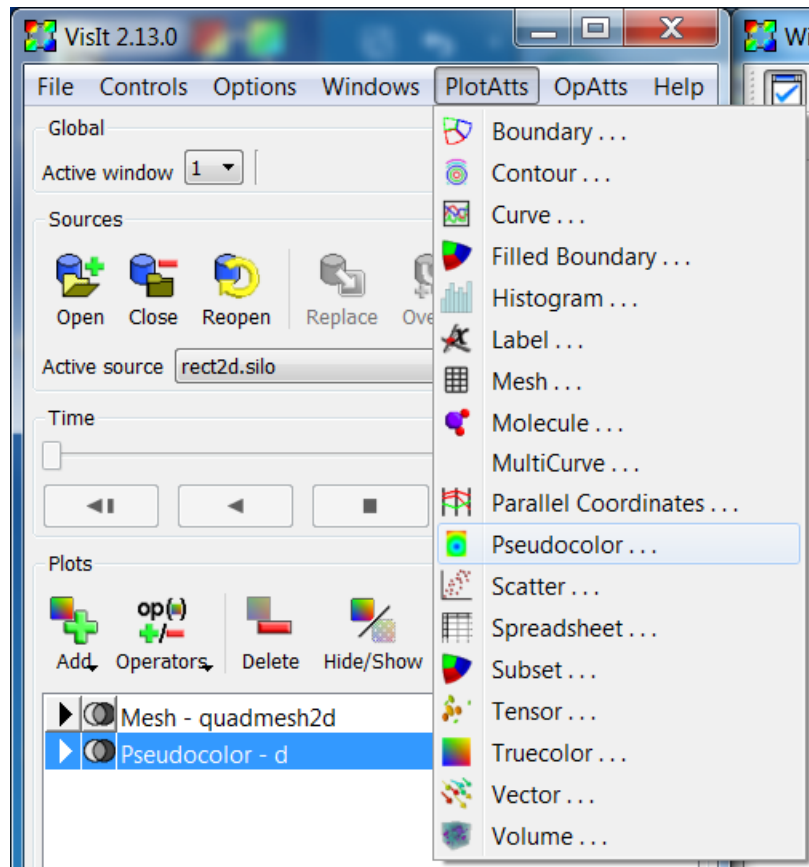Then, attributes specific to each plot type are described in the remaining sections.
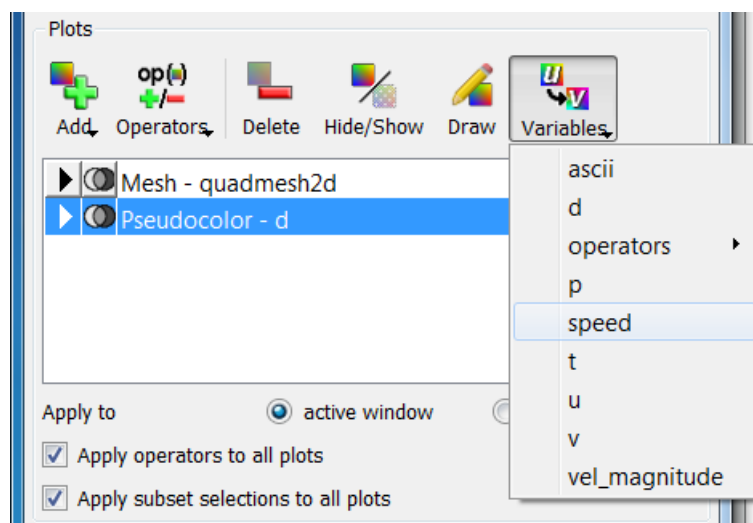
Fig. 1.20: The PlotAtts menu

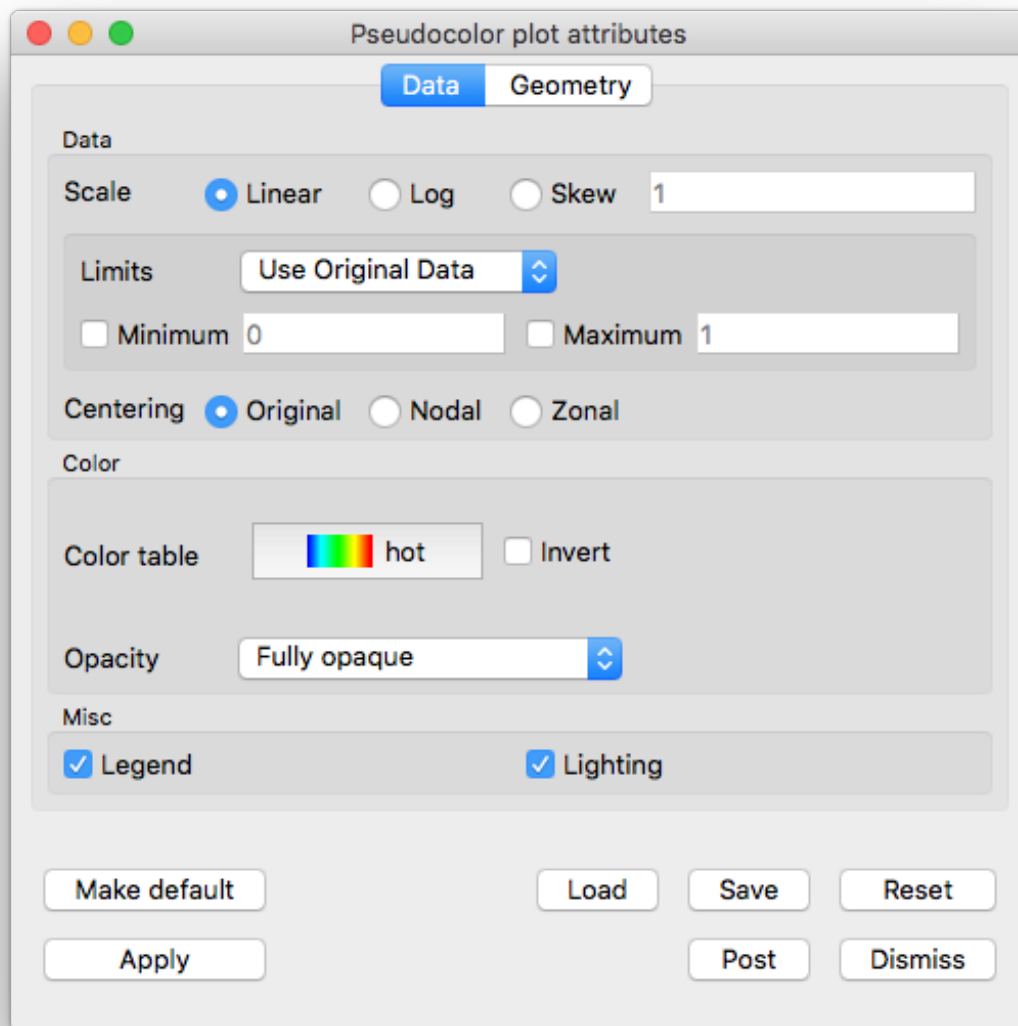

Fig. 1.21: The Variables menu

Fig. 1.22: Example of **Pseudocolor plot attribute window**

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 1.23). 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.

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 in Figure 1.25, displays the surface or lines that separate materials.

The FilledBoundary plot (see Figure 1.26) 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.
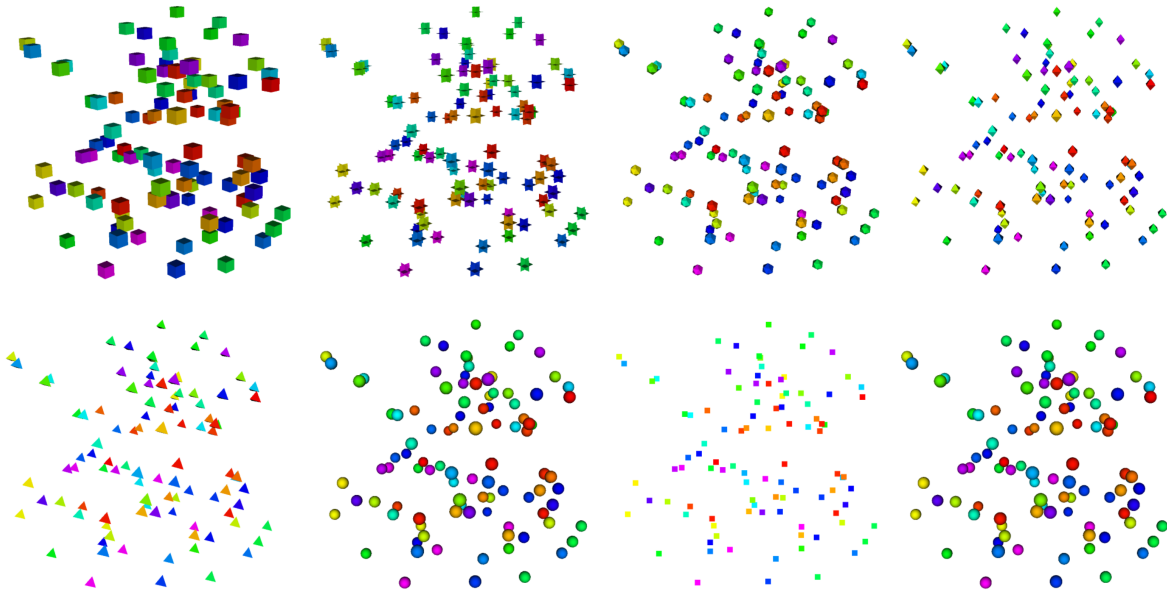
Fig. 1.23: Point types: Box, Axis, Icosahedron, Octahedron, Tetrahedron, Sphere Geometry, Point, Sphere
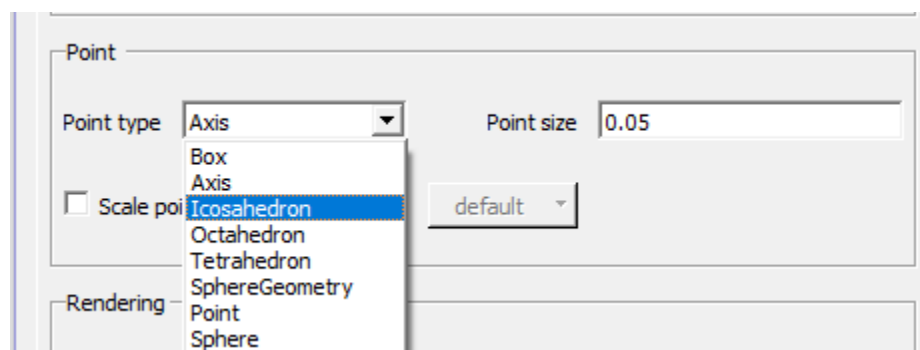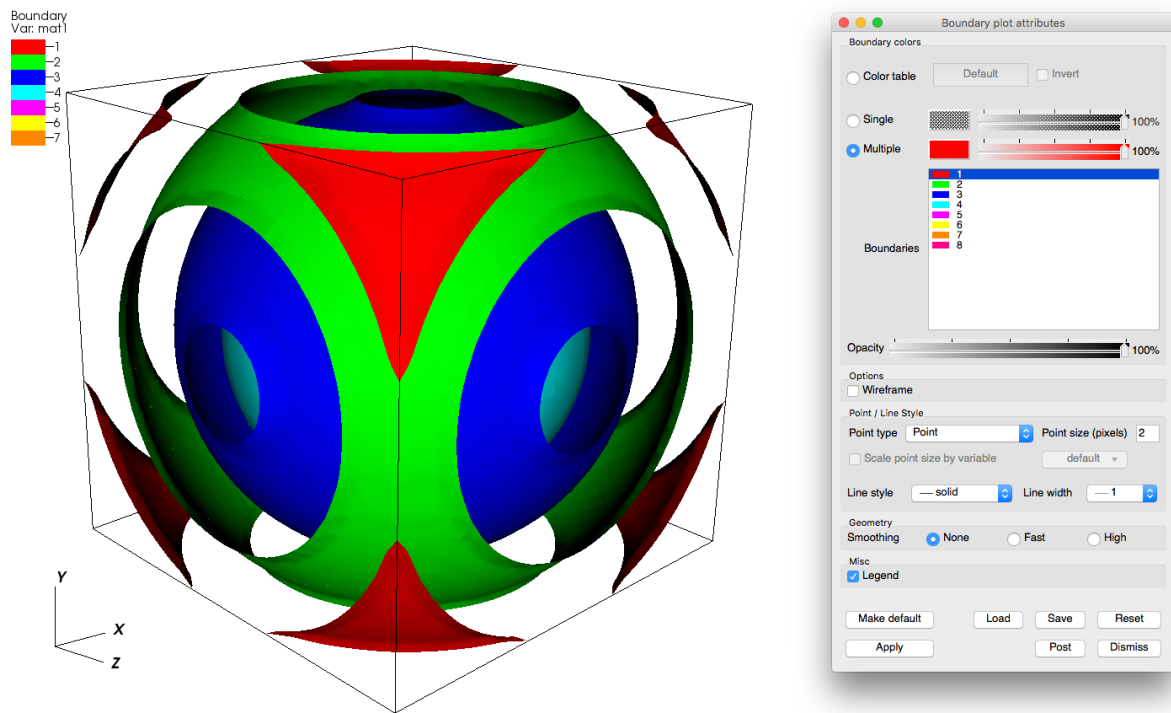

Fig. 1.24: Point type menu, expanded

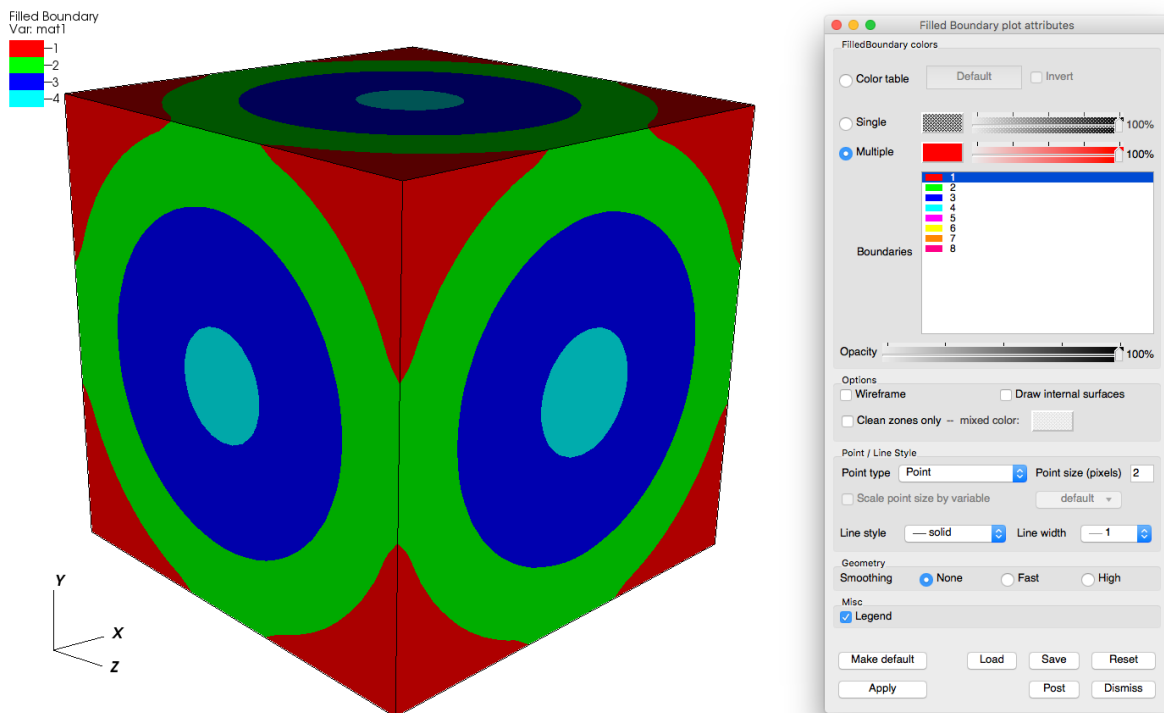Fig. 1.25: Boundary plot and its plot attributes window

Fig. 1.26: FilledBoundary plot and its plot attributes window
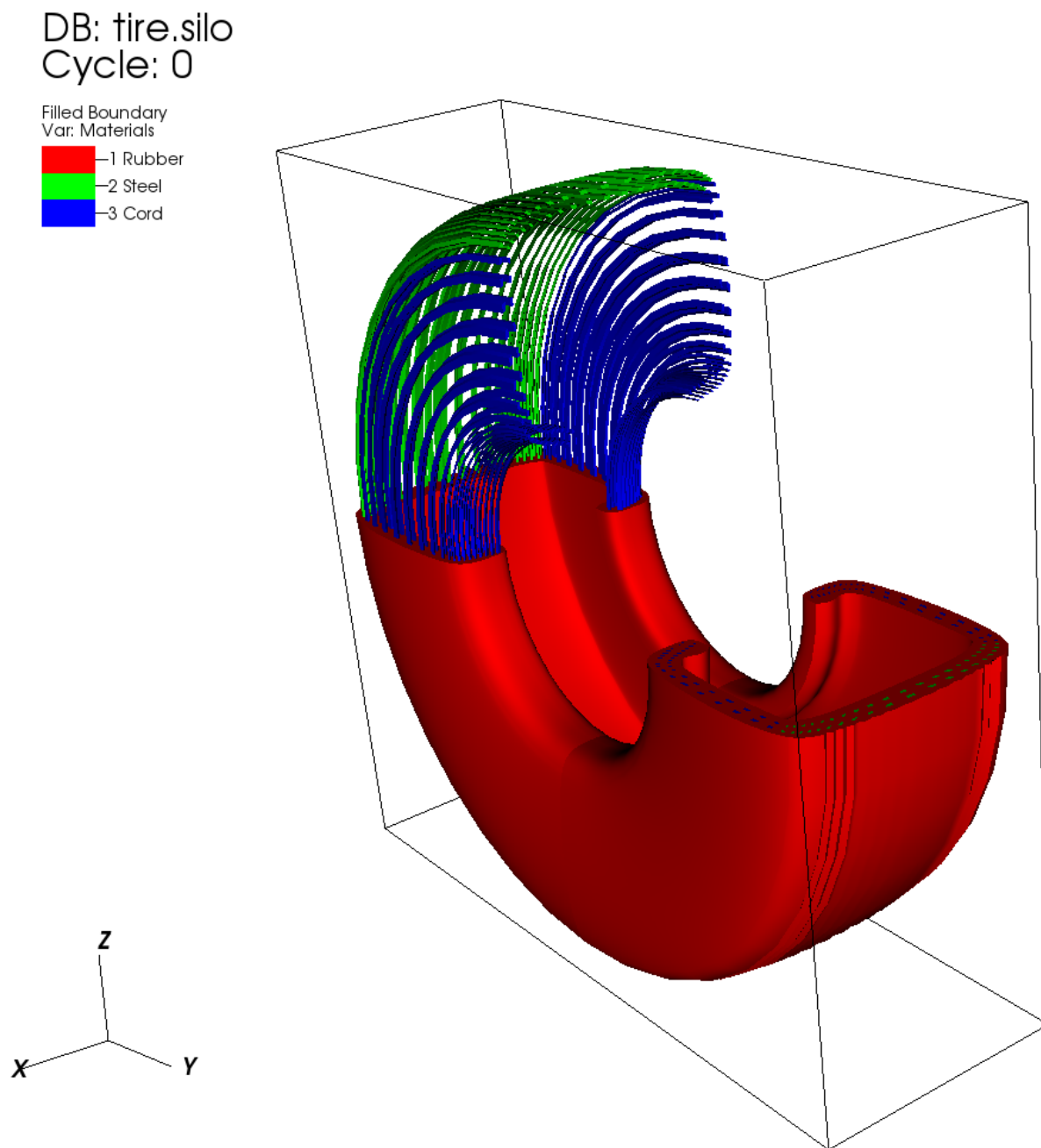
Fig. 1.27: FilledBoundary plot combined with subsets

Combining the FilledBoundary plot with subsets (see Figure 1.27) 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 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.

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
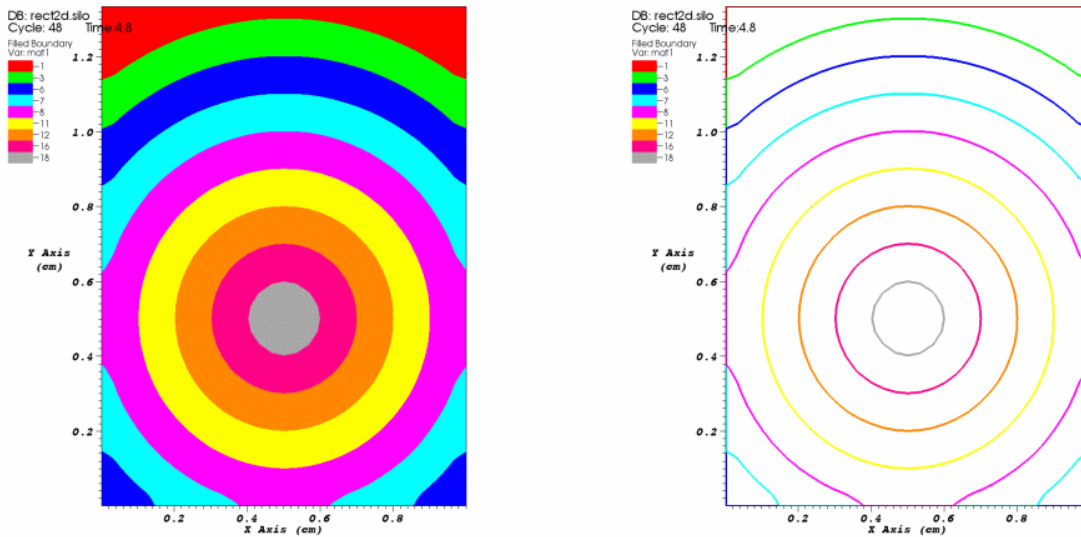
Fig. 1.28: Filled mode and wireframe mode

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**
- **Tetrahedron**
- **Point**
- **Sphere**
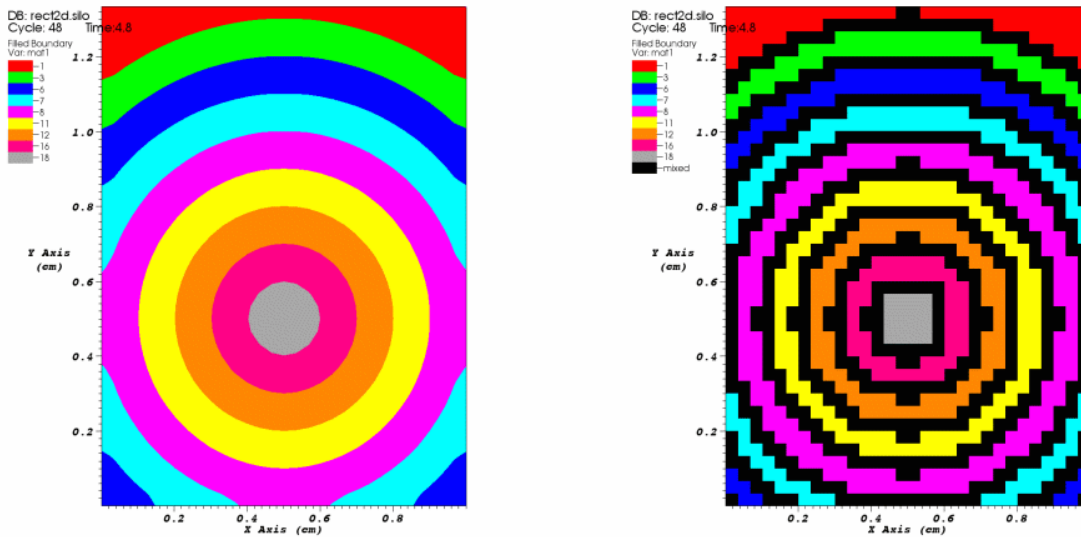
Fig. 1.29: All zones and clean zones

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**.
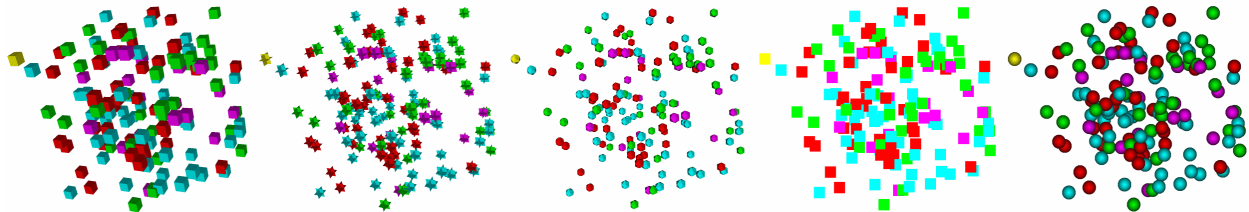
Fig. 1.30: Point types (left-to-right): Box, Axis, Icosahedron, Point, Sphere

Contour Plot

This plot, shown in Figure 1.31, 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.
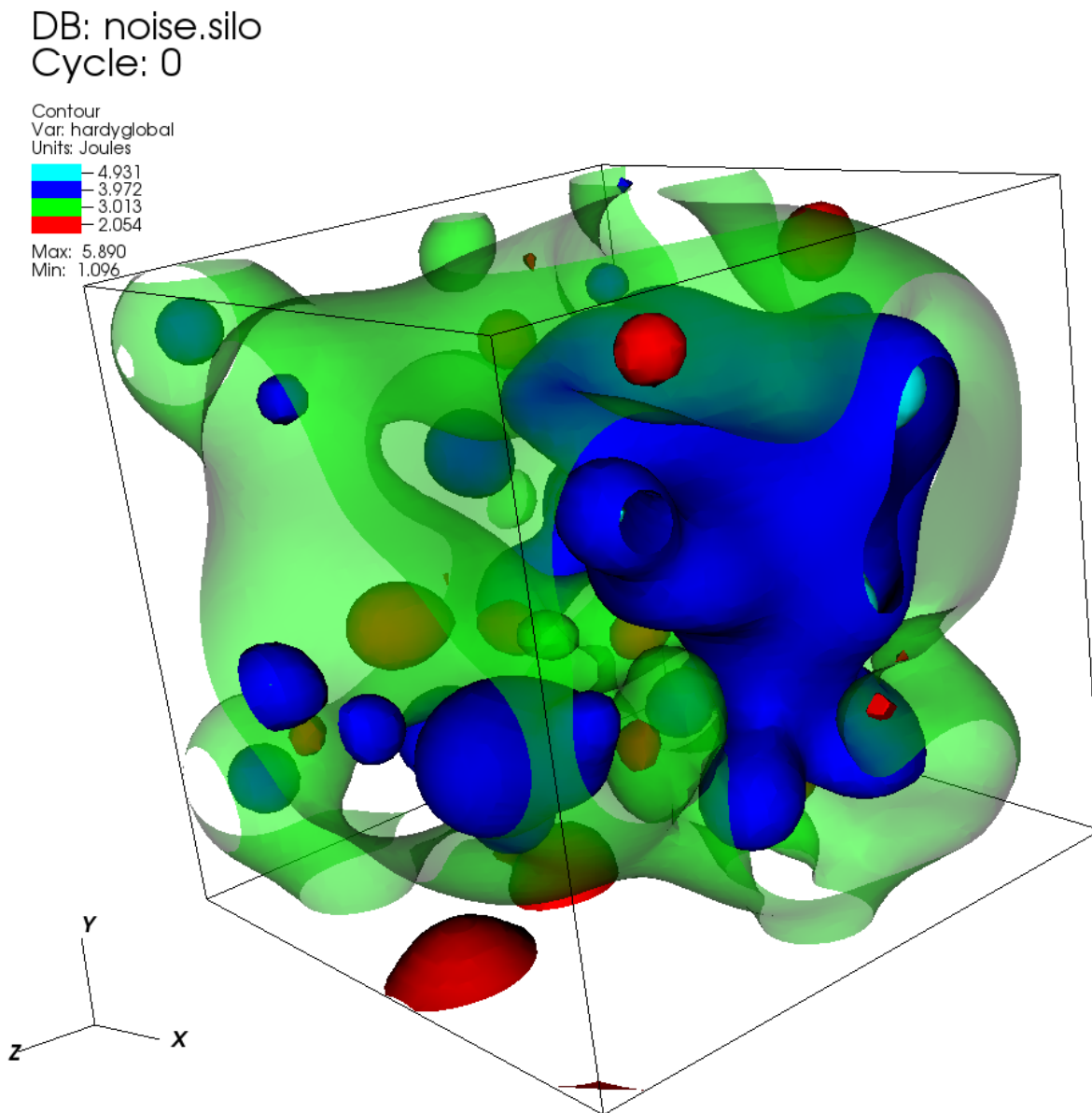
Fig. 1.31: Example of Contour plot
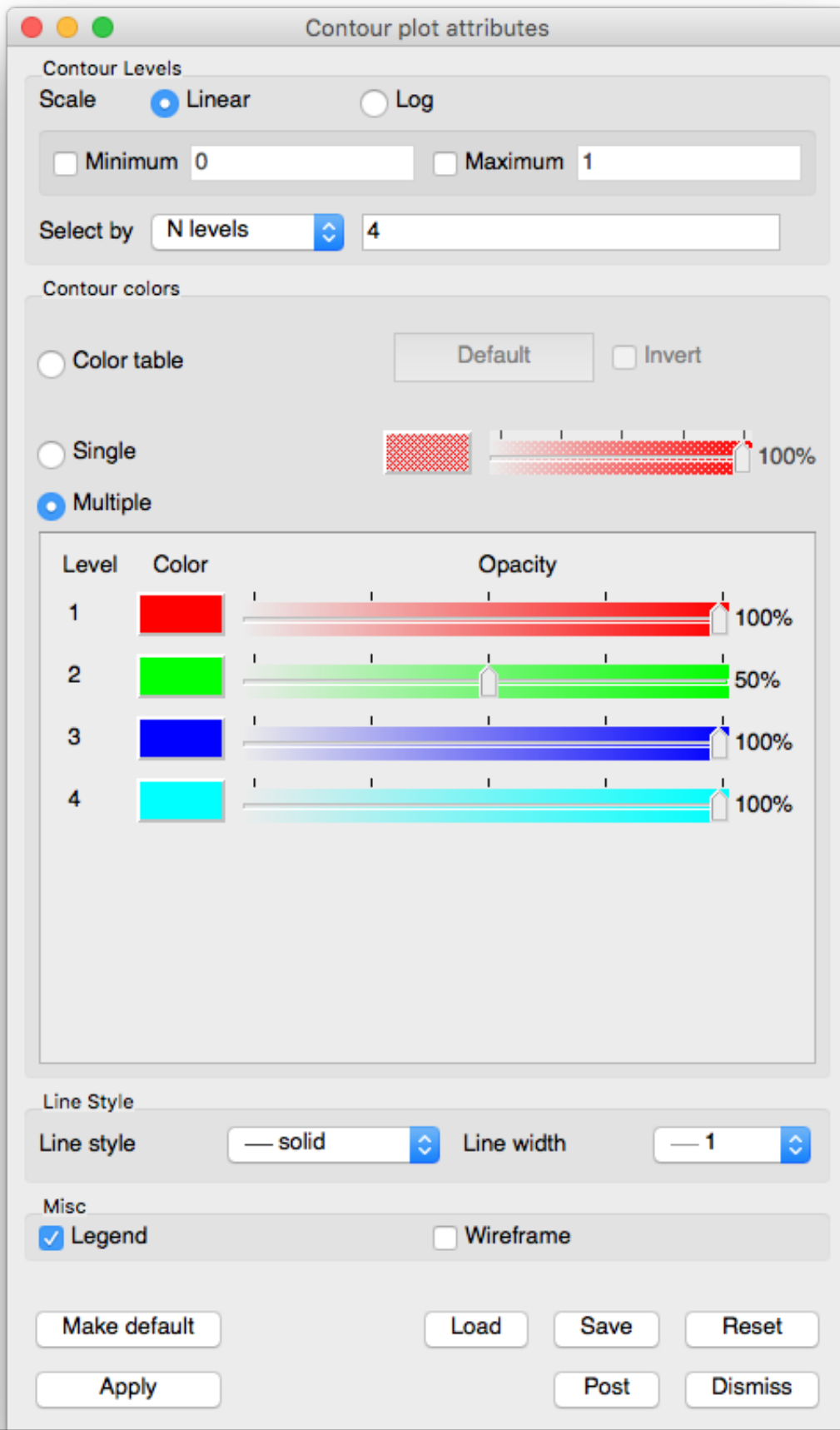
Setting the number of contours

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 1.33, 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.
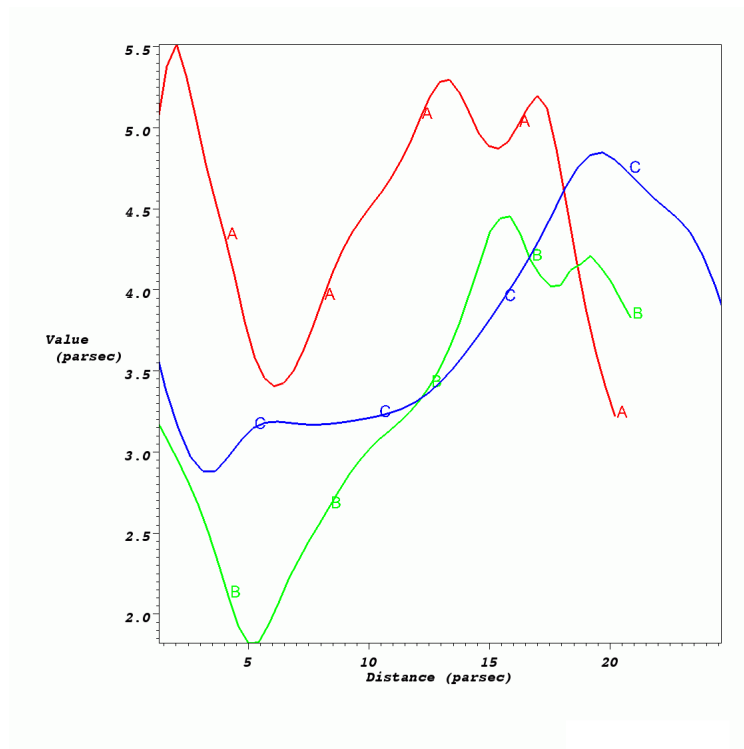
Fig. 1.33: Curve plot

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.
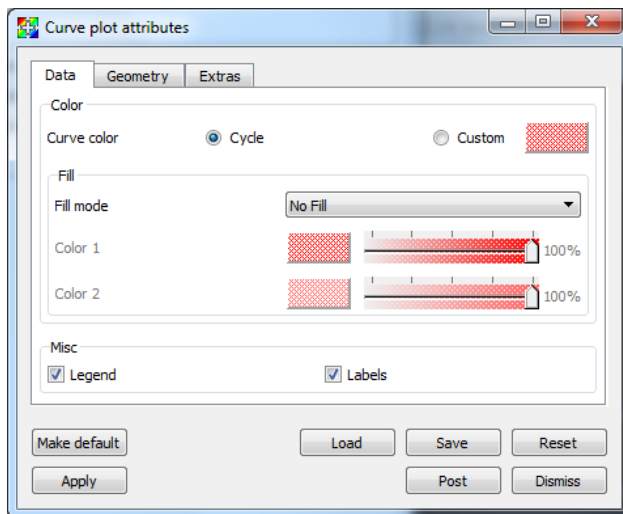
Fig. 1.34: Curve plot attributes, data tab

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.
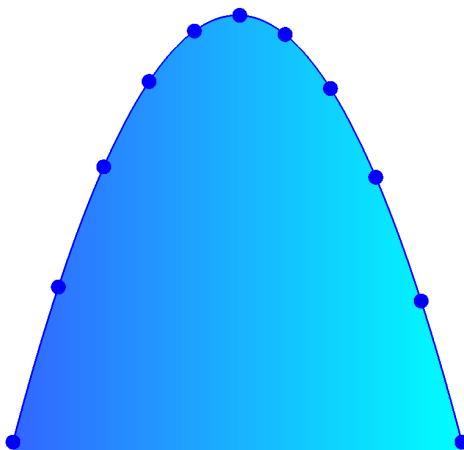
Fig. 1.35: Curve, space-filled with points

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 1.36). The line width, which determines the boldness of the curve, is set by making a

selection from the **Line Width** combo box.
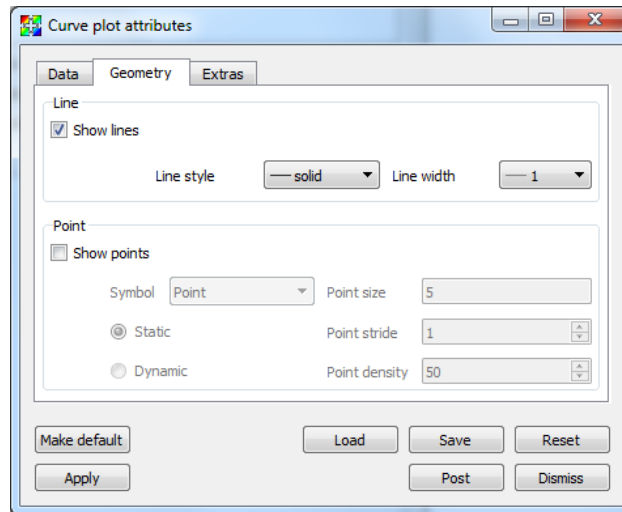
Fig. 1.36: Curve plot attributes, geometry tab

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 1.37). 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 1.38 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 1.39 shows an example.
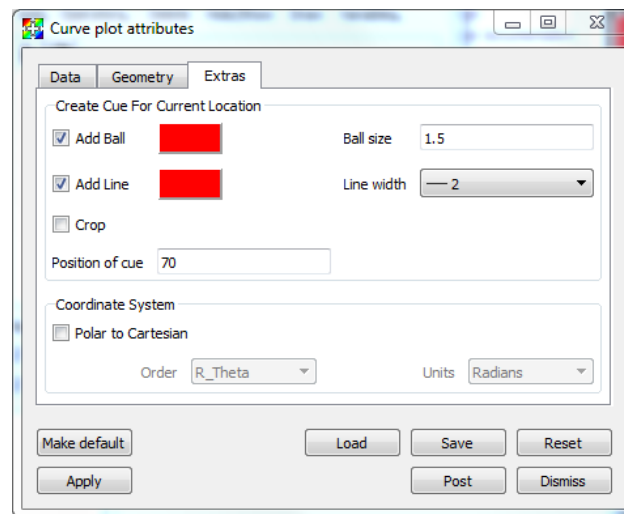
Fig. 1.37: Curve plot attributes, extras tab
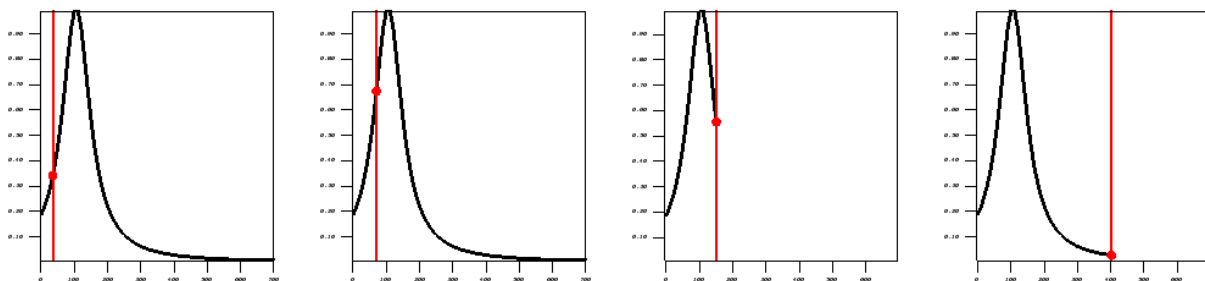
Fig. 1.38: Curve plot with time cues added at different positions, both uncropped and cropped.
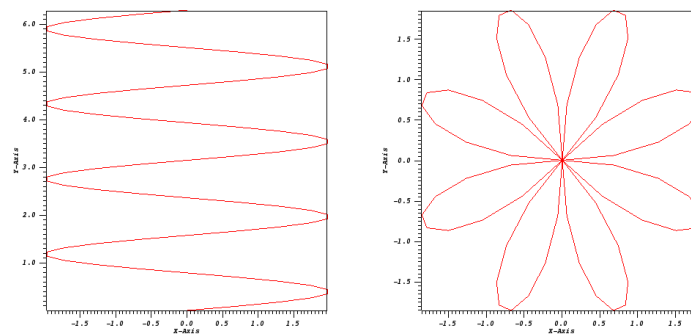
Fig. 1.39: Curve plot before and after Polar coordinate transform (R-theta, radians)

Histogram Plot

The Histogram plot divides the data range of a scalar variable into a number of bins and groups the variable's values, weighted by cell area or revolved volume, into different bins. 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 1.40.
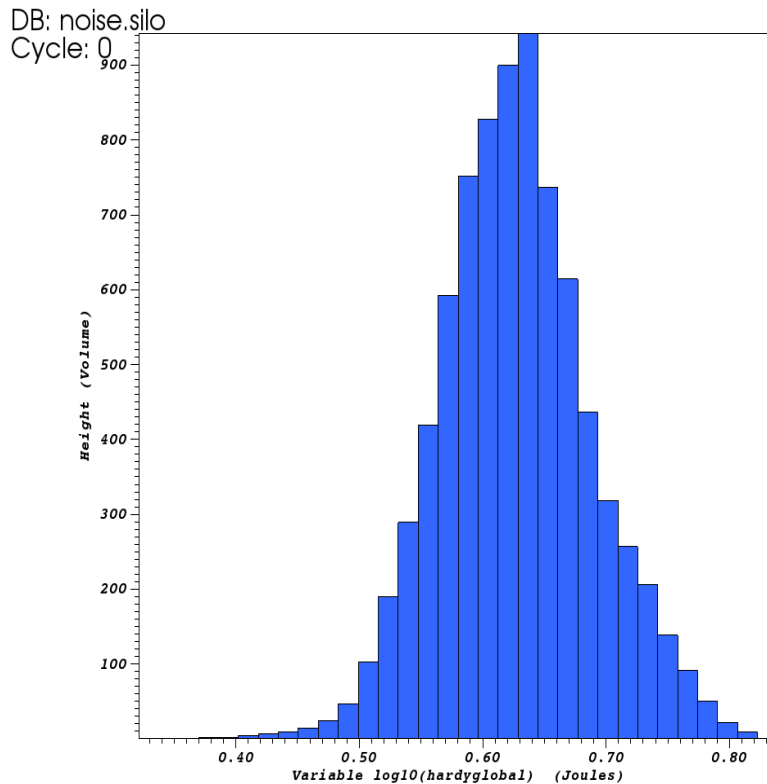
Fig. 1.40: Histogram plot

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 **Specify Range** 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 1.41) before clicking its **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.

Setting the number of bins

The Histogram plot divides a variable's data range into a number of bins and then counts the weighted 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

When the Histogram plot groups data values into bins, it weights the data value by the surface area or revolved volume of the cell so contributions from different sizes of cells are compared fairly. To change the calculation method used to weight the cells, click on the **Area** radio button to make VisIt use surface area or click on the **Revolved volume** radio button to make VisIt use the revolved volume of a 2D cell as the weighting multiplier used to group cells into the right bins.

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.

Label Plot

The Label plot, shown in Figure 1.42, 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 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
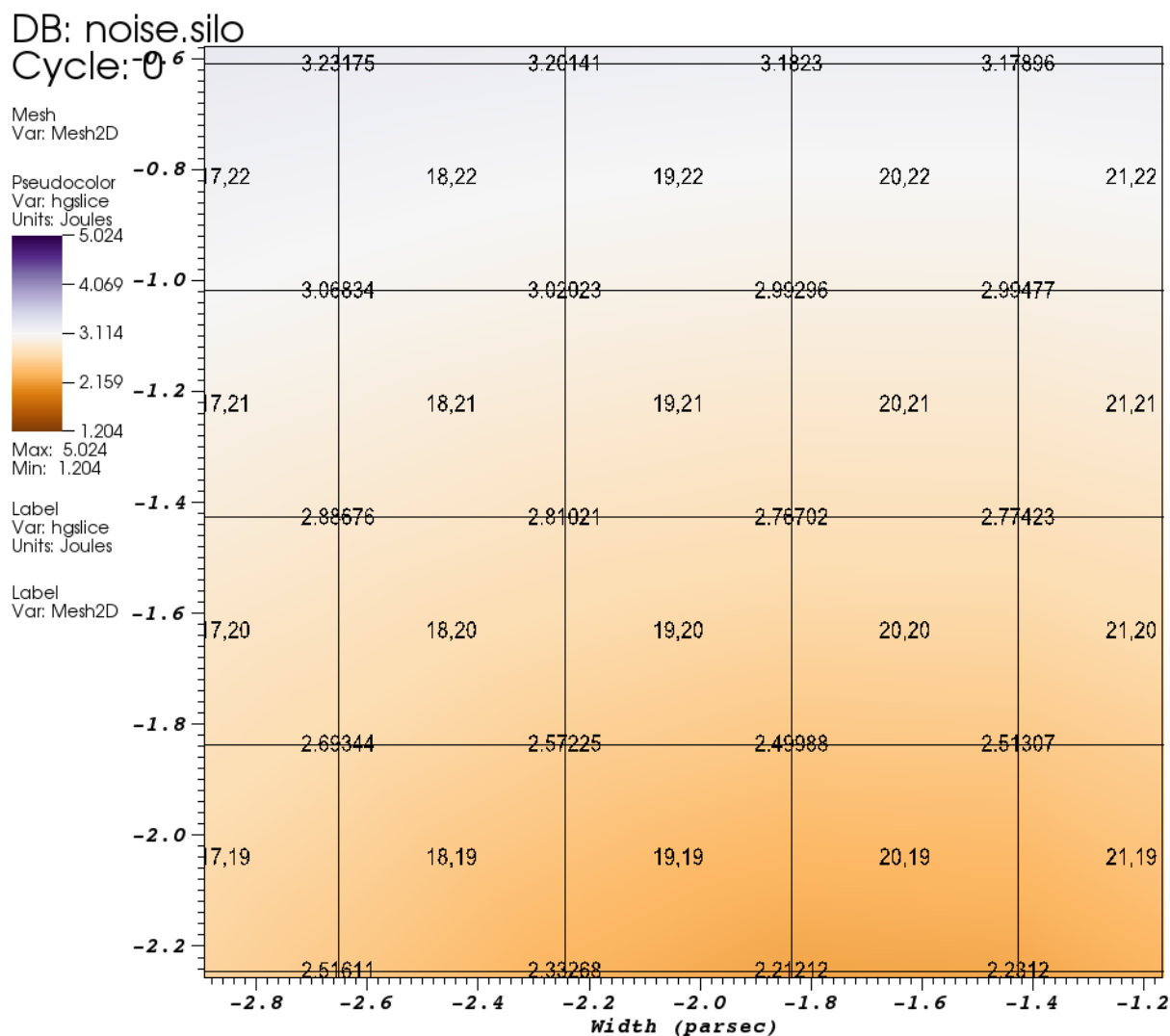
Fig. 1.42: Label plot of the mesh overlayed on Pseudocolor and Mesh plots
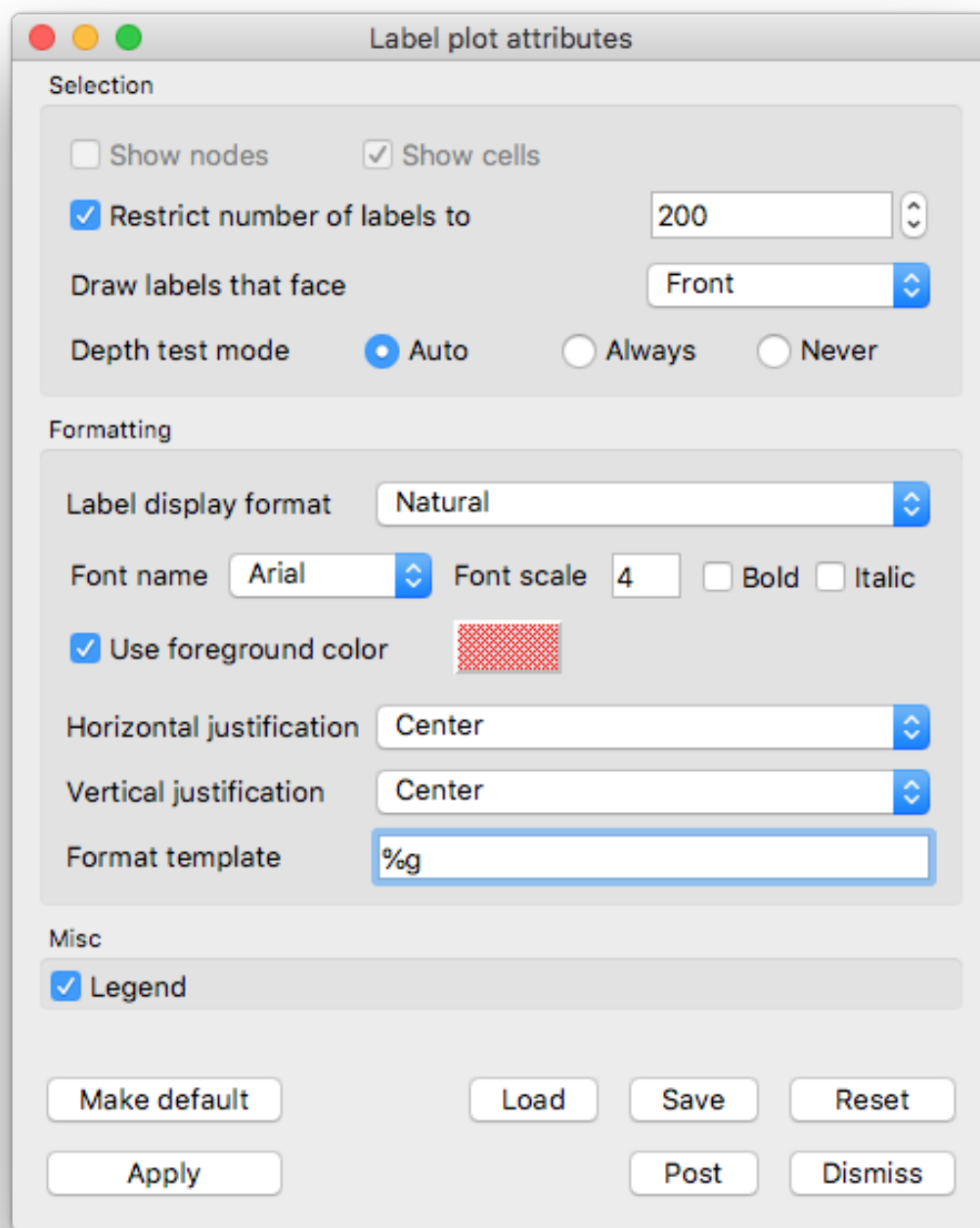
Fig. 1.43: Label plot attributes window

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**.

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.
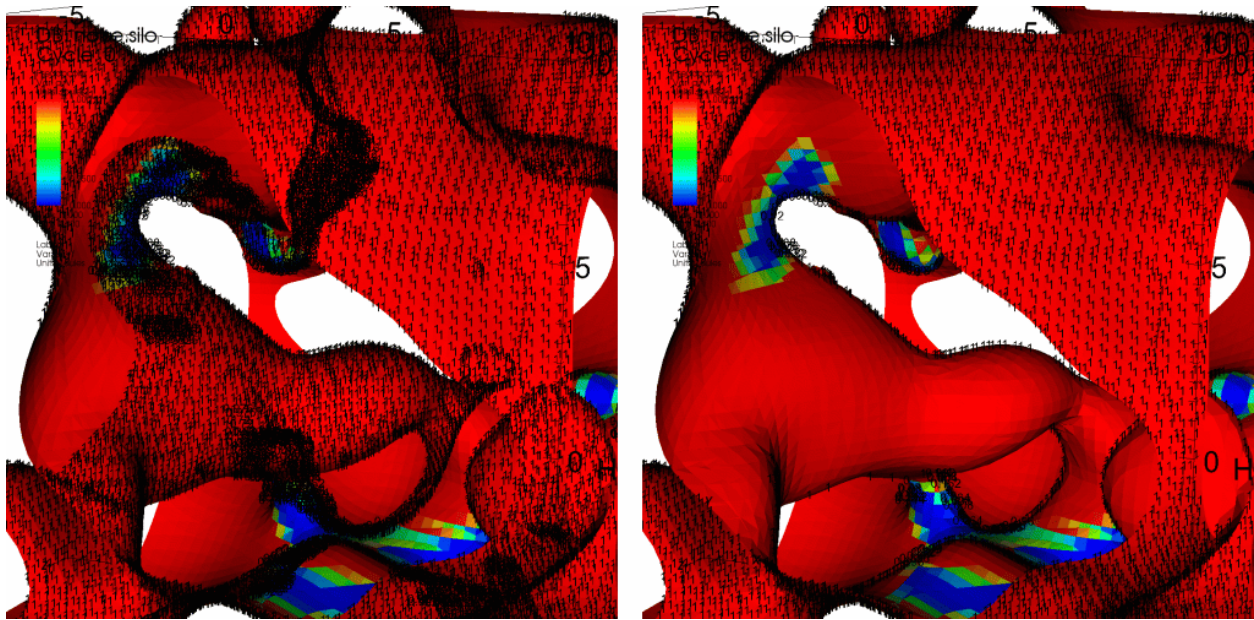
Fig. 1.44: Removing extra labels (left) with depth test (right)

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 **??**).

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 1.46.

Mesh Plot

The **Mesh** plot, shown in Figure 1.47, 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. 1.45: Cell and node labels can be different colors when labeling mesh variables using additional controls in the
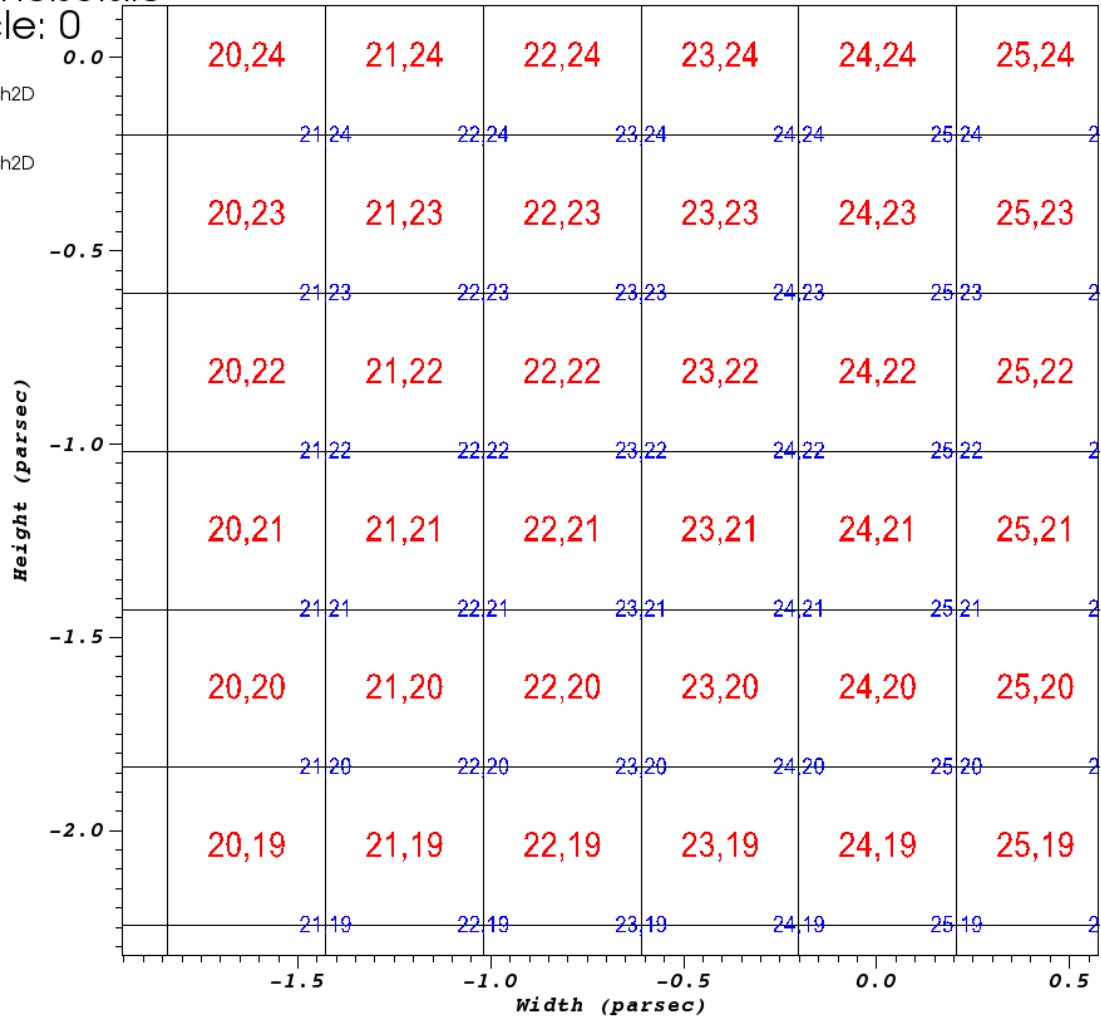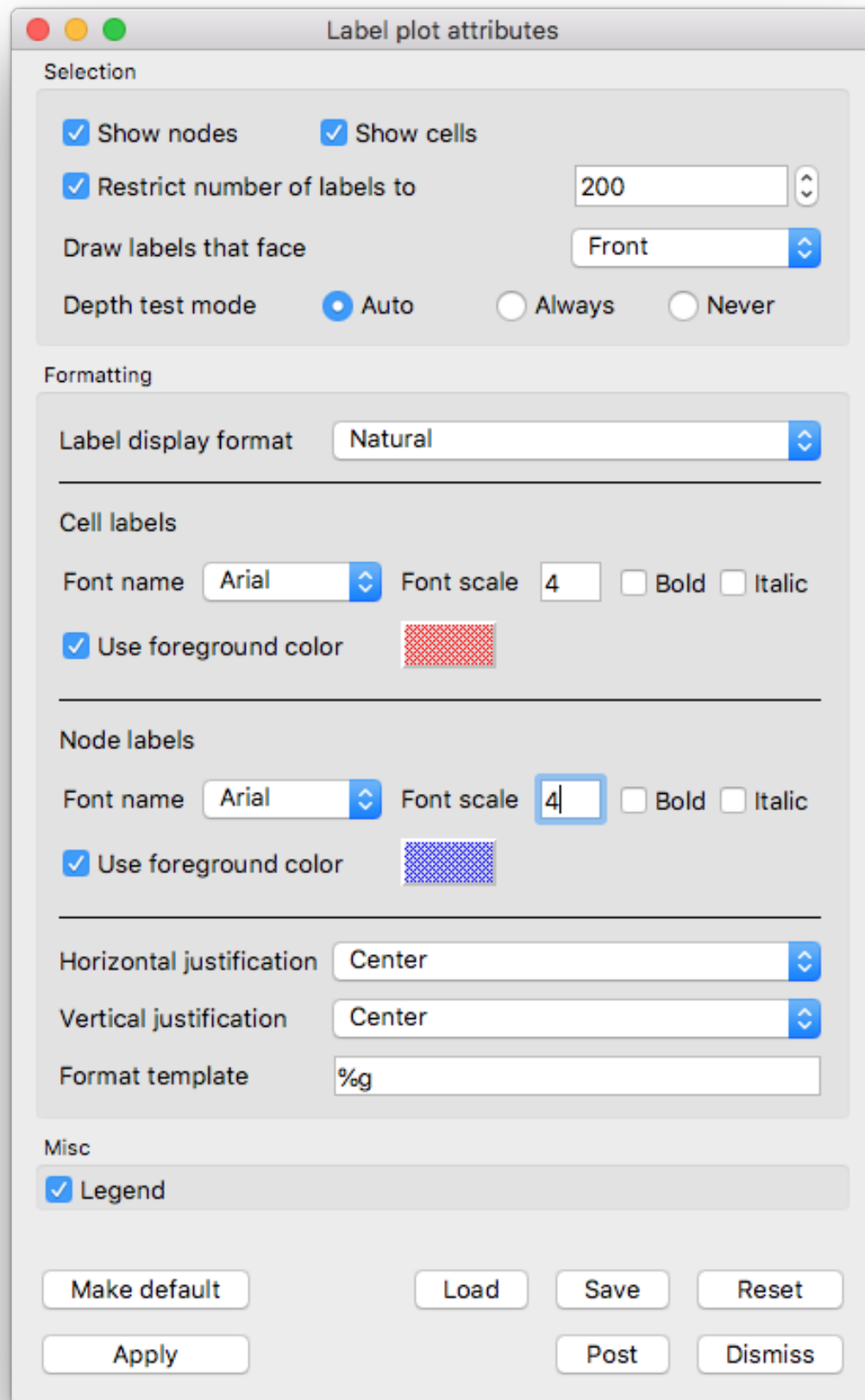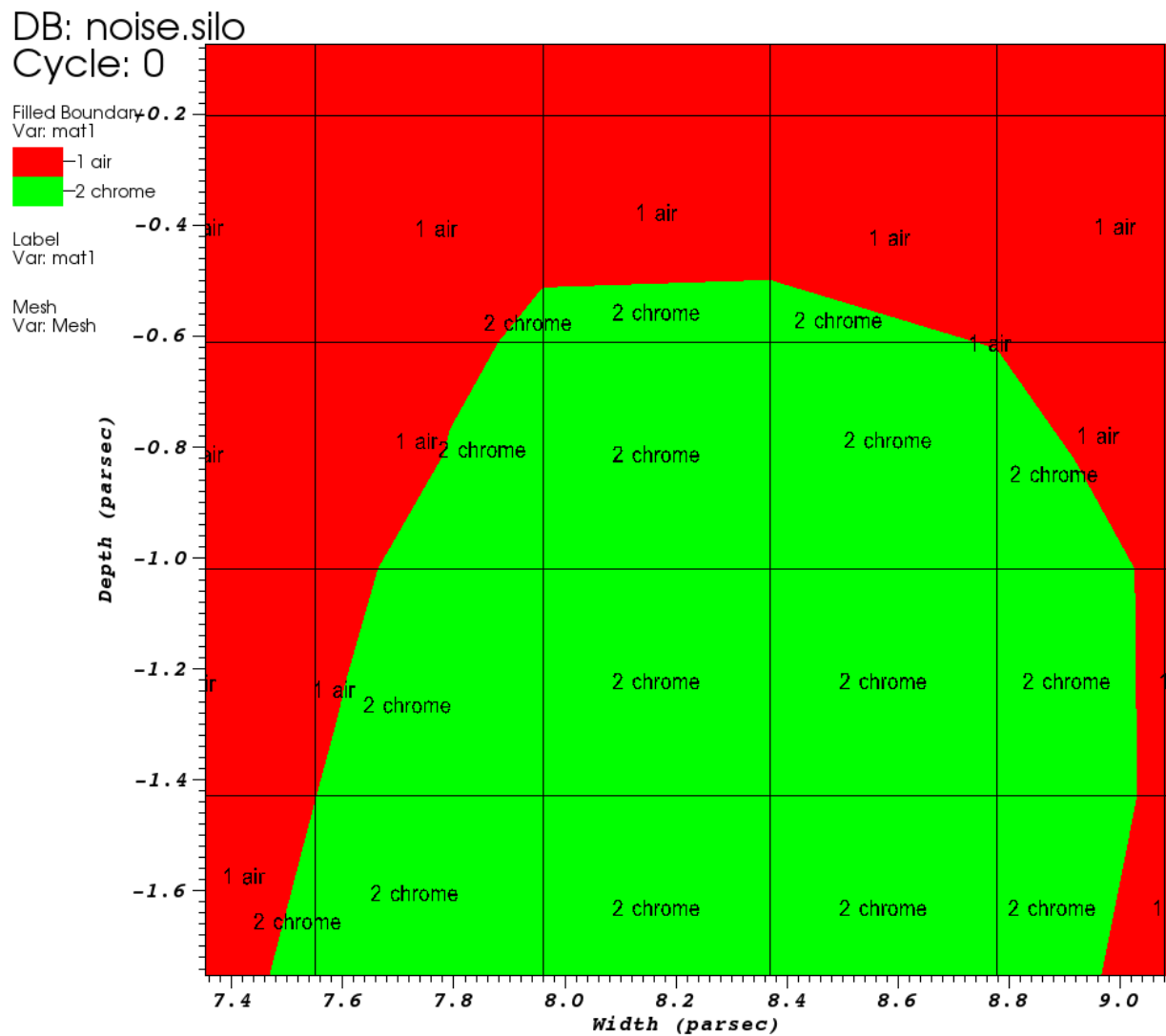
Fig. 1.46: Label plot of materials

DB: f5e_05.obj

Mesh
Var: OBJMesh
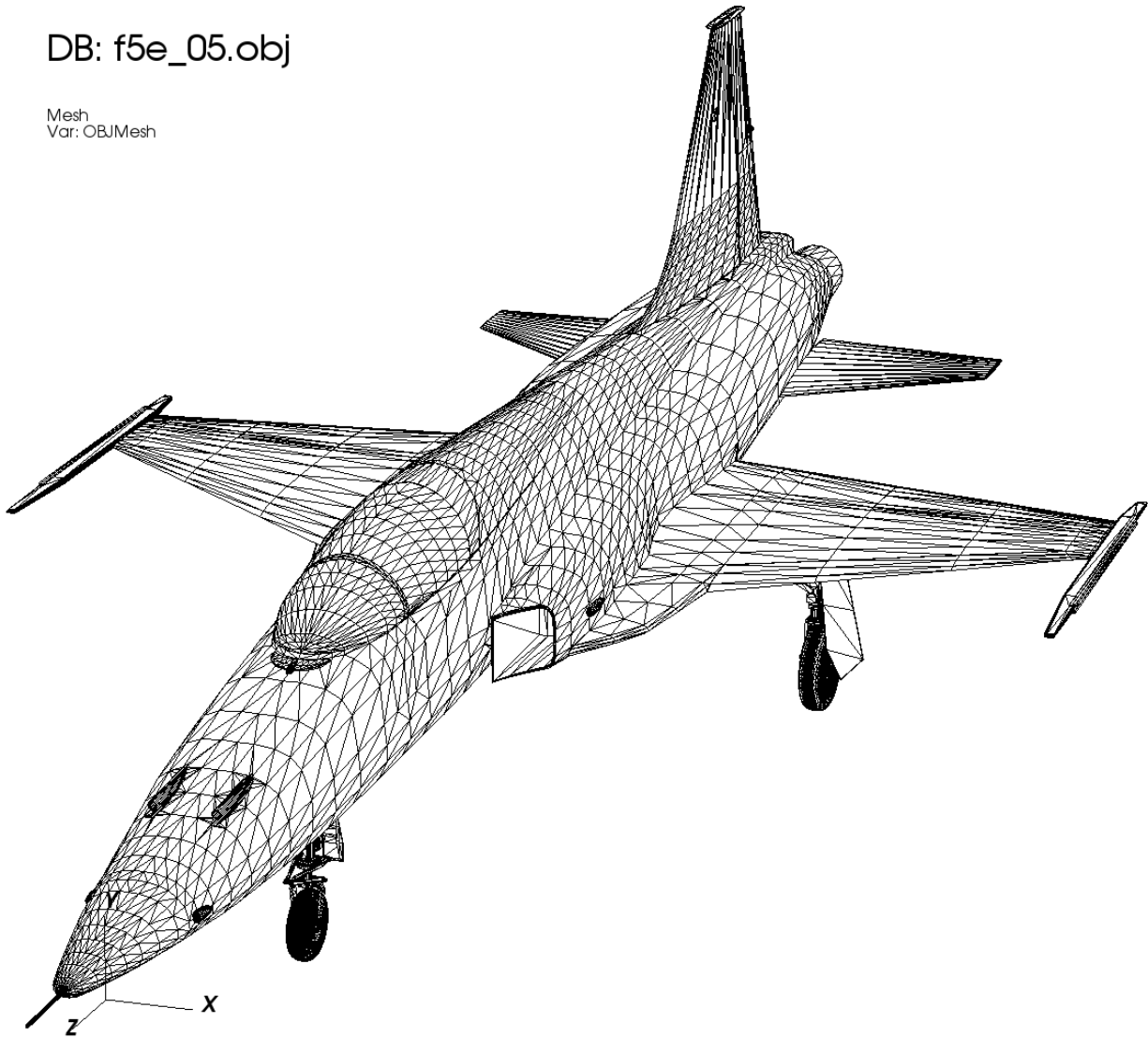
Fig. 1.47: Mesh plot
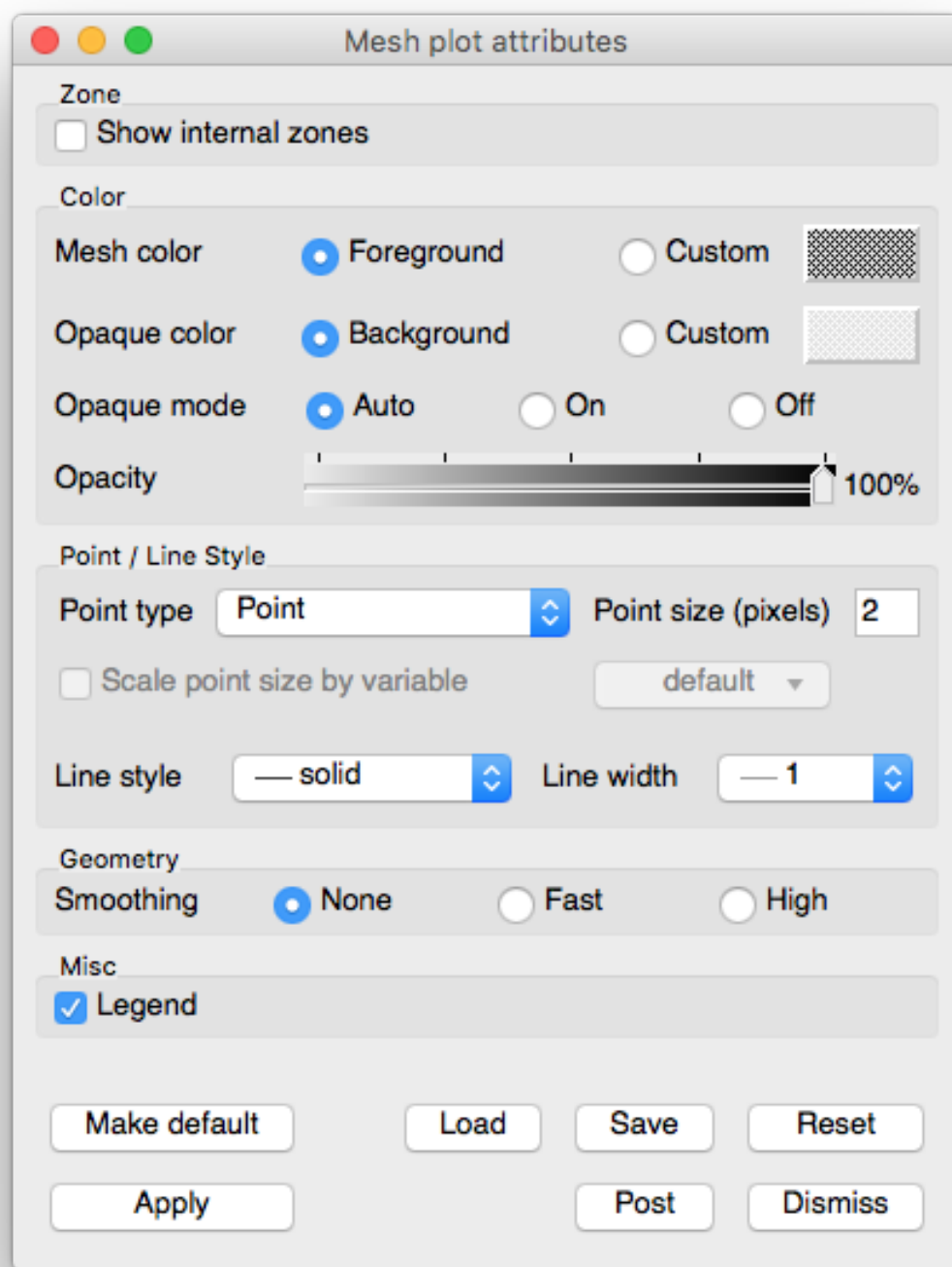
Fig. 1.48: Mesh plot window

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 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 the opaque color

An opaque **Mesh** plot uses the background color of the visualization window for the **Mesh** plot faces. To set the opaque color to a color other than the visualization window's background color, uncheck the **Use background** check box and click on the **Opaque color** button and select a new color from the **Popup color menu**.

Changing the mesh color

The mesh color is the color used to draw the mesh lines. The mesh lines normally use the visualization window's foreground color. To use a different color, uncheck the **Use foreground** check box, click the **Mesh color** button, and select a new color from the **Popup color menu**.

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.

Pseudocolor plot

The **Pseudocolor** plot, shown in Figure 1.49, 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.
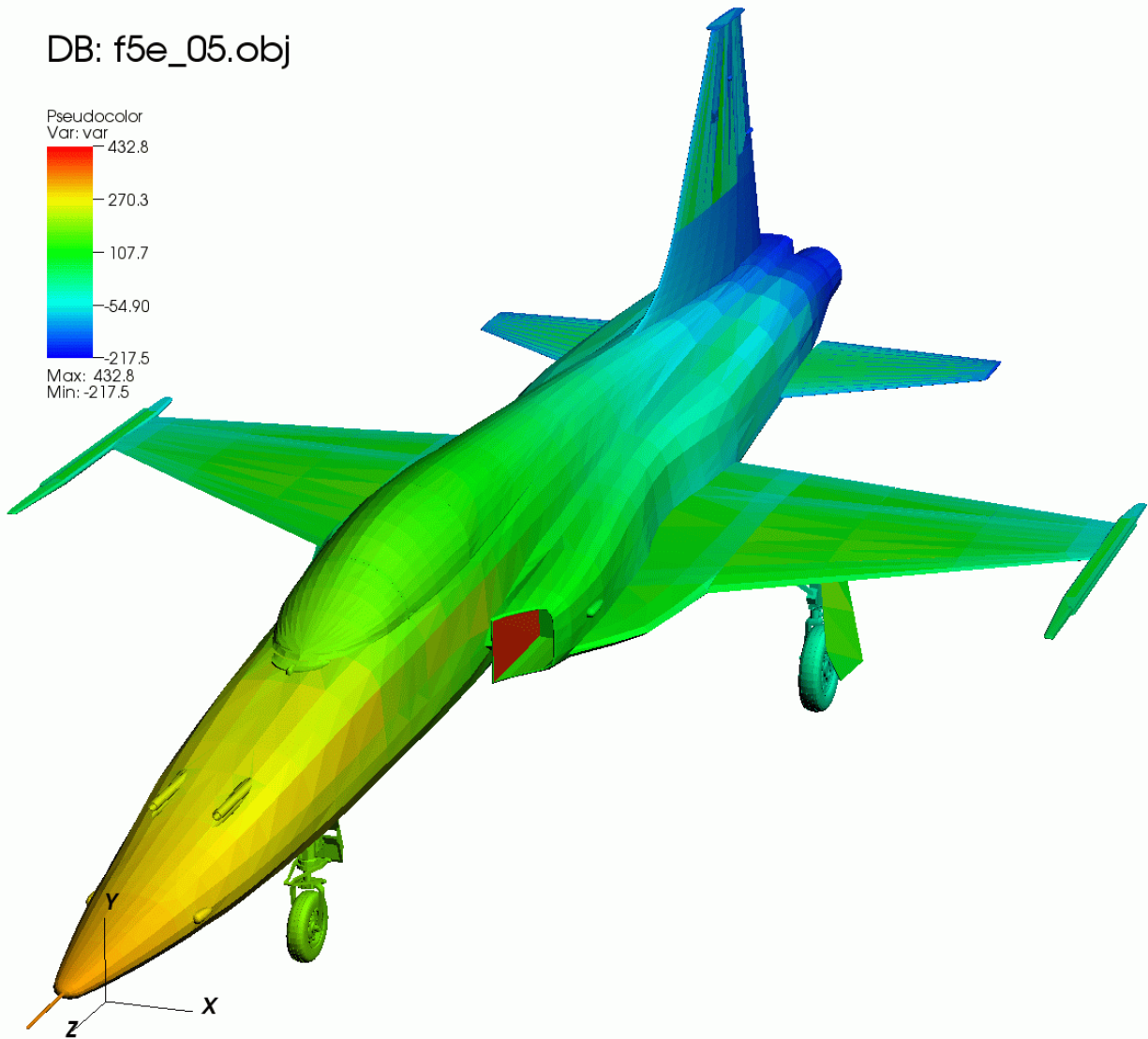
Fig. 1.49: Pseudocolor plot

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 1.50)
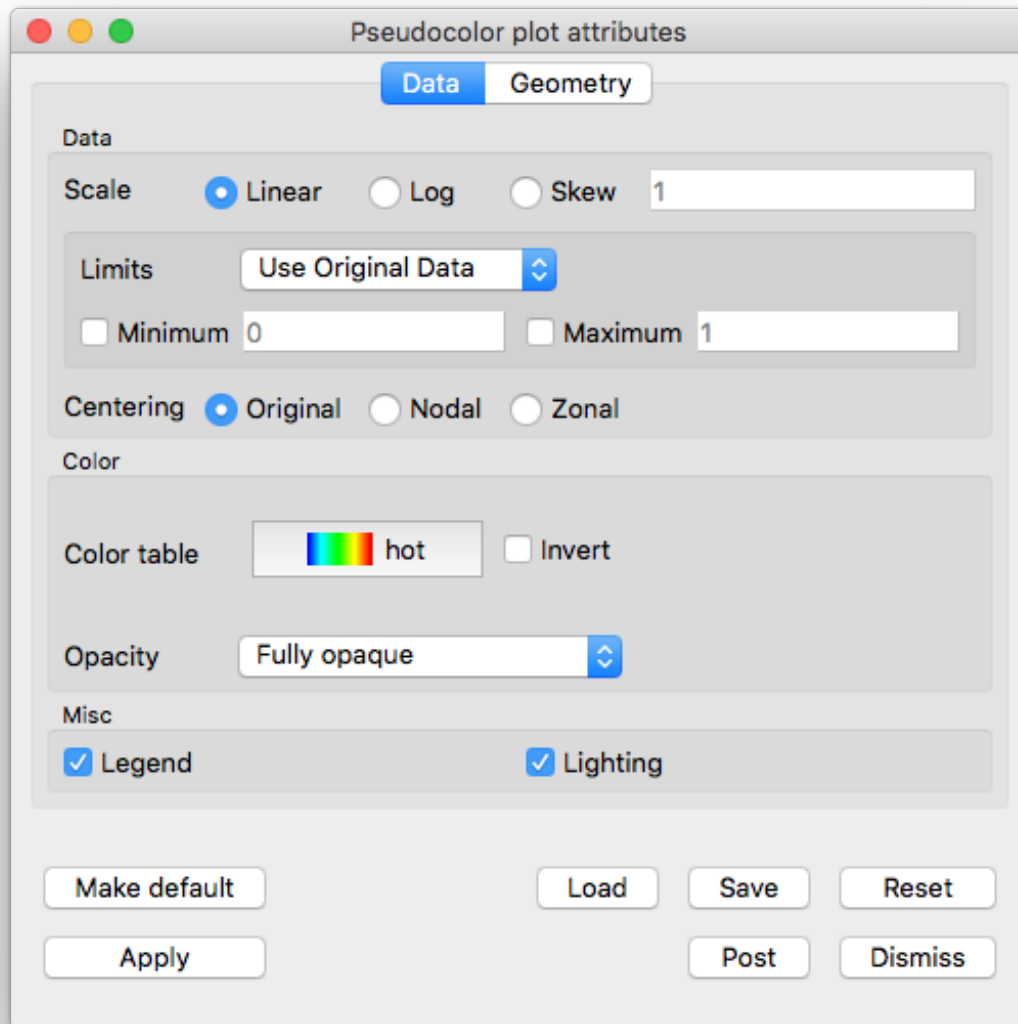
Fig. 1.50: Pseudocolor plot attributes window Data tab

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 current plot 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 Current Plot** from the **Limits** menu.

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 1.51, 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,

Default

Accent
Blues
BrBG
BuGn
BuPu
Dark2
GnBu
Greens
Greys
OrRd
Oranges
PRGn
Paired
Pastel1
Pastel2
PiYG
PuBu
PuBuGn
PuOr
PuRd
Purples
RdBu
RdGy
RdPu
RdYlBu
RdYlGn
Reds
Set1
Set2
Set3
Spectral
YlGn
YlGnBu
YlOrBr
YlOrRd
amino_rasmol
amino_shapely
bluehot
caleblack
calewhite
contoured
cpk_jmol

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 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 1.52)

Lines

The lines section can be useful when visualizing the results of a **Poincare** or **Integral Curve** 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.
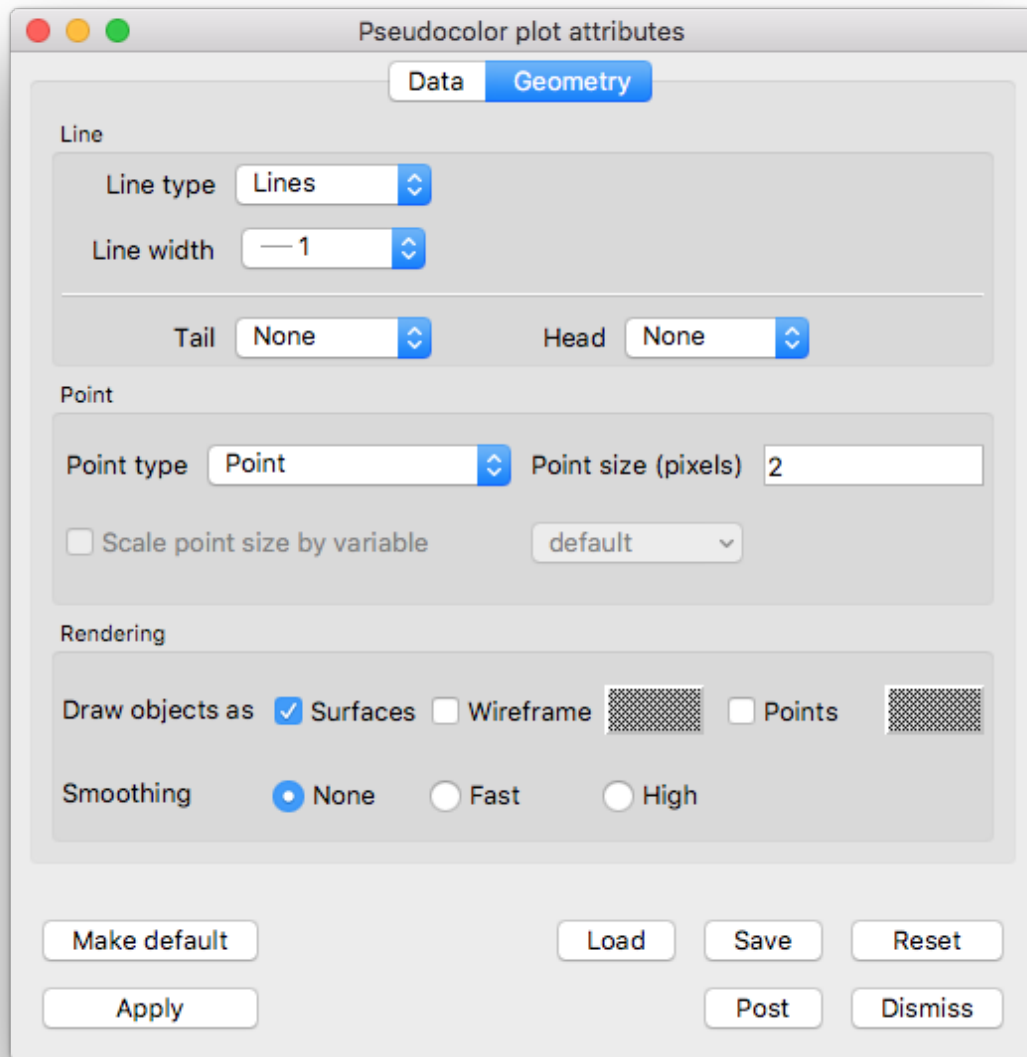
Fig. 1.52: Pseudocolor plot attributes window, geometry tab

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.

Scatter Plot

The **Scatter** plot (see Figure 1.53) 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.

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

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).
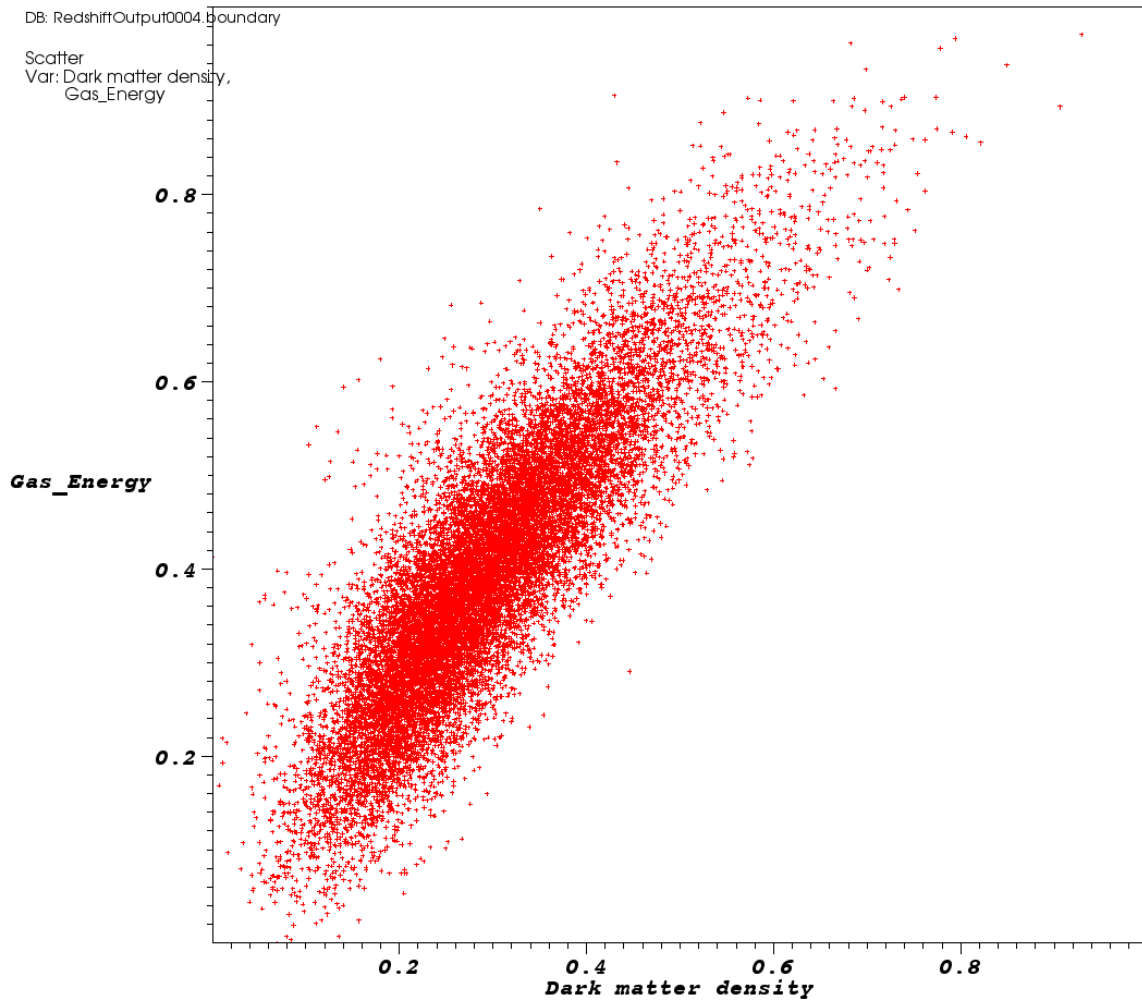
Fig. 1.53: Example of Scatter plot
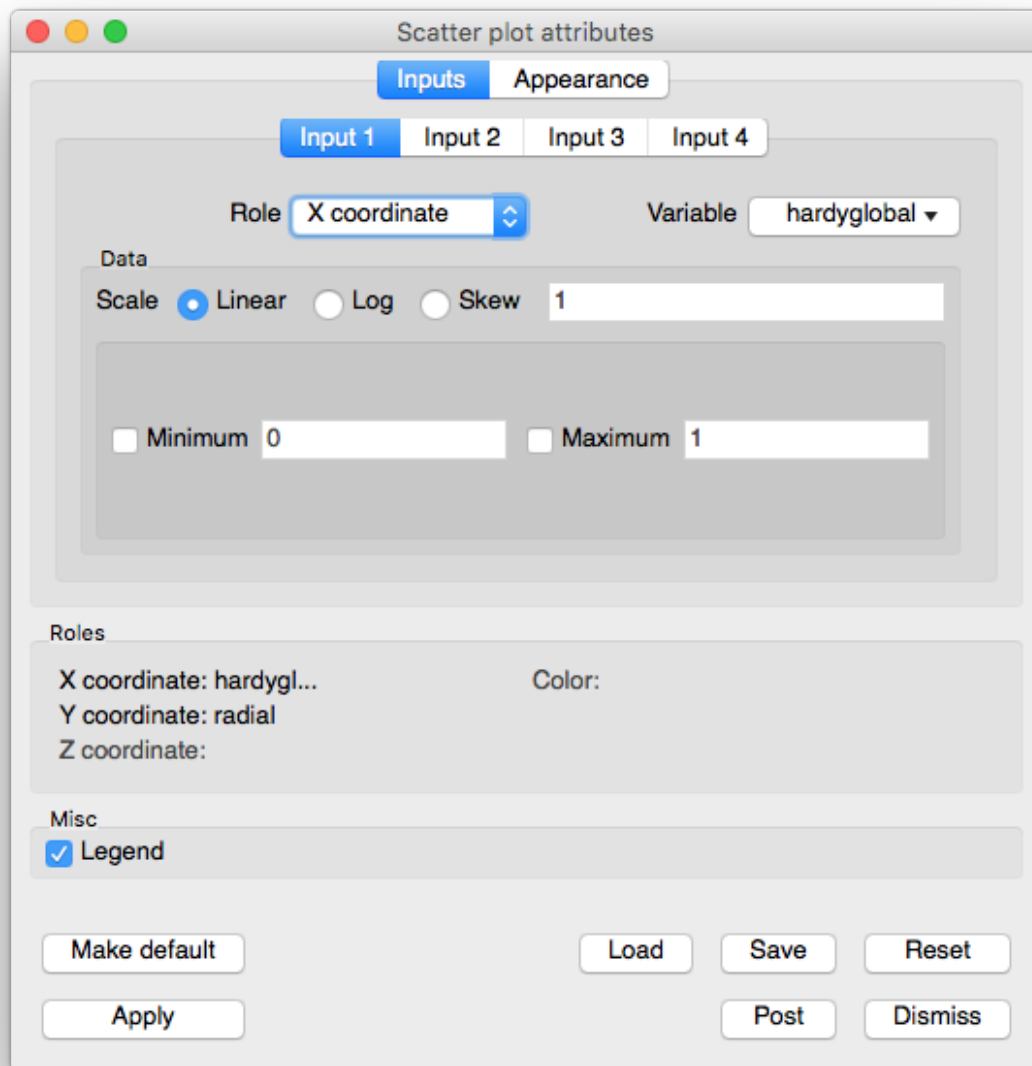
Fig. 1.54: Scatter plot attributes window
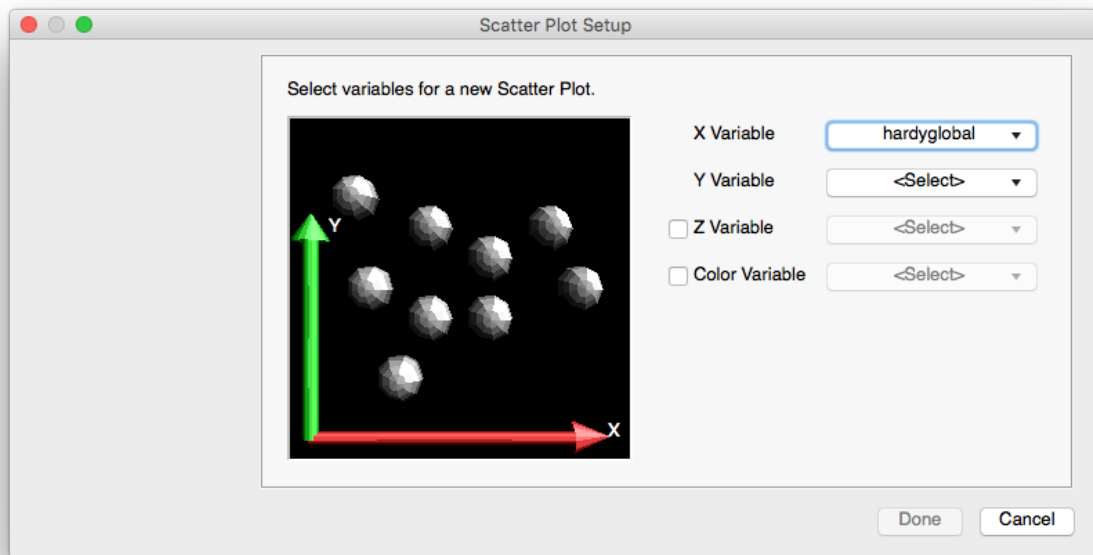
Fig. 1.55: Example of the Scatter plot wizard

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.

Setting point properties

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

Subset Plot

The Subset plot (example in Figure 1.57) 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.

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
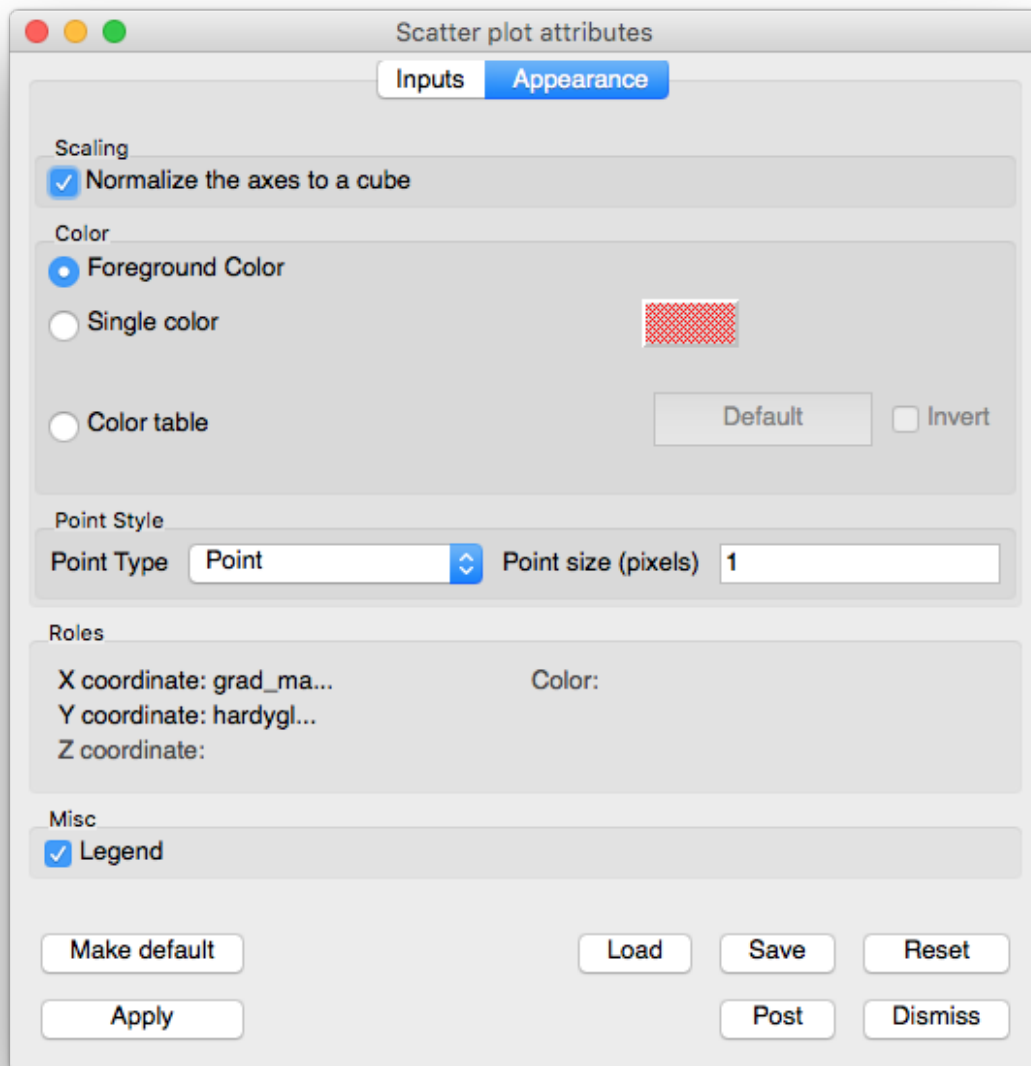
Fig. 1.56: Scatter plot attributes window's Appearance tab
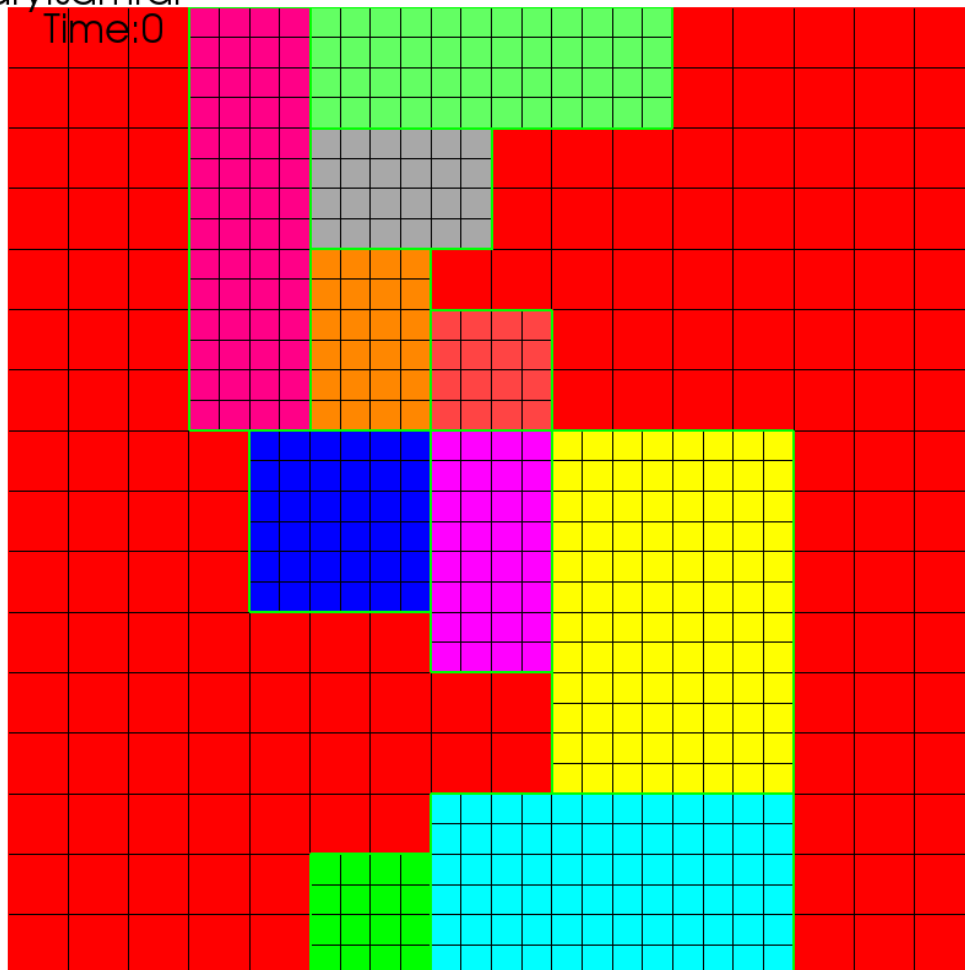
Fig. 1.57: Example of Subset plot of an AMR Mesh
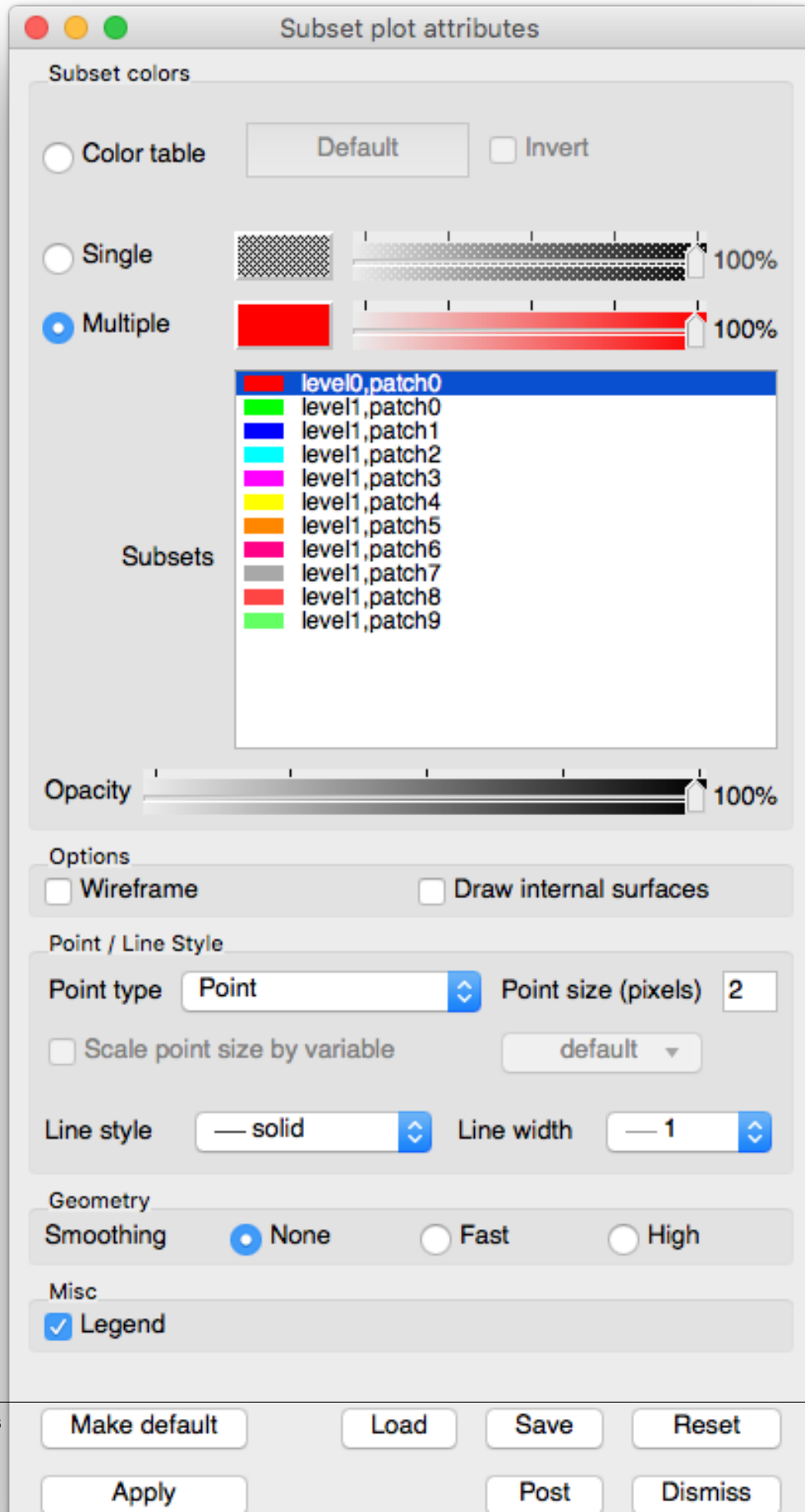
colors can be assigned three different ways, the first of which uses a color table. A color table is a named palette of colors that you can customize to suite your needs. When the Subset plot uses a color table to color subsets, it selects colors that are evenly spaced through the color table based on 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 1.59, 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 1.61, 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

---
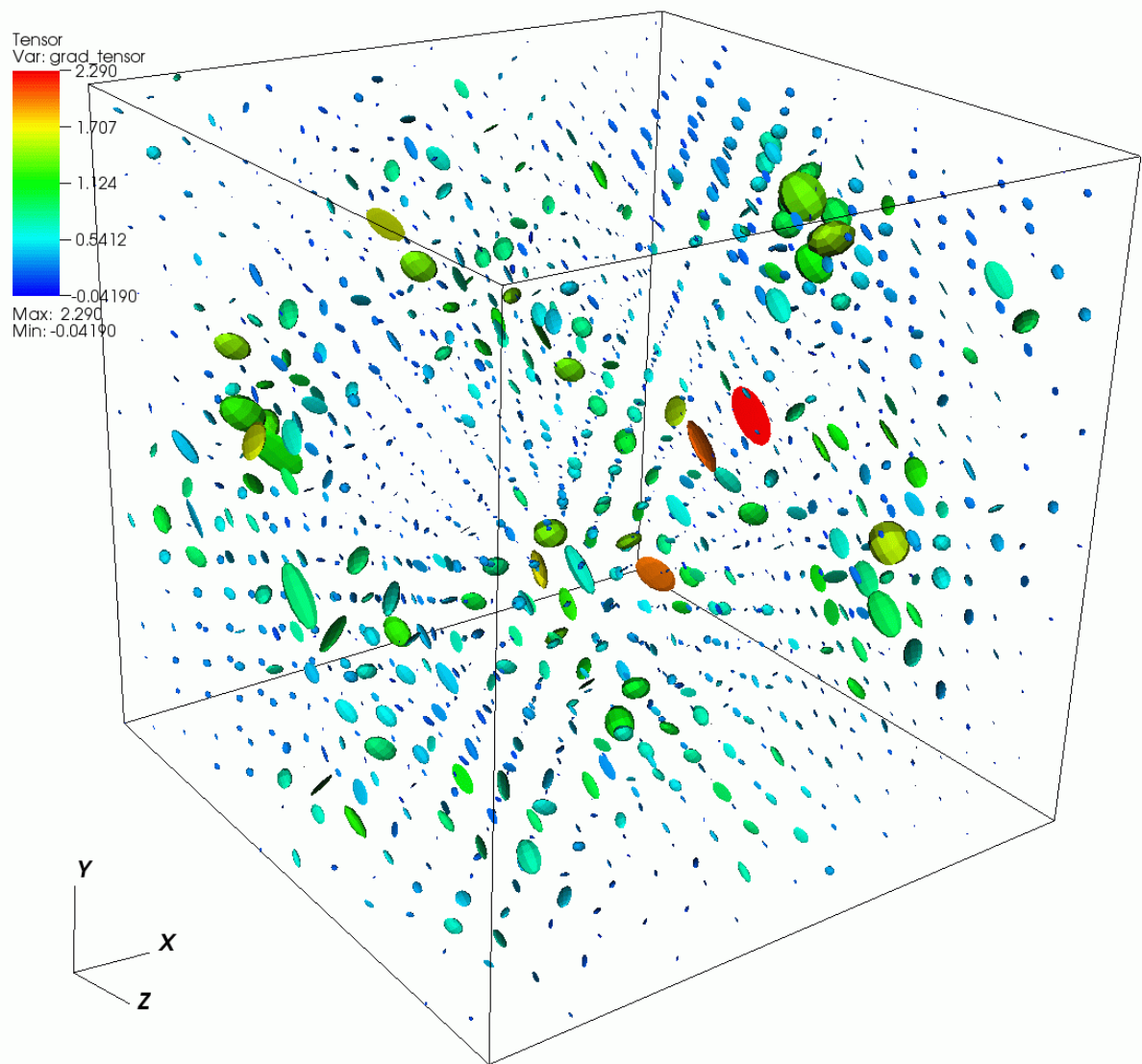
Fig. 1.59: Example of Tensor plot
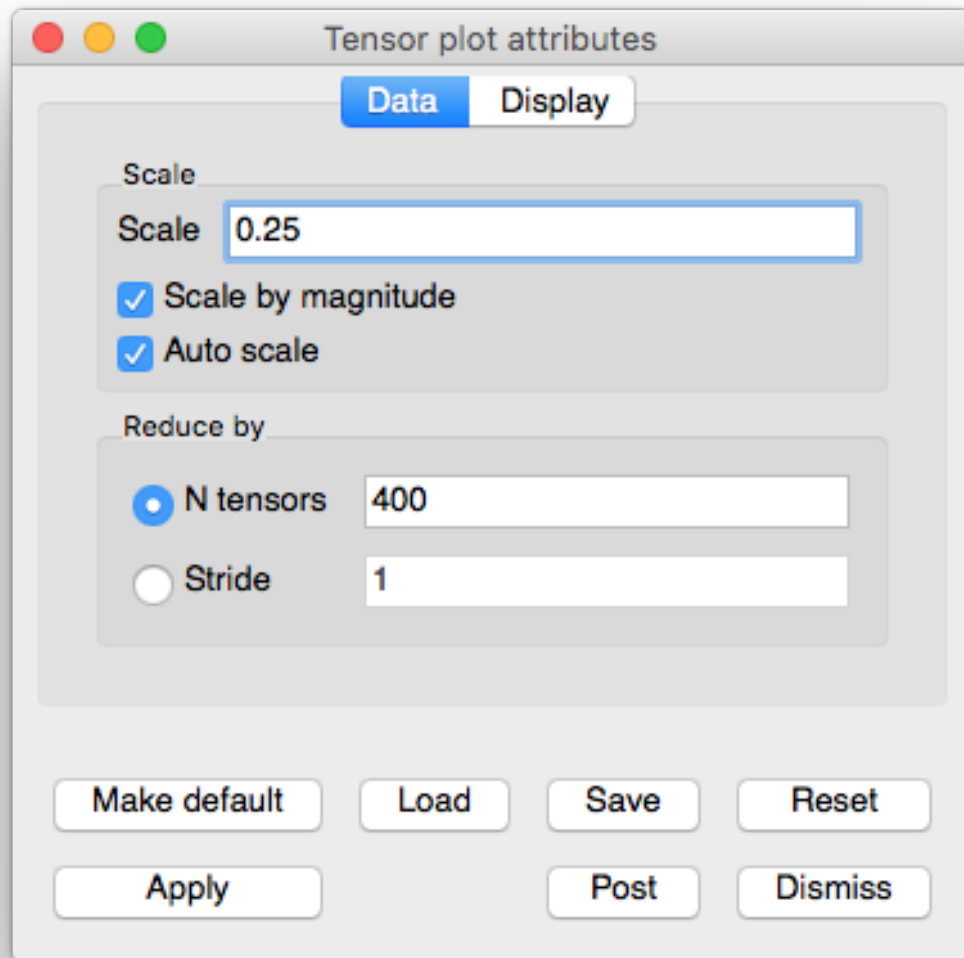
Fig. 1.60: Tensor plot attributes window
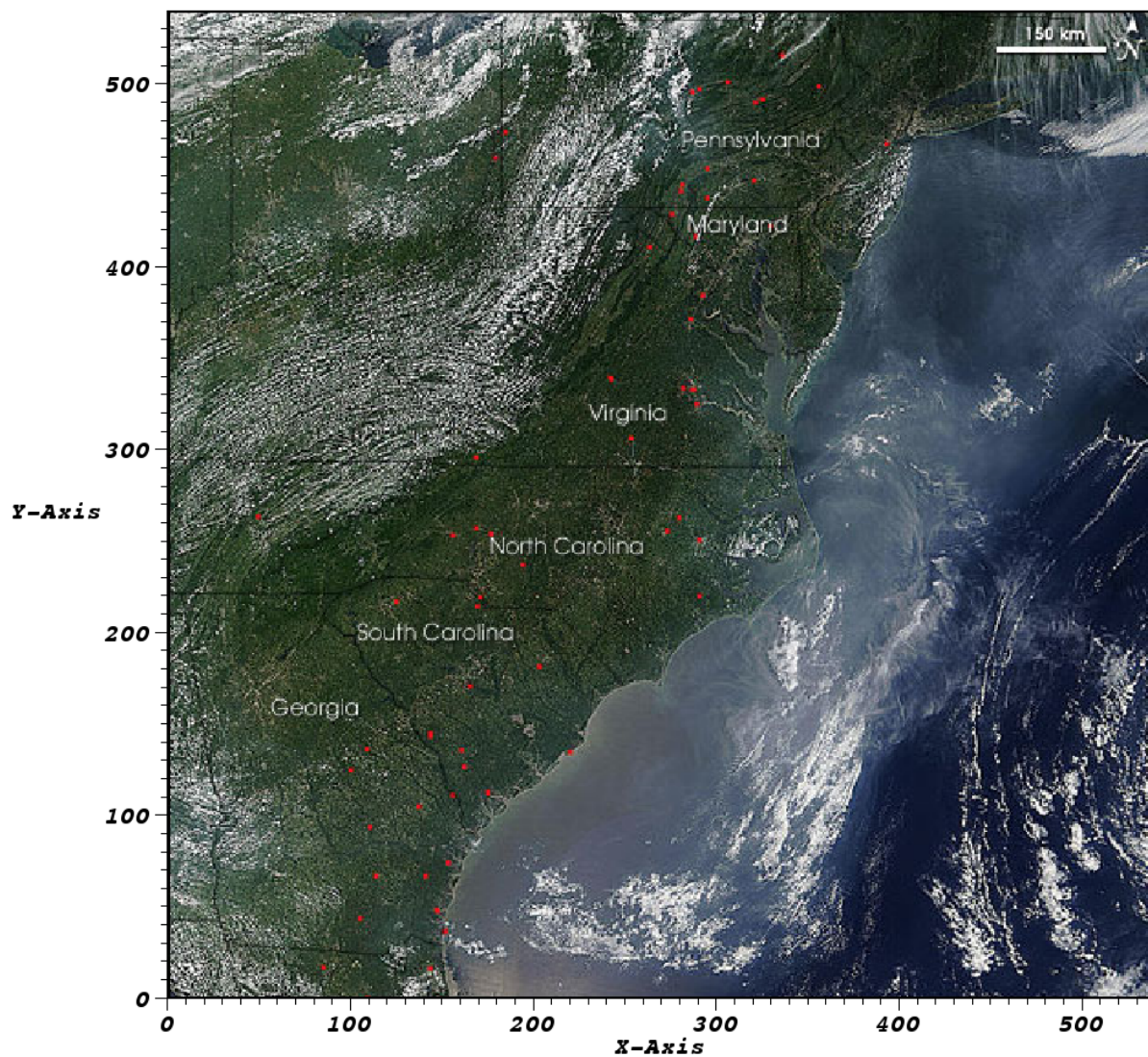
Fig. 1.61: Truecolor Plot
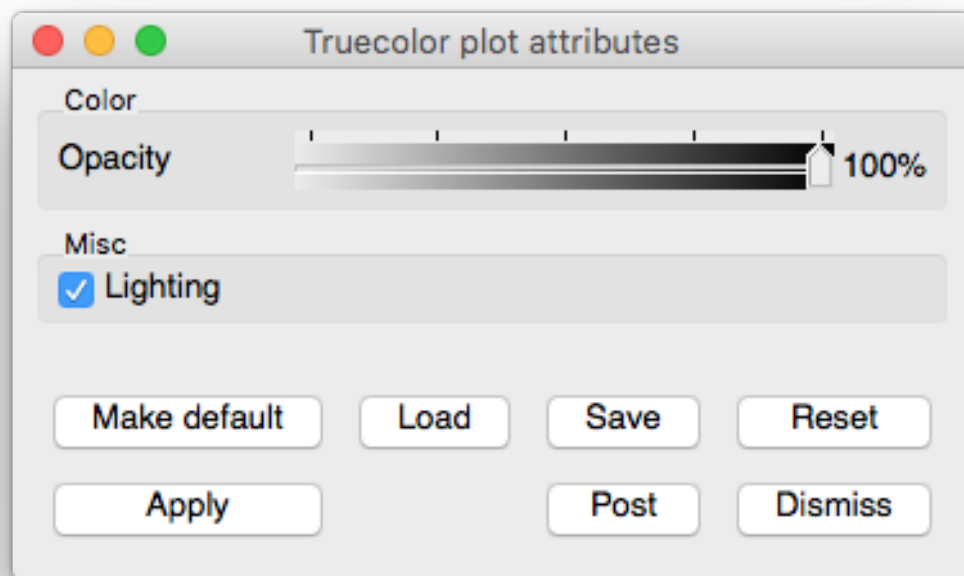
Fig. 1.62: Truecolor Plot Attributes

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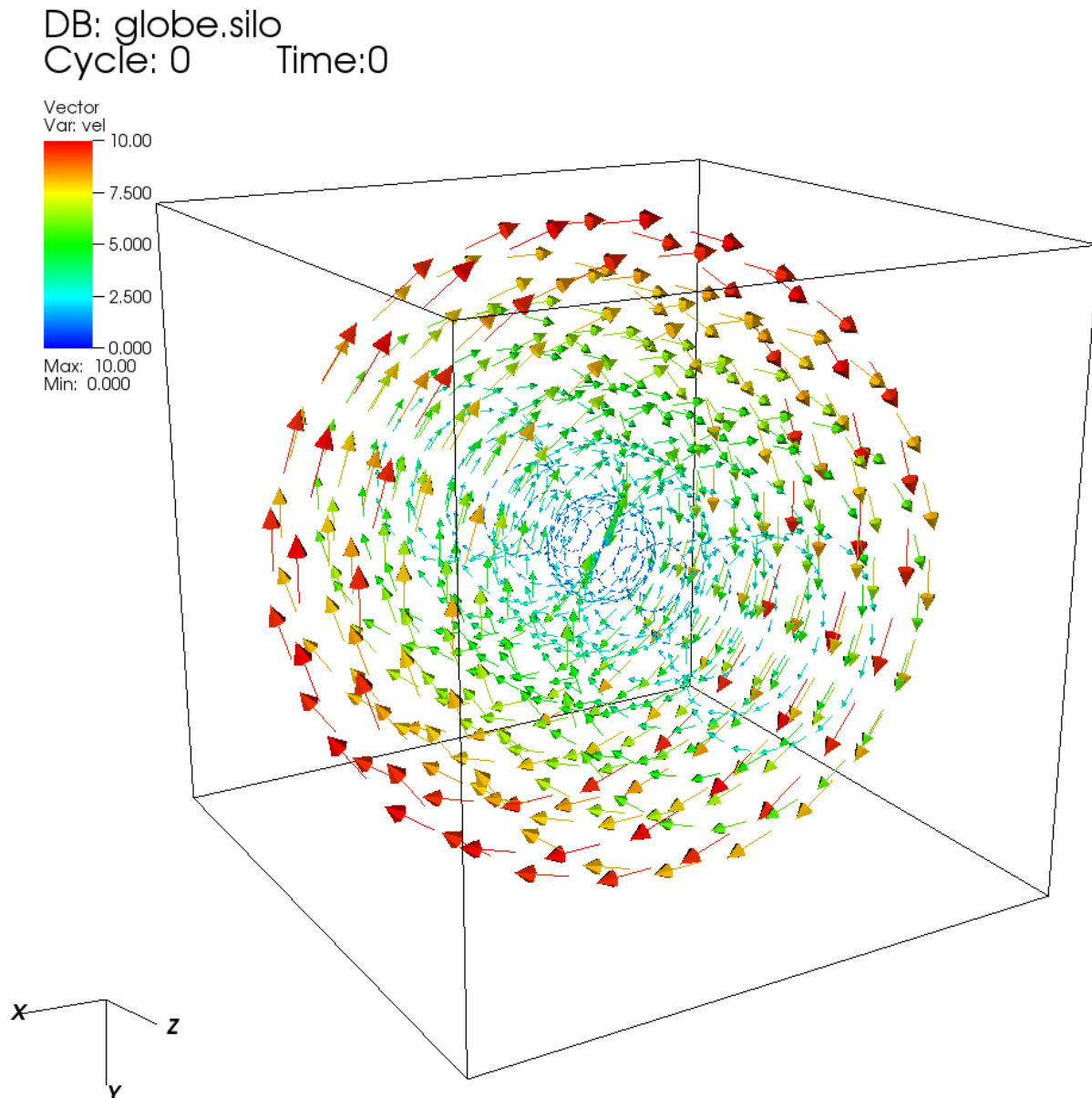
The Vector plot (example shown in Figure **??**) displays vector variables using glyphs that indicate the direction and magnitude of vectors in a vector field.

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
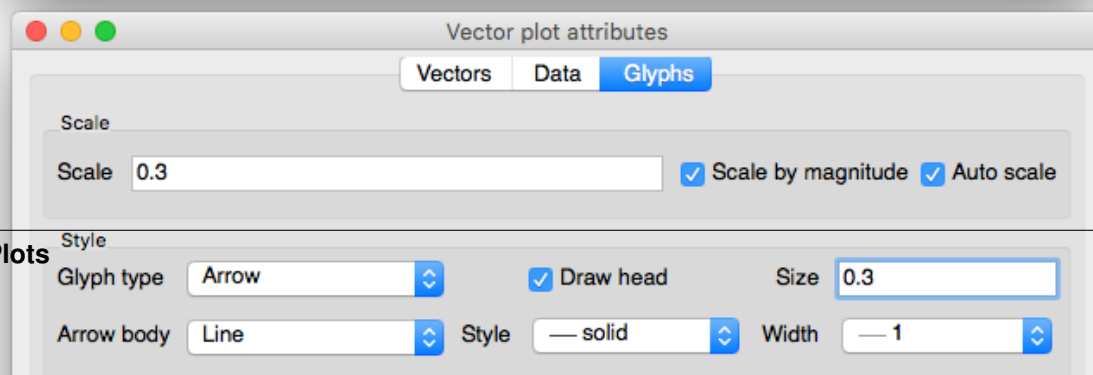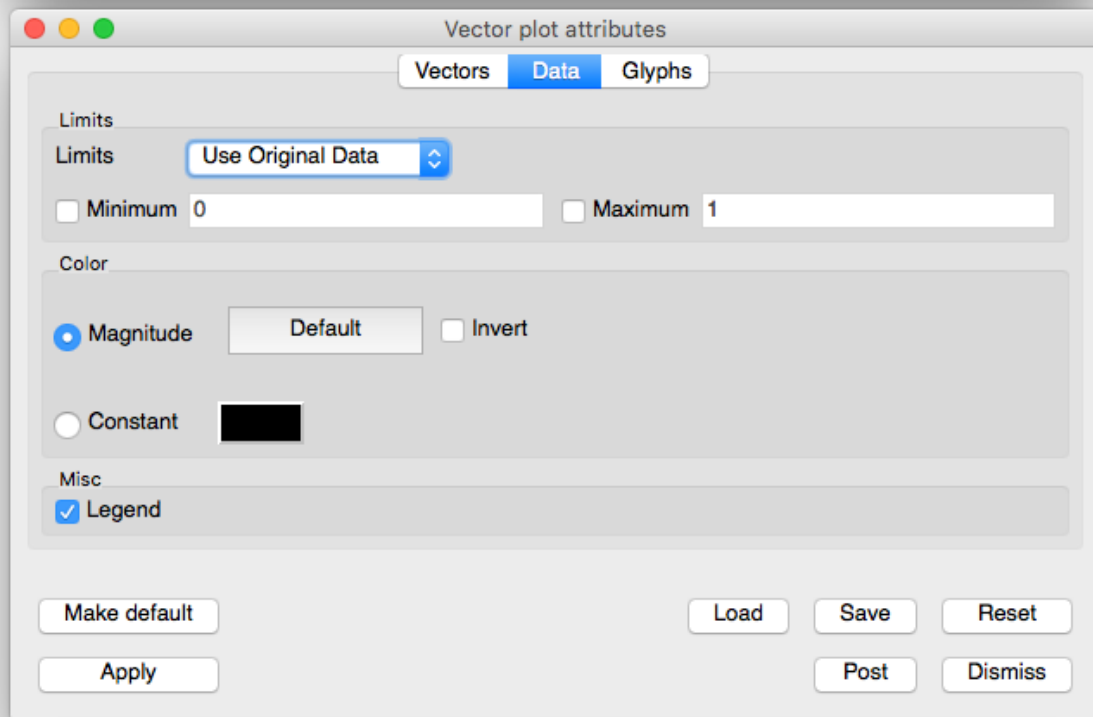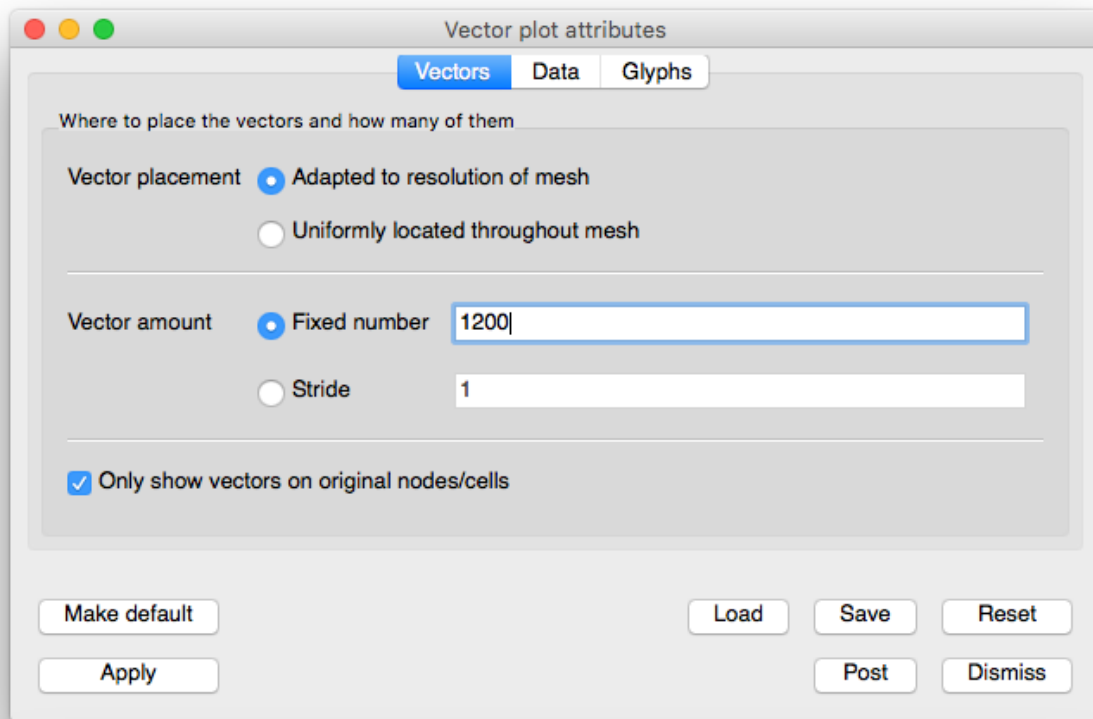
button. When vectors are colored by a constant color, you can change the color by clicking on the color button next to the **Constant** radio button and choosing a new color from the **Popup color menu**. When vectors are colored by magnitude, the color is determined by one of VisIt's color tables, which 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
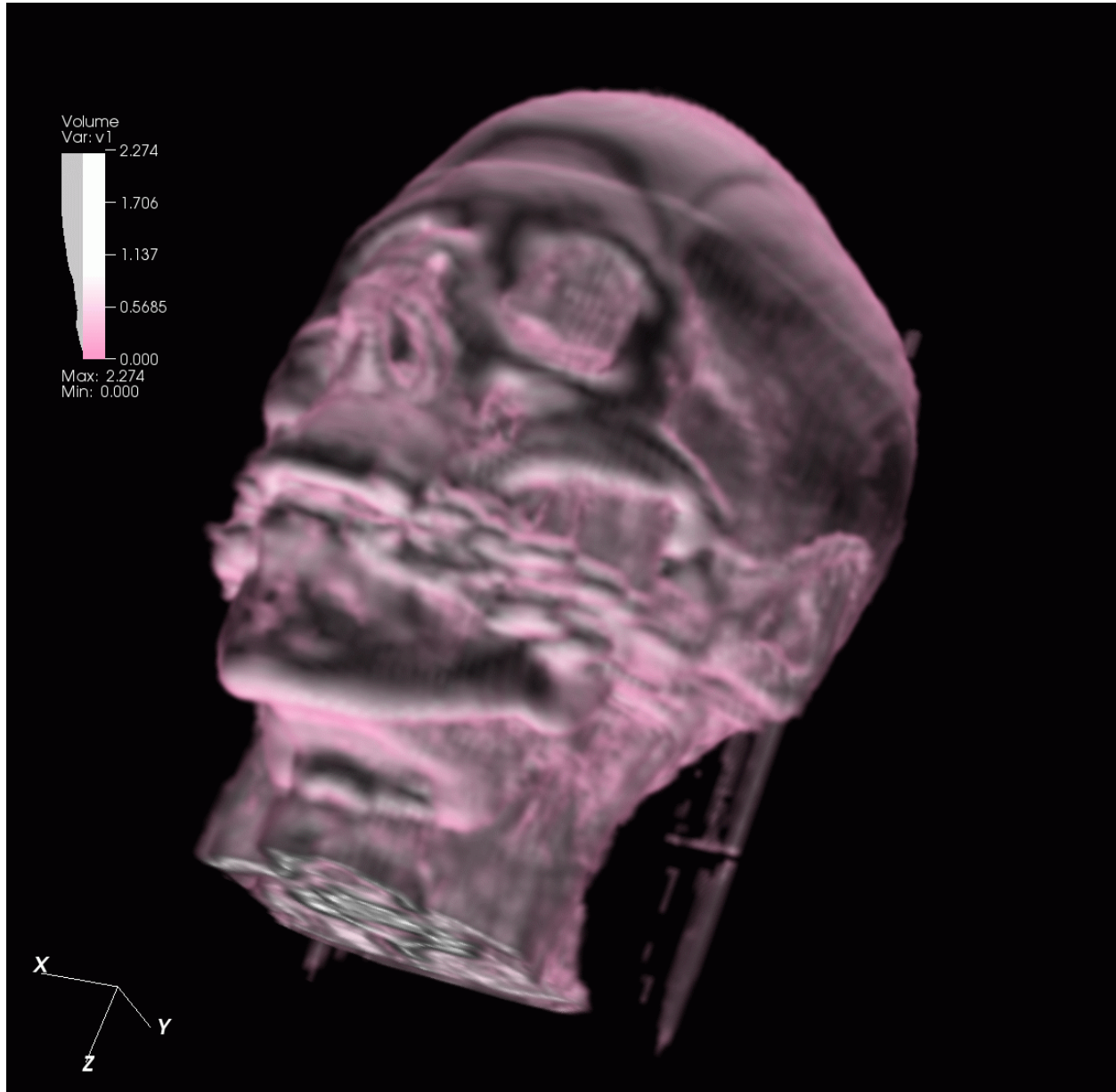
Fig. 1.64: Volume Plot Example

This plot, shown in (Figure 1.64), uses both color and transparency to visualize 3D scalar fields. The values in the data range have associated color and opacity values that allow parts of the dataset to become partially or completely transparent. This plot captures internal details while keeping the whole dataset at least partially visible.

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 Volume plot uses three types of volume-rendering to visualize data.

The first volume rendering method, hardware-accelerated splatting, resamples the entire database onto a small rectilinear grid and then, at each node in the grid, draws a small textured polygon. The polygon gets its colors and opacity from the transfer function. This method is fast due to its use of graphics hardware but it can require a large number of points in the resampled mesh to look accurate.

Like the first volume rendering method, the second method, hardware-accelerated 3D texturing, resamples the entire database onto a small rectilinear grid. Once the data has been resampled, it is converted into a 3D texture using the Volume plot's volume-transfer function and gets loaded into the video card's texture memory. The Volume plot then draws a set of planes that are perpendicular to the view vector from back to front, with each plane getting the pre-loaded texture mapped onto it. The resulting image is very crisp and captures details not evident when the splatting method is used.

The third volume-rendering technique, called ray-casting, used by the Volume plot is not hardware accelerated. 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.

The **Volume Attributes Window**, shown in (Figure **??**), is divided into two main tabs. The **Rendering Options** tab controls the rendering setting. Each volume rendering method has a different set of inputs. For example, splatting and 3D Texturing both specify the total number of samples, while ray-casting settings are specified in samples-per-ray. Additionally, the **Rendering Options** tab contains controls for lighting. **1D transfer function tab** controls how the data is mapped onto colors and the opacities to use for different scalar values.

Setting colors

You can design the color component of the volume transfer function using the controls in **1D Transfer Function** tab of the **Volume 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.

Limits

The **1D 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 are set by first clicking the **Min** or **Max** check box next to the **Min** or **Max** text field. Clicking a check box enables a text field into which you 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.
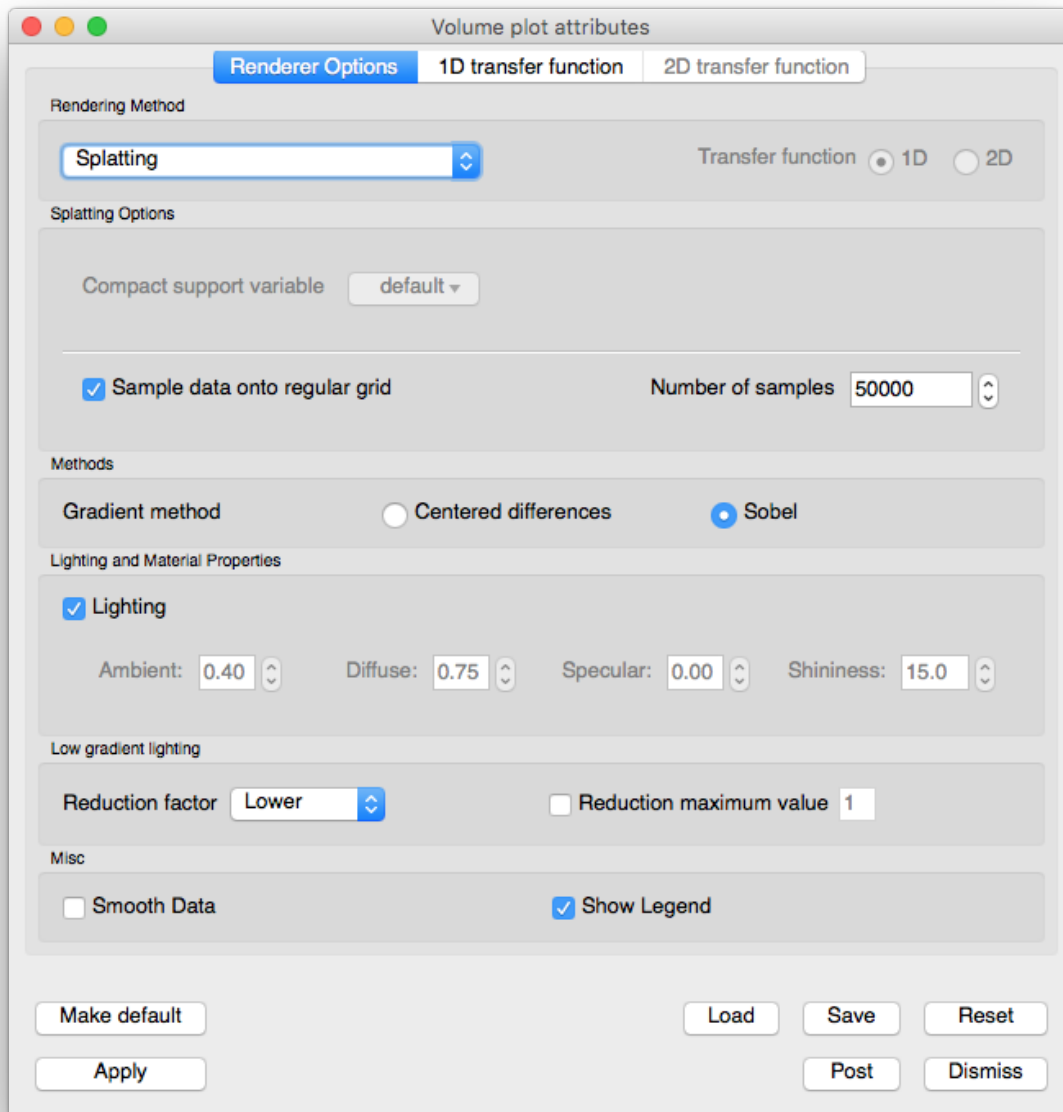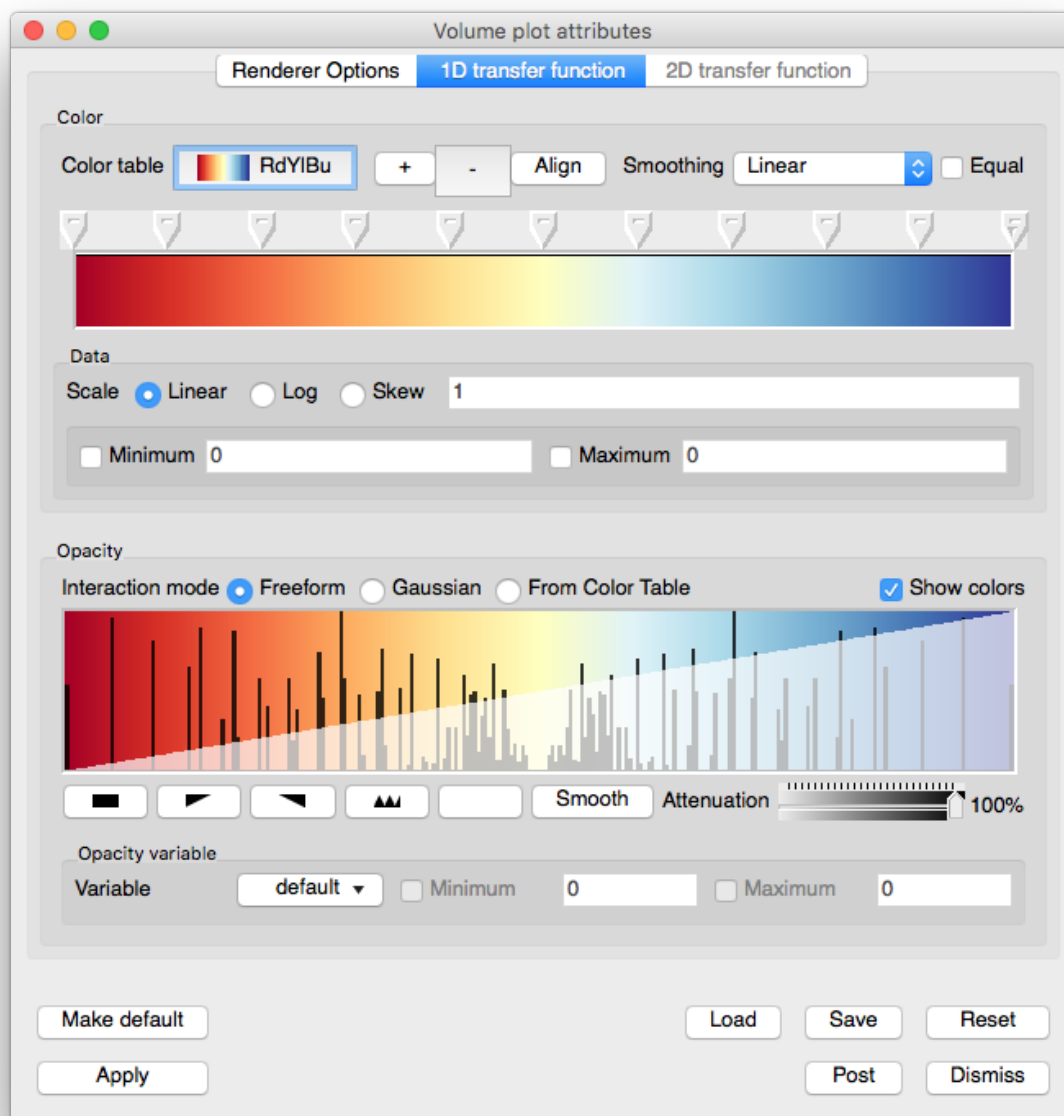
Fig. 1.65: Volume Attributes Window

Setting opacities

The **1D transfer function** tab provides several controls that allow you 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.
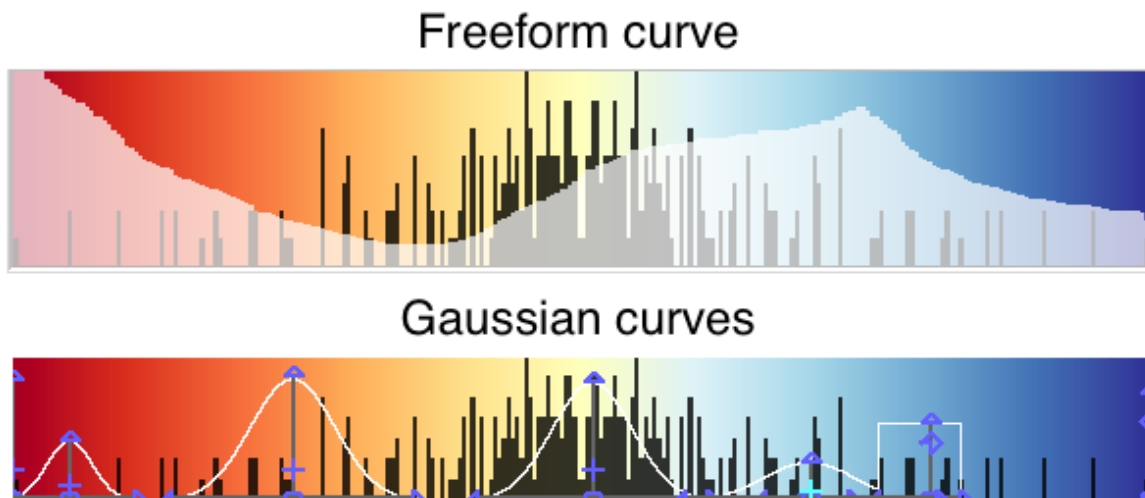
Fig. 1.66: Volume Plot Opacity Options

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 **??**.

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.
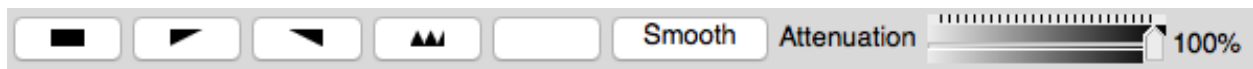
Fig. 1.67: Volume Plot Freeform Opacity Options

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 you can draw the desired opacity curve. Immediately under the **Freeform** control, there are four buttons, shown in (Figure 1.67), 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. 1.68: 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 in along with the shapes that a curve can assume. A control point changes color when it becomes active so there you know 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 you want 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 and slider. To increase the number of sample points, enter a larger number into the **Number of samples** text field or move the slider to the right. Note that the slider is on an exponential scale and moving it to the right increases the number of sample points exponentially.

In addition to setting the number of samples, when the Volume plot is using the 3D texturing method, you can set the number of planes to be drawn from back to front. Increasing the number of planes can help to reduce the amount of aliasing in the resulting image. However, as the Volume plot uses a higher number of planes, more work must be done to draw the plot and it takes a little longer to draw. To set the number of planes, enter a new number of planes into the **Number of slices** 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. You should adjust this number until you are 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.

Software rendered images

The Volume plot uses hardware-accelerated graphics by default. While you will want to operate in this mode most of the time, since it's faster, images drawn by software are more accurate. To get a more accurate image, select **Ray casting** from the **Rendering method** combo box. When the Volume plot is set to use ray casting as its rendering mode, VisIt recalculates what the image should look like in software mode. Note that this can be a time-consuming process if the database being used is large or if the visualization window is large. We recommend shrinking the size of the visualization window before changing the rendering method to ray casting to reduce the time and resources required to draw the plot. It is worth noting that if you have a large dataset with intricate details, the software volume rendering method is the best method to use 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 equal-sized pieces.

Lighting

The Volume plot can use lighting to enhance the look of the plot. Lighting is enabled by default but you can disable it by unchecking the **Lighting** check box near the bottom of the window. Note that lighting is not currently available when the Volume plot is using the ray casting volume renderer.

1.4 Operators

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

1.4.1 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. See the wiki for more details on creating new operator plugins or send an e-mail inquiry to visit-users@elist.ornl.gov.

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.
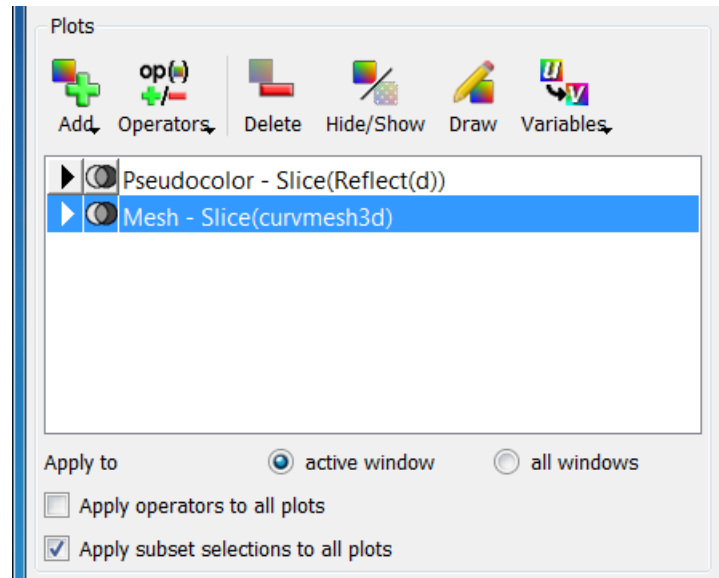
Fig. 1.69: The plots area

Adding an operator

Operators are added by selecting an operator from the **Operators** menu, shown in Figure 1.70. 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 1.71).

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 1.72. 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.
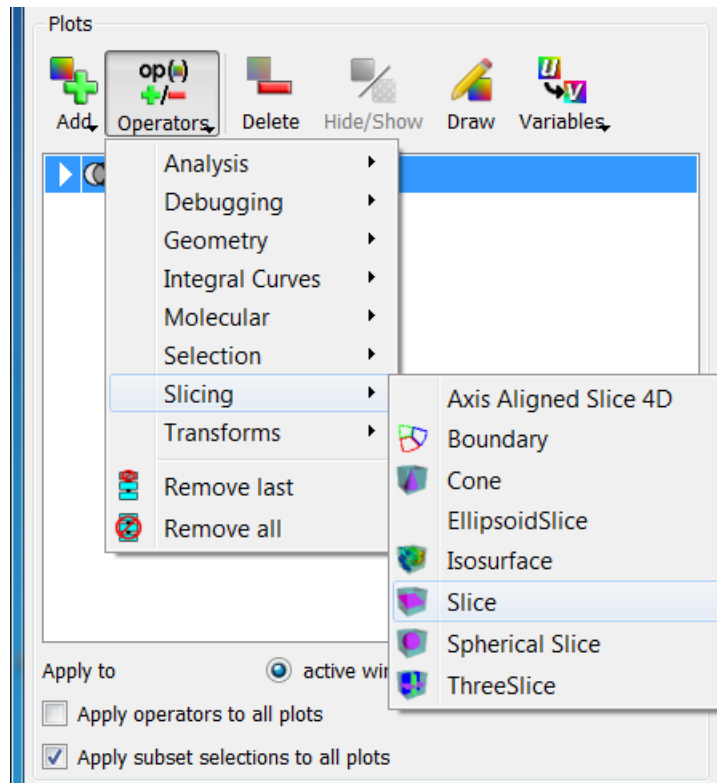
Fig. 1.70: The operators menu
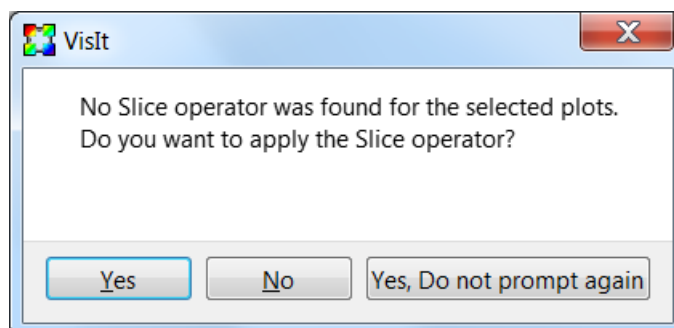
Fig. 1.71: The add operator dialog
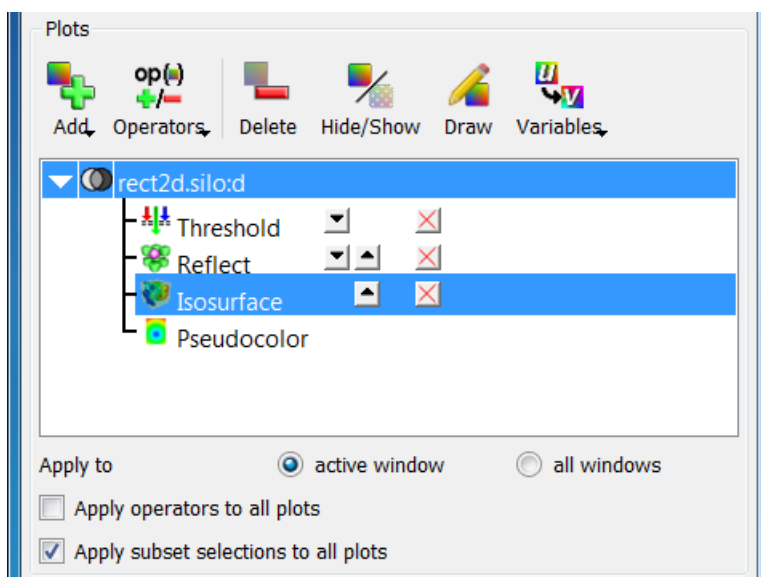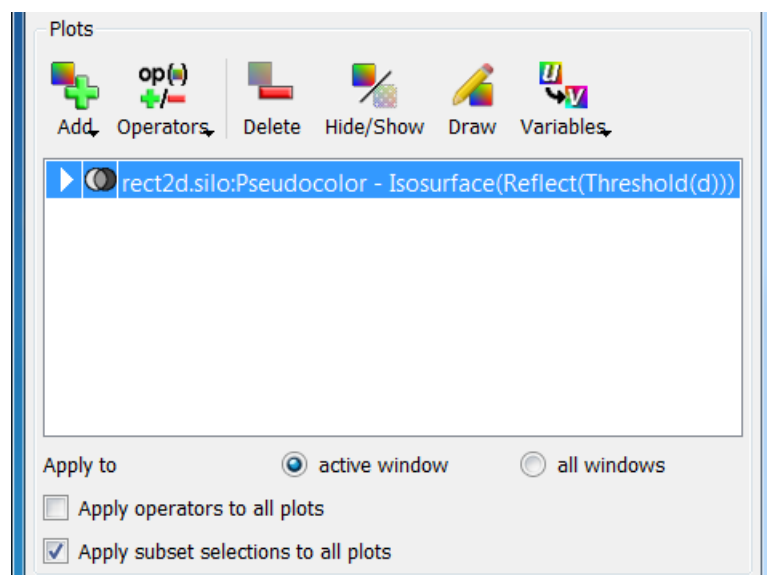
Fig. 1.72: A plot list entry before and after being expanded

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.
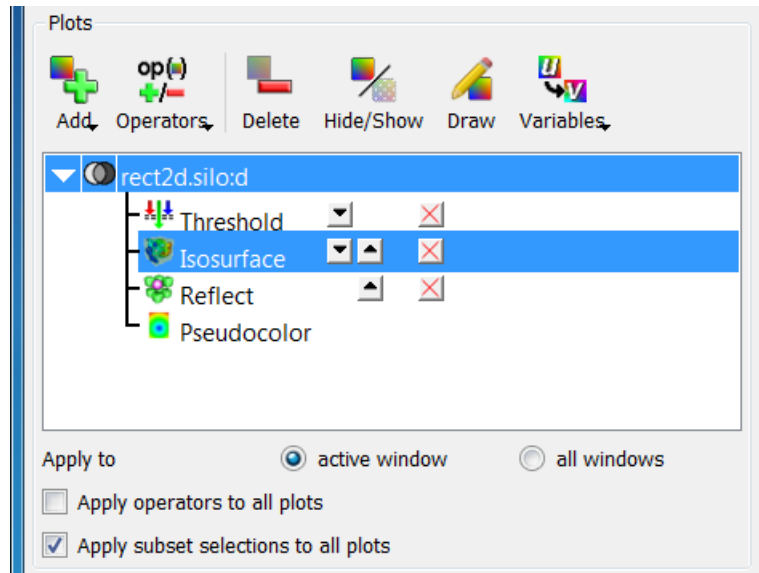
Fig. 1.73: The controls for changing operator order

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 1.74 for an example of deleting an operator from the middle of a plot's operator list.

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 1.75.

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
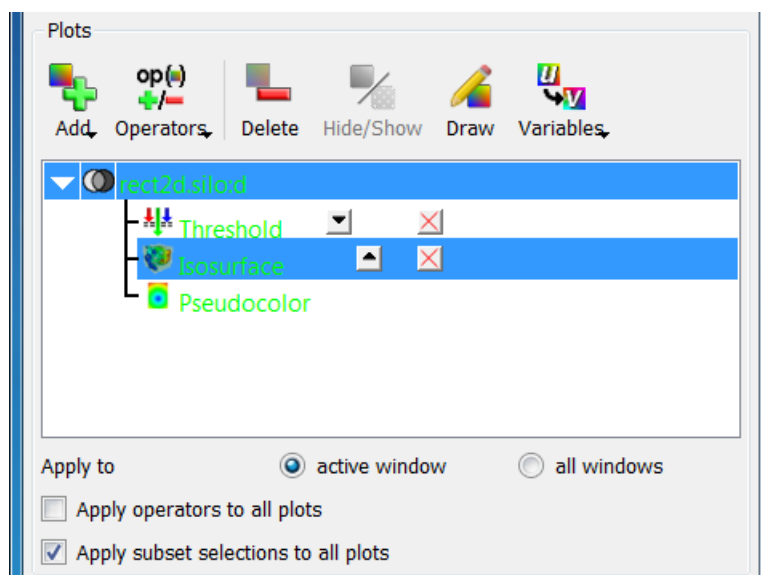
Fig. 1.74: After removing an operator from the middle of the pipeline
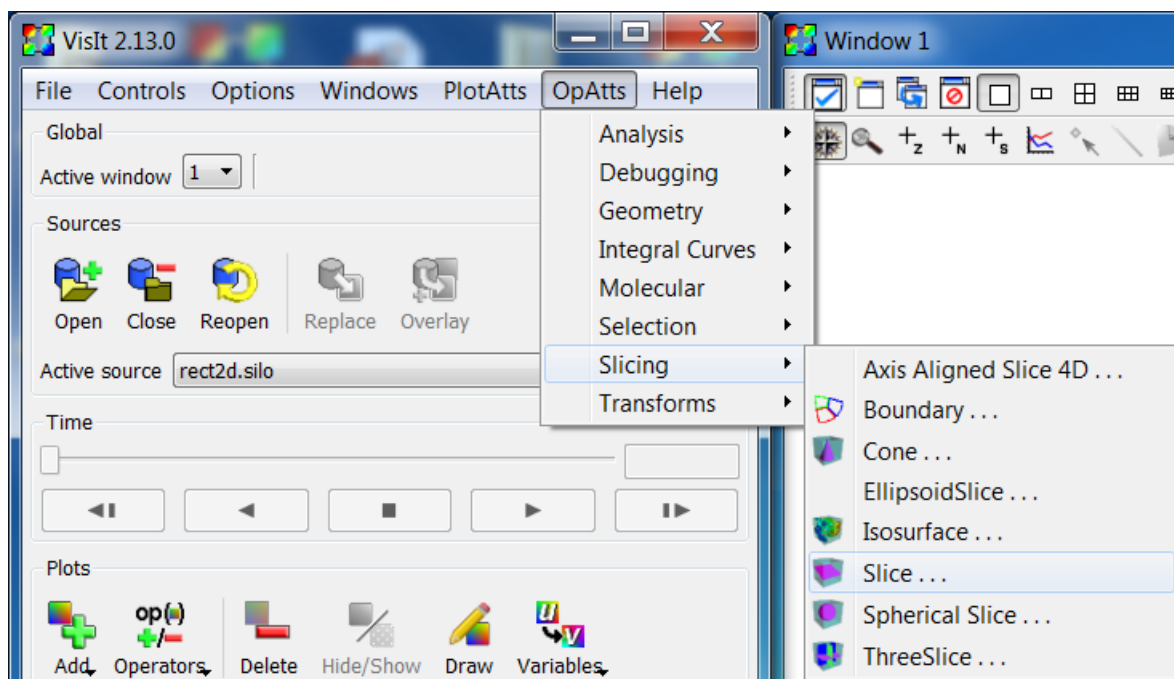
Fig. 1.75: The operator attributes menu

be identified in an expanded plot entry by the highlight that is drawn around it (see Figure 1.76). To set the active operator, expand a plot entry and then click on an operator in the expanded plot entry's operator list.
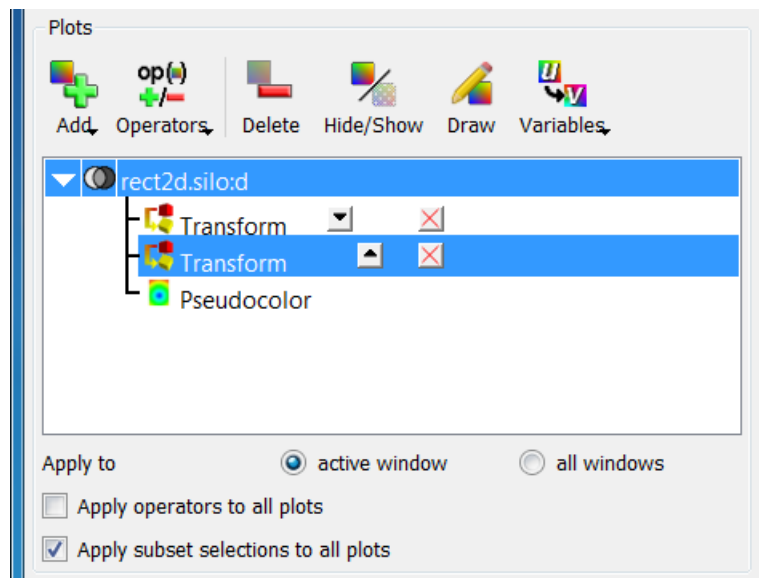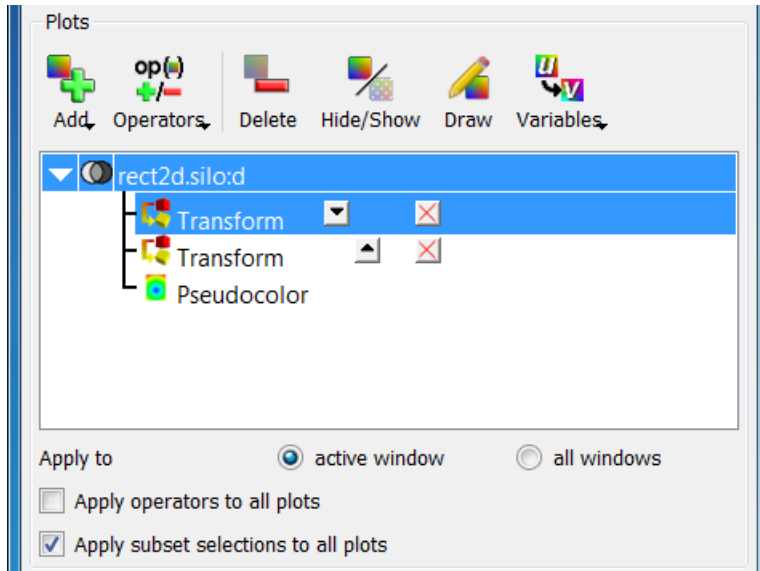
Fig. 1.76: Setting the active operator

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.

1.4.2 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

---

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 1.77,
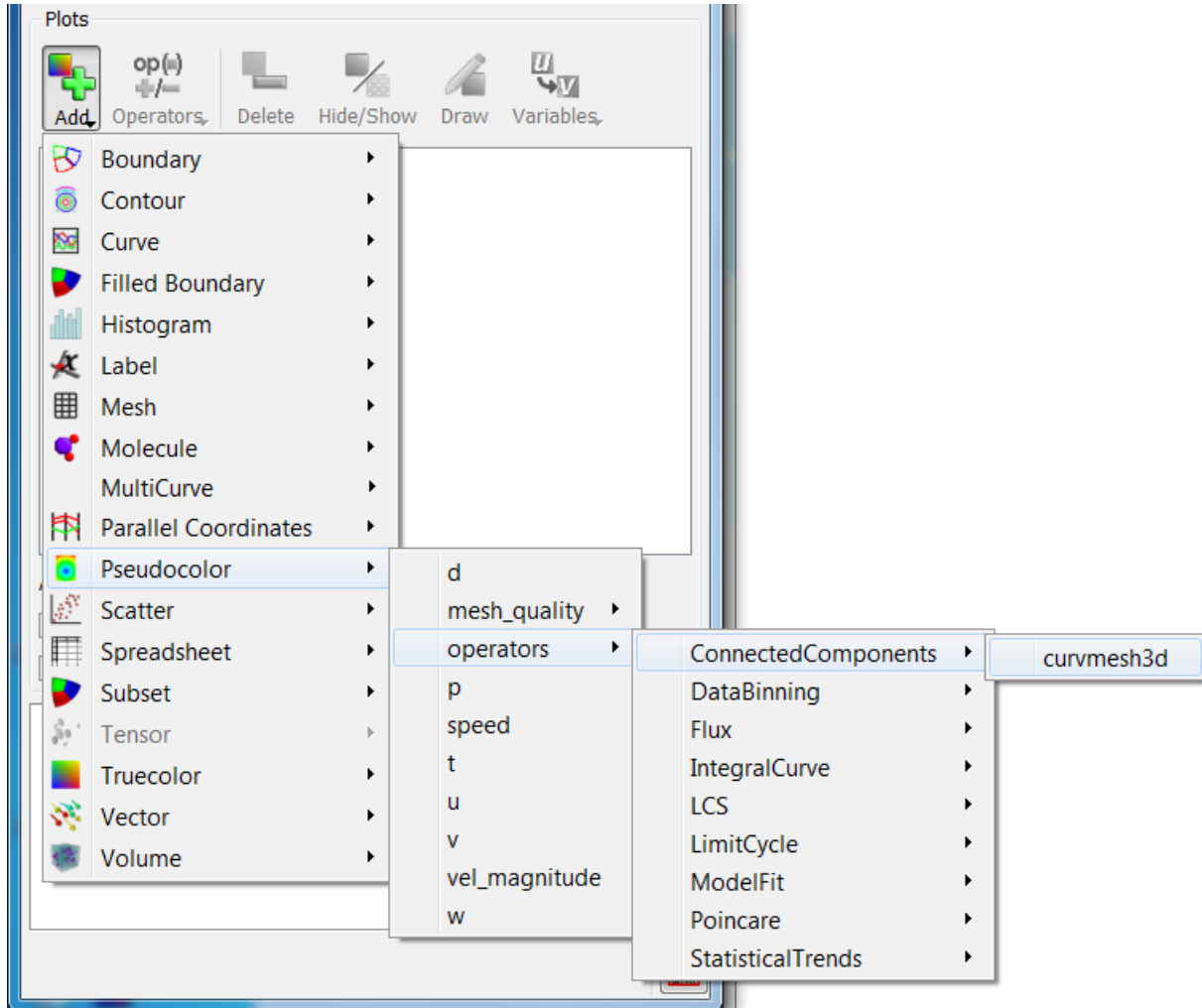
Fig. 1.77: The menu for applying an operator that generates a new variable.

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. IntegralCurve

5. LCS

6. LimitCycle

7. ModelFit

8. Poincare

9. StatisticalTrends

Operators that generate Vectors:

1. LCS

2. SurfaceNormal

Operators that generate Curves:

1. DataBinning

2. Lagrangian

3. LimitCycle

4. Lineout

1.4.3 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.
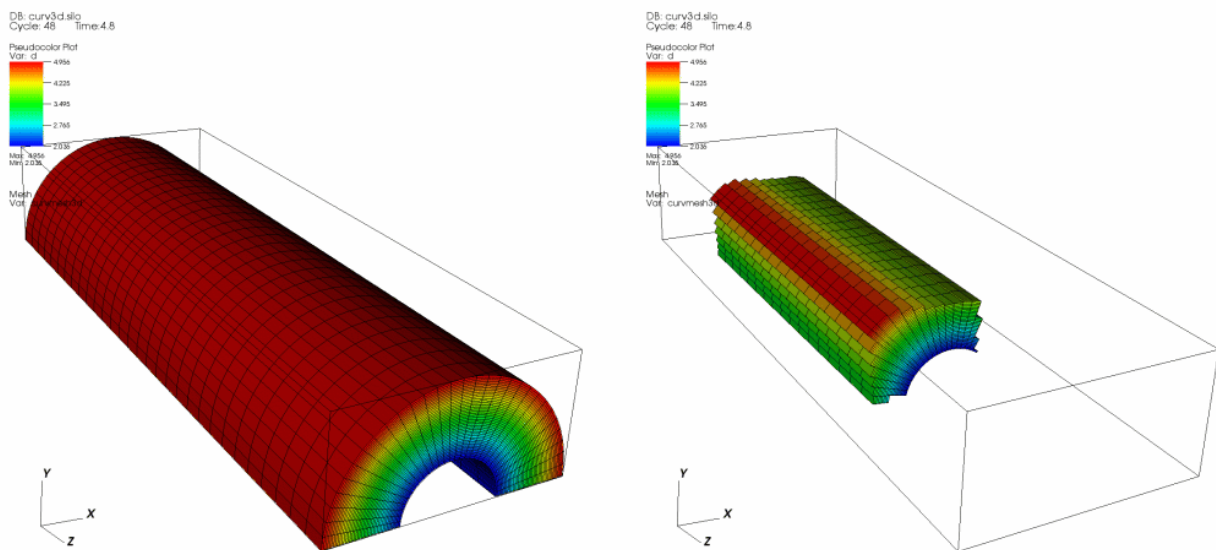
Fig. 1.78: Box operator example (original on left, with Box operator applied on right)

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 1.78). Selecting the **Inverse** option will return everything in the mesh except those cells bounded by the selected box.
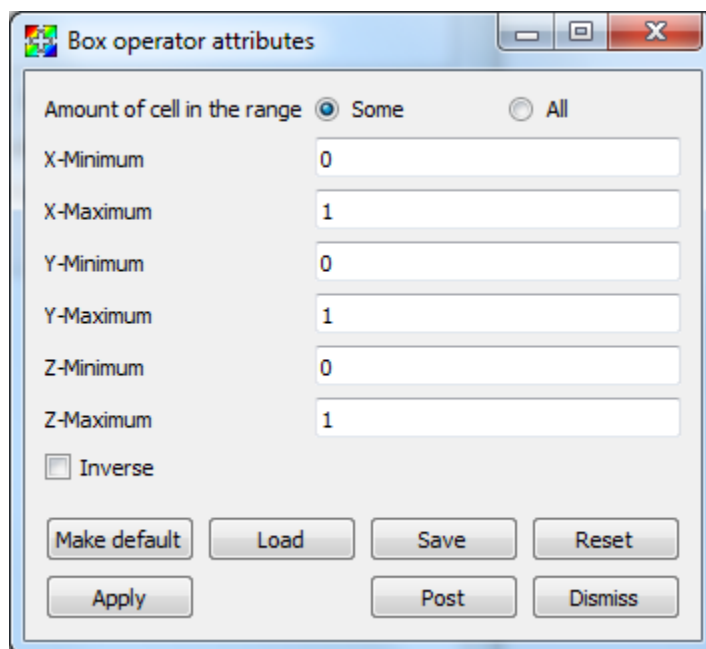
Fig. 1.79: Box attributes window

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 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 1.80 shows a Pseudocolor and Mesh plots with a Clip operator.
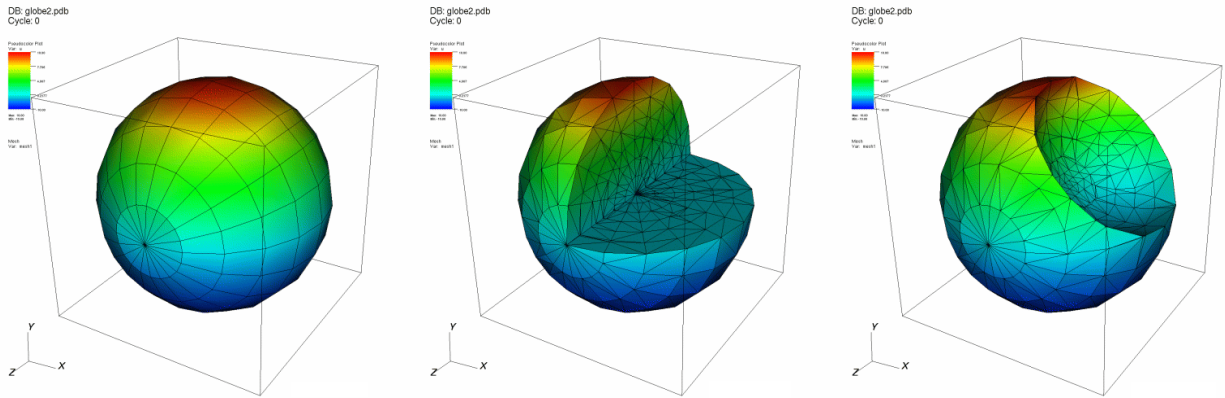
Fig. 1.80: Clip operator example: original plot; clipped with planes; clipped with sphere

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 1 then click the **Plane 1** check box for plane 1 (see Figure 1.81). 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 1.82. 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.
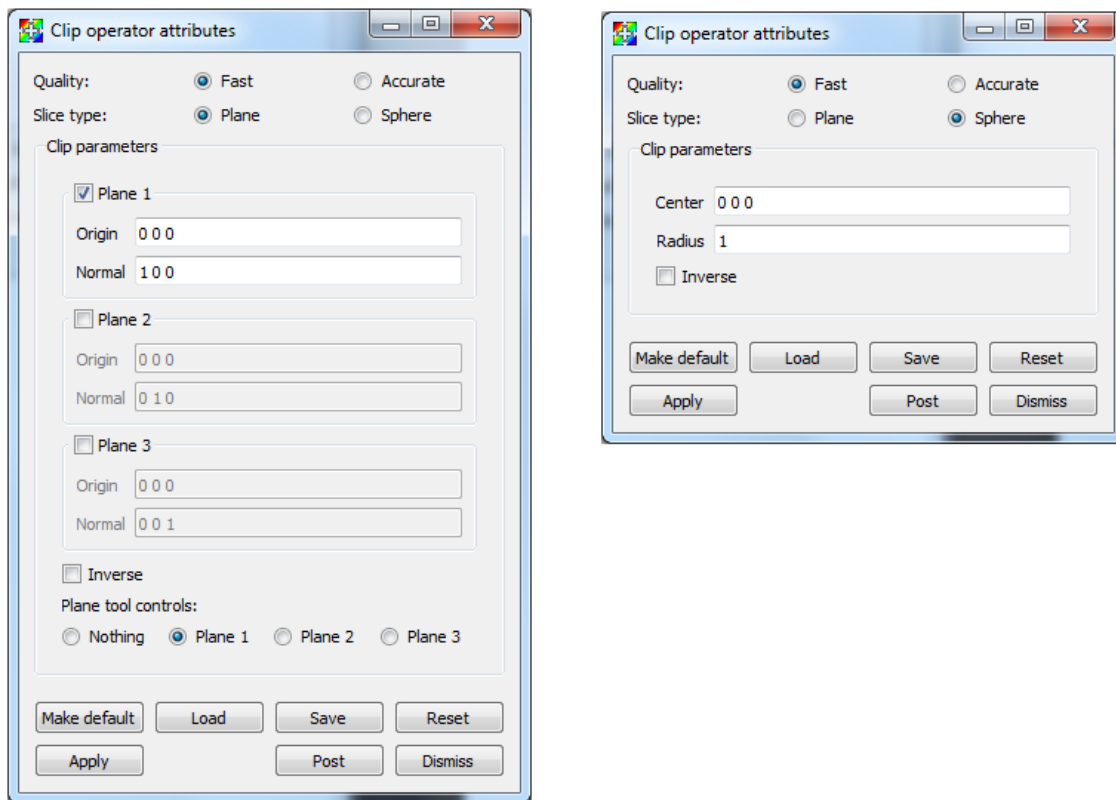
Fig. 1.81: Clip attributes window
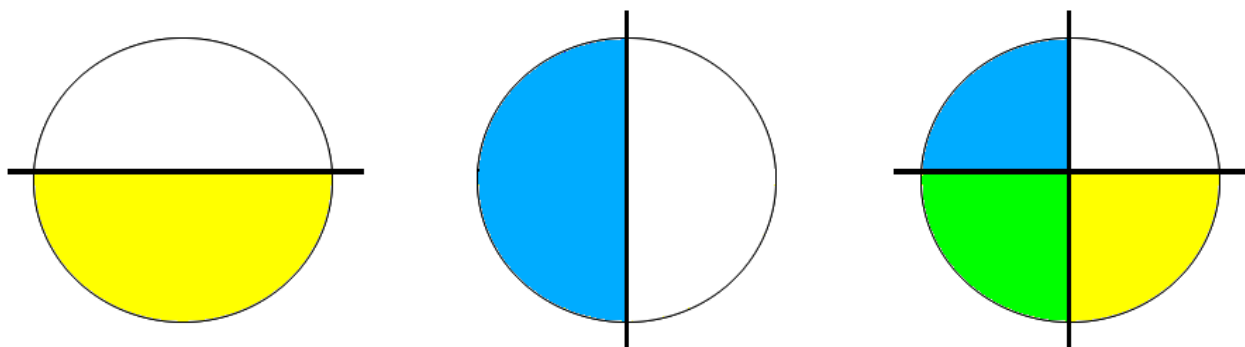
Fig. 1.82: Removing one quarter of a plot using two clip planes: Plane1 clipped region + Plane2 clipped region = One quarter removed

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.

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 1.83.
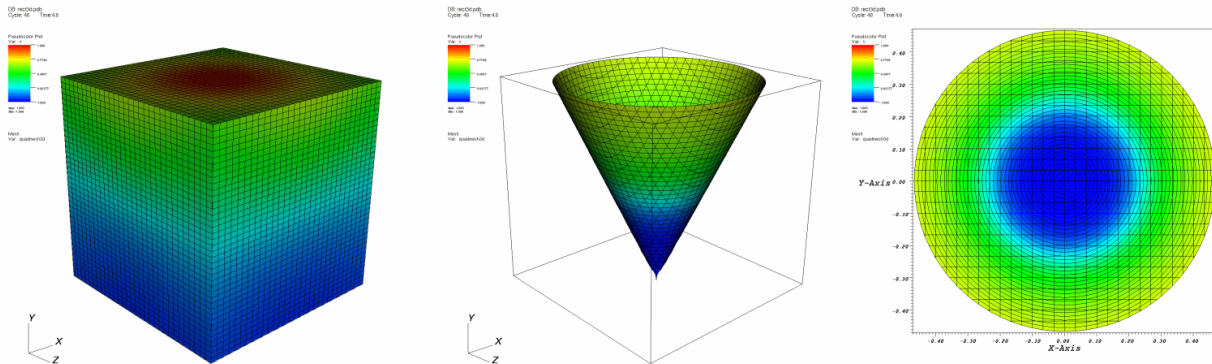
Fig. 1.83: Cone operator example: original plot; sliced with cone; sliced with cone and projected to 2D

Specifying the slice cone

You can specify the slice cone by setting various fields in the **Cone attributes window**, shown in Figure 1.84. 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.

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
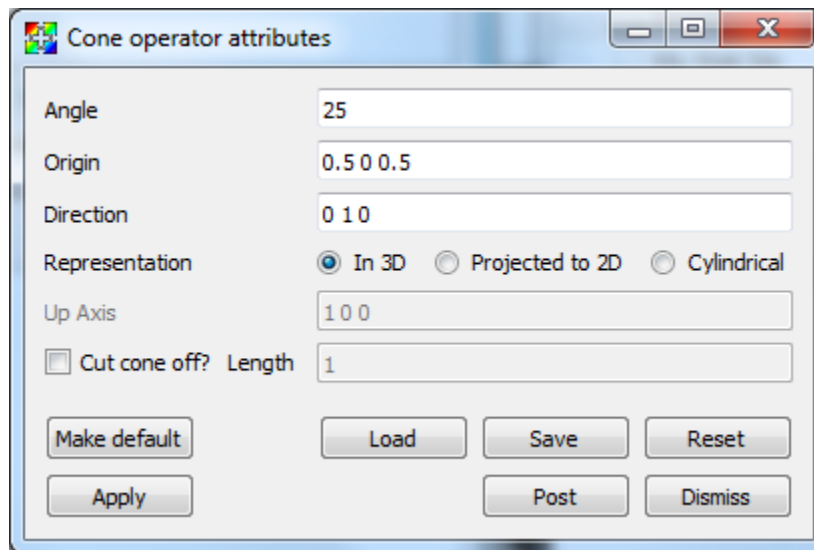
Fig. 1.84: Cone operator window.

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 1.85,

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 1.86

Cylinder operator

The Cylinder operator, shown in Figure 1.87, slices a dataset with a cylinder whose size and orientation are specified by the user. The result is a cylindrical surface.

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 1.88) 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.
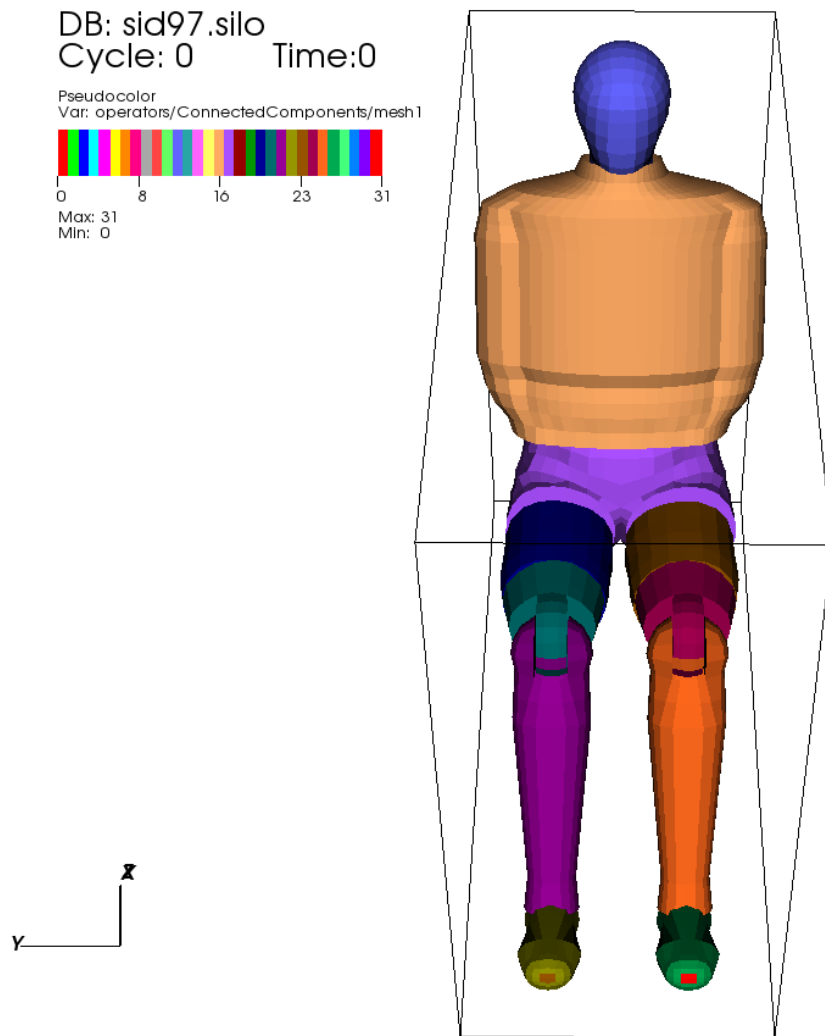
Fig. 1.85: Connected Components operator shown with Pseudocolor Plot.
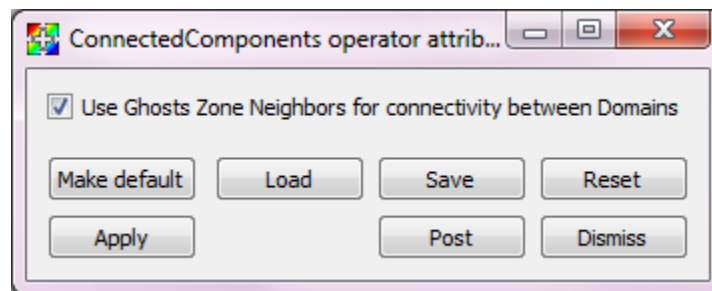
Fig. 1.86: Connected Components operator window.
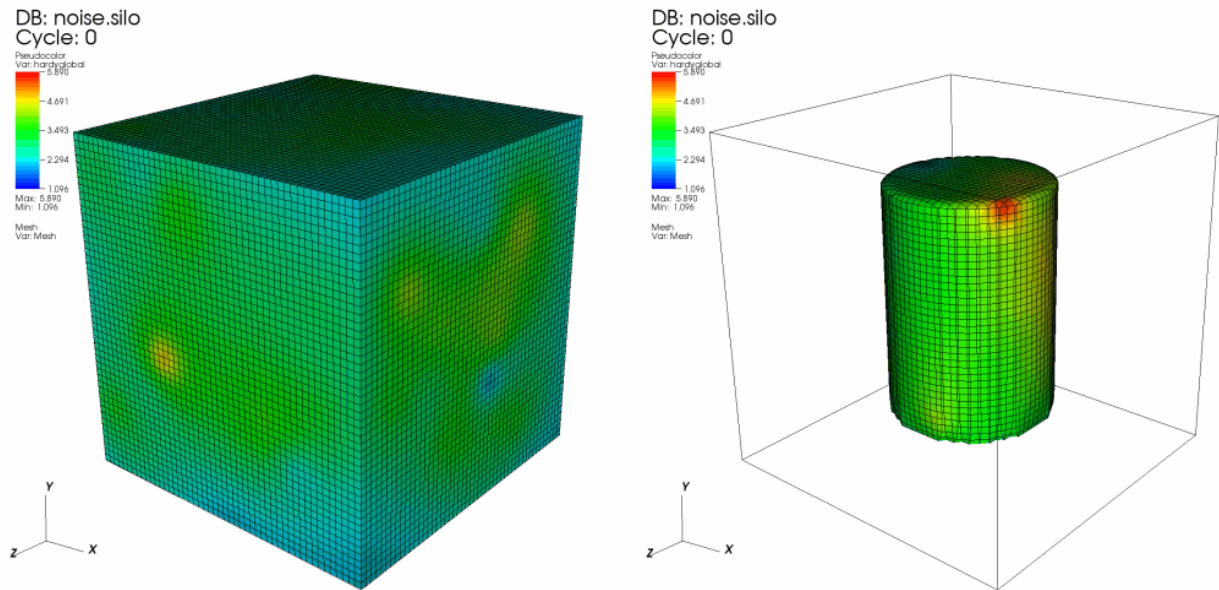
Fig. 1.87: Cylinder operator example: original plot; plot clipped by cylinder

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** .
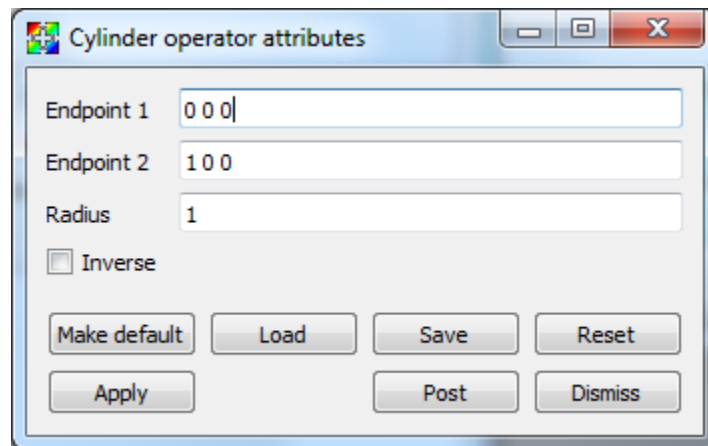
Fig. 1.88: Cylinder operator window.

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.
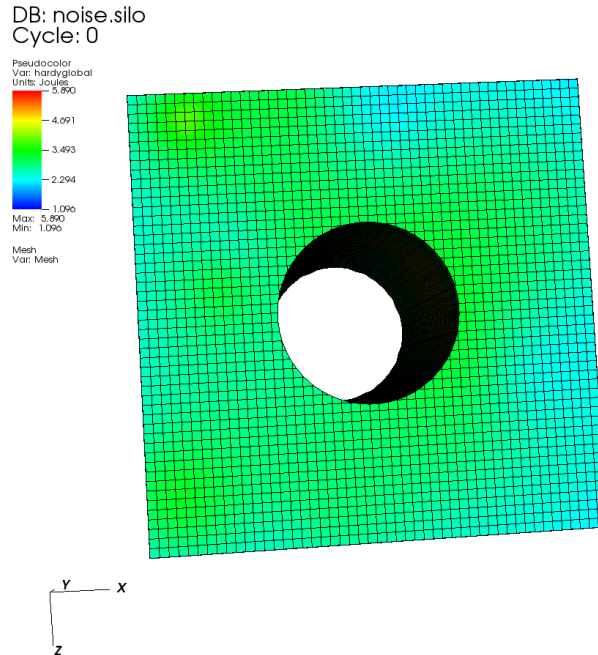
Fig. 1.89: Cylinder with inverse applied

Decimate operator

The Decimate operator, shown in Figure 1.90, 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 1.91). 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 1.90, higher values for **Target Reduction** value cause VisIt to simplify the mesh even more.
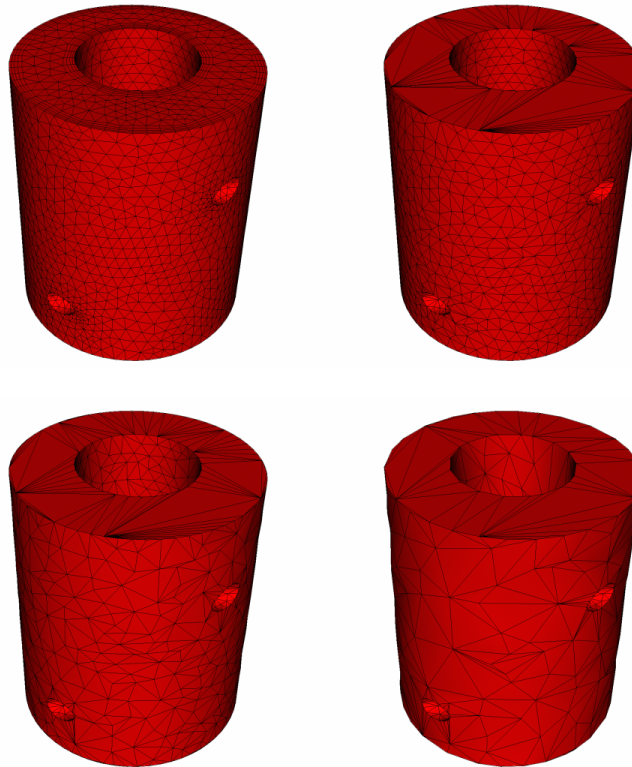
Fig. 1.90: Decimate operator applied to reduce the number of cells in the mesh. (Left-to-right, top-to-bottom): Original Mesh, Reduction = 0.1, Reduction = 0.5, Reduction = 0.75
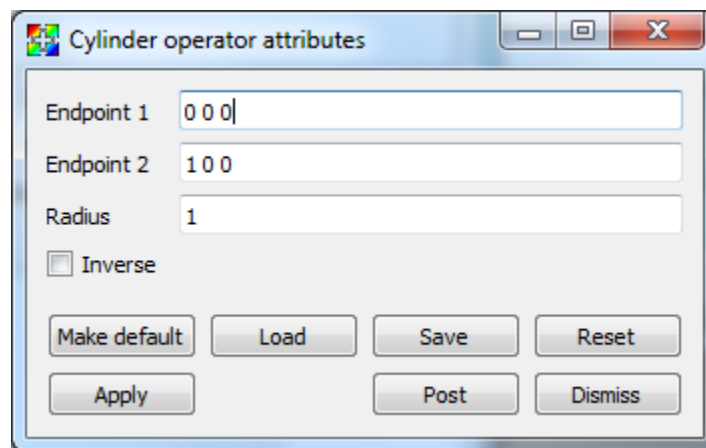
Fig. 1.91: Decimate attributes window

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 execution taking place before any operators are applied, expression execution can instead take place after operators have been applied.

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.
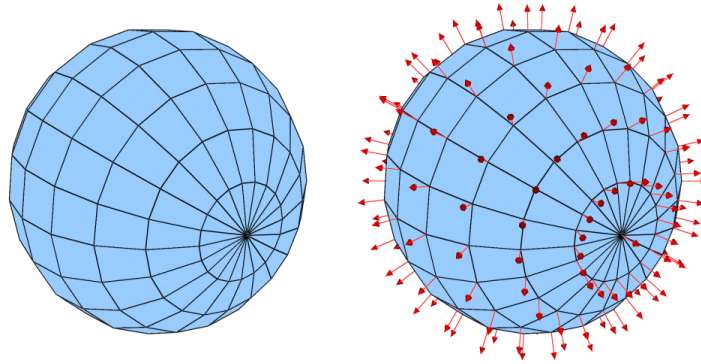
Fig. 1.92: DeferExpression operator example

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 1.93.
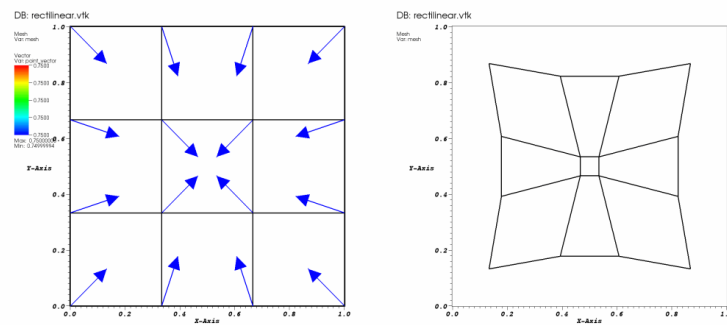
Fig. 1.93: Mesh and Vector plots and a Mesh plot that uses the Displace operator to deform the mesh using a vector field.

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 1.94). 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.
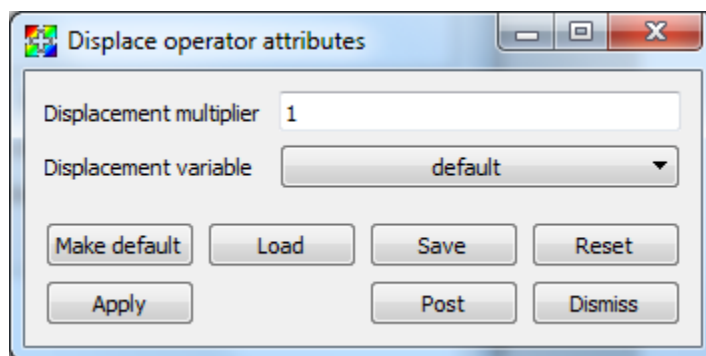
Fig. 1.94: Displace attributes window

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 1.95).
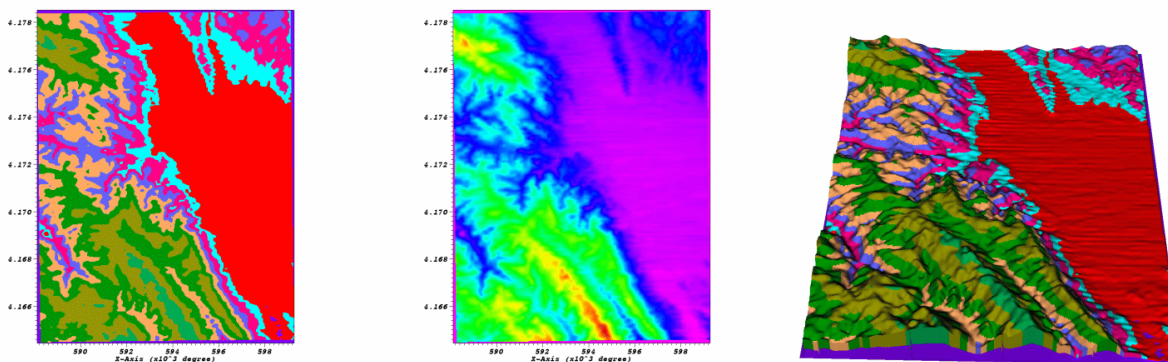
Fig. 1.95: Elevate operator example: 2D plot of rainfall; 2D plot of elevation; Plot of rainfall elevated by elevation

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 1.96) 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.
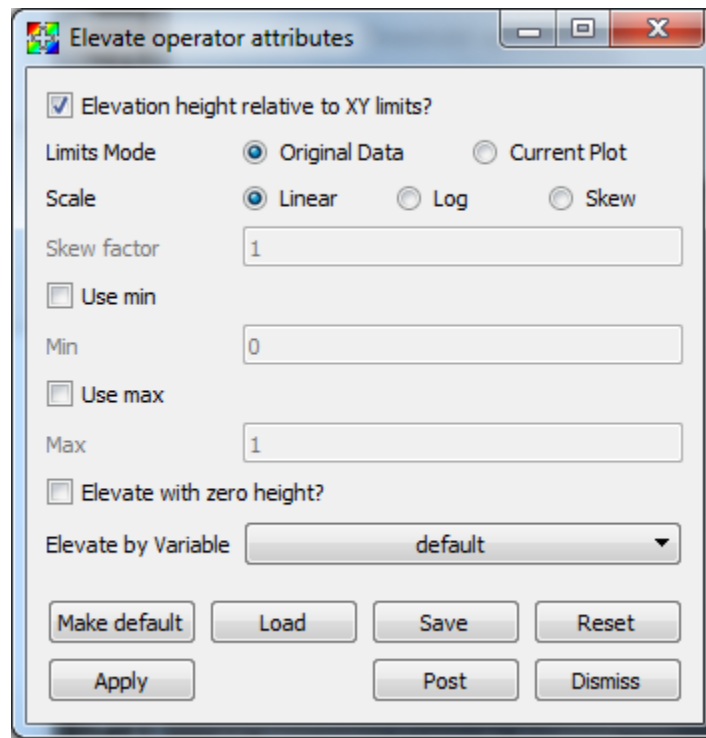
Fig. 1.96: Elevate window

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**.
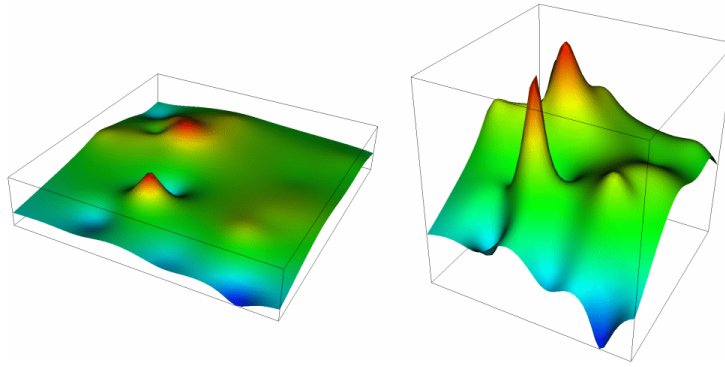
Fig. 1.97: Effect of scaling relative to XY limits

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.
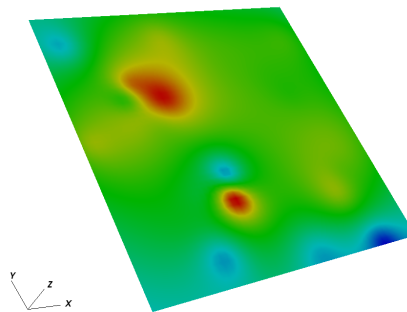
Fig. 1.98: Effect of elevating with zero height

Explode operator

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.
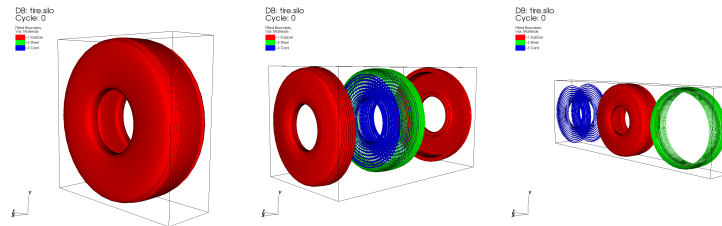
Fig. 1.99: Explode operator example: original plot; exploding cells of a material; exploding materials.
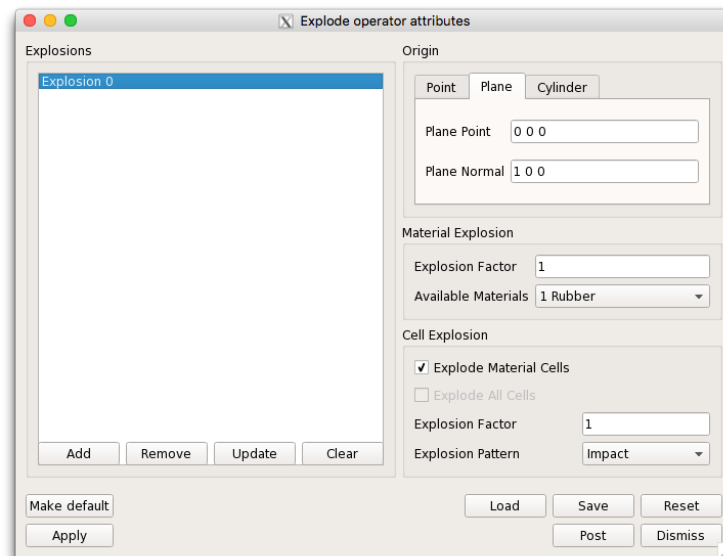
Fig. 1.100: Explode attributes window

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 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 1.99, 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 1.99, 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.

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
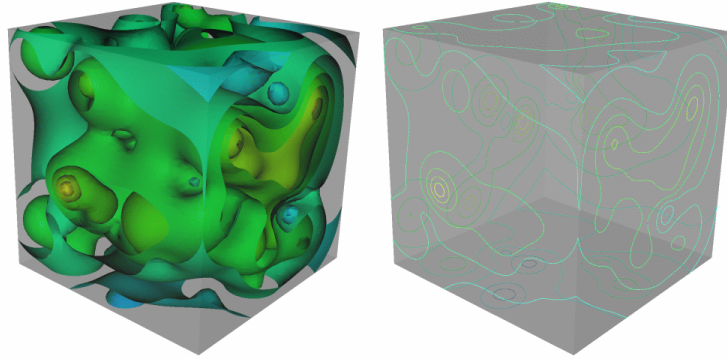
Fig. 1.101: ExternalSurface operator example

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 1.102.
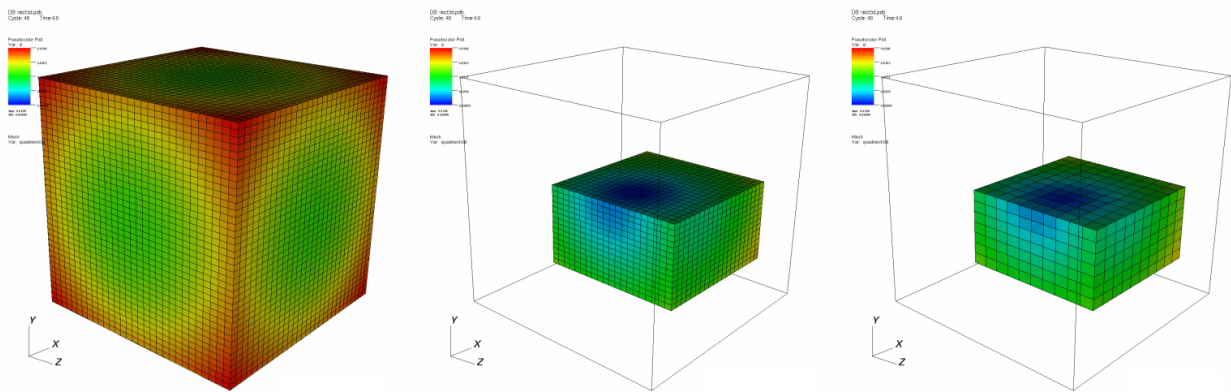
Fig. 1.102: Index Select operator example: original plot; index selected (stride=1); index selected (stride=2)

Setting a selection range

The **Index Select attributes window**, shown in Figure 1.103, 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.
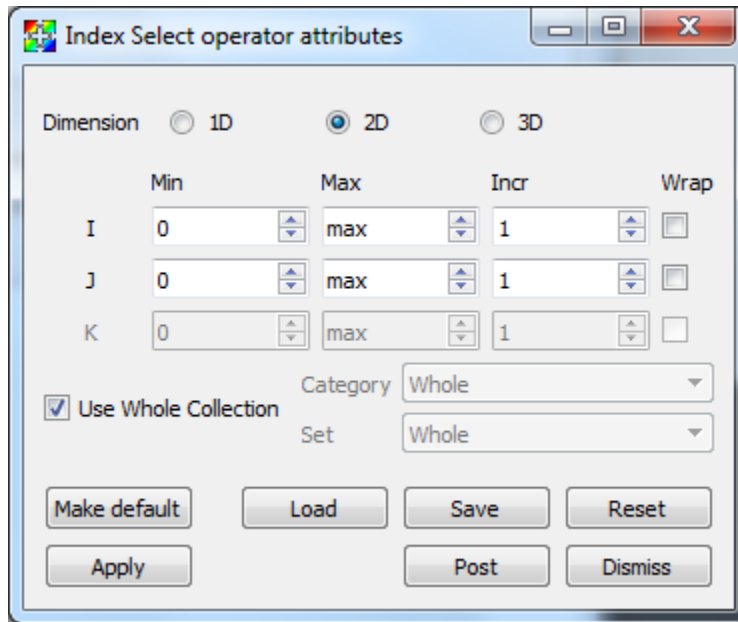
Fig. 1.103: Index Select attributes window

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 **??** 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 1.105) has various **Show** options allowing you to select which types of ghost cells are returned. By default all options are turned on.

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
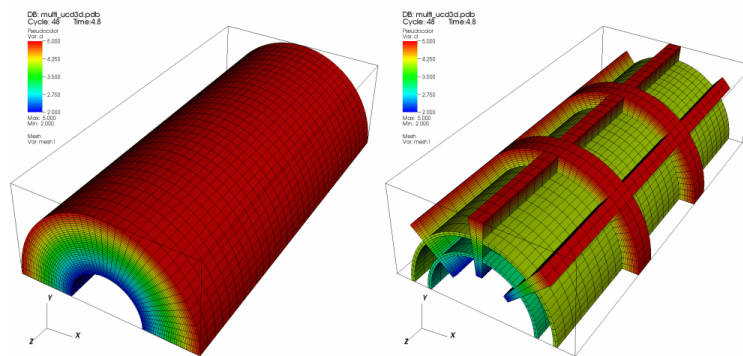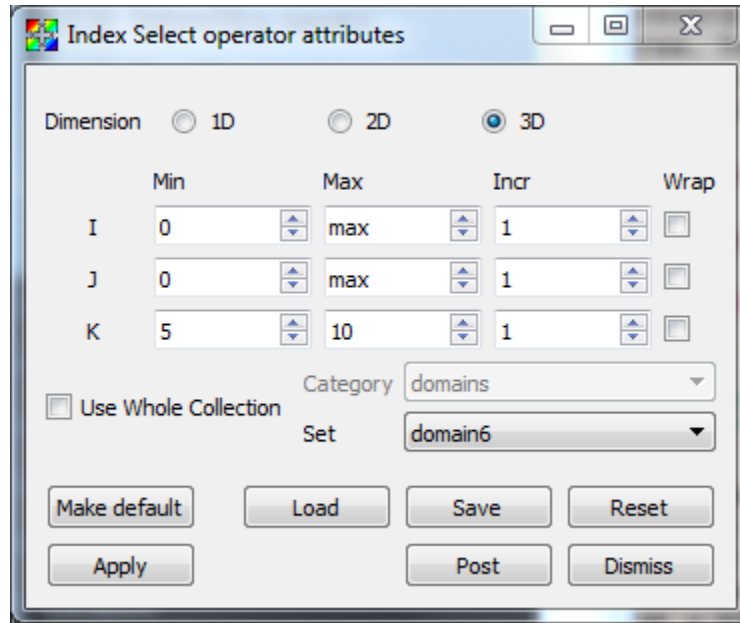
Fig. 1.104: InversetGhostZone example
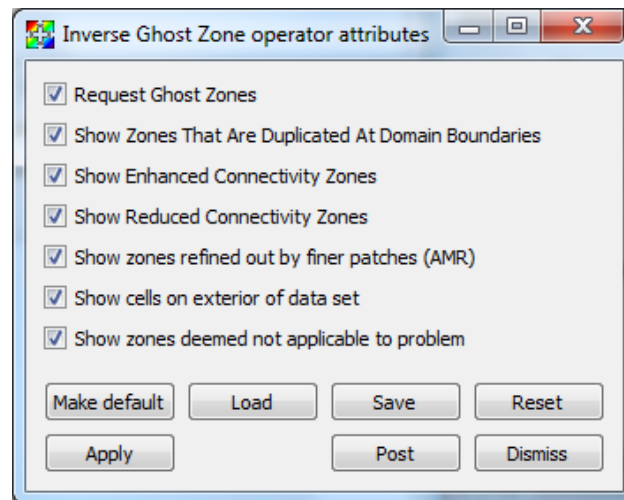
Fig. 1.105: InversetGhostZone window

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 1.106.
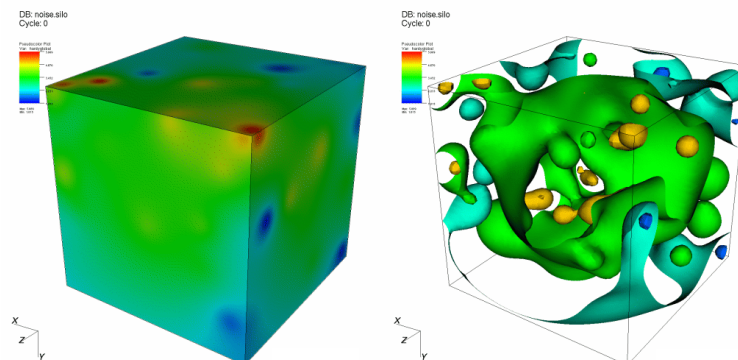
Fig. 1.106: Isosurface operator example

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.
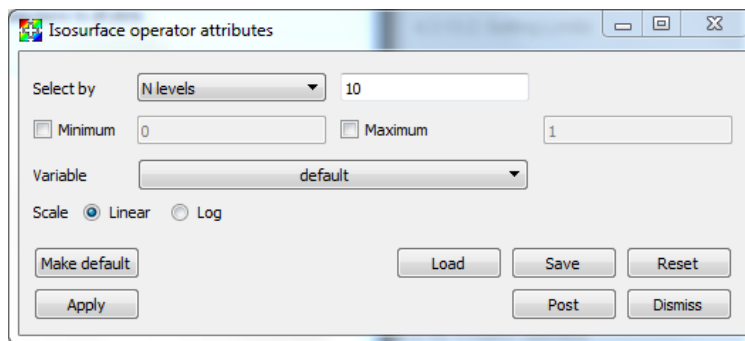
Fig. 1.107: Isosurface attributes

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 1.107, 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
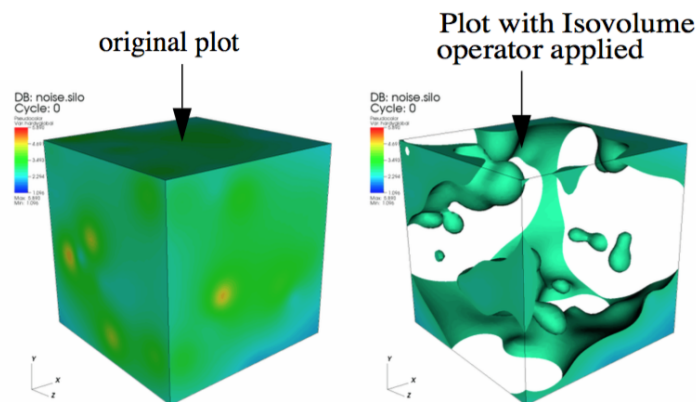
Fig. 1.108: Isovolume Operator Example

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
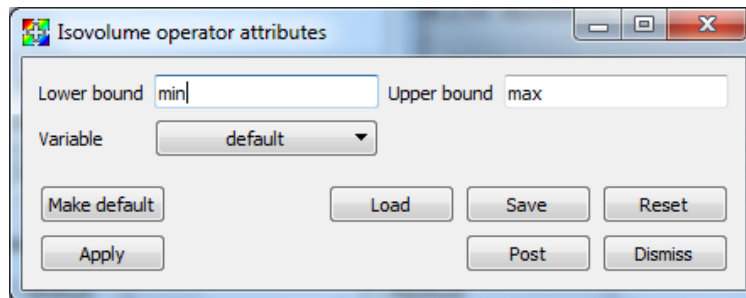
Fig. 1.109: Isovolume Attributes Window

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 1.109.

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 database from databases 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.

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 1.110.
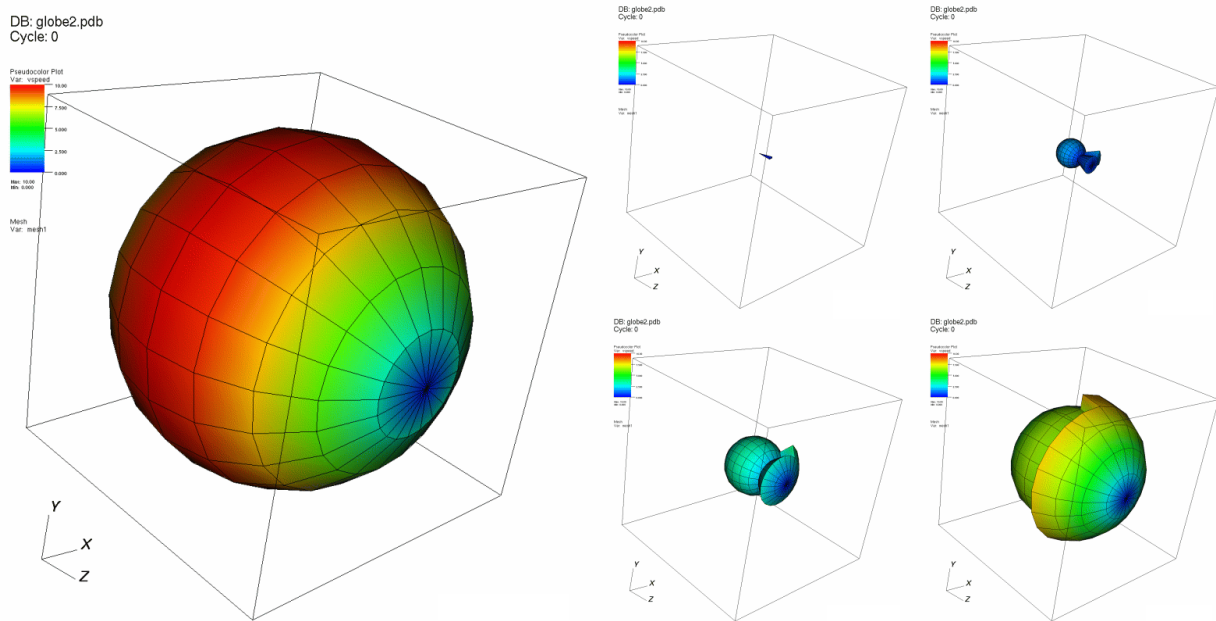
Fig. 1.110: Onion peel operator example

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

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.
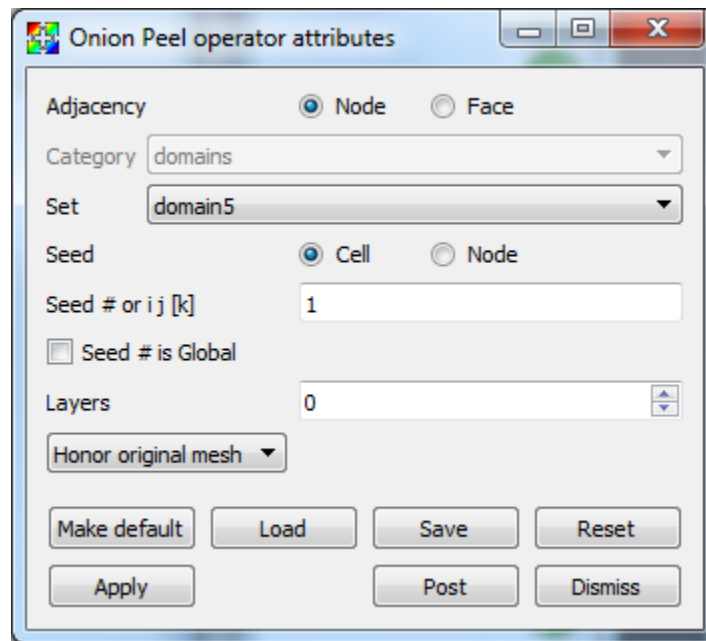
Fig. 1.111: Onion peel attributes

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 1.111.

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 1.112.

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 1.113). 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.
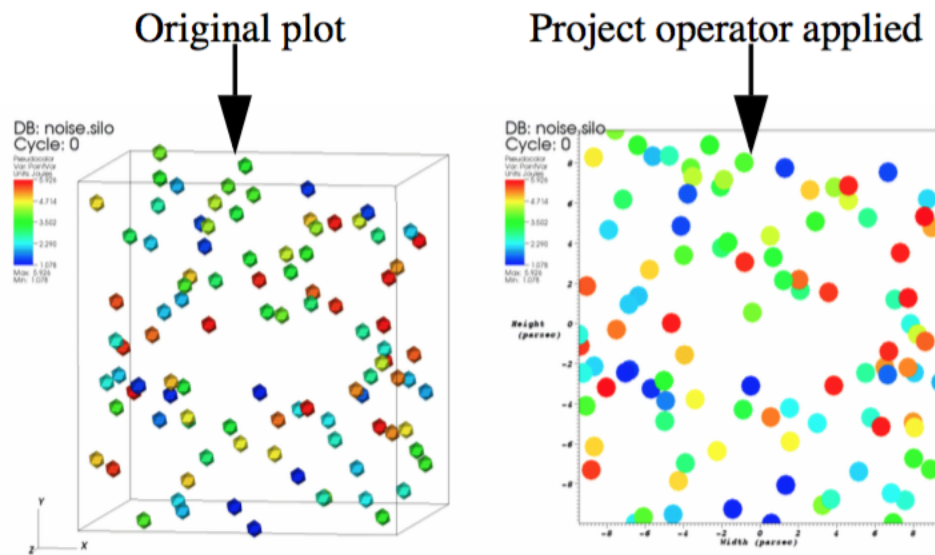
Fig. 1.112: Project Operator Example
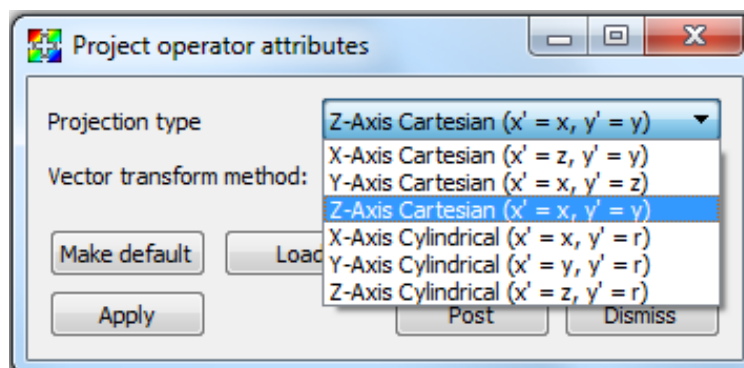
Fig. 1.113: Project Attributes Window showing available projection types

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 1.114) The default is **Treat as instantaneous directions**.
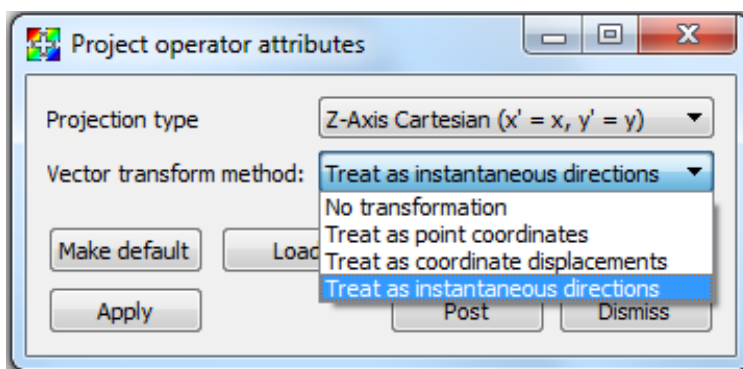
Fig. 1.114: Project Attributes Window showing available vector treatments

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 1.115.
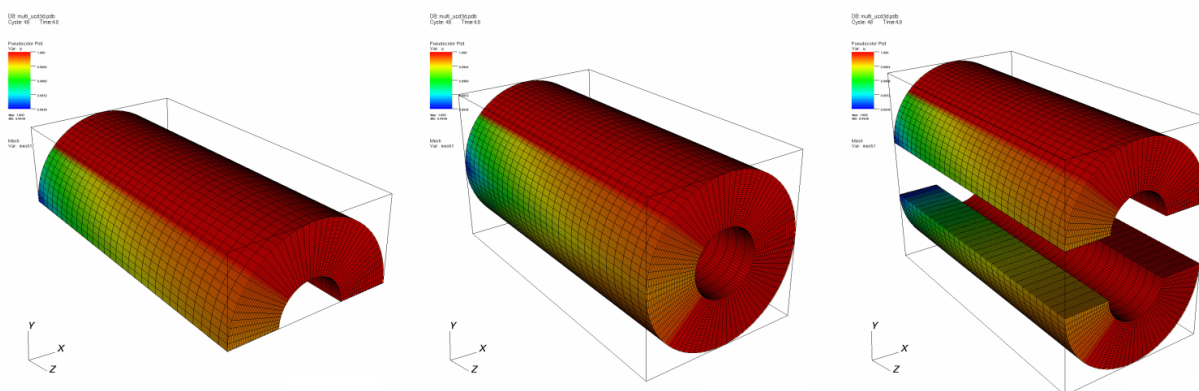
Fig. 1.115: Reflect operator example

Setting the Reflect attribute window's input mode

The **Reflect attributes window**, shown in Figure 1.116, 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.
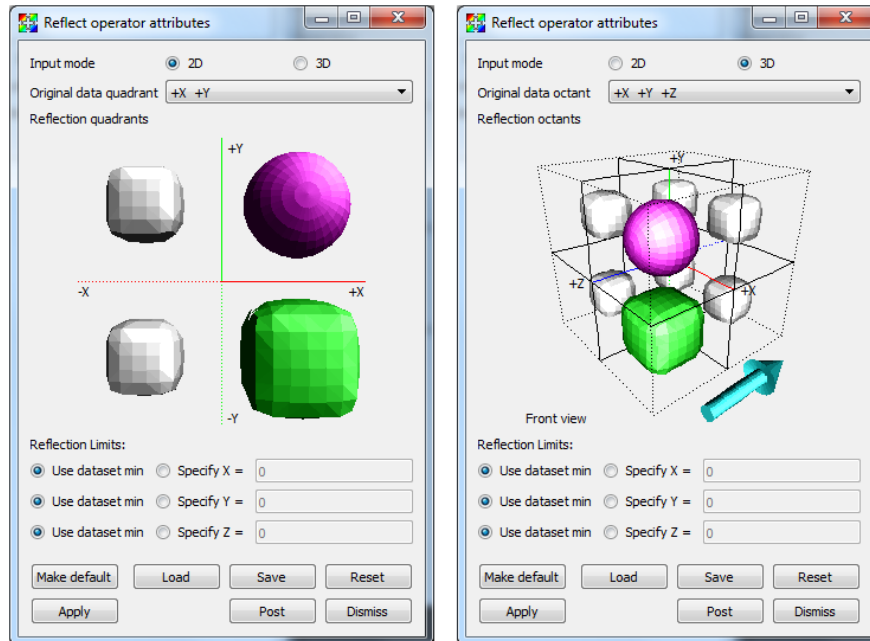
Fig. 1.116: Reflect attributes window

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 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.

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 1.117), rendering Constructive Solid Geometry (CSG) meshes, or mapping multiple datasets into a common grid for comparison purposes.
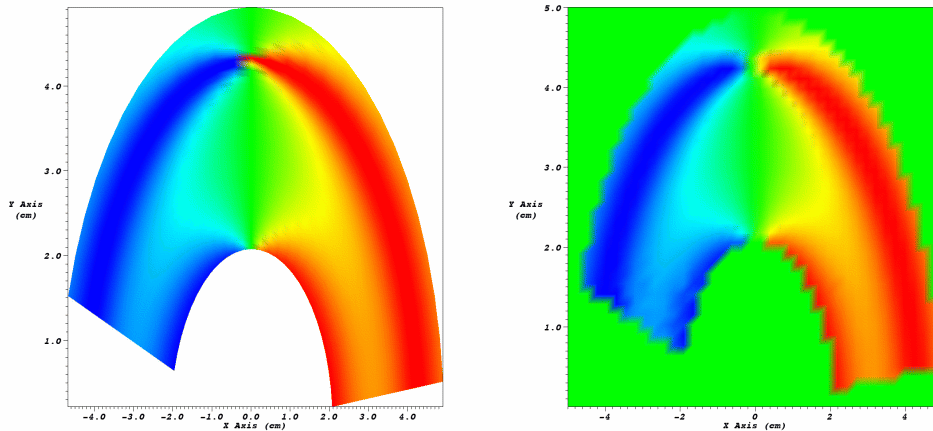
Fig. 1.117: Resample operator example

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.
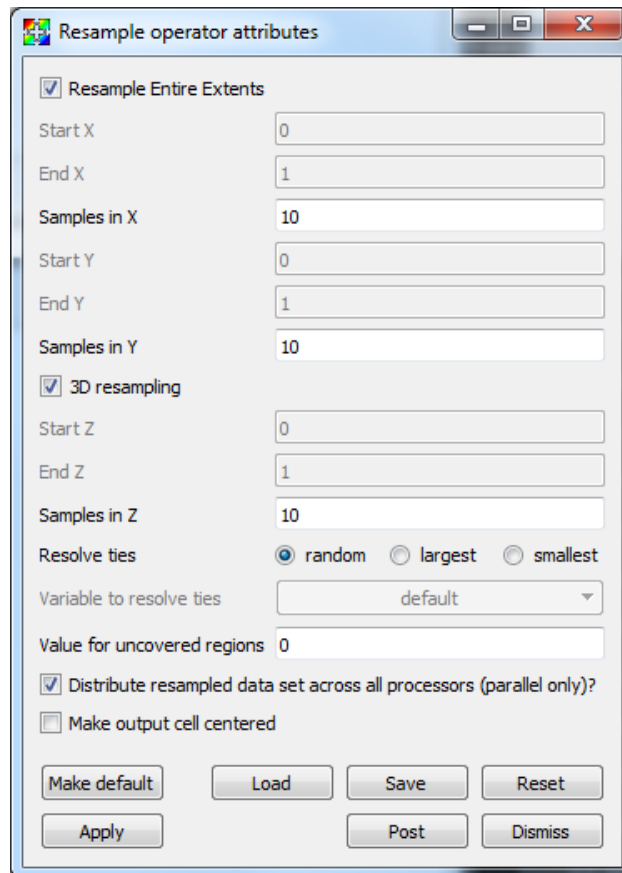
Fig. 1.118: Resample attributes window

Naturally, higher numbers of samples in the Resample operator produce a more faithful representation of the original CSG mesh. Figure 1.119 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.
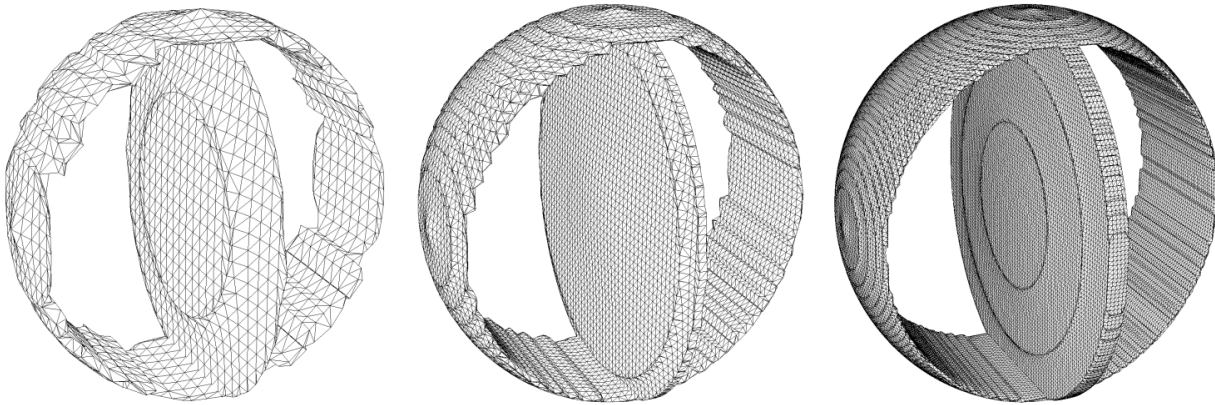
Fig. 1.119: The Resample operator can be used to control the resolution of CSG meshes. Resolution is increased from left to right.

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 1.120. 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.

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 1.121.

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 1.122) 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. Finally, the number of steps determines how many times the initial 2D geometry is revolved along the way from the
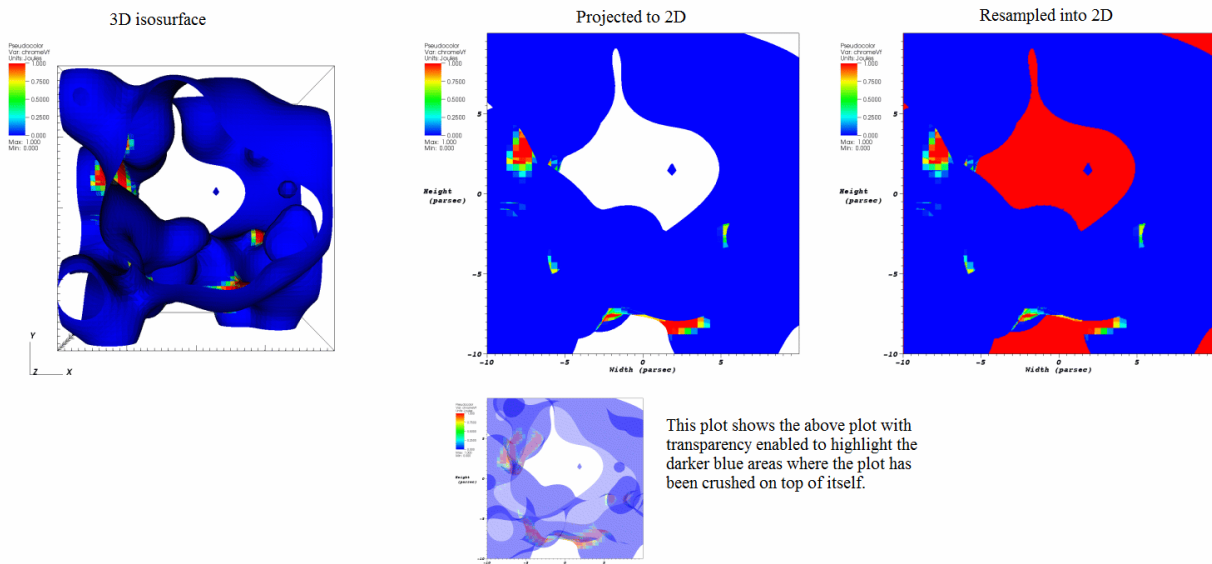
Fig. 1.120: Using the Resample operator to create a 2D projection
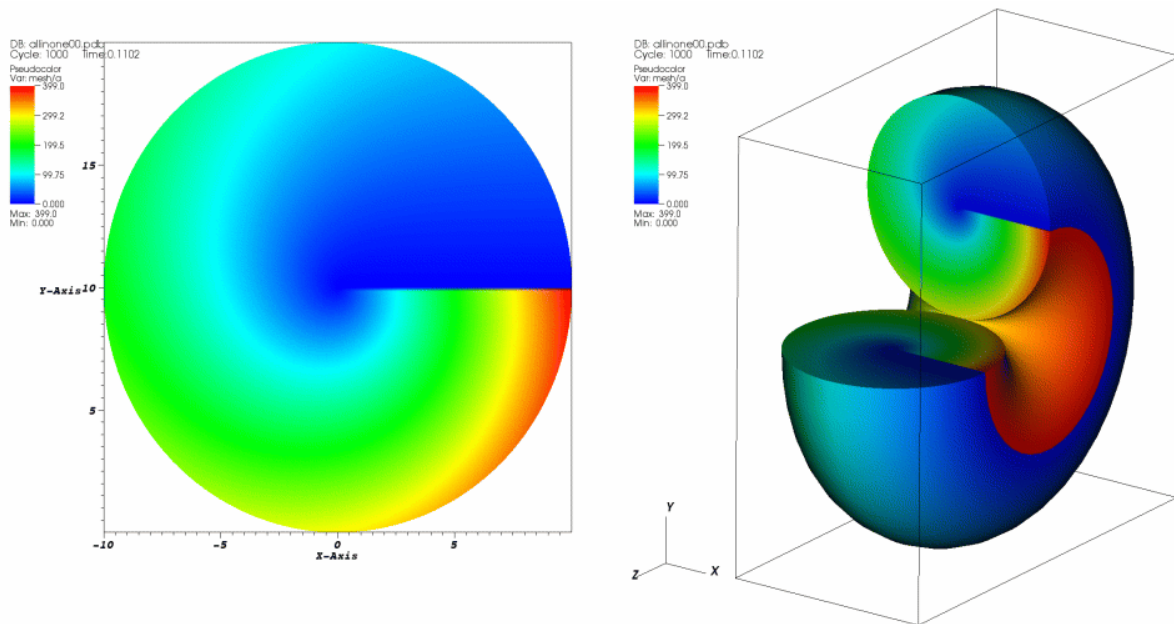
Fig. 1.121: Revolve operator example

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.
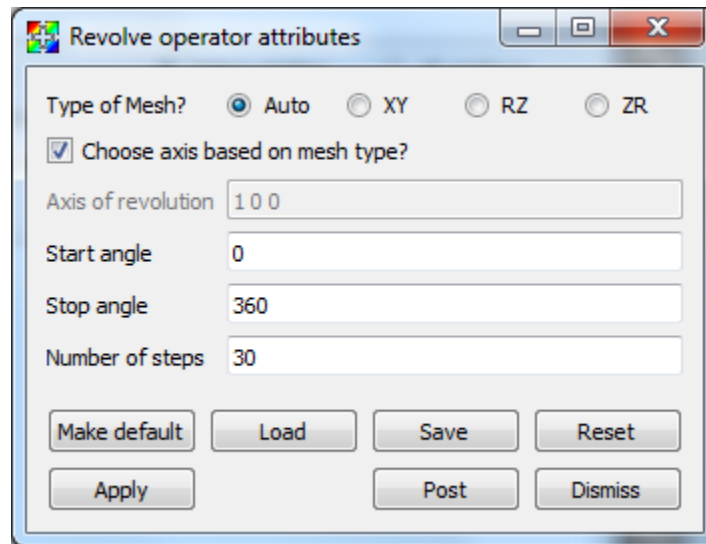
Fig. 1.122: Revolve attributes window

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 1.123.
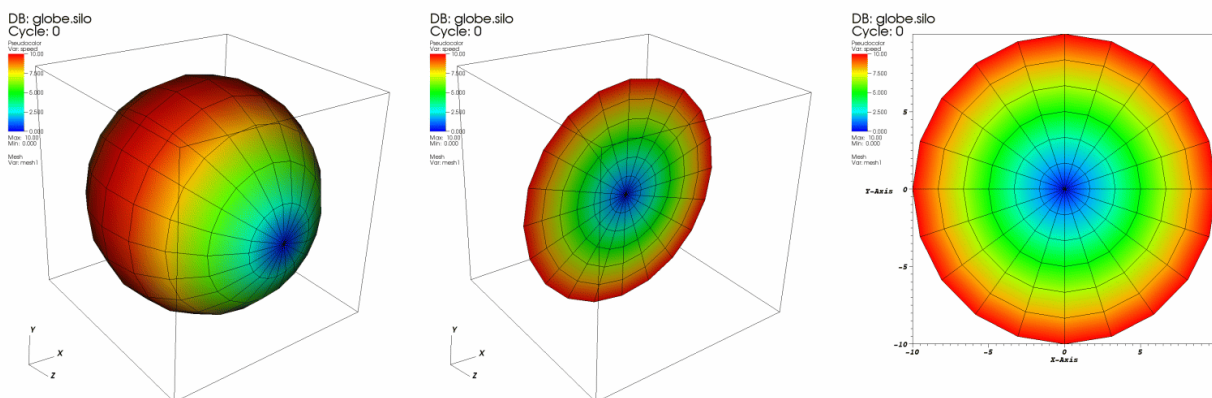
Fig. 1.123: Slice operator example

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 1.124 . 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.
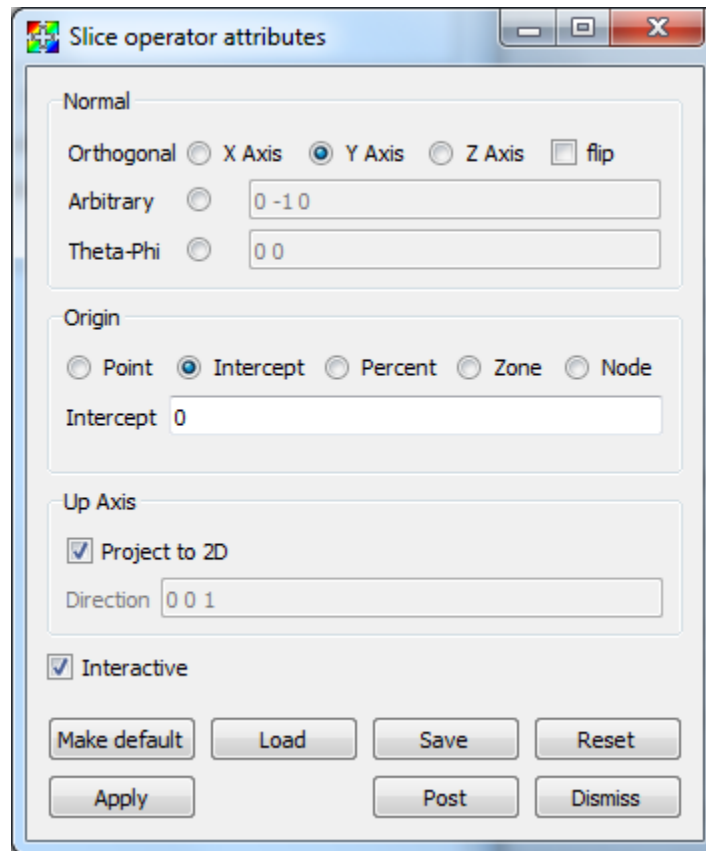
Fig. 1.124: Slice attributes window

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
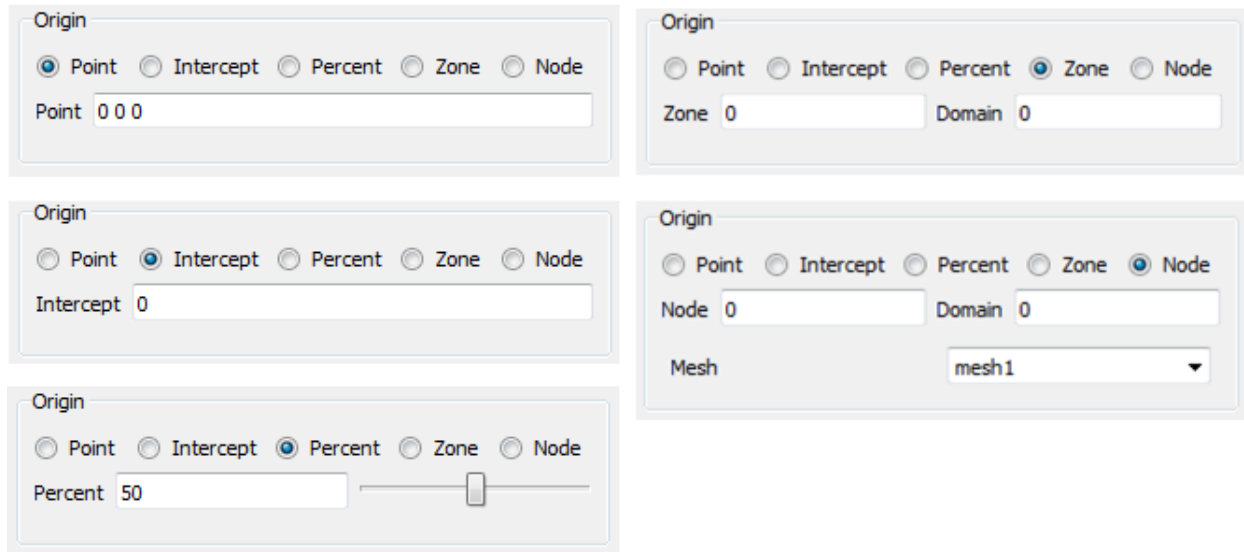
Fig. 1.125: Origin area appearance

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 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 1.125.

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 smooths a mesh to improve areas plagued by jagged edges or sharp peaks.
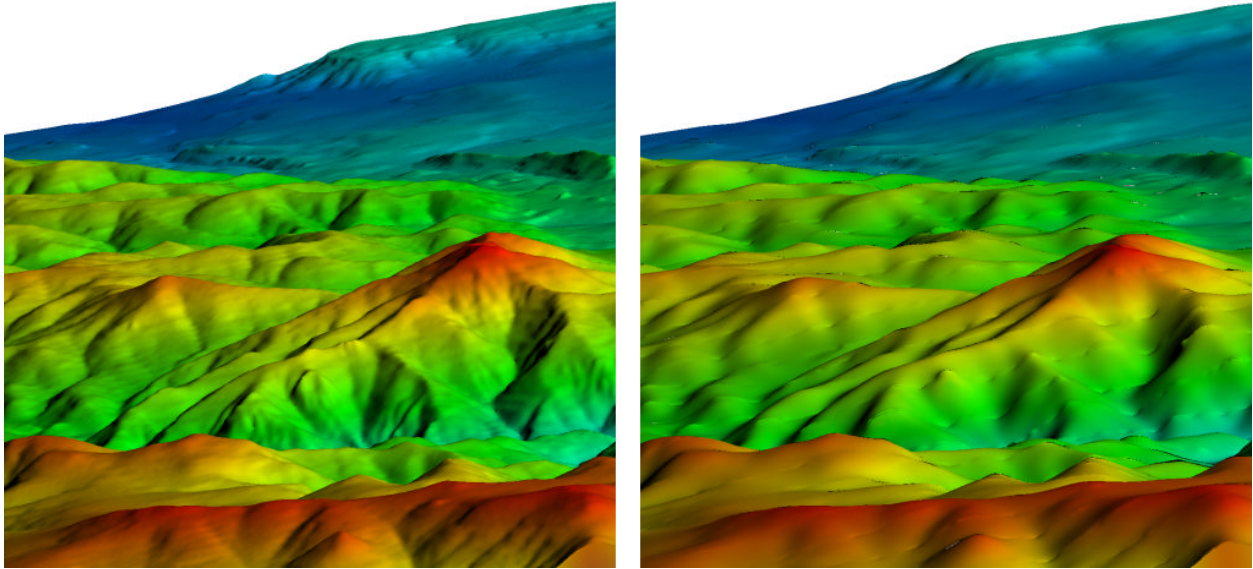
Fig. 1.126: Smooth operator example

Using the Smooth operator

The Smooth operator has a number of controls that can be used to tune mesh smoothness. One such control is the number of iterations, which controls the 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. To change the number of iterations, type a new integer value into the **Maximum number of iterations** text field in the **Smooth attributes window** (see Figure 1.127). The relaxation factor is a floating point number in the range [0,1] and it controls how much the mesh is relaxed. Values near 1 produce a mesh that is very smooth relative to the input mesh. To use a new relaxation factor, type a floating point number into the **Relaxation Factor** text field. The **Maintain Features** check box allows you to tell VisIt to preserve sharp peaks in the mesh while still smoothing out most of the mesh. The angle in the **Feature Angle** text field determines which features are kept. Any mesh angles less than the feature angle are preserved while others are smoothed.

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 1.128.

Positioning and resizing the slice sphere

You can position the slice sphere by setting its origin in the **SphereSlice attributes window** shown in Figure 1.129 . The slice sphere is specified by a center point and a radius. To change the slice sphere's center, enter a new point into
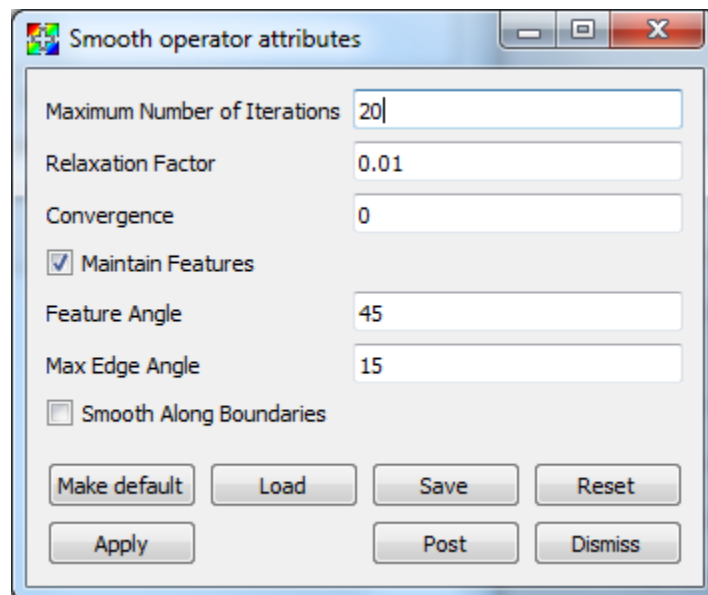
Fig. 1.127: Smooth attributes
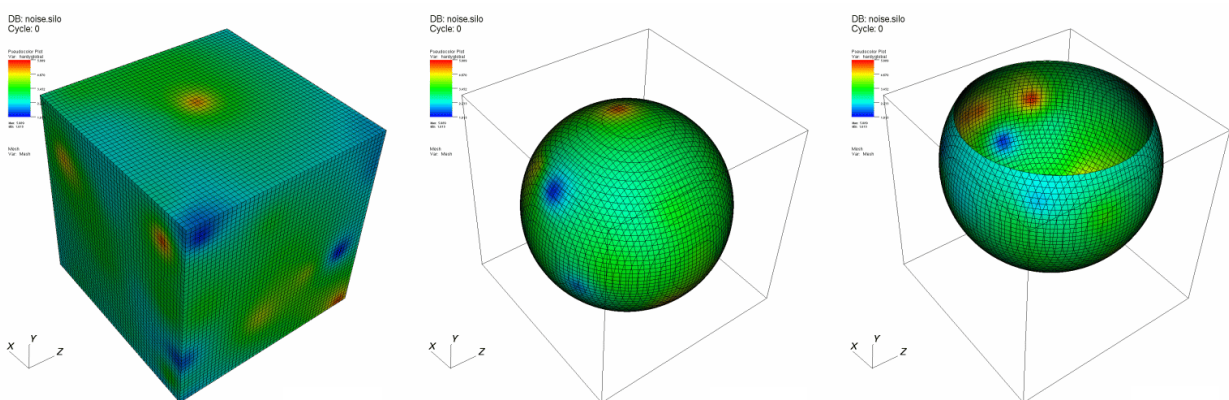
Fig. 1.128: SphereSlice operator example

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.
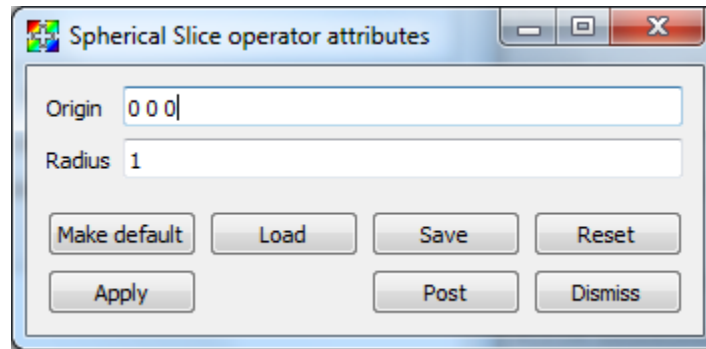
Fig. 1.129: SphereSlice attributes window

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.

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 1.131. 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 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
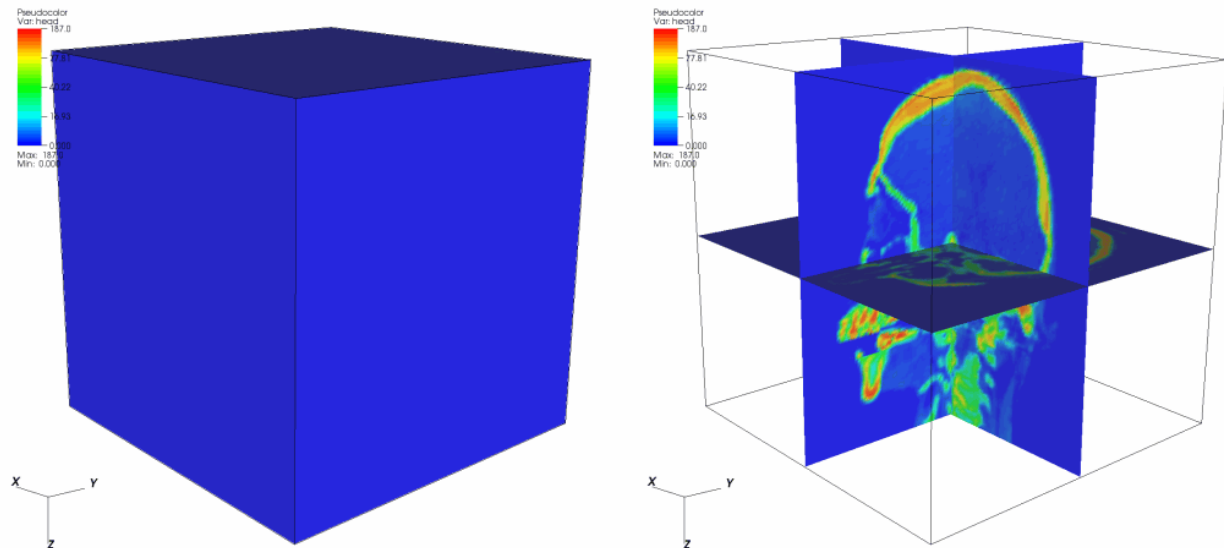
Fig. 1.130: ThreeSlice operator example
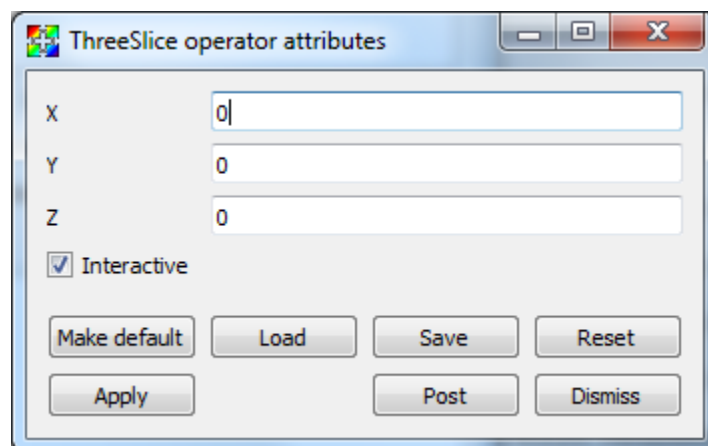
Fig. 1.131: ThreeSlice attributes window

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 1.132.
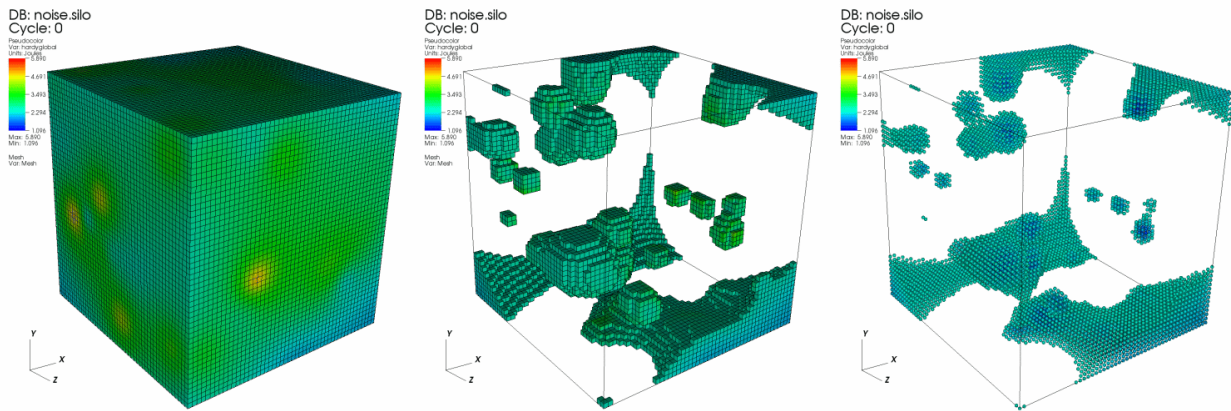
Fig. 1.132: Threshold operator example

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 1.133) **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 1.134 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.

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.
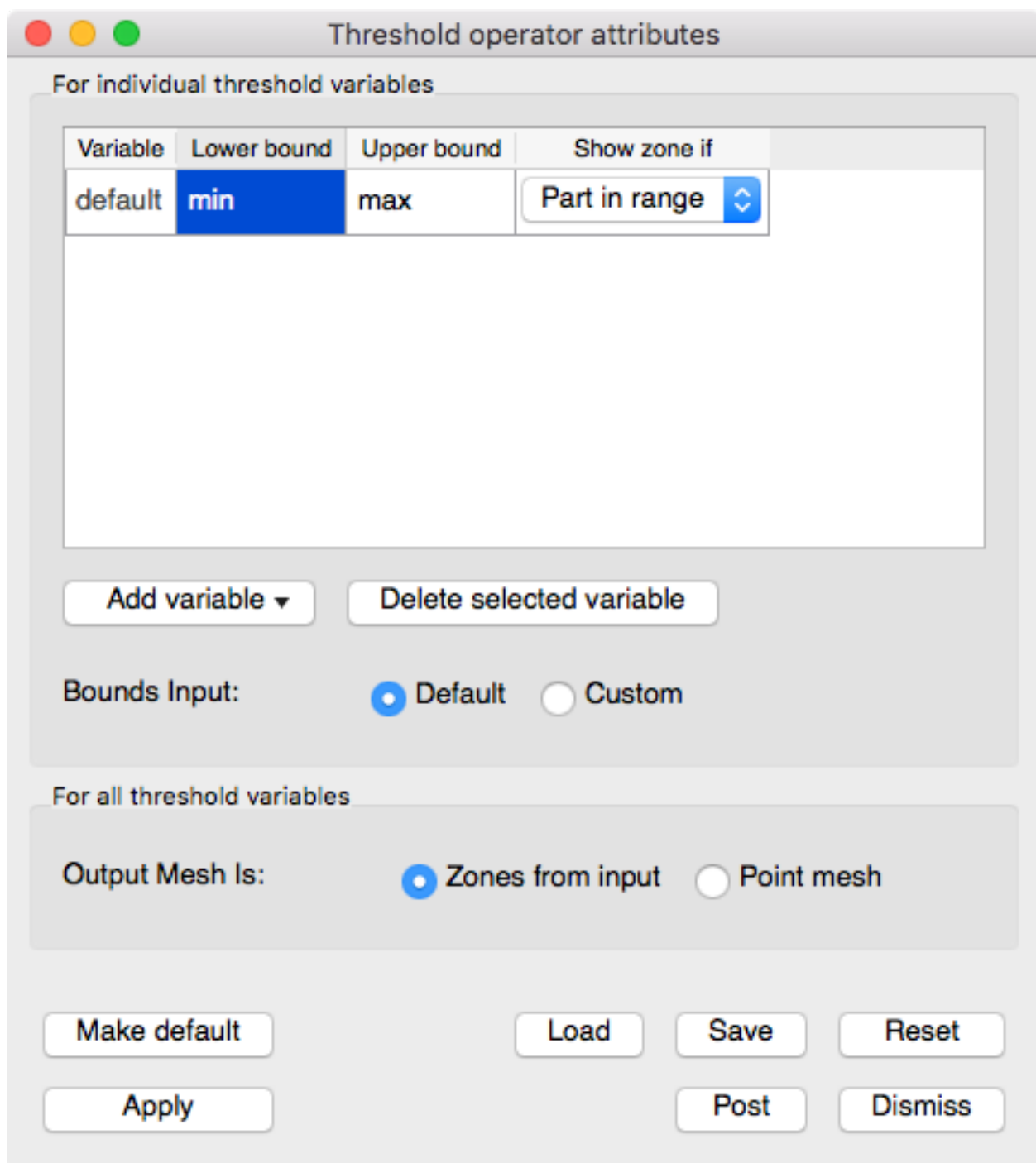
Fig. 1.133: Threshold attributes window - Default
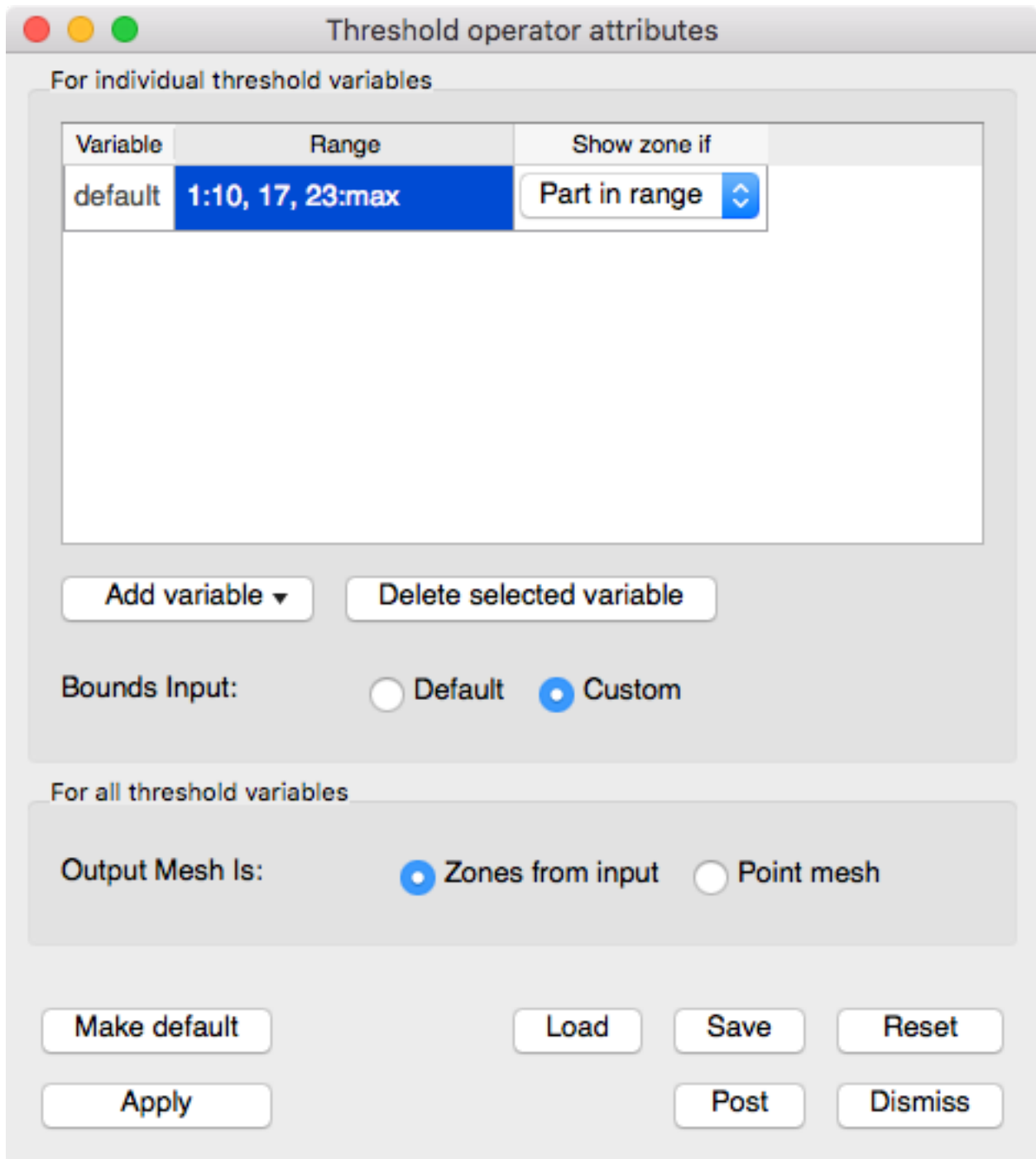
Fig. 1.134: Threshold attributes window - Custom

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 1.135.
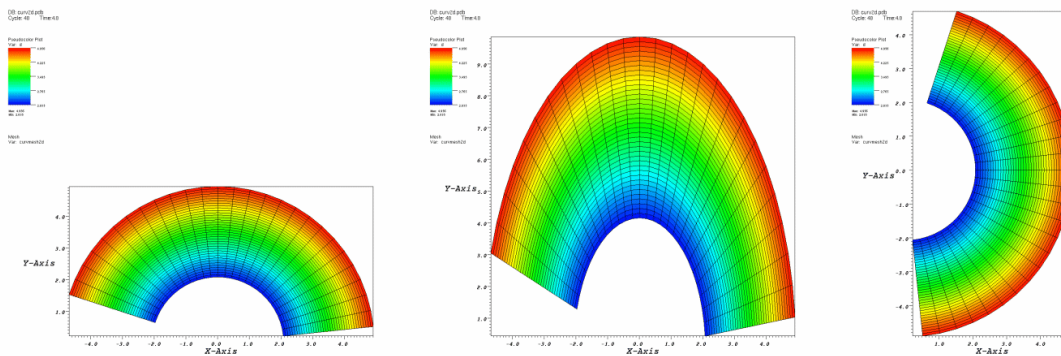
Fig. 1.135: Transform operator example

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 1.136). 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 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.
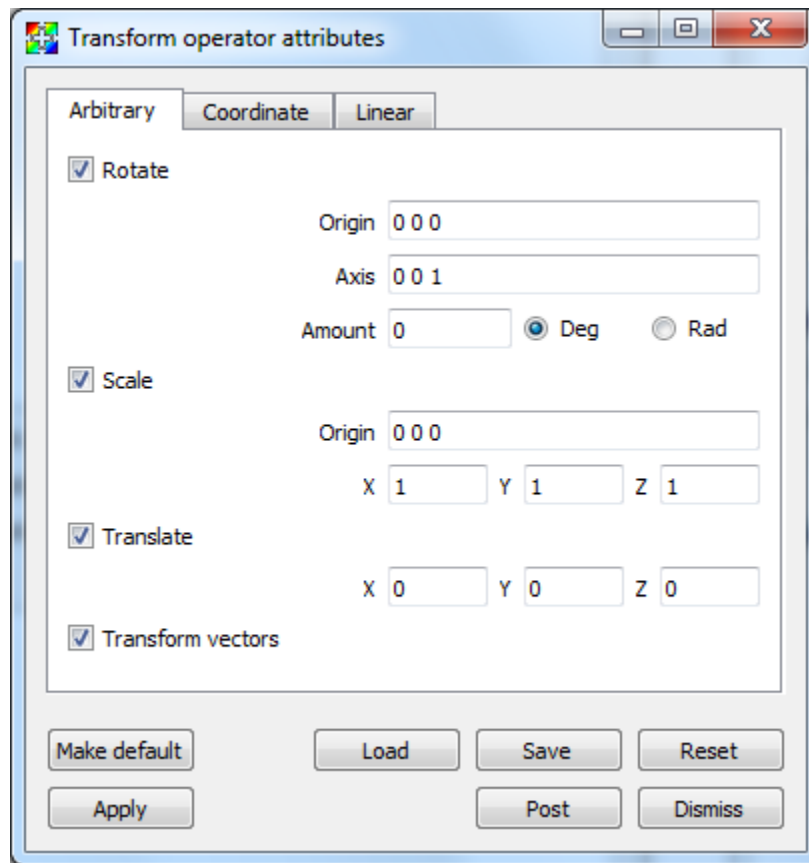
Fig. 1.136: Transform attributes window

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 1.137). 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.
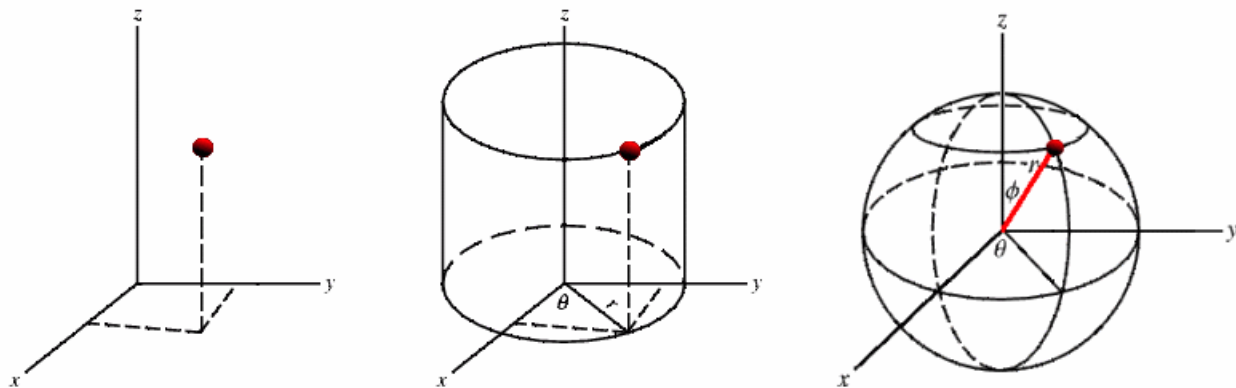
Fig. 1.137: Cartesian, Cylindrical, Spherical coordinate systems

Figure 1.138 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.

The Transform operator allows coordinate system transformations between any of the three supported coordinate systems, shown in Figure 1.139 . 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 1.138, 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.
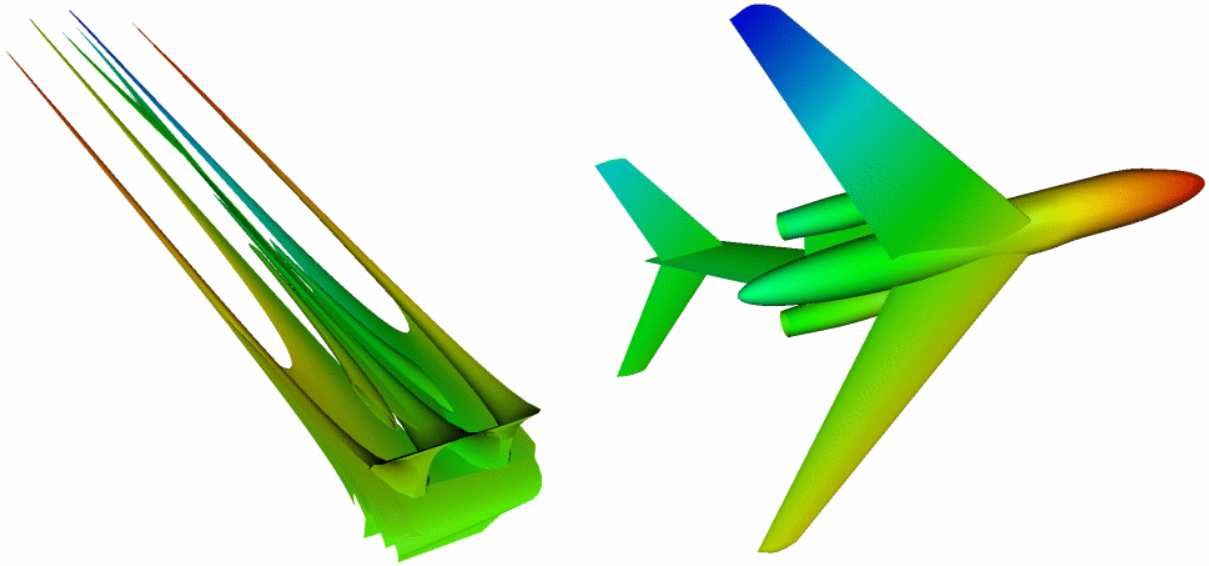
Fig. 1.138: Coordinate system conversion using the Transform operator
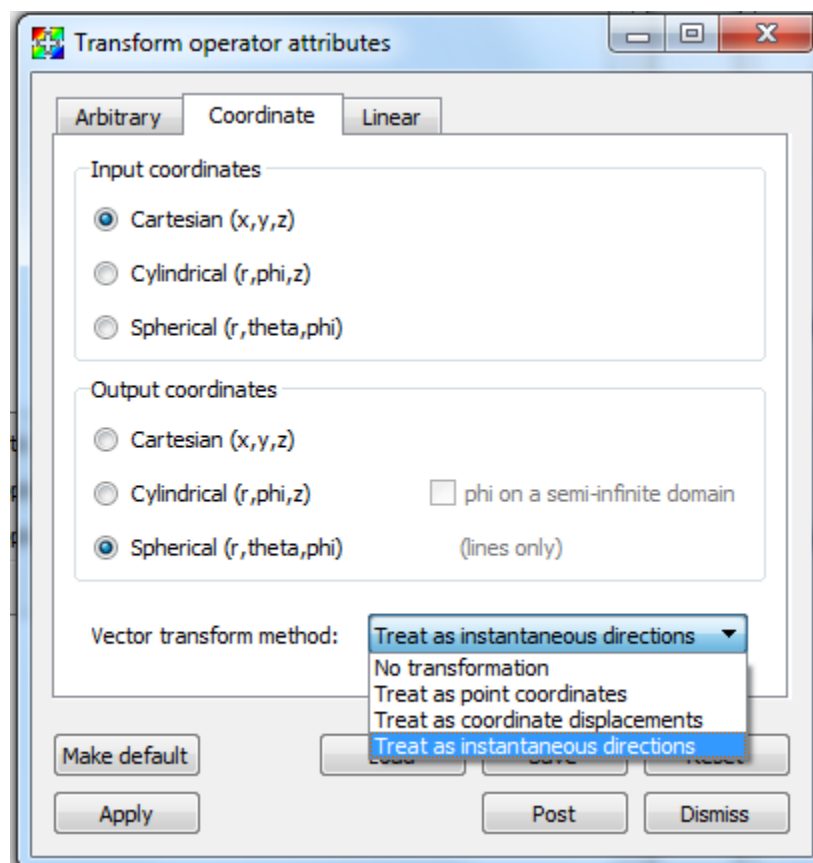
Fig. 1.139: Supported coordinate systems

Linear transforms

Linear transforms can be specified via a 4x4 matrix as shown in Figure 1.140. 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**.
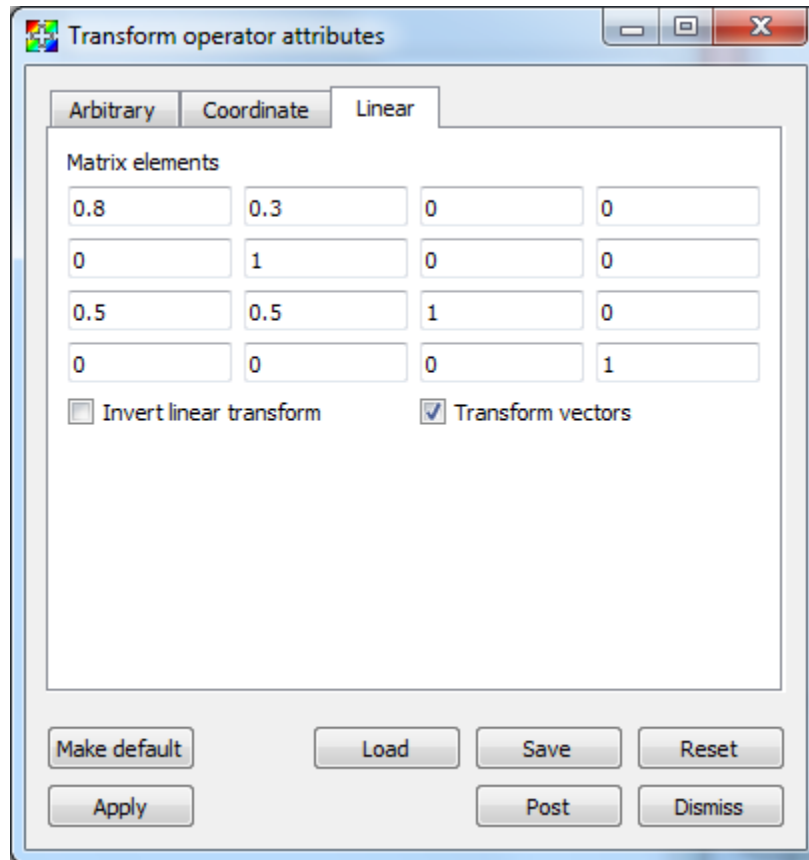
Fig. 1.140: Linear transformation options

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 1.142). 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.

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 1.143 for result of capping.
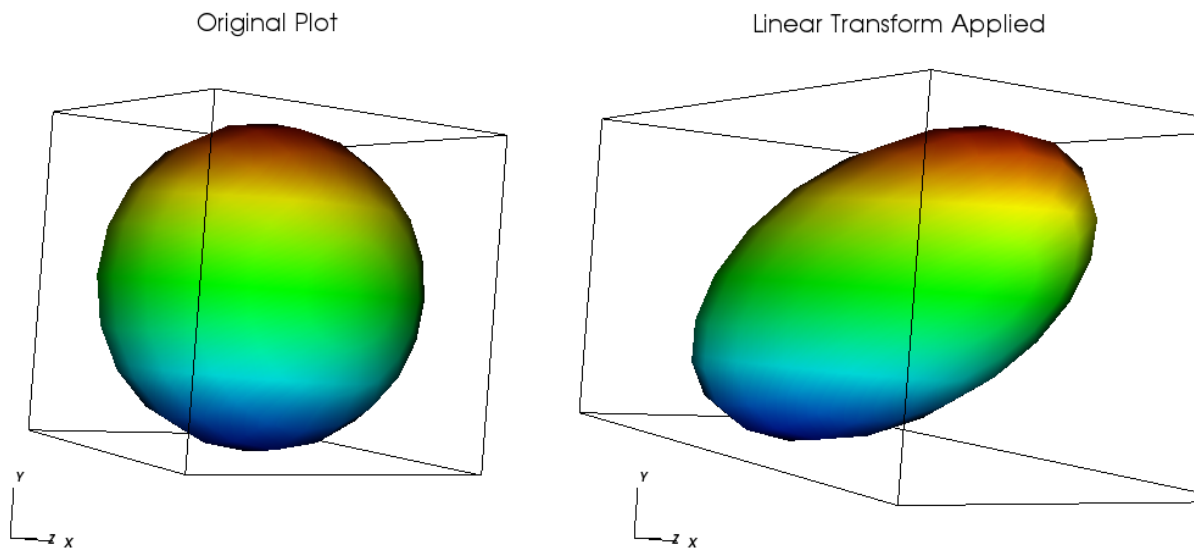
Fig. 1.141: Linear transformation example
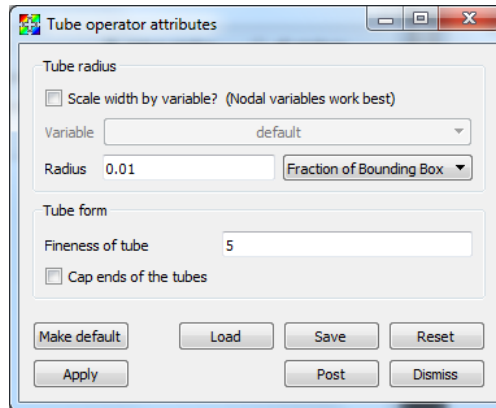
Fig. 1.142: Tube attributes window
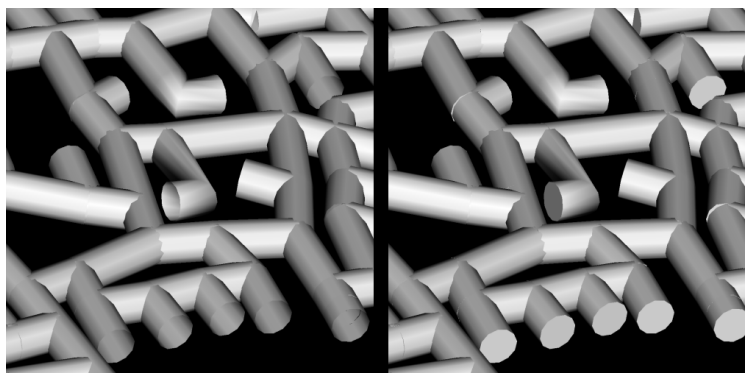
Fig. 1.143: Uncapped and capped tubes

1.5 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**.

1.5.1 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.

1.5.2 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 1.144, 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 `"My images"` directory, which is a directory under the VisIt installation directory. 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 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"*.
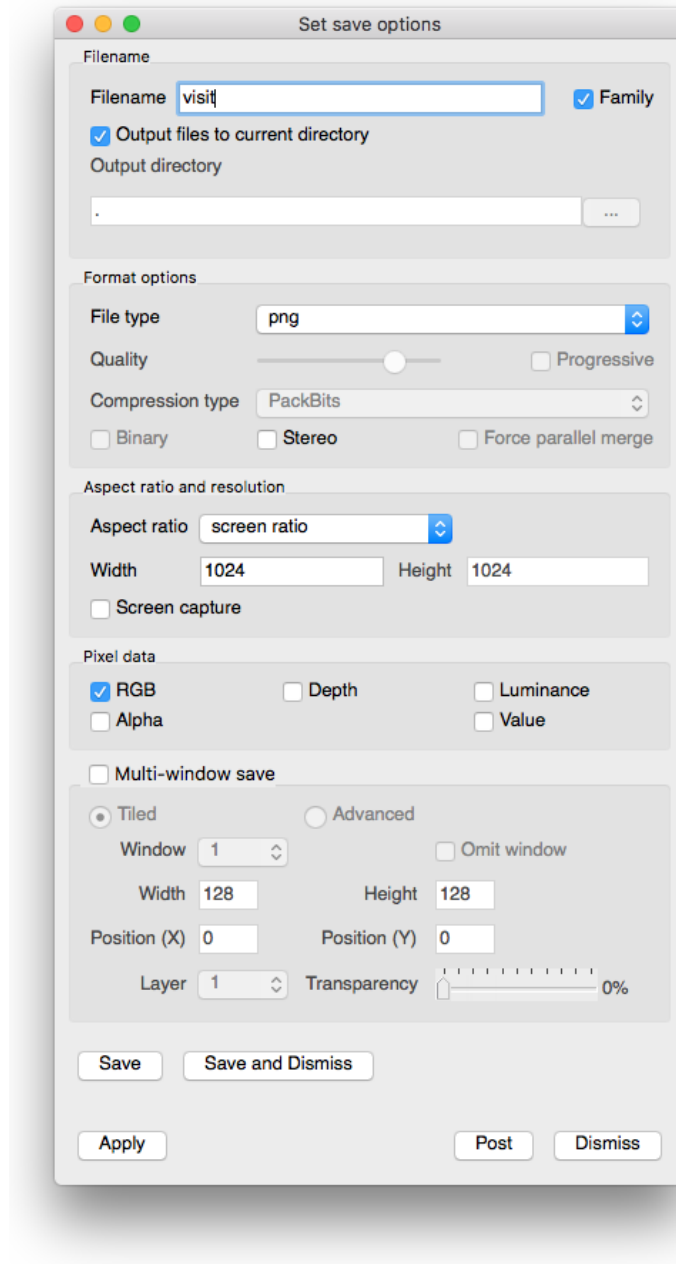
Fig. 1.144: Save Window

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 1.145). 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).
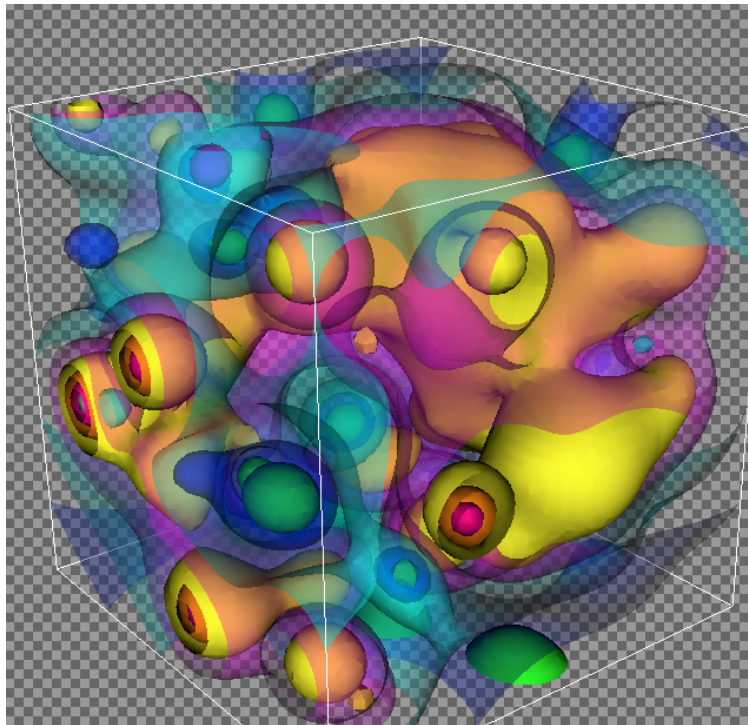
Fig. 1.145: Partially transparent plot saved to PNG with alpha channel

Saving tiled images
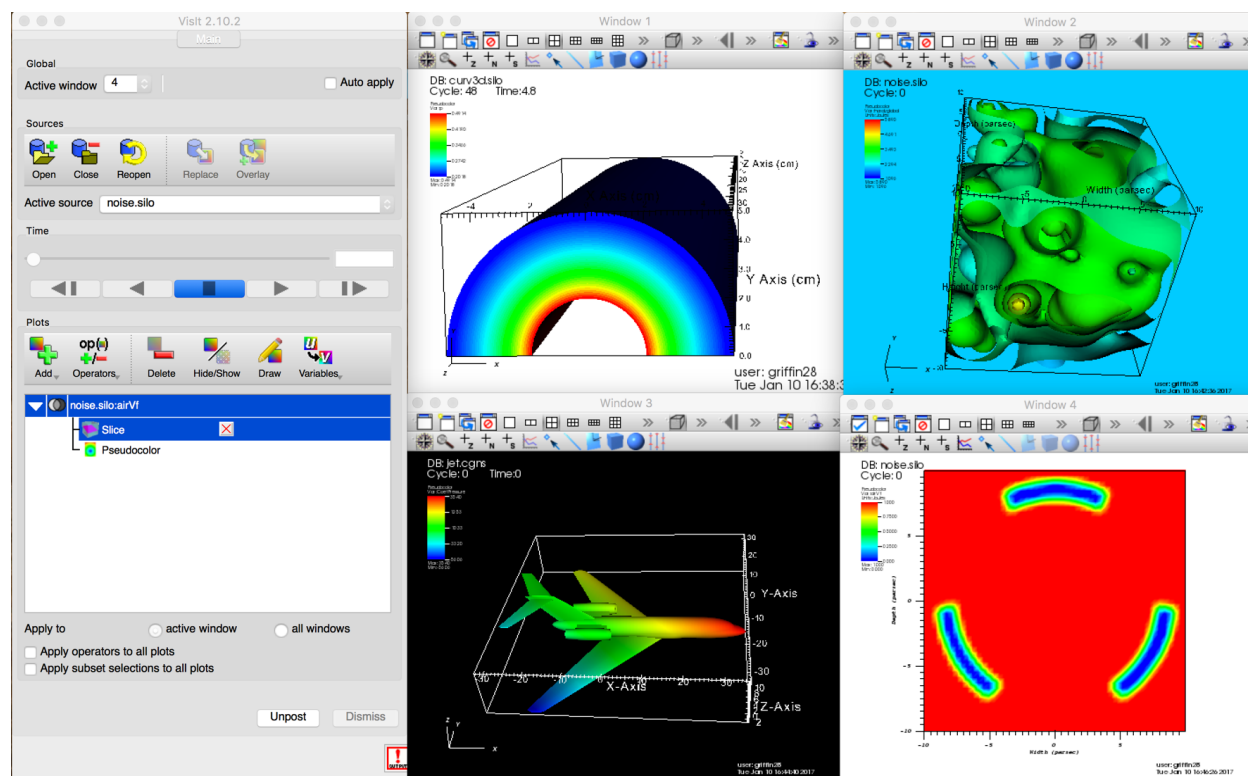
Fig. 1.146: Saving tiled images example (before)

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 1.146 and Figure 1.147.

1.5.3 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.

The **Save movie wizard** (see Figure 1.148) 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 OSX)** 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.
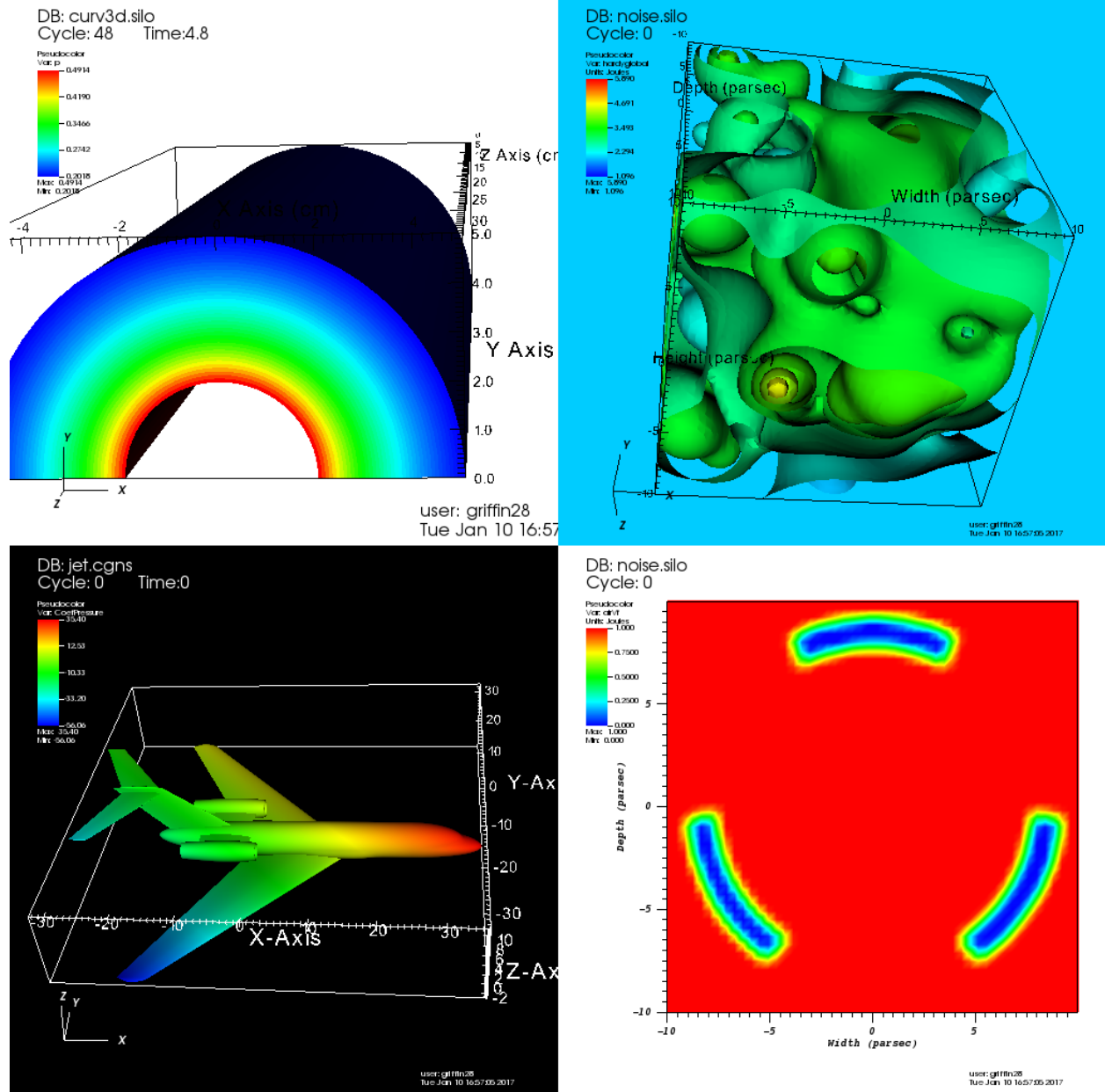
Fig. 1.147: Saving tiled images example (after)
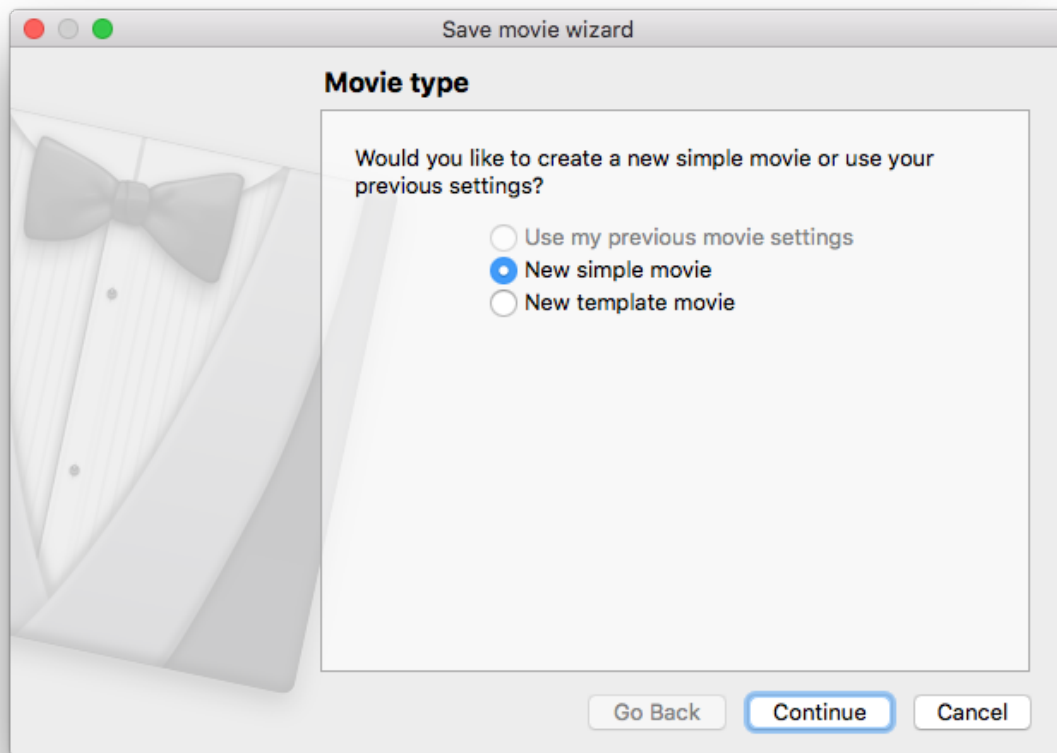
Fig. 1.148: Save movie wizard (screen 1)
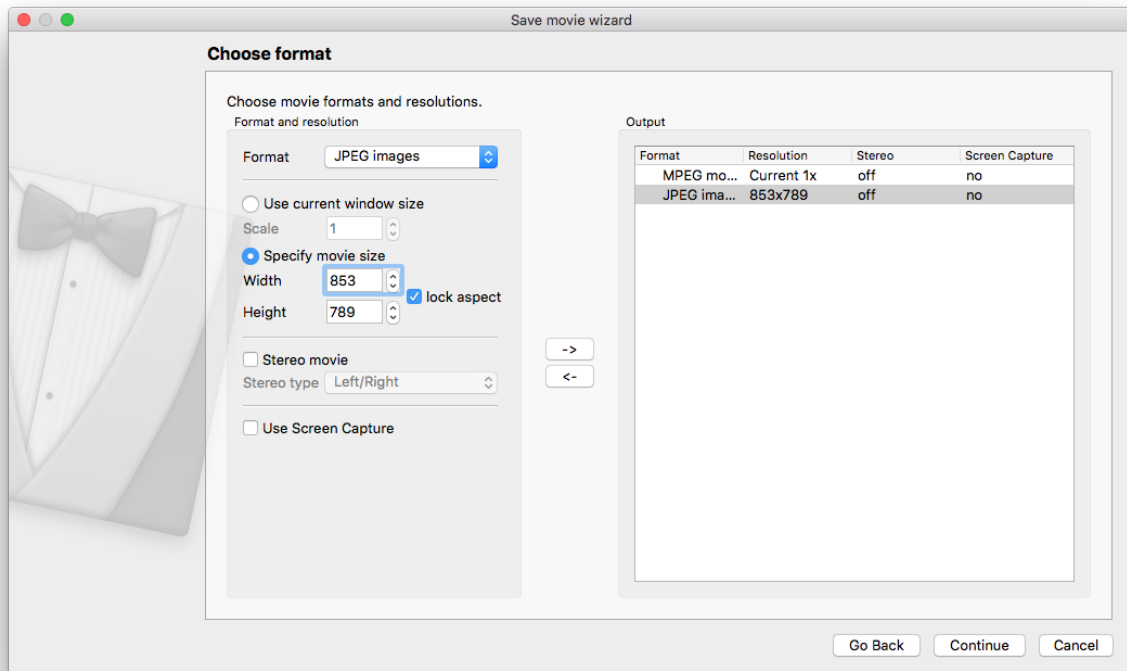
Fig. 1.149: Save movie wizard (screen 2)

Choosing movie formats

The **Save movie wizard's** second screen, shown in Figure 1.149, 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
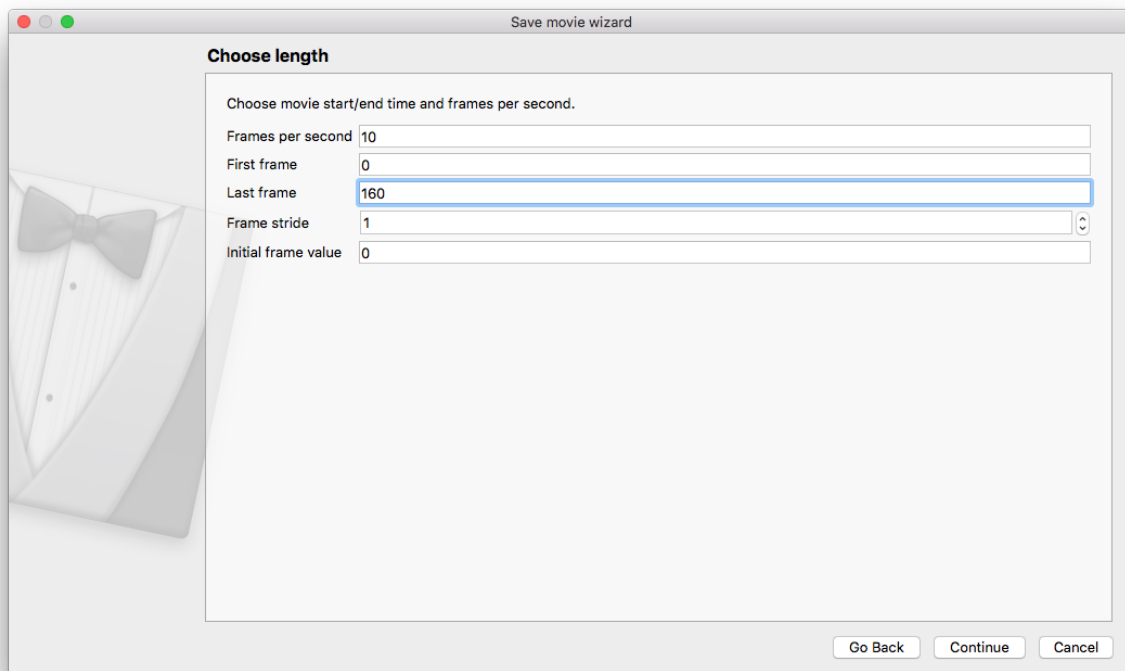
Fig. 1.150: Save movie wizard (screen 3)

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 1.150). 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

---
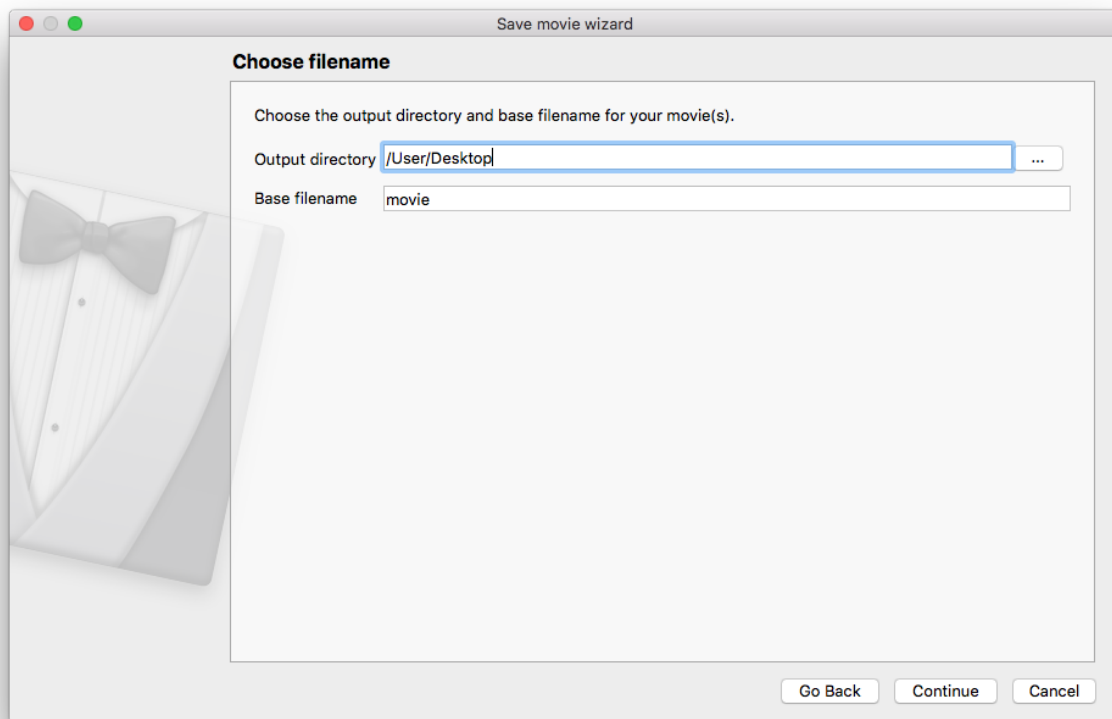
Fig. 1.151: Save movie wizard (screen 4)

want to specify an alternate directory, you can either type in a new directory path into the **Output directory** text field (see Figure 1.151) 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
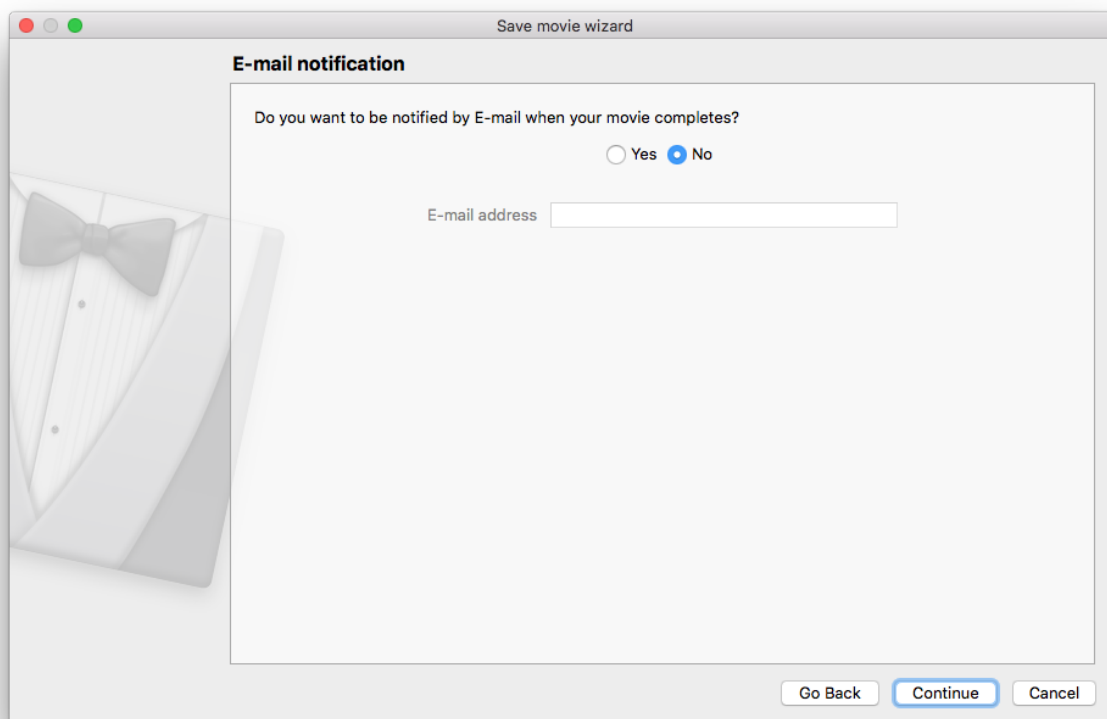
Fig. 1.152: Save movie wizard (screen 5)

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 1.152). 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 VisIt 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 1.153 .

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
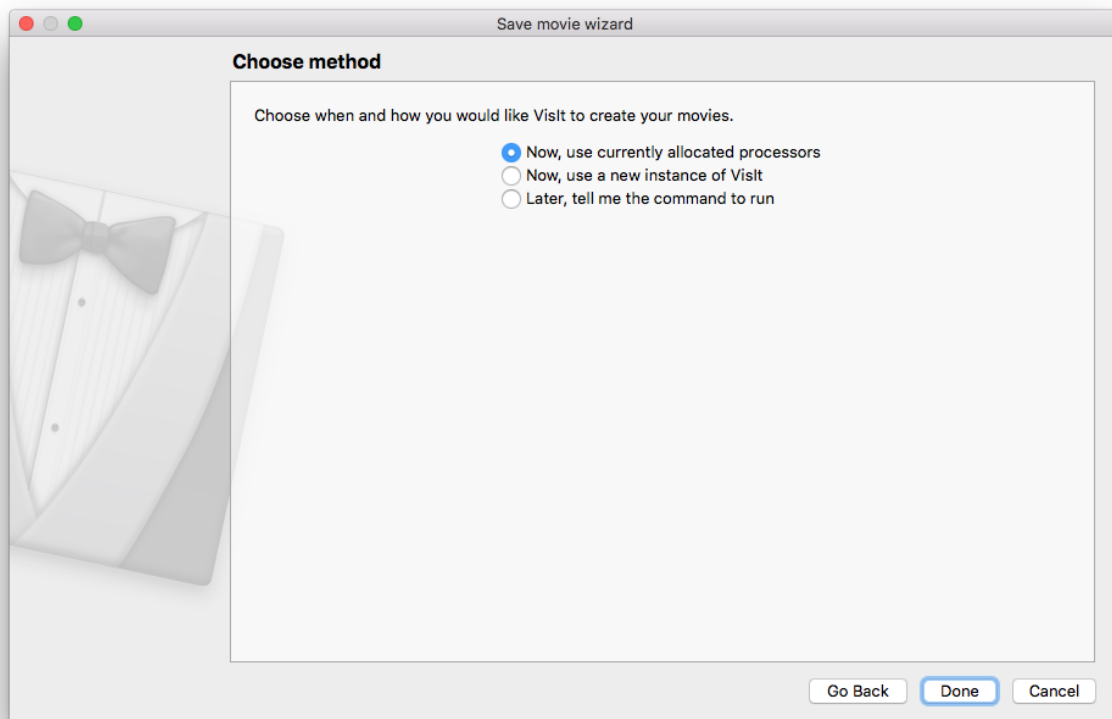
Fig. 1.153: Save movie wizard (screen 6)

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 1.154 . 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.
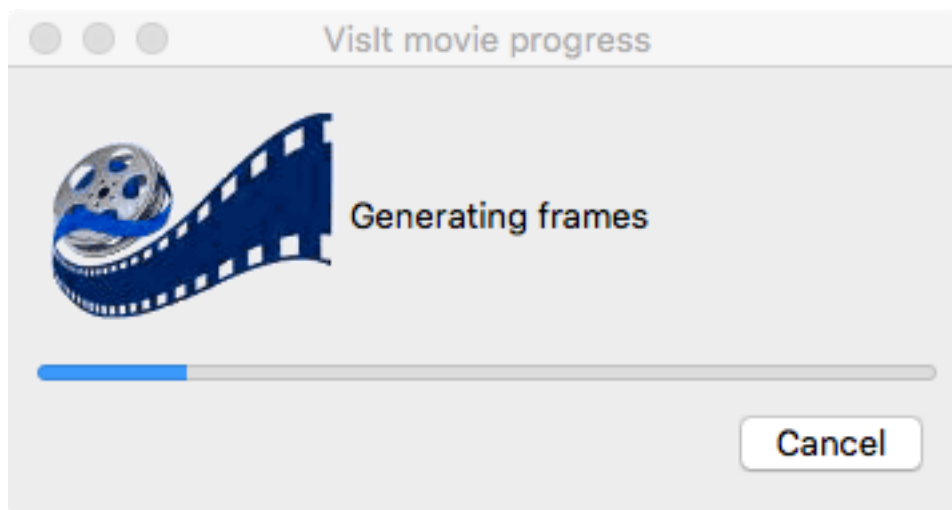
Fig. 1.154: Movie progress dialog

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.

1.5.4 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 1.155) 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.
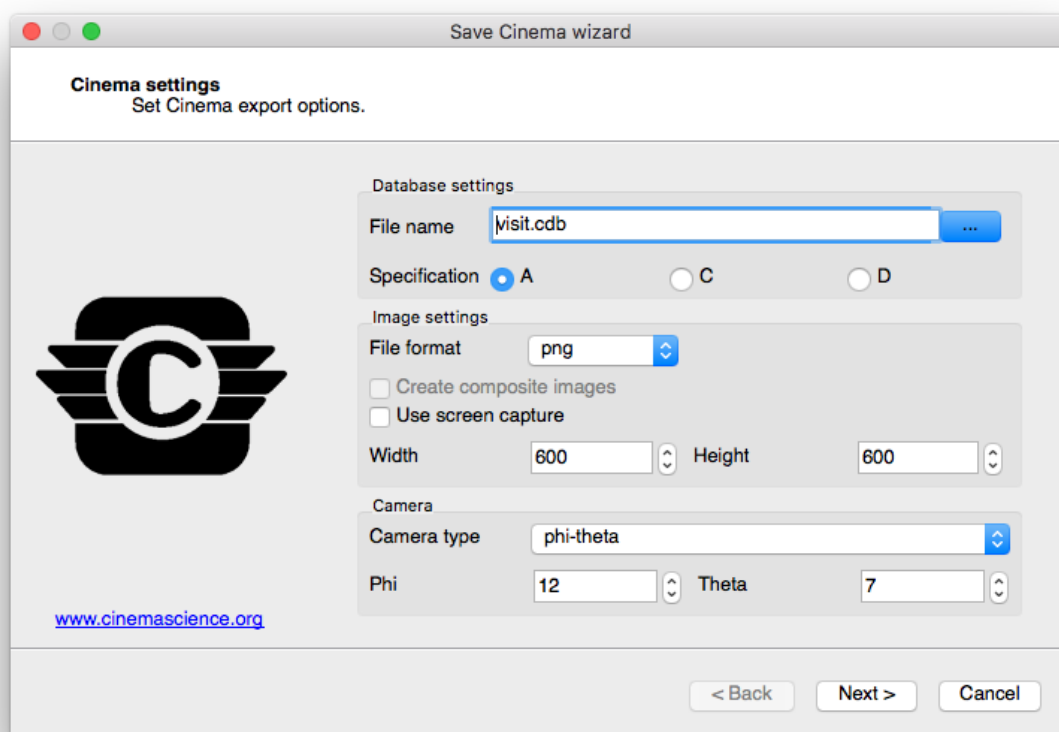
Fig. 1.155: Save Cinema wizard (screen 1)

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 1.157 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.
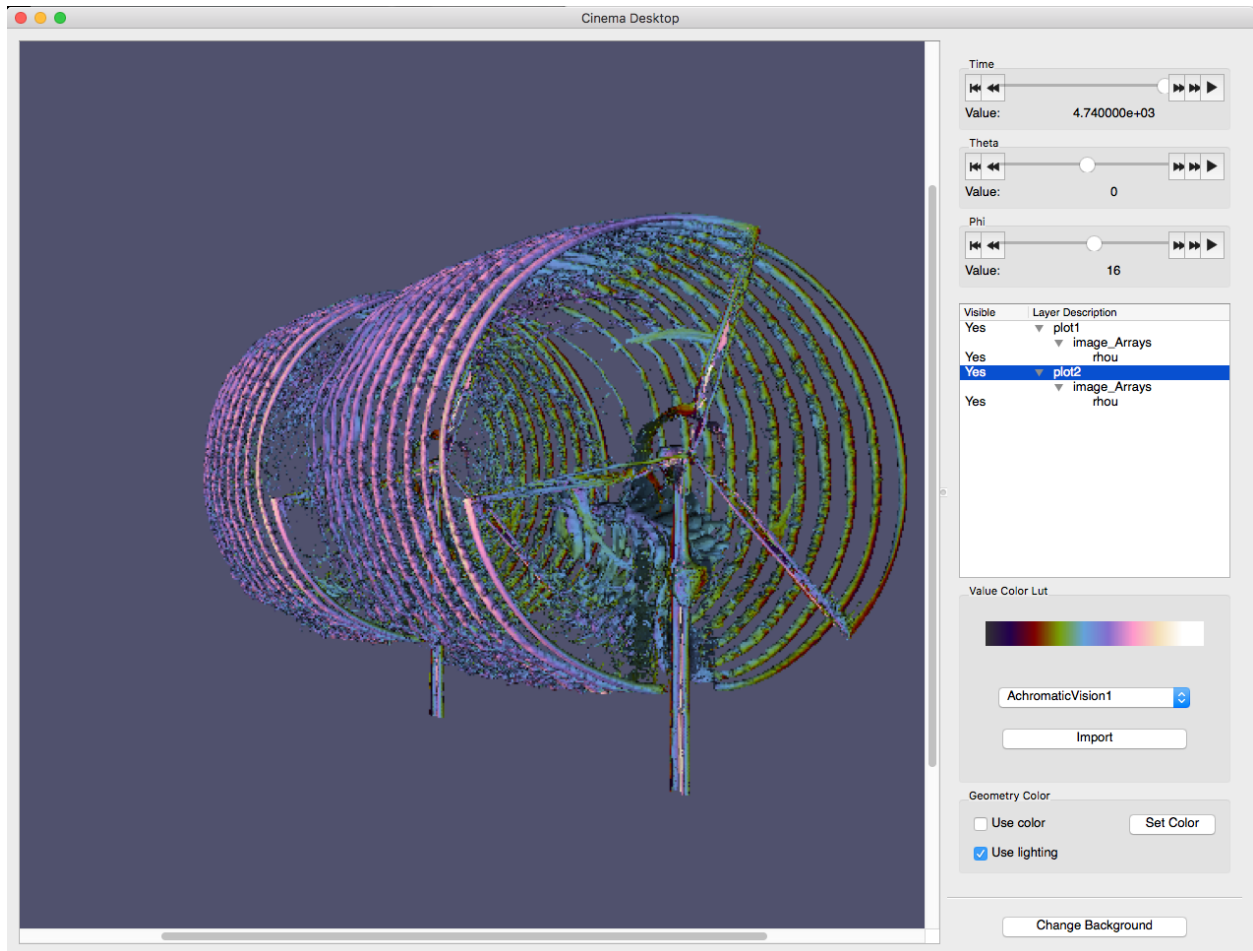
Fig. 1.156: Cinema viewer with composite layers

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 1.157) 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.
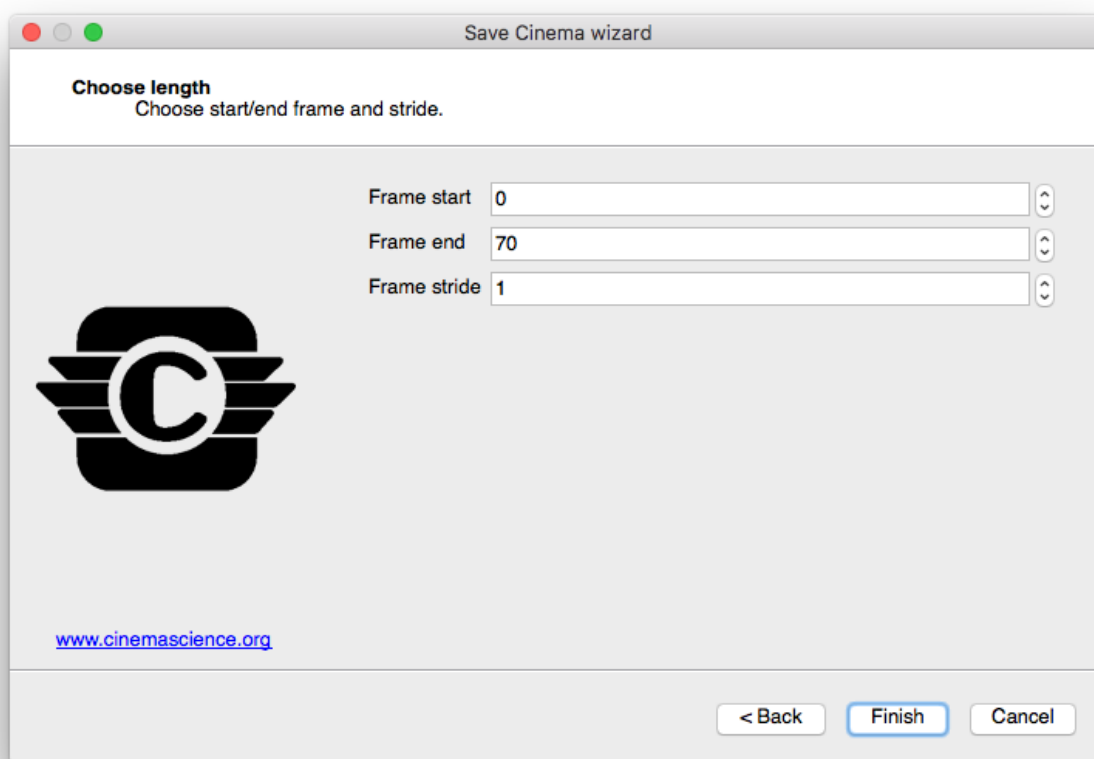
Fig. 1.157: Save Cinema wizard (screen 2)

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");
```

1.5.5 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 1.158 .
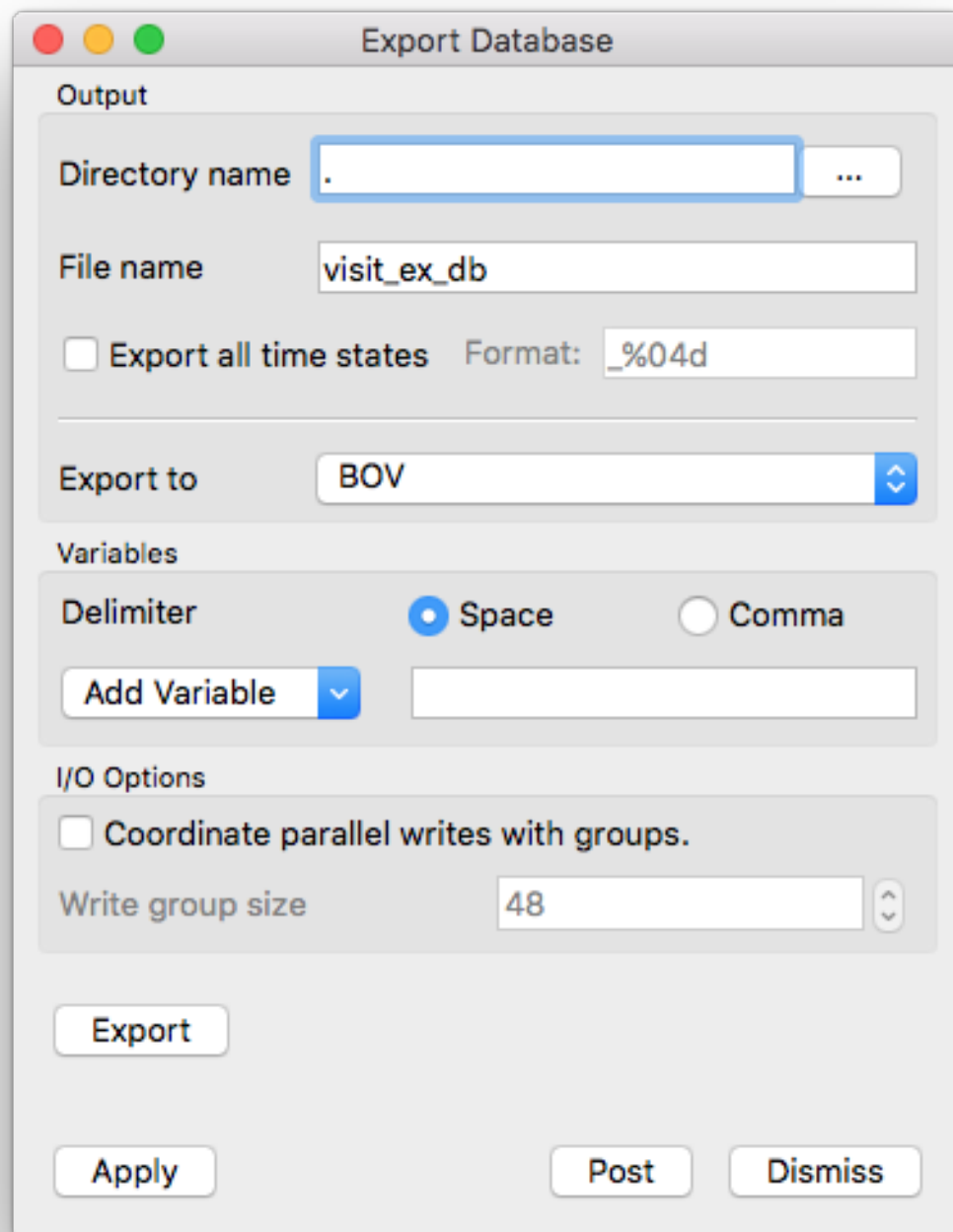
Fig. 1.158: Export Database Window

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
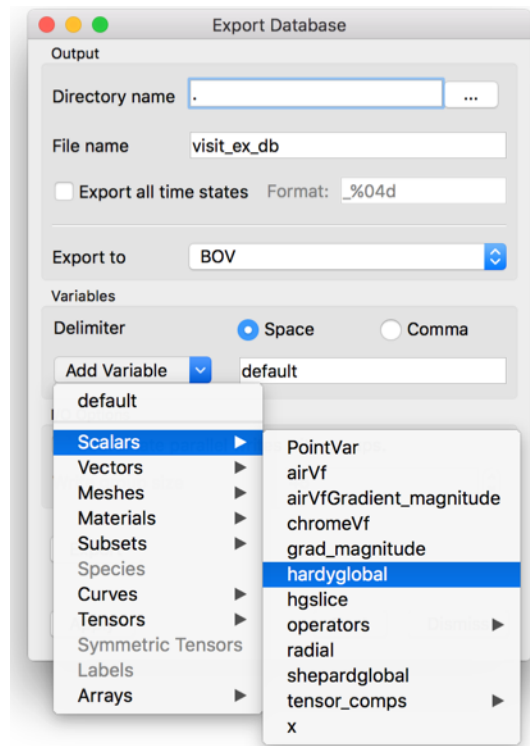
Fig. 1.159: Variables menu

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 1.159 . 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 1.160 .
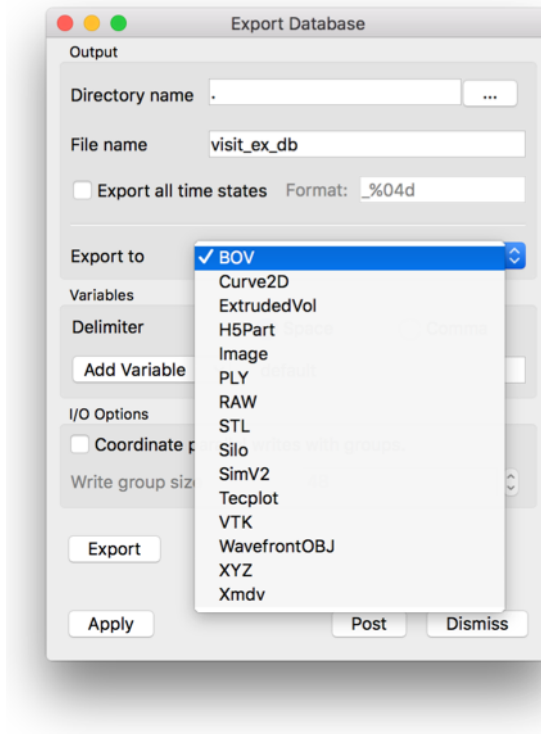
Fig. 1.160: Export file types

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.

1.5.6 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 1.162 .
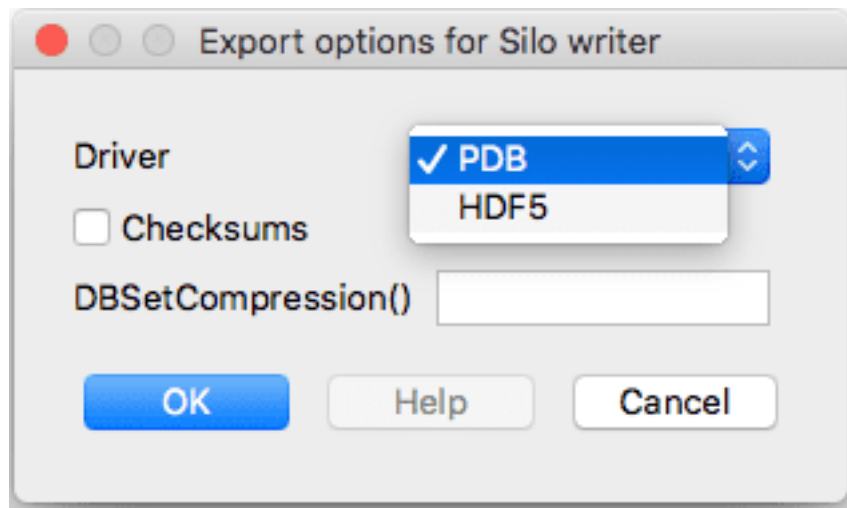
Fig. 1.161: Export options example (for Silo)
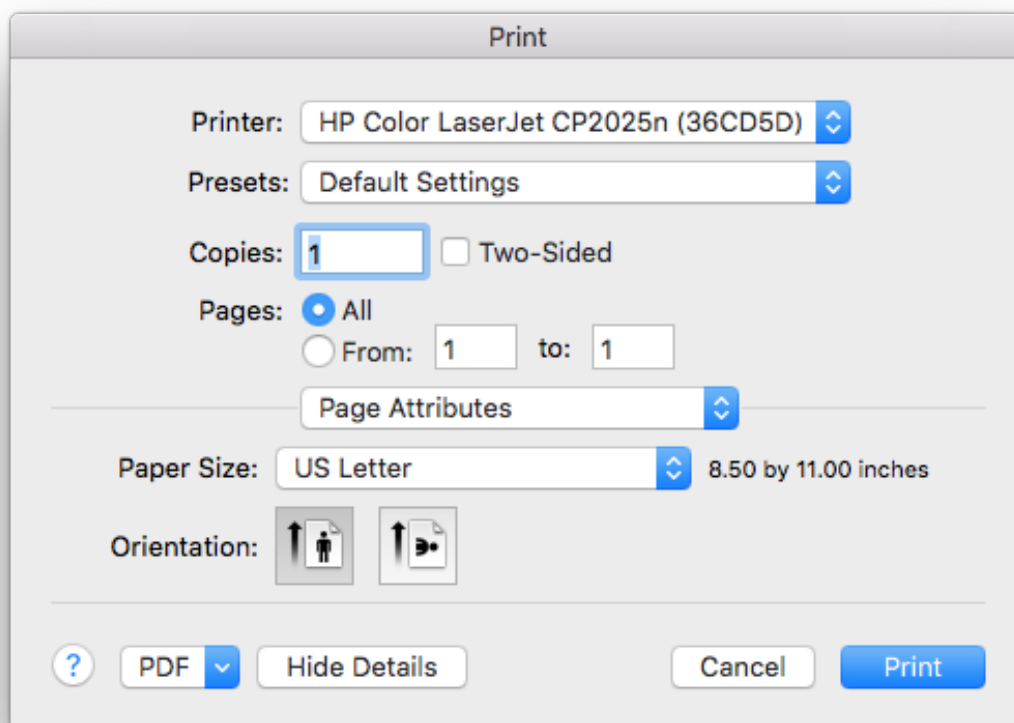
Fig. 1.162: Printer window

1.6 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.

1.6.1 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 1.163), as well as in the vis window's **Toolbars** and **Popup menu**.
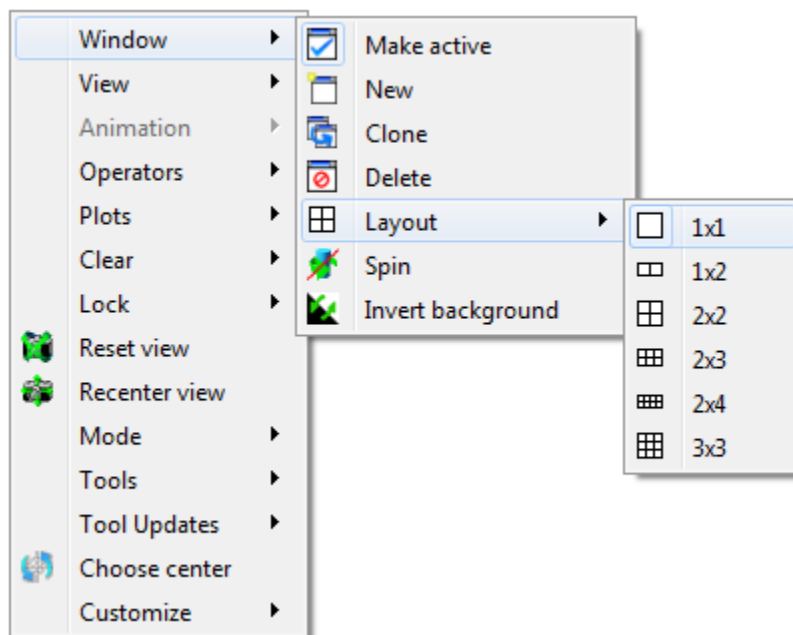
Fig. 1.163: Window menu

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 1.164.

Fig. 1.164: 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 1.165). 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 1.166. 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**.
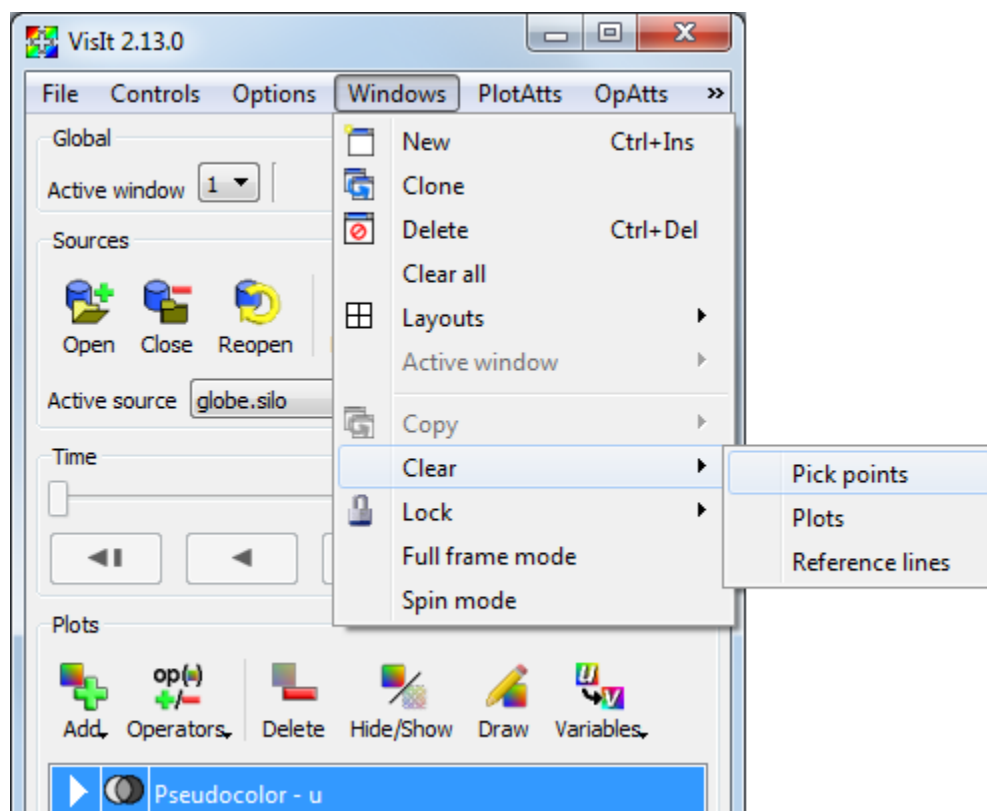
Fig. 1.165: Clear menu
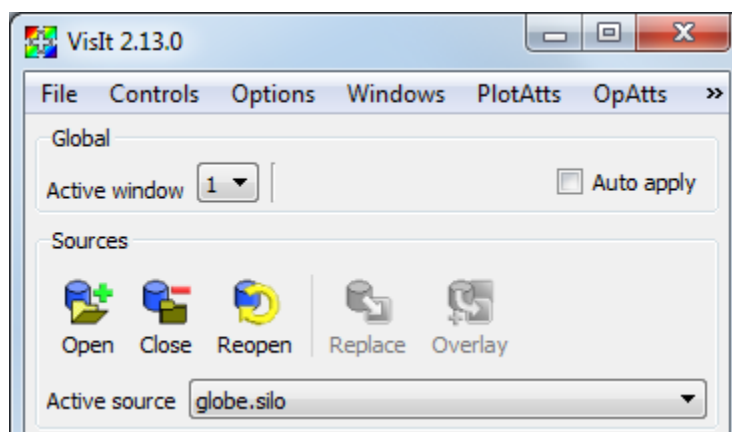
Fig. 1.166: Active window menu

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 1.167, 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.
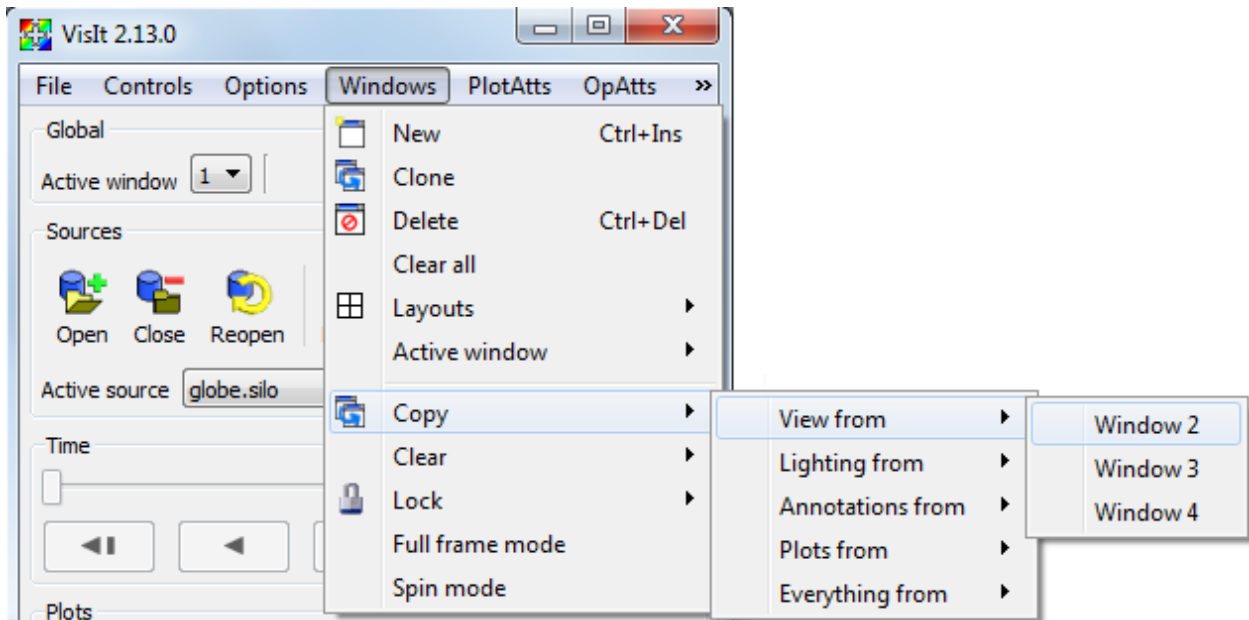
Fig. 1.167: Copy menu

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 1.168) 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
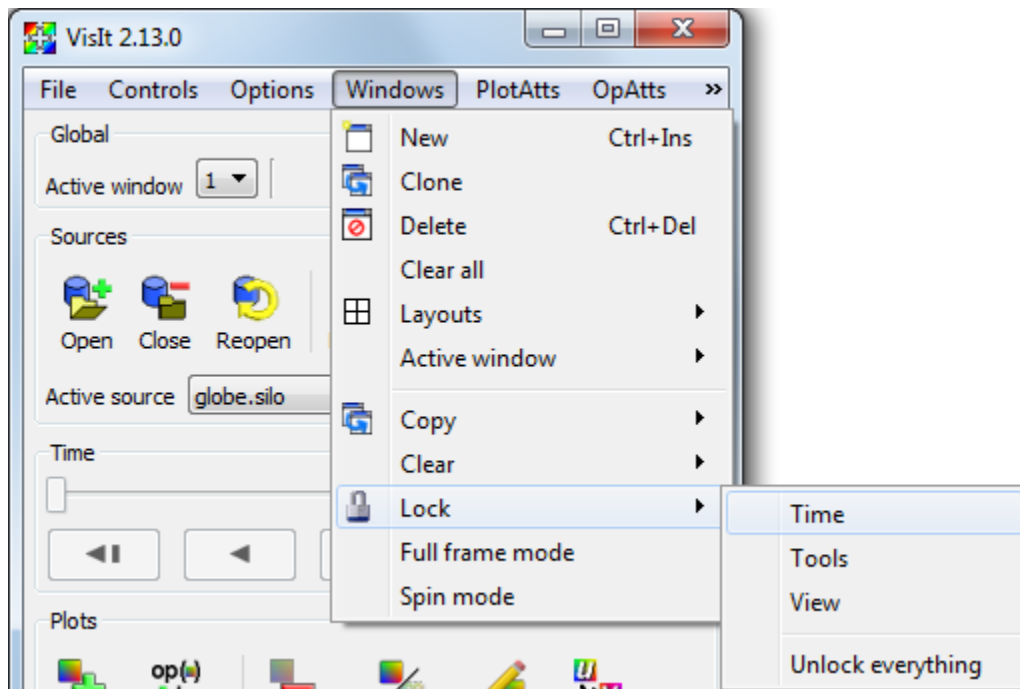
Fig. 1.168: Lock menu

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**.

1.6.2 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 1.169. 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*.
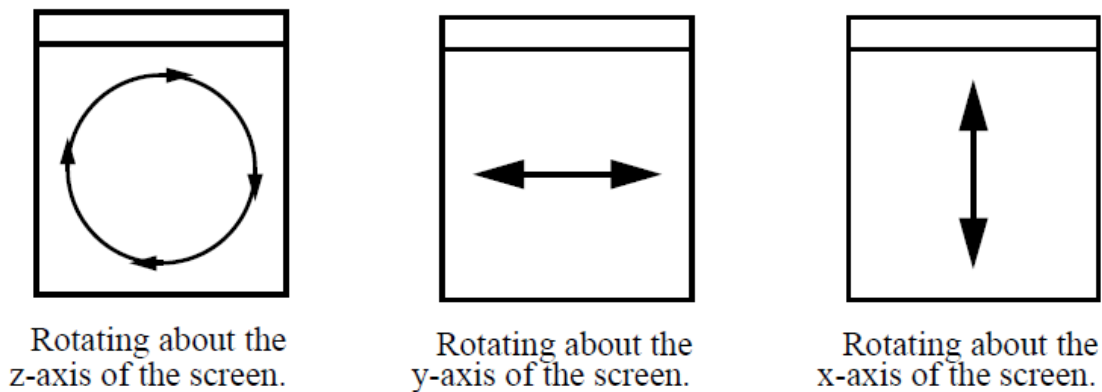
Fig. 1.169: Mouse motions used to rotate plots in navigate mode

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**.

1.6.3 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 1.170.
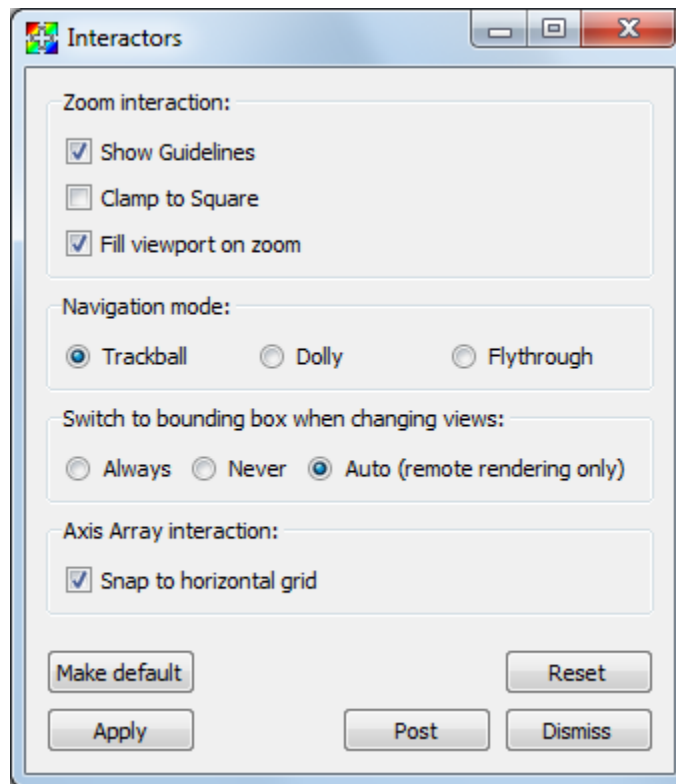
Fig. 1.170: Interactors window

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.

1.6.4 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.

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 1.172) 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.
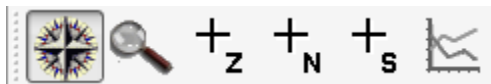
Activating tools

The **Popup menu** contains a **Tools** menu (see Figure 1.173) 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
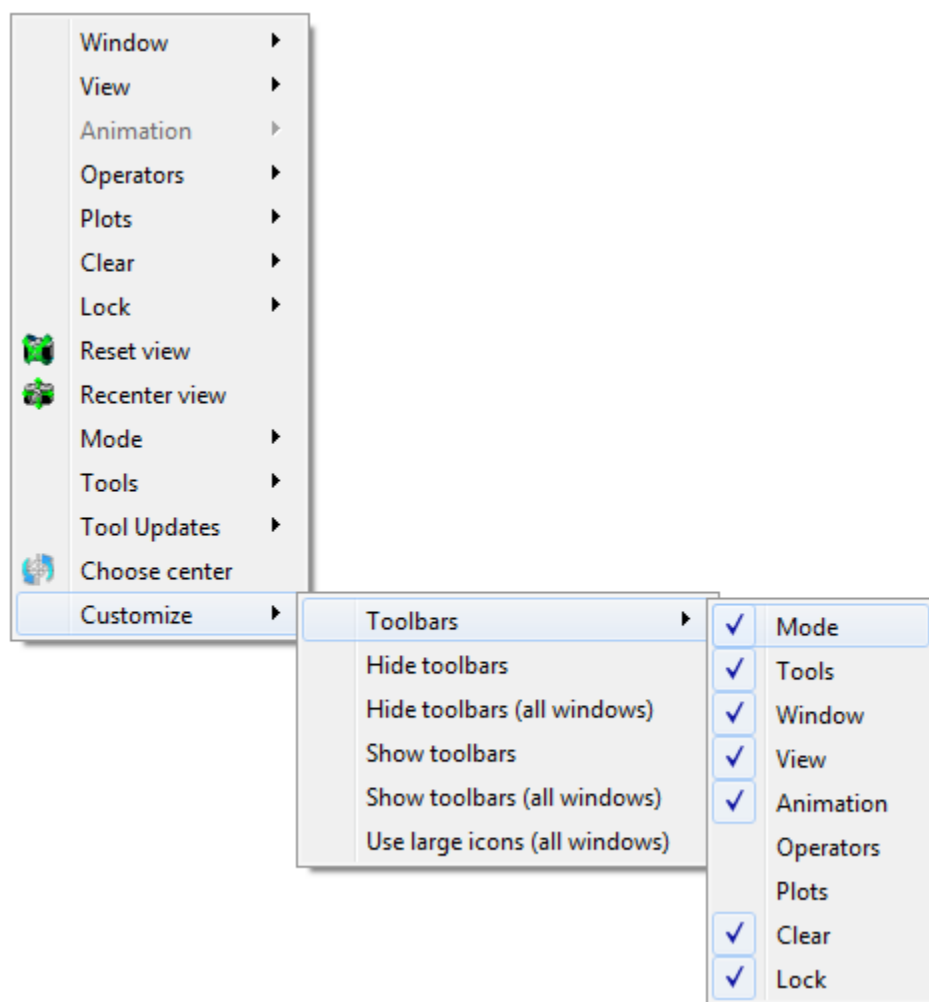
Fig. 1.171: Customize menu
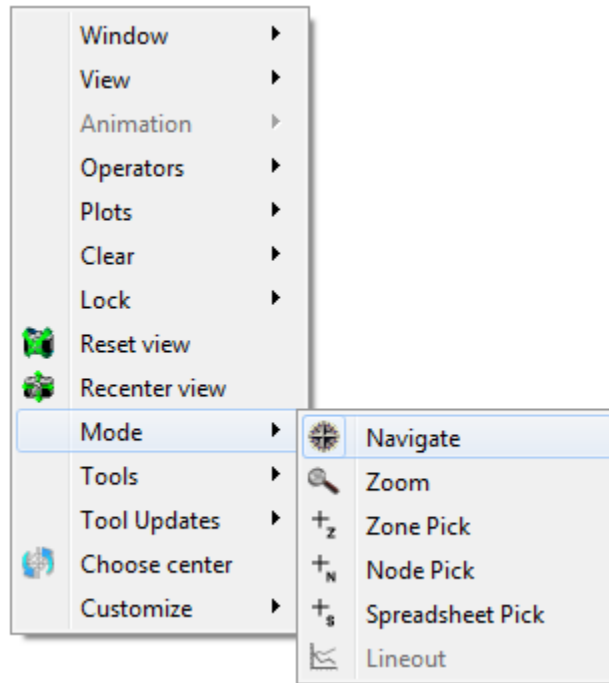
Fig. 1.172: Mode toolbar and menu

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**.
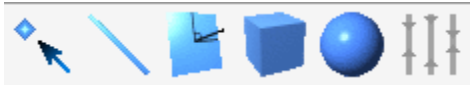
View options

VisIt's **Popup menu** and **Toolbar** (see Figure 1.174) 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 1.174) 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 1.174) to calculate a new center of rotation for the plots so they are drawn in the center of the window. You can
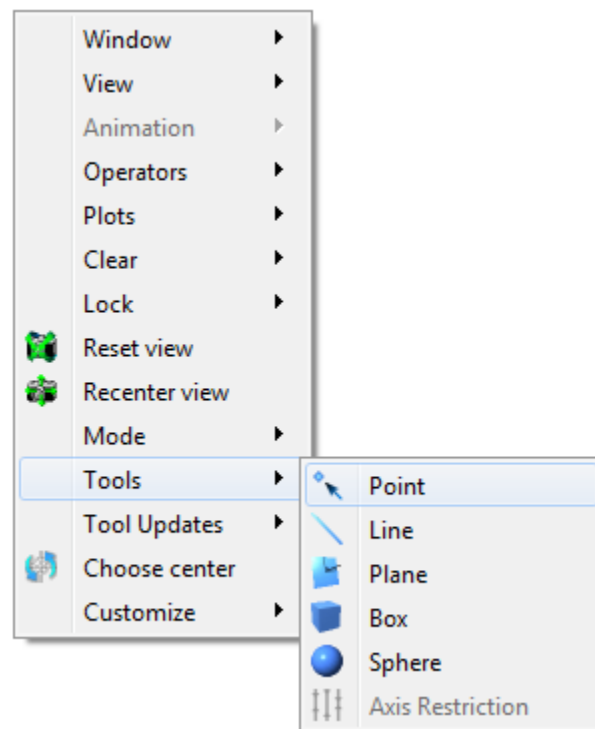
Fig. 1.173: Tool toolbar and menu
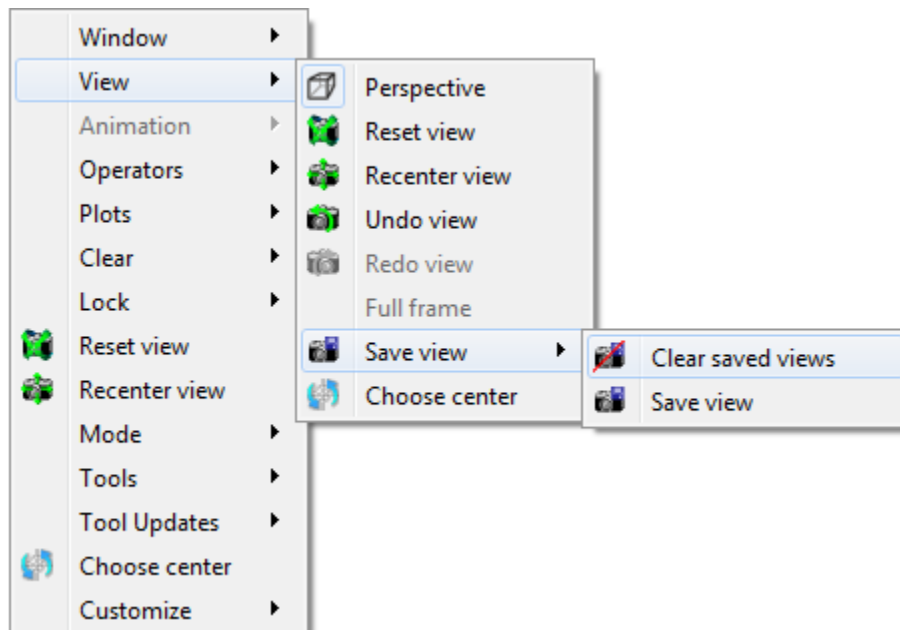
Fig. 1.174: View toolbar and menu

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** .

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 1.174).

Changing view perspective
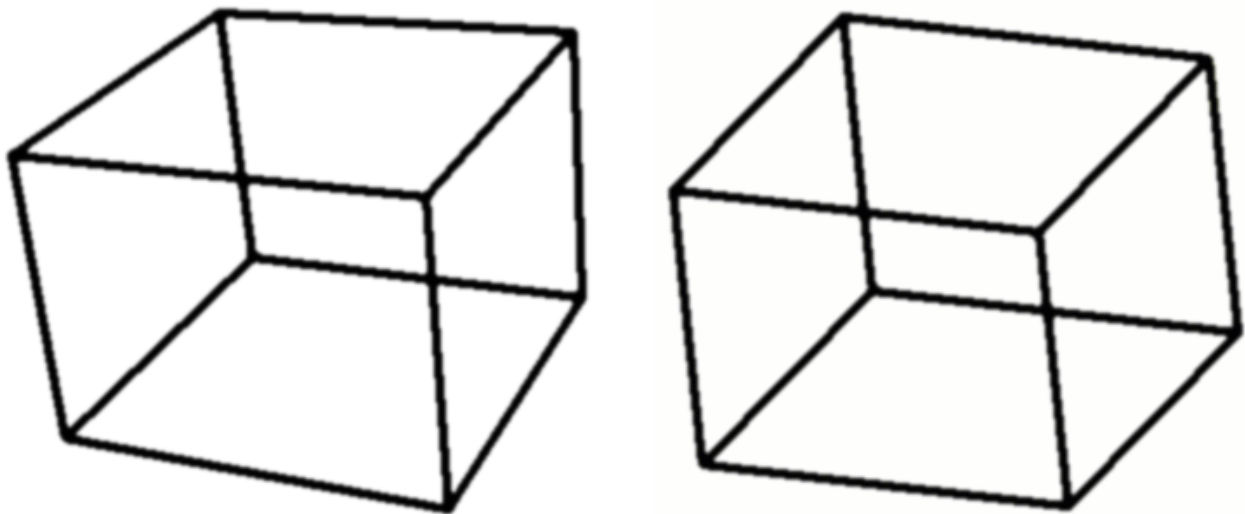
Fig. 1.175: Perspective examples

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 1.174). 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 1.175.

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 1.174) 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 1.176. 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.

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 1.177).
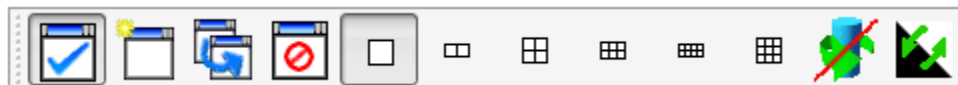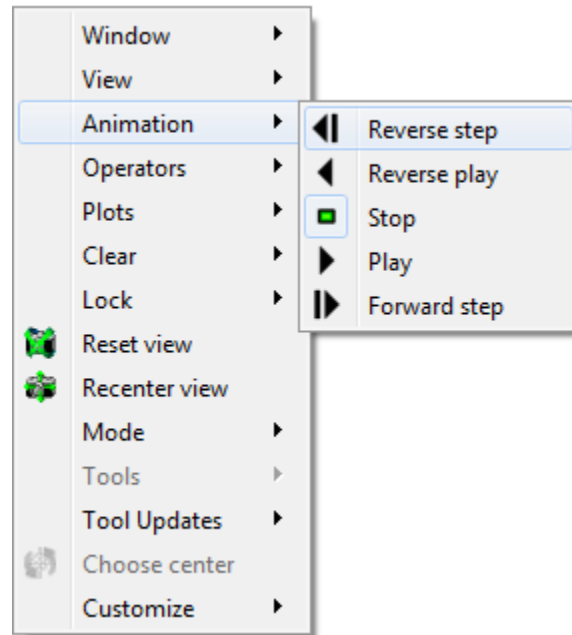
Fig. 1.176: Animation toolbar and menu
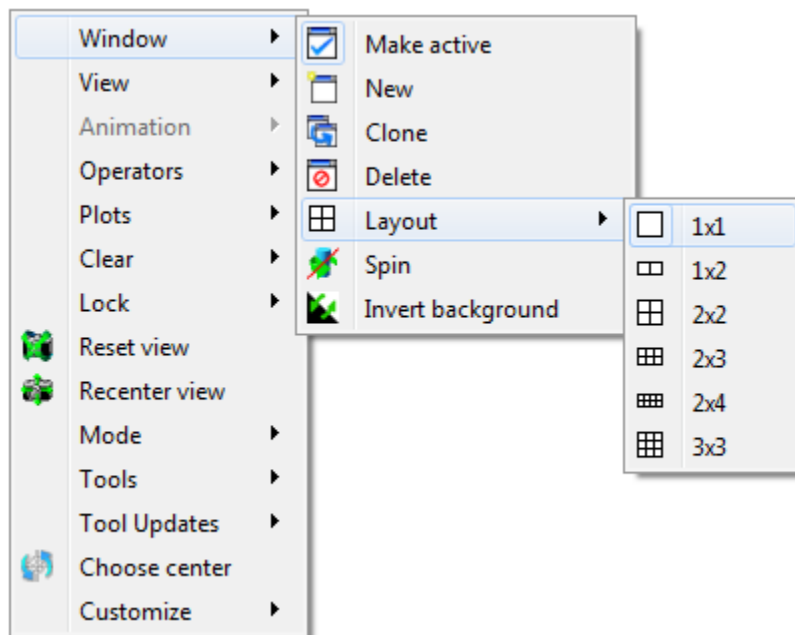
Fig. 1.177: Window toolbar and menu

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 1.177. You can also change the bounding-box mode by clicking on the **Bounding-box icon** in the **Toolbar**.

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 1.178) 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.
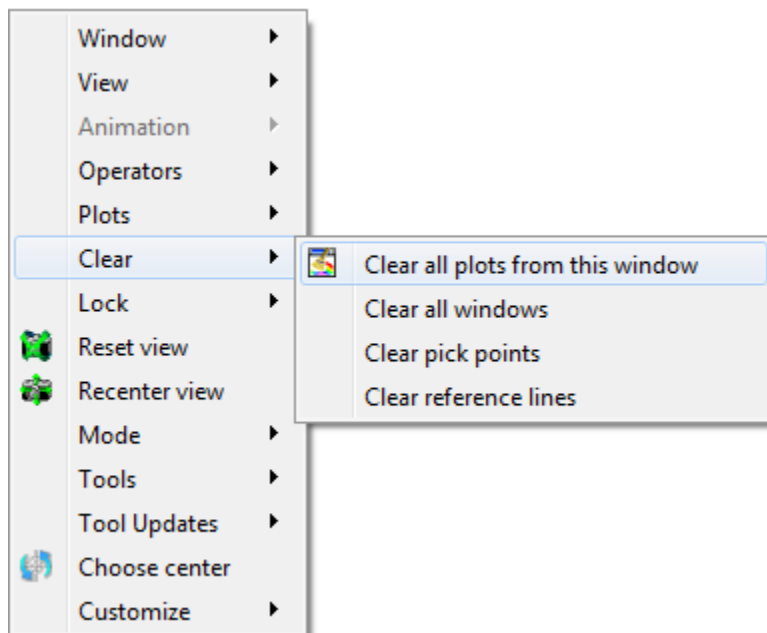
Fig. 1.178: Clear menu

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 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 1.179.

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.
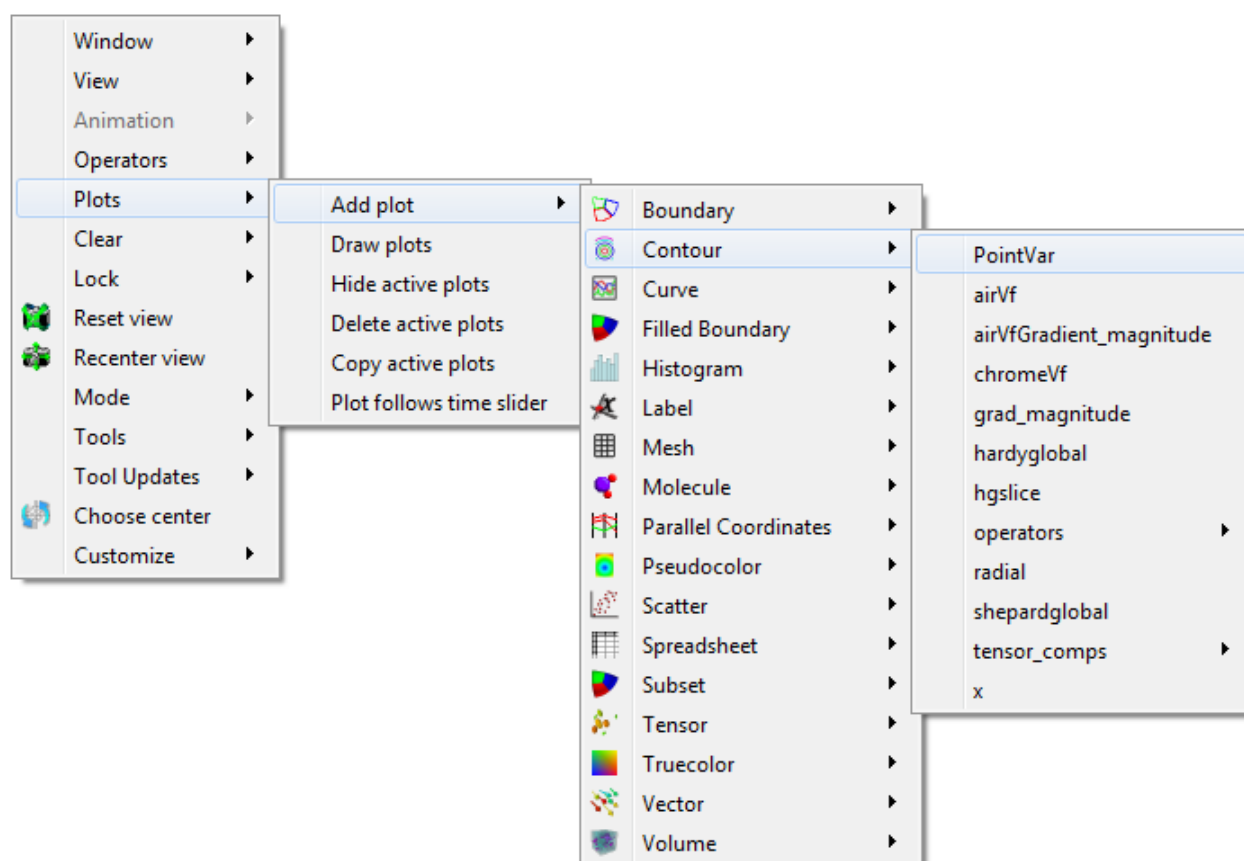
Fig. 1.179: Plot toolbar and menu

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 1.180.

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.

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 1.181, 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.

1.7 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.

	Window	▶
	View	▶
	Animation	▶
	Operators	▶
	Plots	▶
	Clear	▶
	Lock	▶
	Reset view	
	Recenter view	
	Mode	▶
	Tools	▶
	Tool Updates	▶
	Choose center	
	Customize	▶

Add operator	▶
Remove last operator	
Remove all operators	

- AMR Dual Grid and Stitch Cells
- Boundary
- Box
- Cartographic Projection
- Clip
- Cone
- CoordSwap
- CreateBonds
- Cylinder
- Displace
- Dual Mesh
- Elevate
- Extrude
- Index Select
- IntegralCurve
- Inverse Ghost Zone
- Isosurface
- Isovolume
- LCS
- LimitCycle
- Onion Peel
- PersistentParticles
- Poincare
- Project
- Reflect
- Replicate
- Resample
- Revolve
- Slice
- Spherical Slice
- StatisticalTrends
- SurfaceNormal
- ThreeSlice
- Threshold
- Transform
- Tube

Fig. 1.180: Operator toolbar and menu

Chapter 1. VisIt GUI User Manual

Fig. 1.181: Lock toolbar and menu

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.

1.7.1 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.

1.7.2 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 1.182 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.
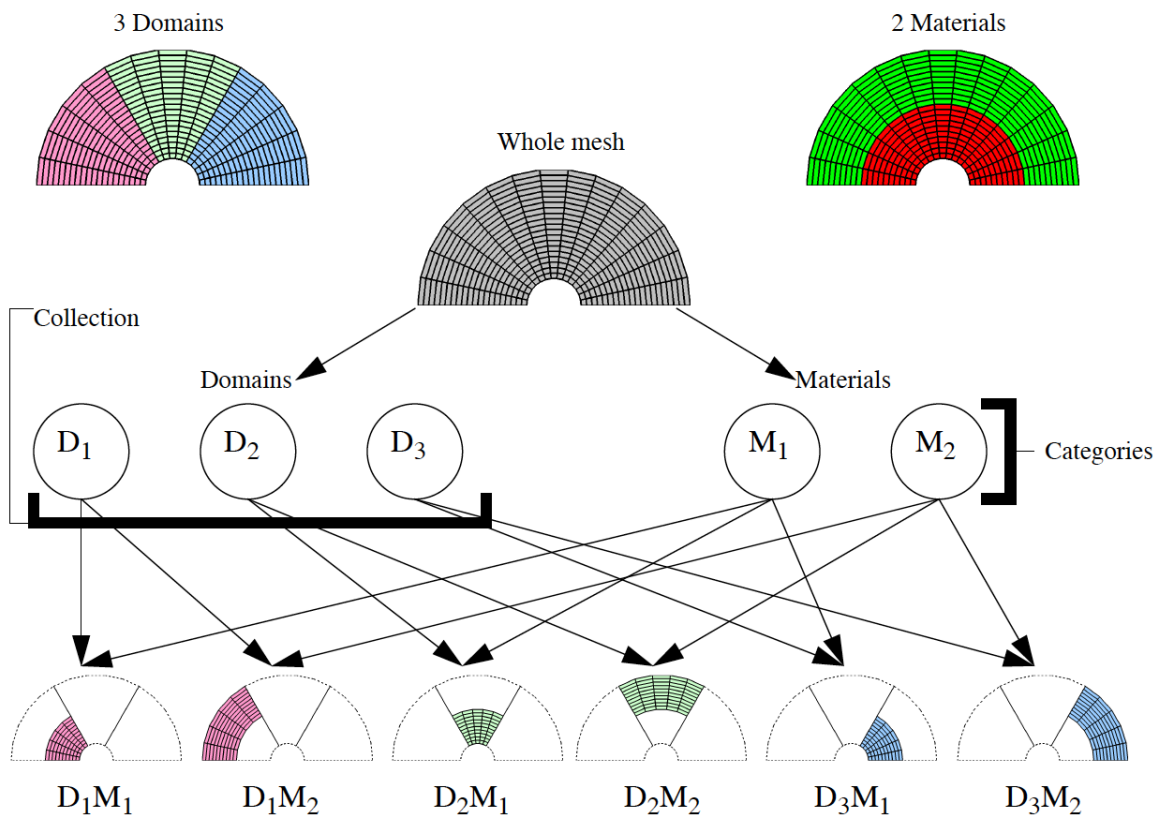
Fig. 1.182: Whole mesh divided up into domains and materials

1.7.3 Using the Subset Window

Users can open the **Subset Window**, shown in Figure 1.183, 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.
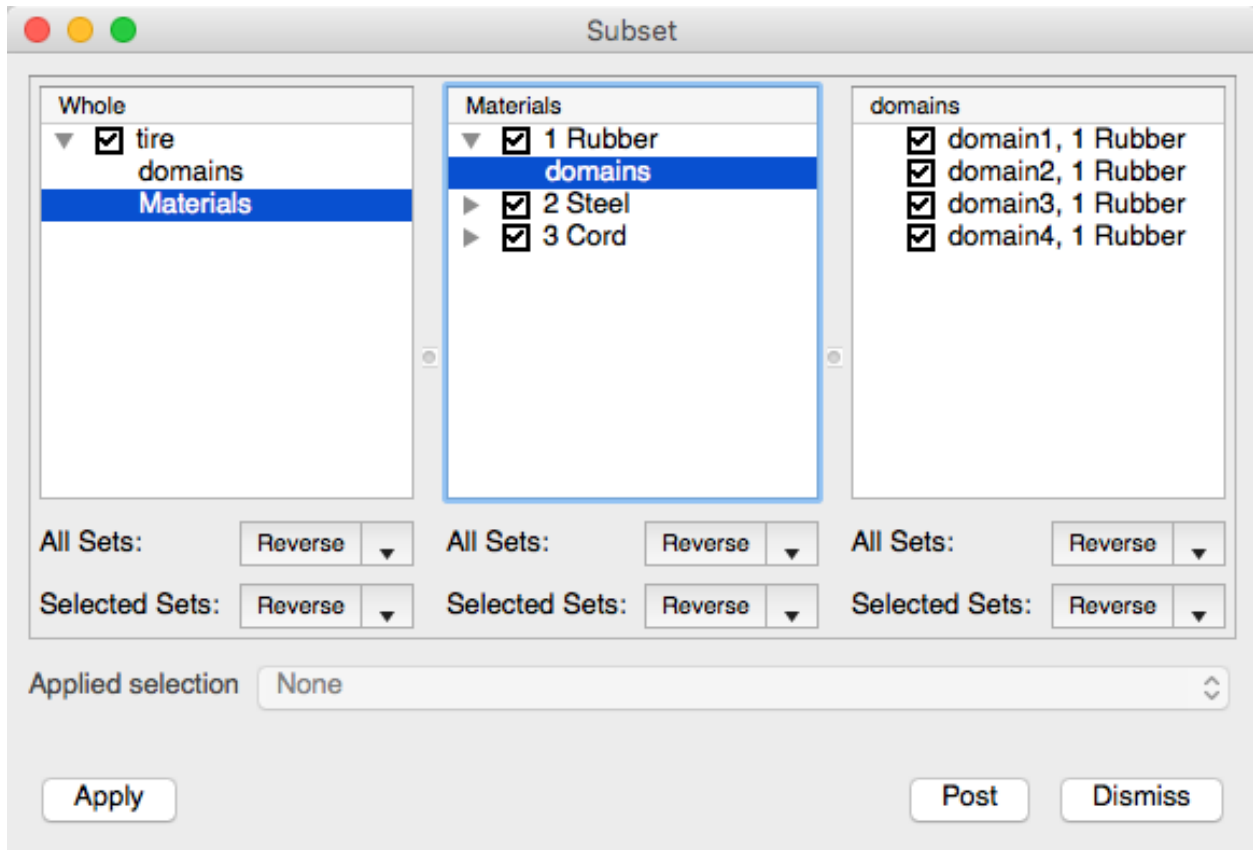
Fig. 1.183: Subset window

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 1.184, 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.
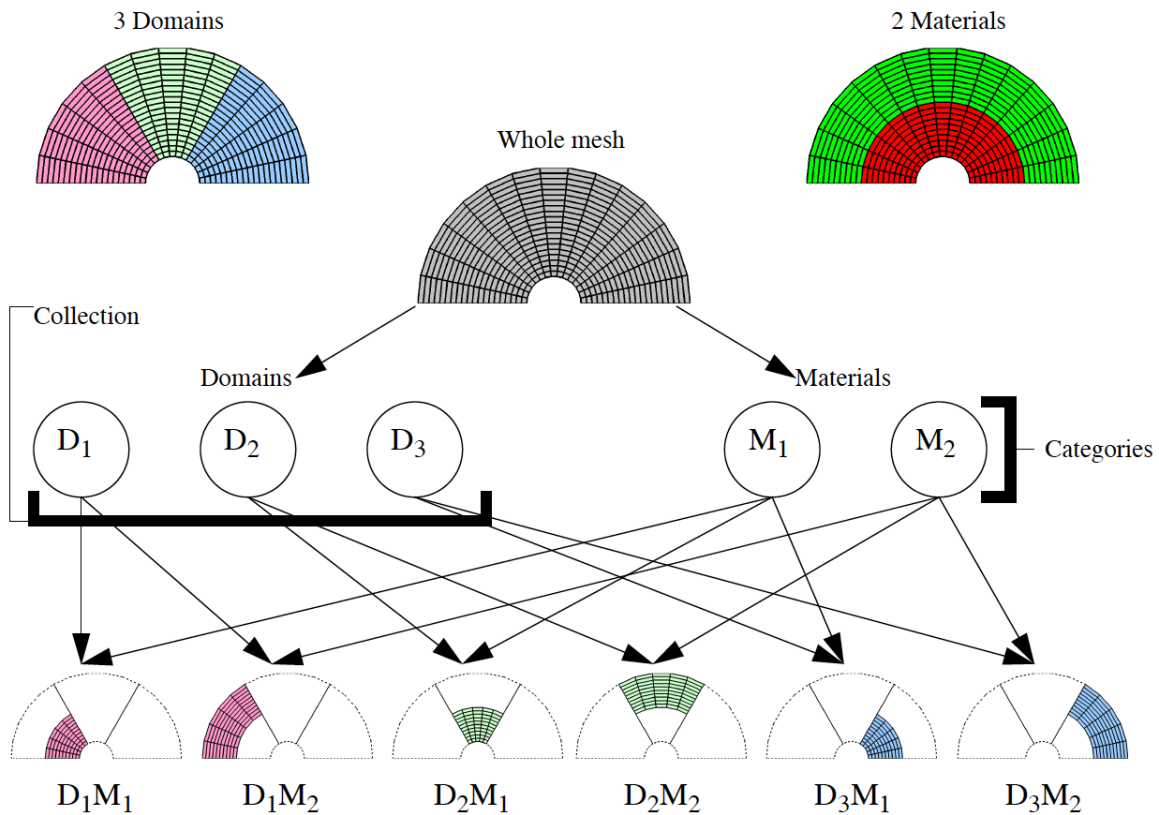
Fig. 1.184: Removing one subset.

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
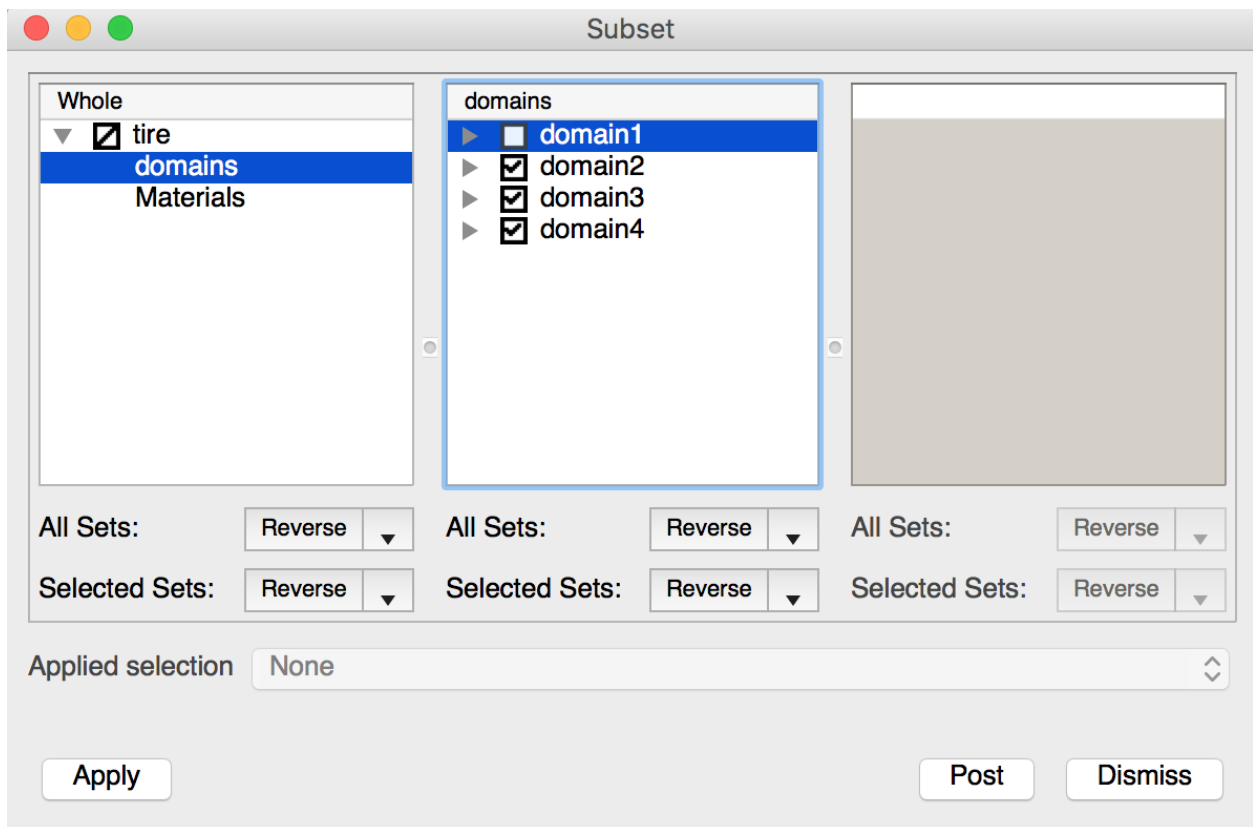
Fig. 1.185: Subset window with one subset removed.

the "domain4" subset. An example of a complex subset selection is shown in Figure 1.186 and the state of the **Subset window** is shown in Figure 1.187.
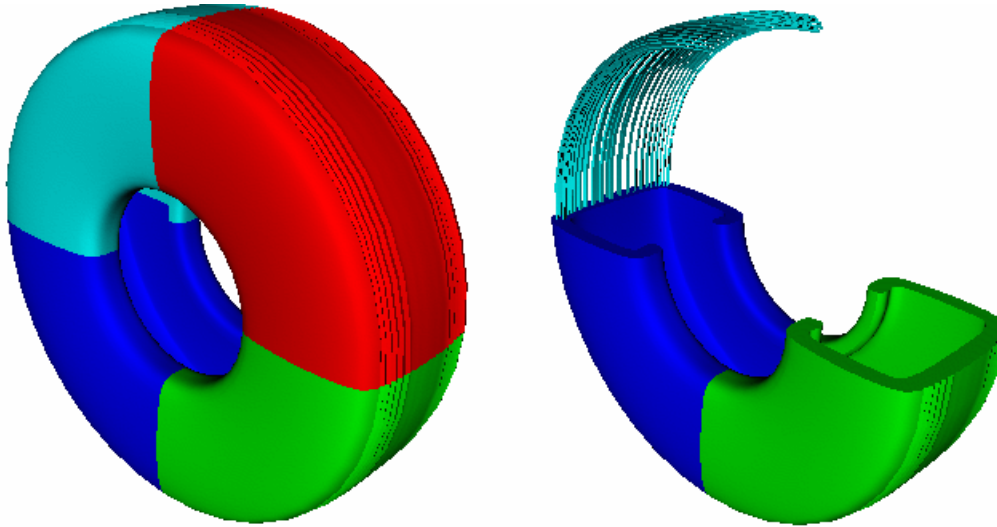
Fig. 1.186: Example of a complex subset.

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.

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).

1.7.4 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
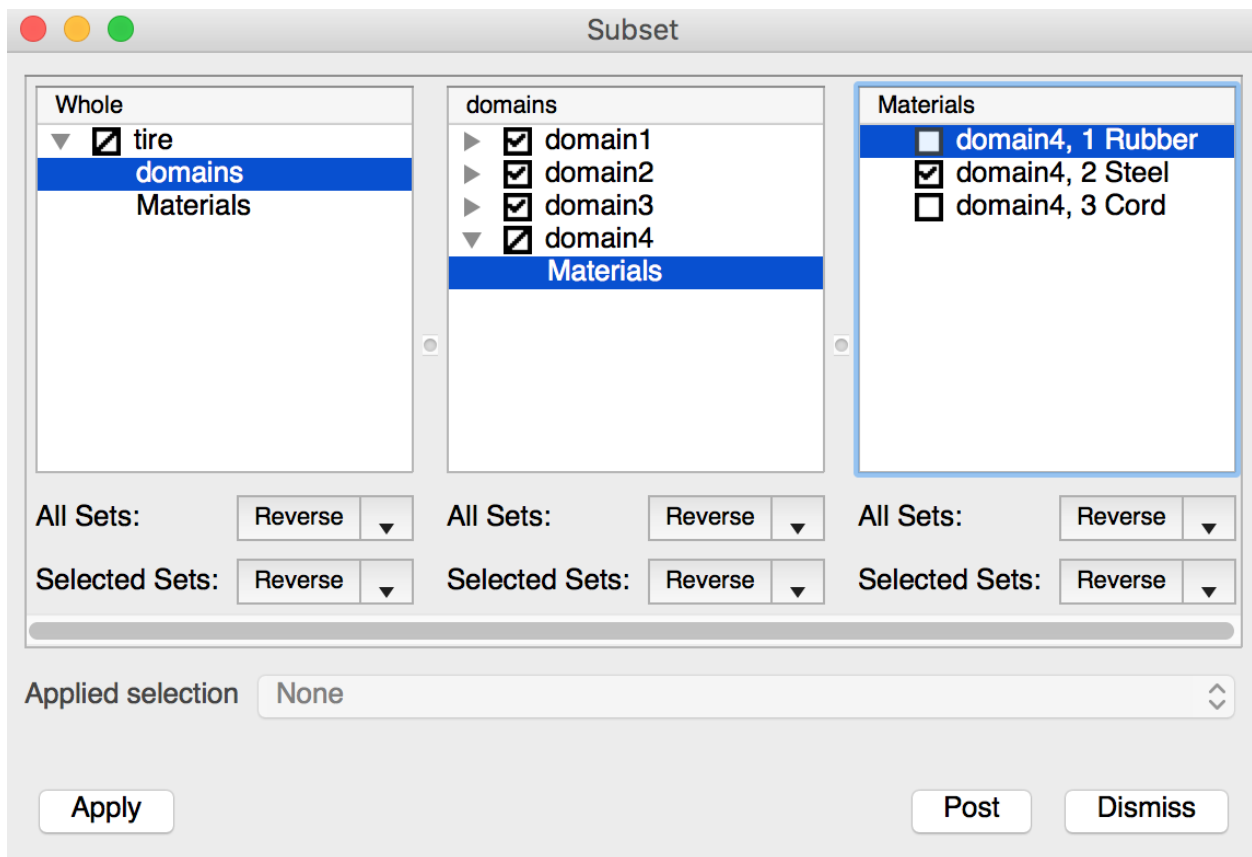
Fig. 1.187: Subset window for complex subset example.

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 1.188), 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**.
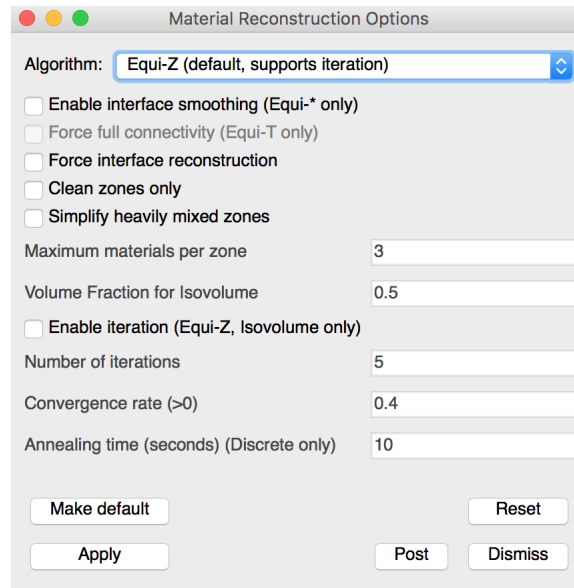
Fig. 1.188: Material Reconstruction Options Window

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.

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].
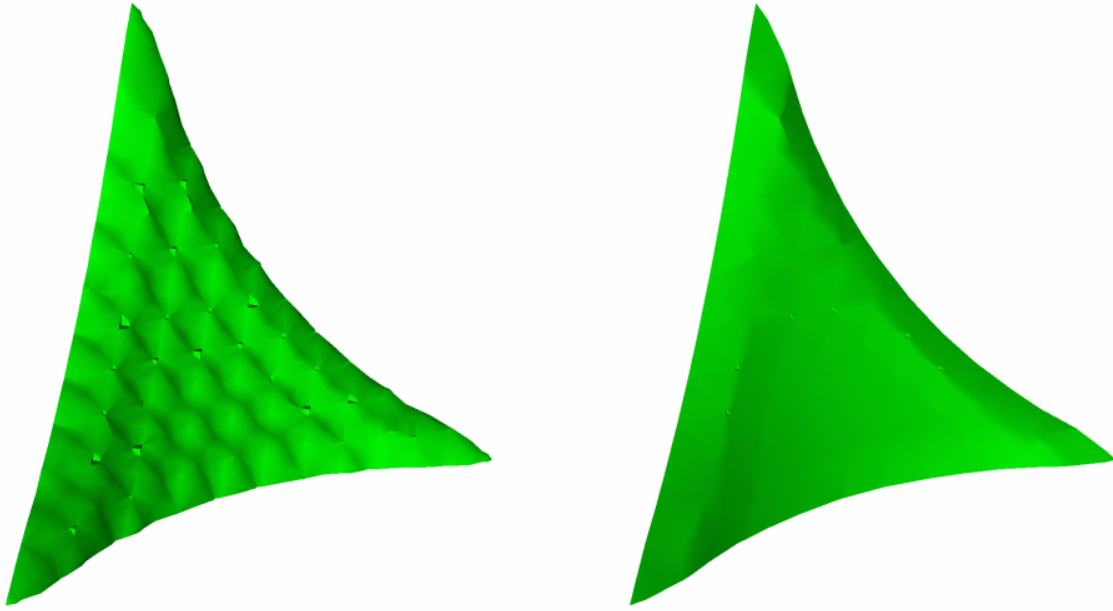
Fig. 1.189: Tetrahedral MIR vs. Zoo-based MIR

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 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 1.191 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.

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.
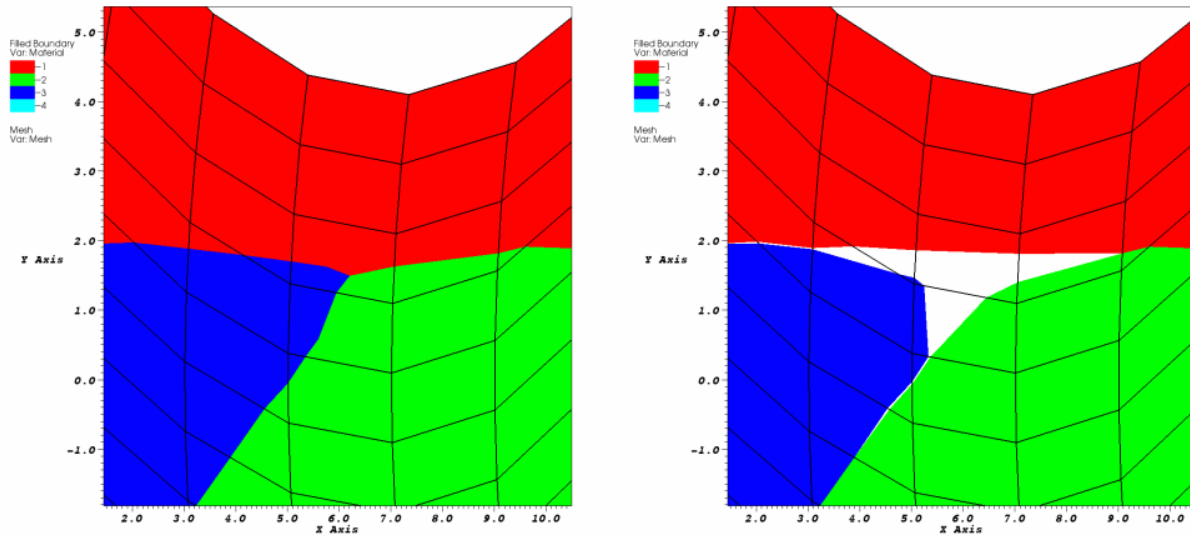
Fig. 1.190: Zoo-based MIR vs. Isovolume MIR
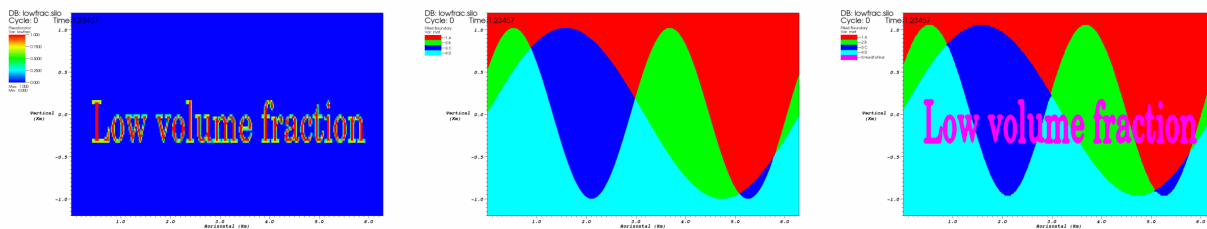
Fig. 1.191: Materials with low volume fractions can be found with the isosurface MIR algorithm

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 1.192 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.
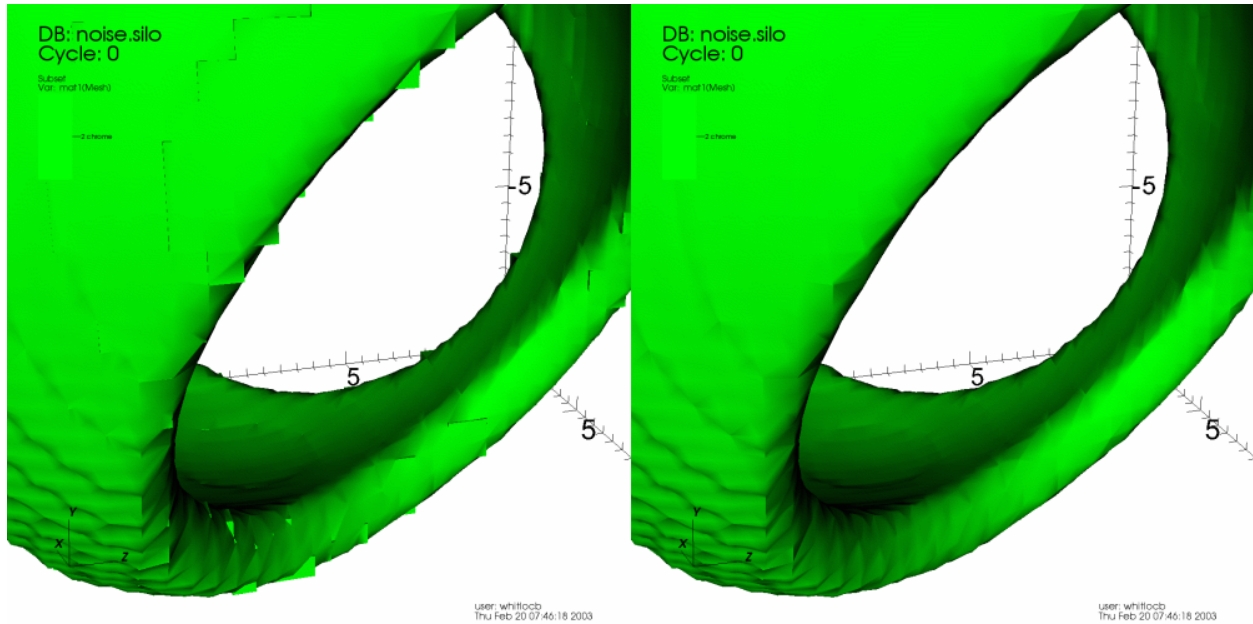
Fig. 1.192: Effect of material interface smoothing

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 1.193). 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.
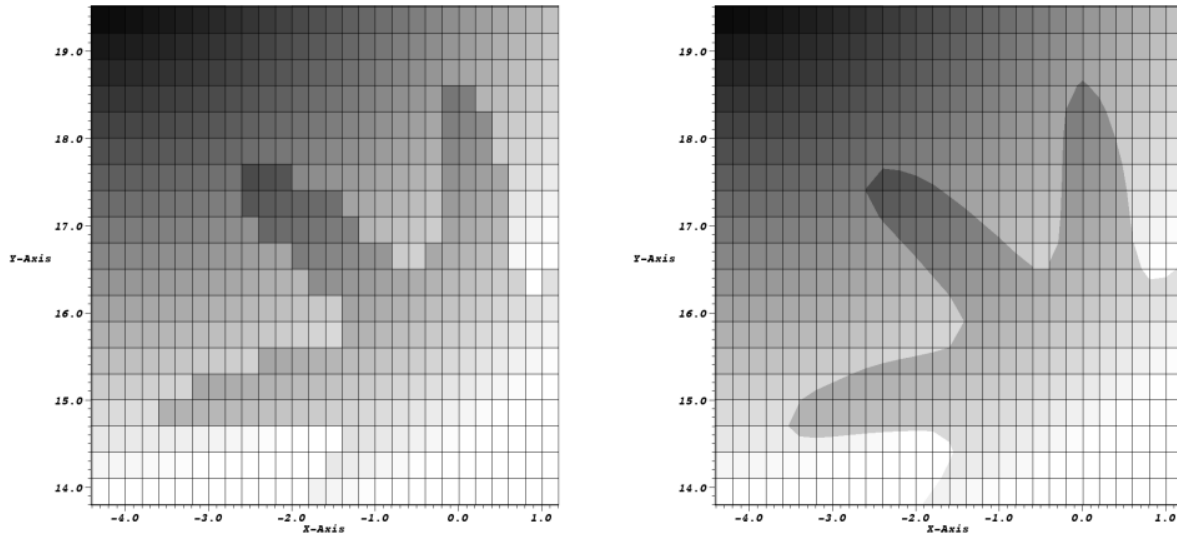
Fig. 1.193: Mixed variables can improve a visualization

1.7.5 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.
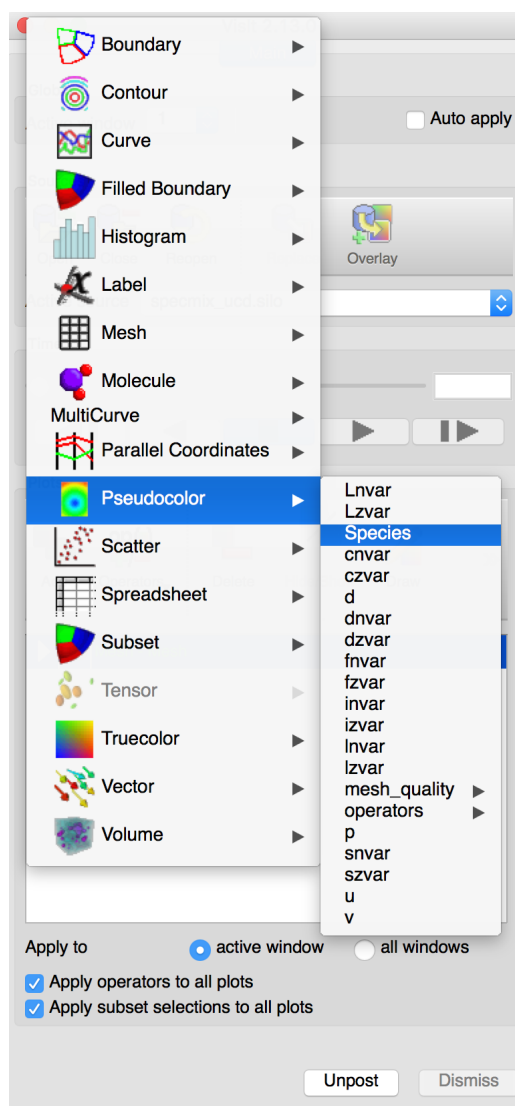
Fig. 1.194: Species variable

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 1.195). 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.
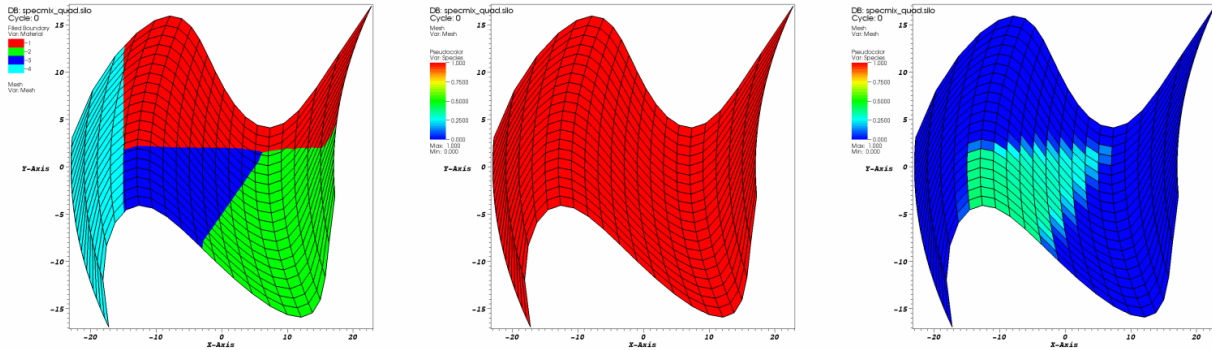
Fig. 1.195: Plots of materials and species

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 1.196) 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.

1.8 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.

1.8.1 Expressions

> **Danger:** Confirm the text here adequately characterizes that an expression has value everywhere over the whole mesh it is defined on. Its a field.

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
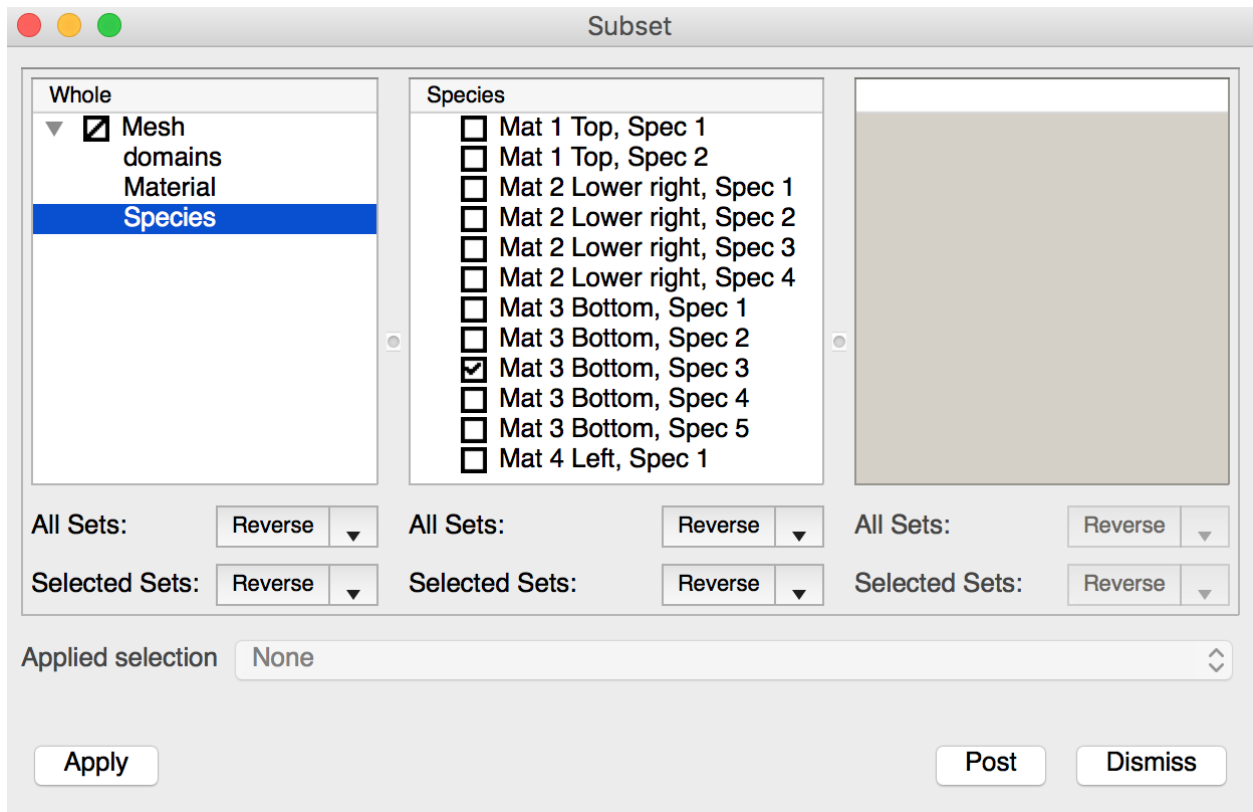
Fig. 1.196: Turning off species in the Subset Window

other variables using a variable expression. VisIt provides variable 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.

Expression Window

VisIt provides an **Expression Window**, shown in Figure 1.197, 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.
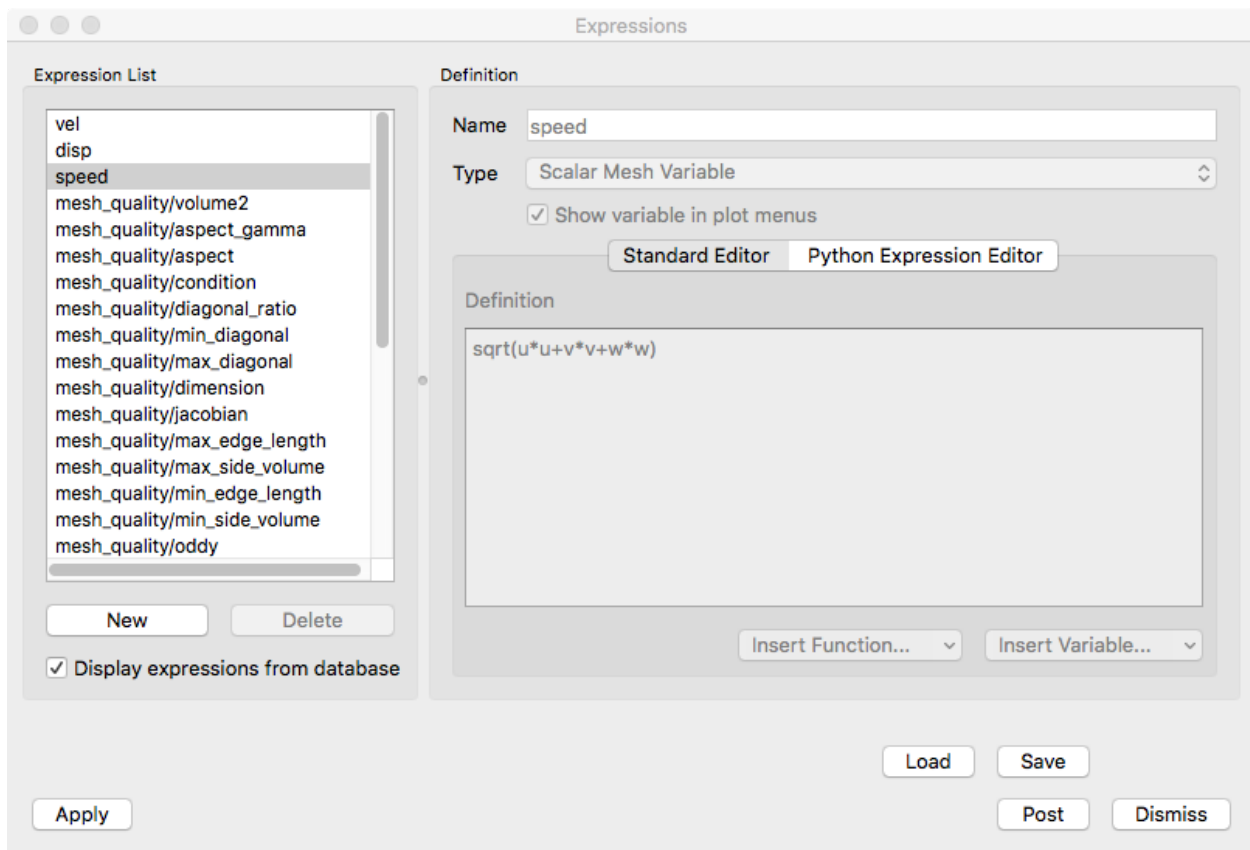
Fig. 1.197: Expression Window

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. As the user types a new name for the expression into the **Name** text field, the expression's name in the **Expression list** will update.

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 1.198, 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 1.198, 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 1.199, 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
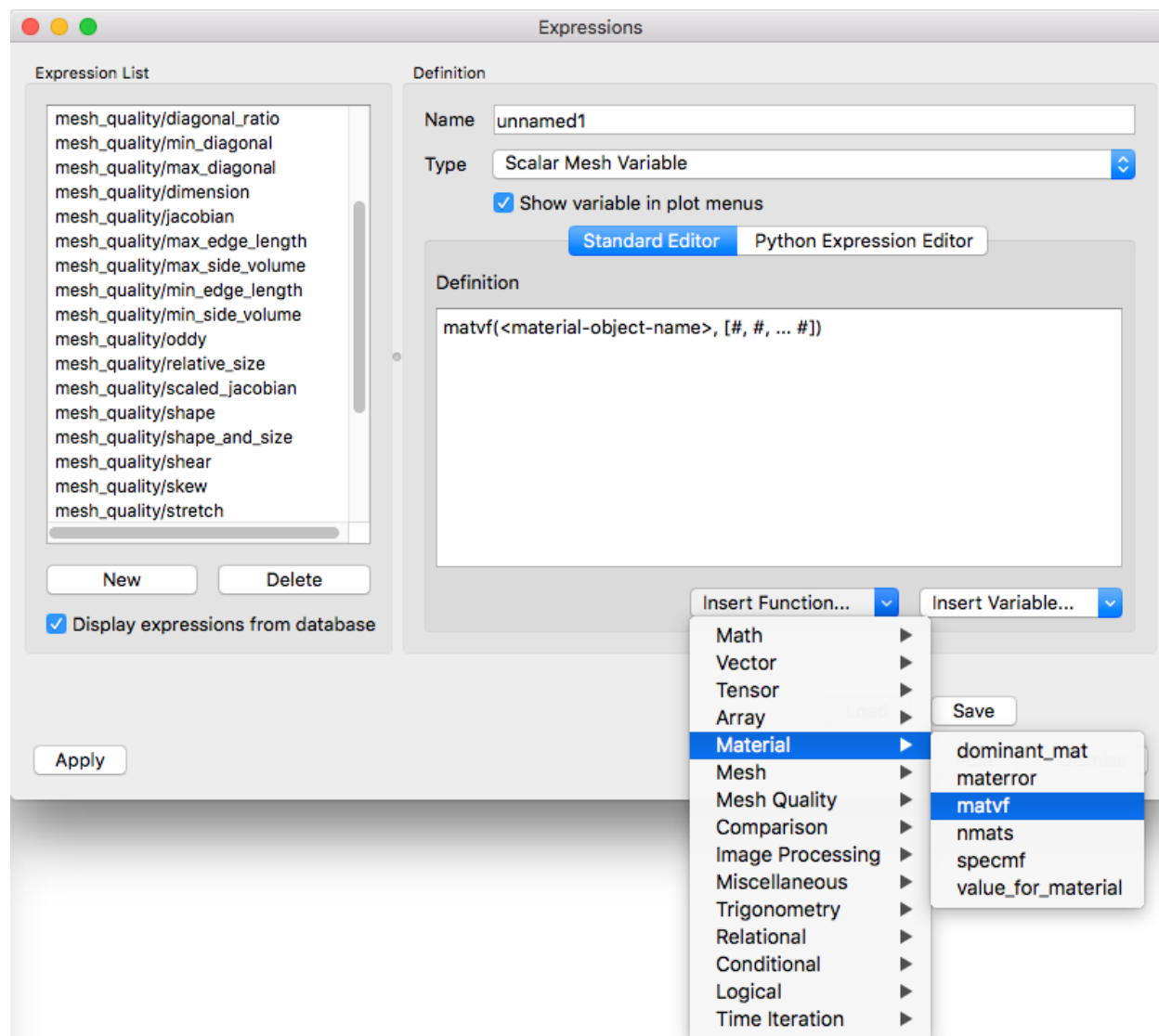
Fig. 1.198: Expression Window's Insert Function... menu
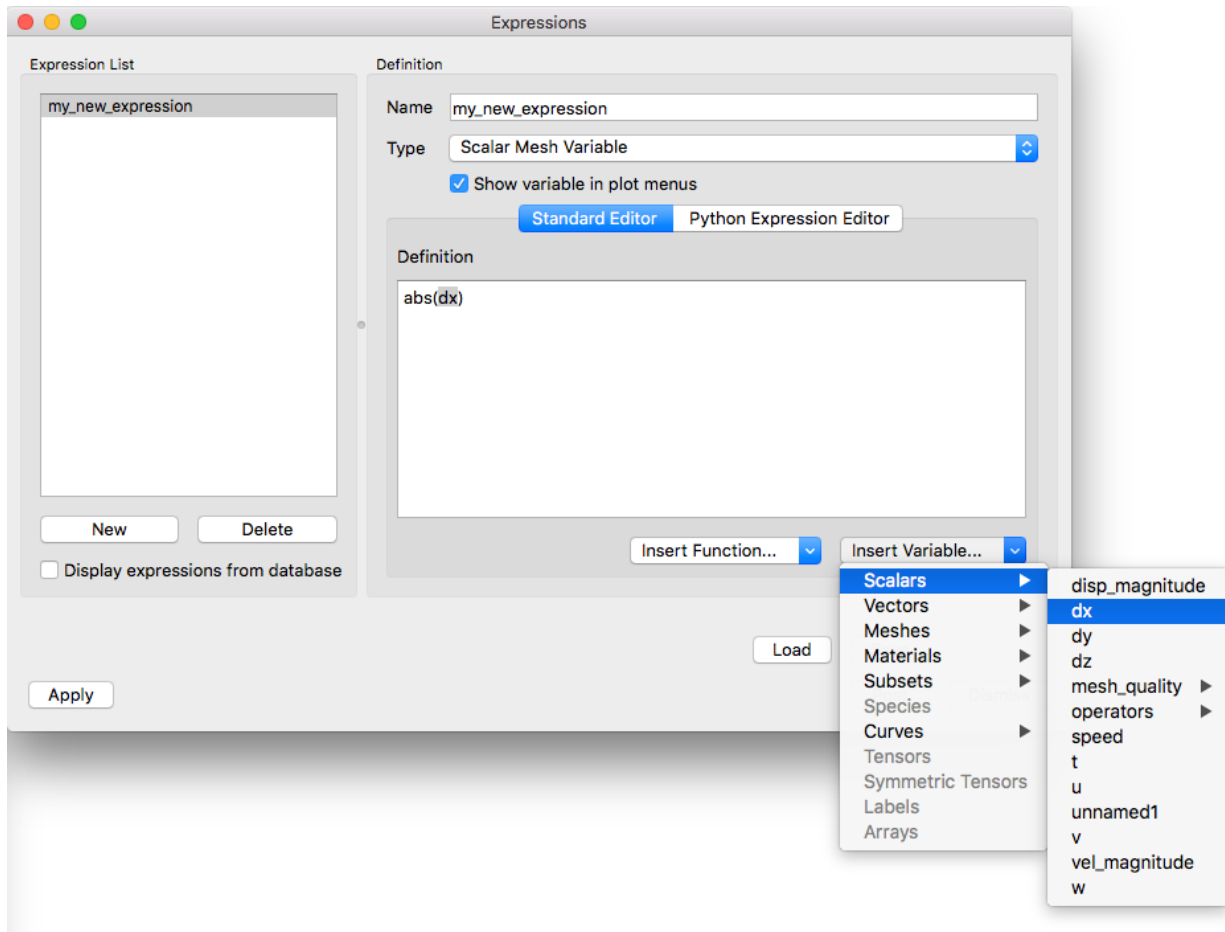
Fig. 1.199: Expression Window's Insert Variable… menu

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 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

> **Danger:** Add examples for some of the more complicated cases.

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 quotient after dividing the exprL expression by the exprR expression.

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.

Pairwise Max Function (max()) [max(expr0,exrp1)] Creates a new expression which is everywhere the pairwise maximum of its two arguments.

Pairwise Min Function (min()) [min(expr0,exrp1)] Creates a new expression which is everywhere the pairwise minimum of its two arguments.

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 (rand()%1024) ÷ 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 relation 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.

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.

> **Danger:** Confirm relational and logical expressions produce new, boolean valued expression variables which are themselves plottable in VisIt. Their original intent may have been only to be used as args in the IF expression and not so much be plottable outputs in their own right.

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*.

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 new expression of higher tensor rank from 2 or more expression of lower tensor rank. A common use is to compose several tensor rank 0 expressions (e.g. scalar expressions) into a tensor rank 1 expression (e.g. a vector expression). The component expressions, `expr0`, `expr1`, etc. must all be the same tensor rank and expression type. For example, they must all be rank 0 (e.g. *scalar* expressions) or they must all be rank 1 (e.g. *vector*) expressions of the same number of components. If they are all scalars, the result is a tensor of rank 1 (e.g. a vector). If they are all vectors, the result is a tensor of rank 2 (e.g. a tensor). The vector compose operator is also used to compose array expressions.

For example, the expression `{u, v, w}` takes three scalar mesh variables named `u`, `v` and `w` and creates a vector mesh variable.

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.

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

Contraction Function: `contraction()` `[contraction(expr0)]` No description available.

Determinant Function: `determinant()` `[determinant(expr0)]` No description available.

Effective Tensor Function: `effective_tensor()` `[effective_tensor(expr0)]` No description available.

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.

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)]` Creates a new vector expression which is everywhere the principal deviatoric stress components of the input argument, which must be a tensor.

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 be a tensor.

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.

Tensor Maximum Shear Function: `tensor_maximum_shear()` `[tensor_maximum_shear(expr0)]` No description available.

Trace Function: `trace()` `[trace(expr0)]` No description available.

Viscous Stress Function: `viscous_stress()` `[viscous_stress(expr0)]` No description available.

Array Expressions

Array Compose Function: `array_compose()` `[array_compose(expr0, expr1, ..., exprN-1)]` 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(expr0,` `...,exprN-1,b0,...bn-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 (you can kinda think of as bin boundaries), that should be used by VisIt for certain kinds of operations involving the array variable. If there are N variables in the array, `expr0`, `expr1`, and so on, there are N+1 coordinate values (or bin boundaries), `b0`, `b1`. When such a variable is picked using one of VisIt's pick operations, VisIt can display a bar-graph. Each bar in the bar-graph has a height determined by the associated scalar mesh variable (at the picked point) 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. . .
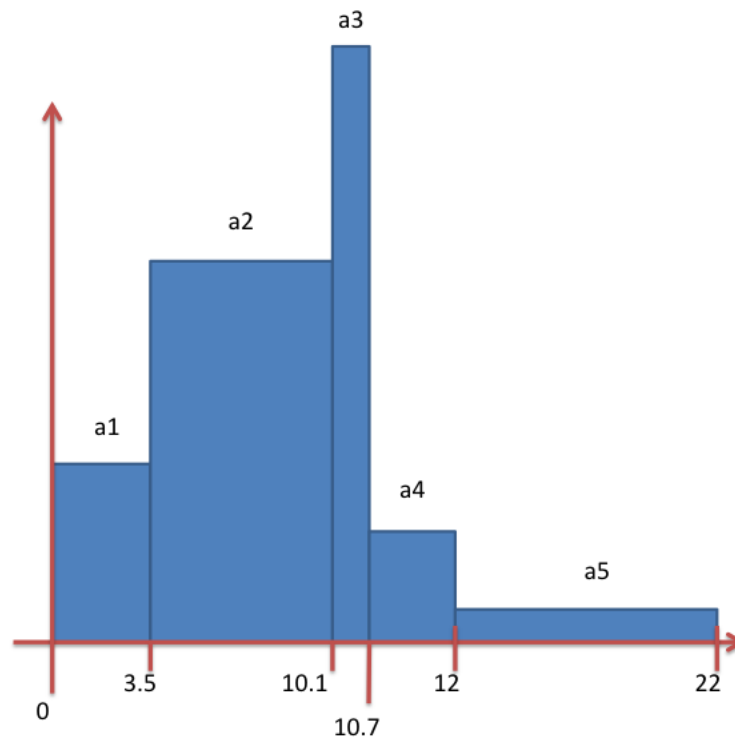
Fig. 1.200: Bar graph created from picking an array variable created with array_compose_with_bins()

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.

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>)] See the Verdict Manual

degree Function: `degree()` [degree(expr0)] No description available.

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. 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

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.

> **Danger:** It is not clear how filtering is handled across different domain boundaries.

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]>)] 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 −1.

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(expr0)] No description available.

lambda2 Function: lambda2() [lambda2(expr0)] No description available.

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.

Laplacian Function: laplacian() [laplacian(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.

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.

> **Danger:** How does this work with changing topology? Also, what is the actual math of the average? Is it an update algorithm or a sum and then division by number of iterations?

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.

Displacement Function: `displacement()` [displacement(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.

Automatic expressions

1.8.2 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 1.201), 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
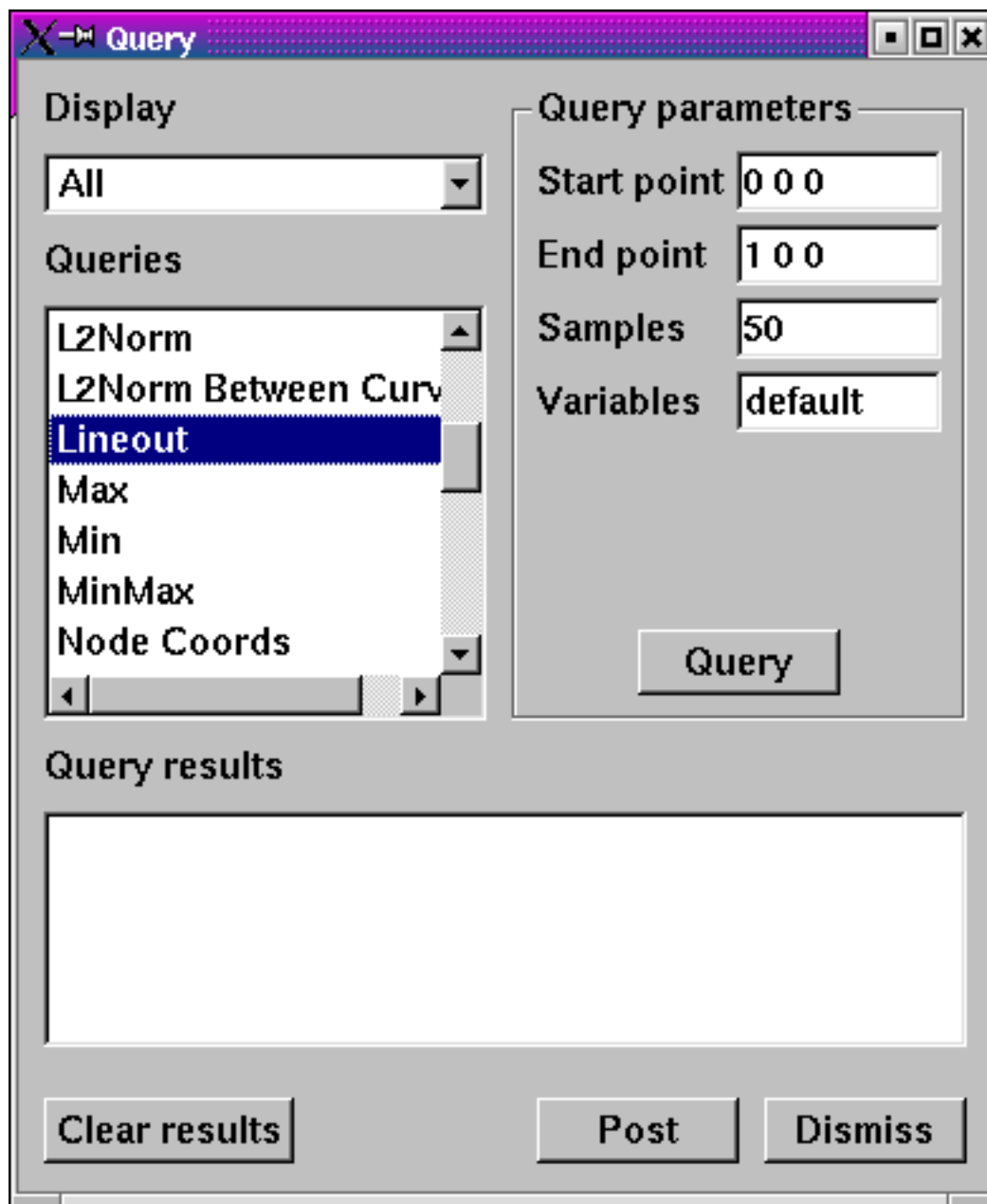
Fig. 1.201: Query window

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 1.202) and as you highlight different queries, the controls in the **Query parameters** area may change.
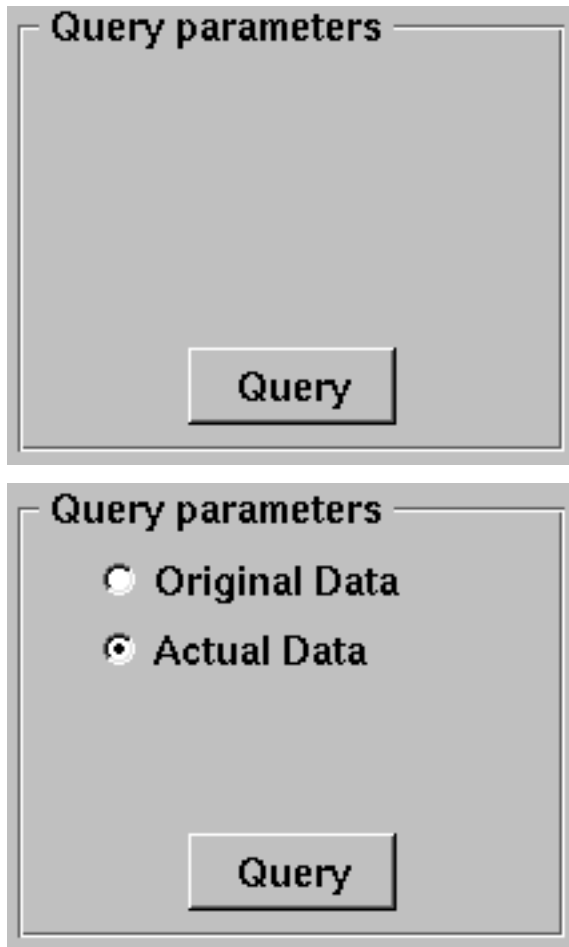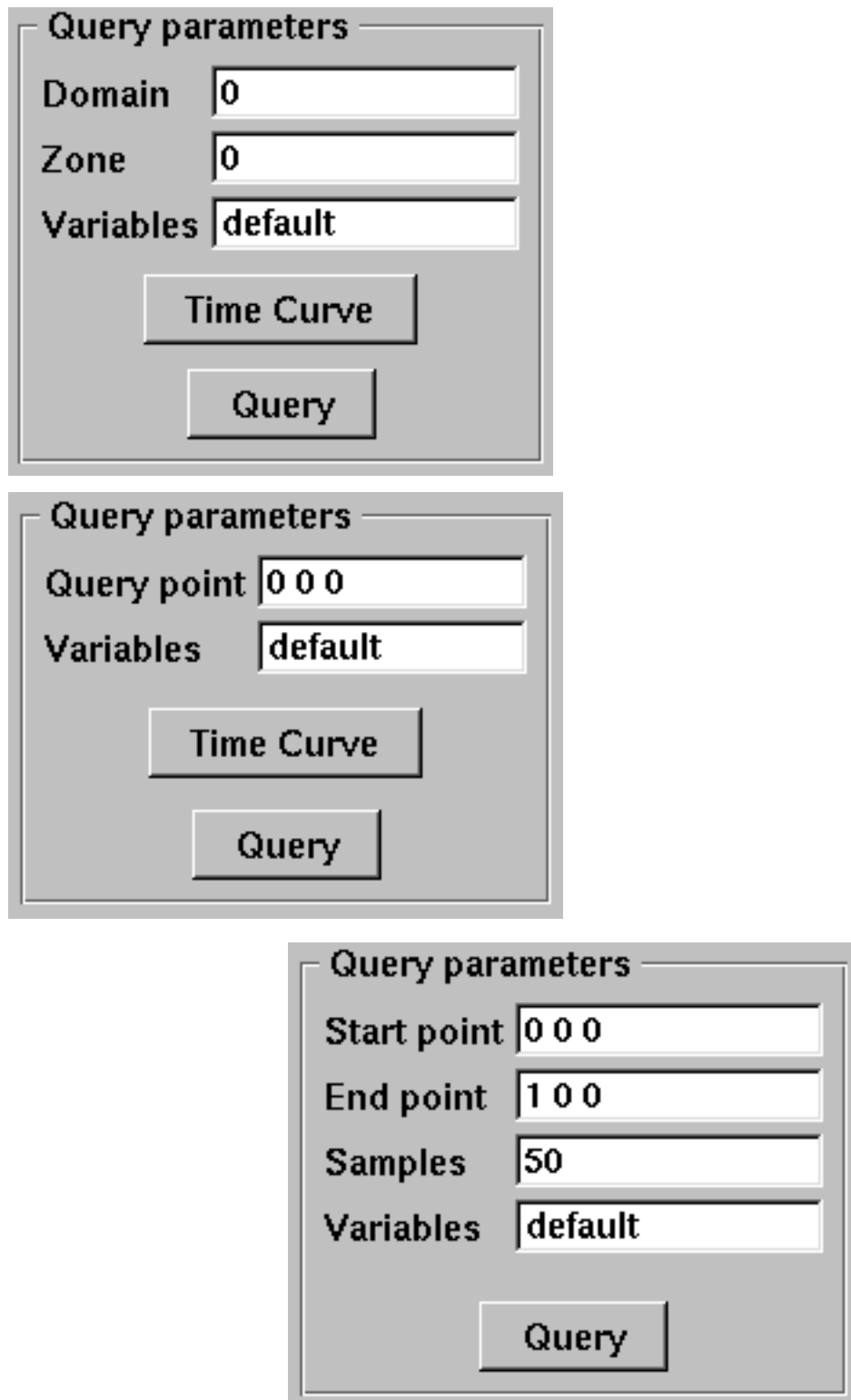
Fig. 1.202: Query parameters area

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.

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

The following table lists the query name, type, and description for the built-in queries that VisIt supports. The query type for database queries, point queries, and line queries are respectively marked as: DB, Point, and Line.

Query name	Query type	Description
2D area	DB	The 2D area query calculates the area of the 2D plot highlighted in the **Plot list** and pr
3D surface area	DB	The 3D surface area calculates the area of the plot highlighted in the **Plot list** and print
Area Between Curves	DB	The Area Between Curves query calculates the area between 2 curve plots. The plots t
Centroid	DB	This query will calculate the centroid of a dataset. The contribution of each cell is calc
Chord Length Distribution	DB	The Chord Length Distribution query calculates a probability density function of chor
Compactness	DB	The Compactness query calculates mesh metrics and prints them in the **Query results**
Cycle	DB	The Cycle query prints the cycle for the plot that is highlighted in the **Plot list** to the **Q**
Distance from Boundary		The Distance From Boundary query calculates how much mass is at a given distance a
Eulerian		The Eulerian query calculates the Eulerian number for the mesh that is used by the hig
Expected Value	DB	The Expected Value query calculates the integral of $xf(x)dx$ for some curve f(x). The
Grid Information	DB	The Grid Information query prints information for each domain in a multi-domain mes
Integrate	DB	The Integrate query calculates the area under the Curve plot that is highlighted in the F
Kurtosis	DB	The Kurtosis query calculates the kurtosis of a normalized distribution function. The n
L2Norm	DB	The L2Norm query calculates the L2Norm, or square of the integrated area, of a Curve
L2Norm Between Curves	DB	The L2Norm query takes two Curve plots as input and calculates the L2Norm between
Lineout	Line	The Lineout query creates a new instance of the highlighted plot in the **Plot list** , appli
Min	DB	The Min query calculates the minimum value for the variable used by the highlighted
Mass Distribution	DB	The Mass Distribution query calculates how much mass occurs at different length scal
Max	DB	The Max query calculates the maximum value for the variable used by the highlighted
MinMax	DB	The MinMax query calculates the minimum and maximum values for the variable use
Moment of inertia	DB	This query will calculate the moment of inertia tensor for each cell in a three-dimensio
NodeCoords	DB	The NodeCoords query prints the node coordinates for the specified node and prints th
NodePick	Point	The NodePick query performs node picking at the specified world coordinate which, i
NumNodes	DB	The NumNodes query prints the number of nodes for the mesh used by the highlighted
NumZones	DB	The NumZones query prints the number of zones for the mesh used by the highlighted
Pick	Point	The Pick query performs zone picking at the specified world coordinate which, if used
PickByNode	Point	The PickByNode query performs node pick using the highlighted plot in the **Plot list** a
PickByZone	Point	The PickByZone query performs zone pick using the highlighted plot in the Plot list an
Revolved surface area	DB	The Revolved surface area query revolves the mesh used by the highlighted plot in the
Revolved volume	DB	The Revolved volume area query revolves the mesh used by the highlighted plot in the
Skewness	DB	The Skewness query calculates the skewness of a normalized distribution function. Th
Spatial Extents	DB	The Spatial Extents query calculates the original or actual spatial extents for the plot th
Spherical compactness factor	DB	This query attempts to measure how spherical a three dimensional shape is. The query
Time	DB	The Time query prints the time for the plot that is highlighted in the Plot list to the **Qu**
Variable Sum	DB	The Variable Sum query adds up the variable values for all cells using the plot highligh
Volume	DB	The Volume query calculates the volume of the mesh used by the plot highlighted in th
Watertight	DB	The Watertight query determines if a three-dimensional surface mesh, of the plot highl

Query name	Query type	Description
Weighted Variable Sum	DB	The Weighted Variable Sum query adds up the variable values, weighted by cell size, f
ZoneCenter	DB	The ZoneCenter query calculates the zone center for a certain cell in the database used

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 1.203). 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**.
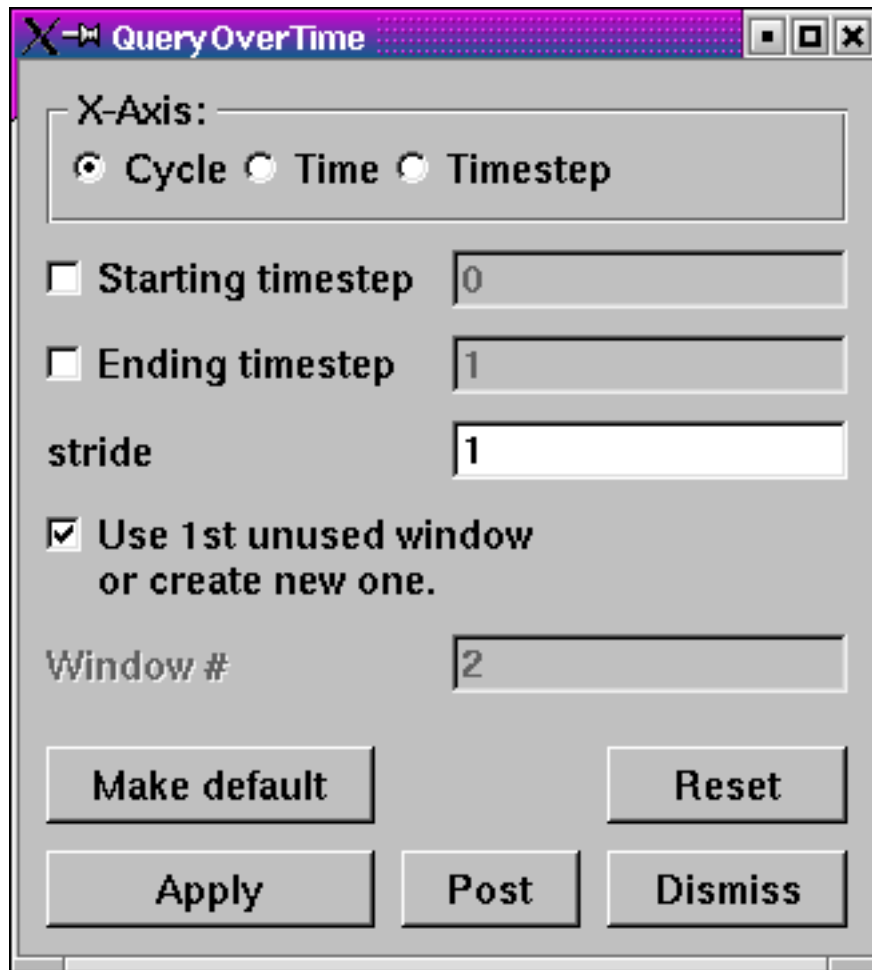
Fig. 1.203: Query Over Time Window

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.

1.8.3 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 1.204 and an example of node pick and zone pick markers is shown in Figure 1.205.

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 1.206) 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.
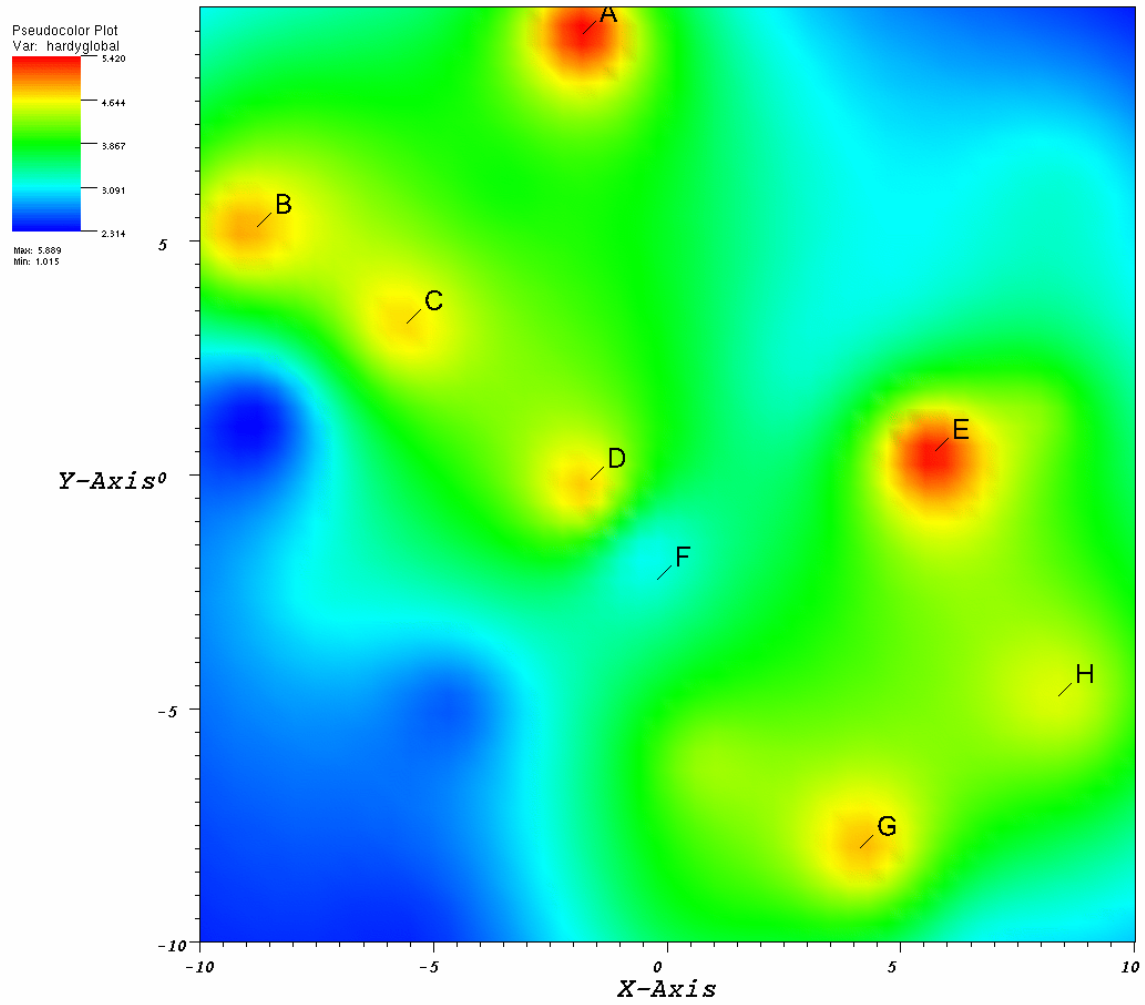
Fig. 1.204: Visualization window with pick points
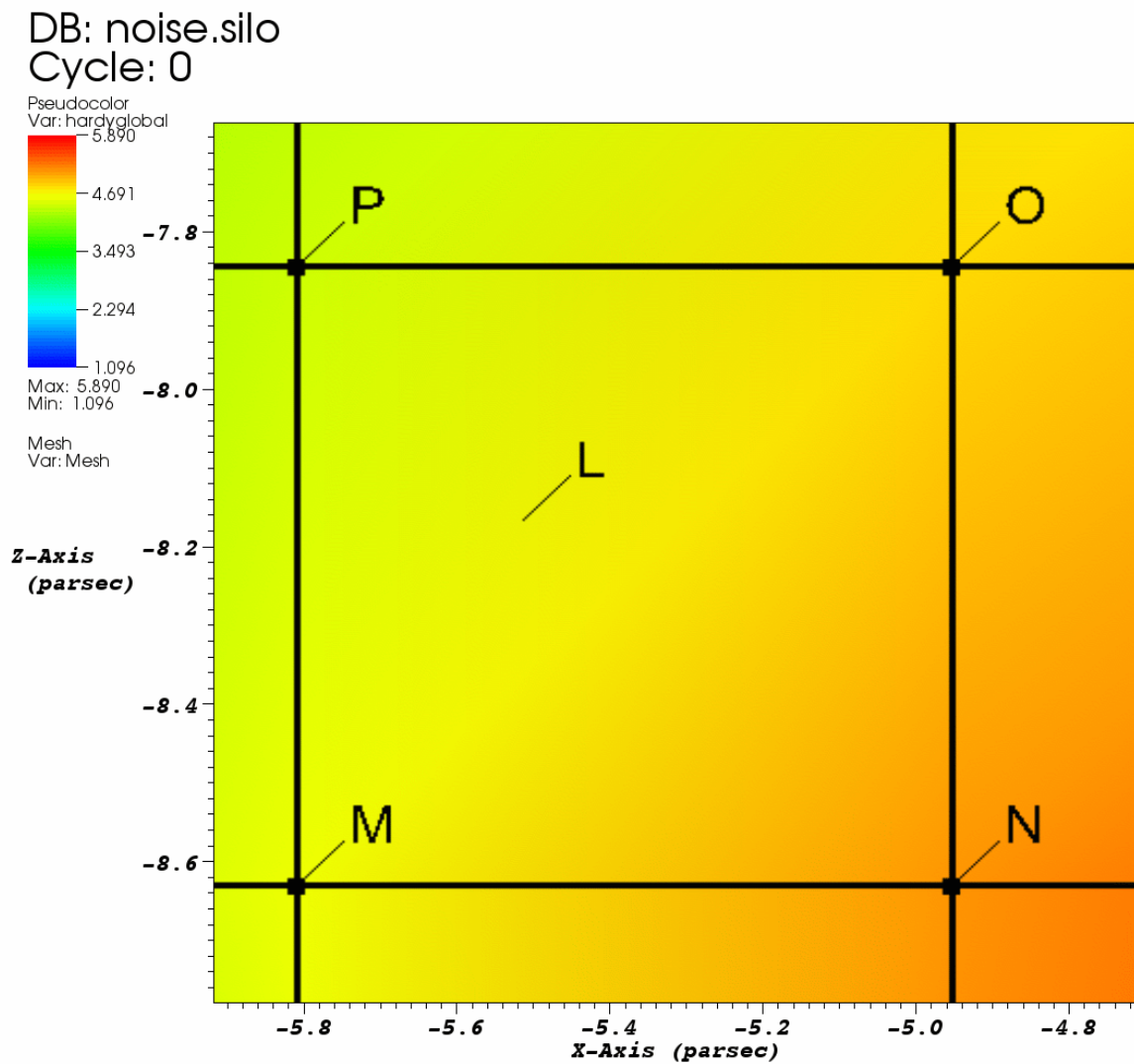
Fig. 1.205: Zone pick marker **L** and node pick markers **M**, **N**, **O**, **P**
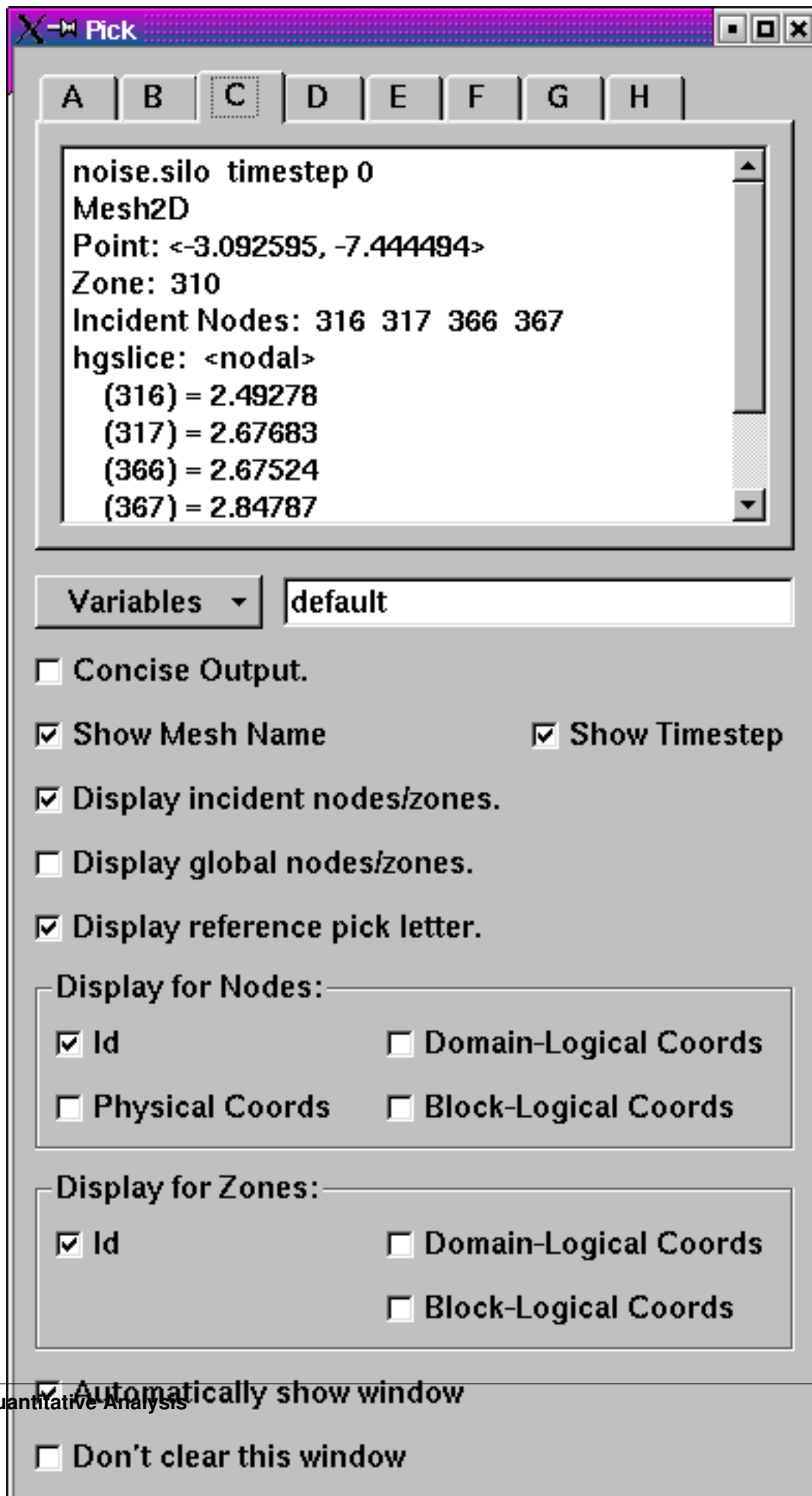
Setting the pick variable

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.

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.

1.8.4 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 1.207) 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 1.208.

Fig. 1.207: Lineout mode toolbar icon
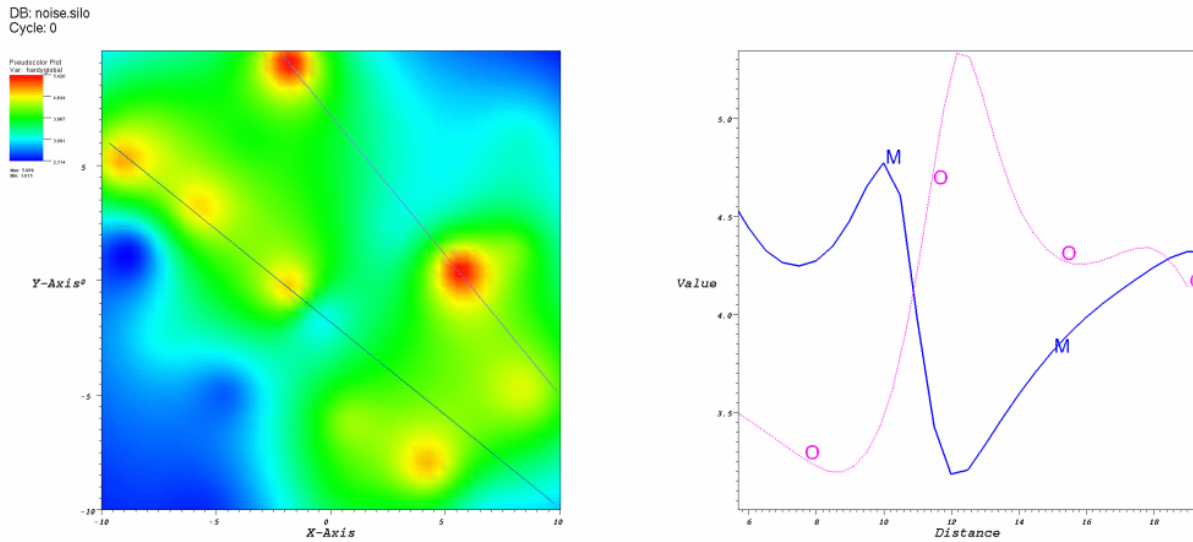
Fig. 1.208: Visualization windows with reference line and Curve plots

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.

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 1.209). 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.
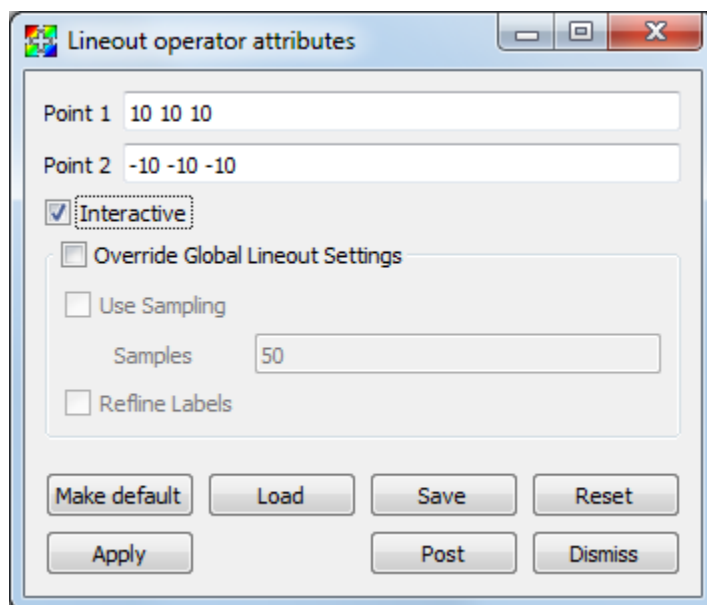
Fig. 1.209: Lineout attributes window

Setting the number of lineout samples

The Lineout operator works by extracting sample points along a line. The sample points are then used to create Curve plots. The Lineout operator's default sampling scheme is to sample data values at the intersections between the sampling line and the cell boundaries encountered along the way. This method gives rise to jagged Curve plots favored by many VisIt users. See Figure 1.210 for a comparison between the sampling methods. If you instead want to smoothly sample the cells along the sampling line with some number of evenly spaced sample points, you can make the Lineout operator use evenly spaced sampling by clicking on the **Override Global Lineout Settings** check box in the **Lineout attributes** window. Then click on the **Use Sampling** check box and enter a number of sample points. The number of sample points taken along the line determine the fidelity of the Curve plot. Generally, it is best to set the number of sample points such that each cell is sampled at least once. To set the number of sample points, type a new number into the **Samples** text field.

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*
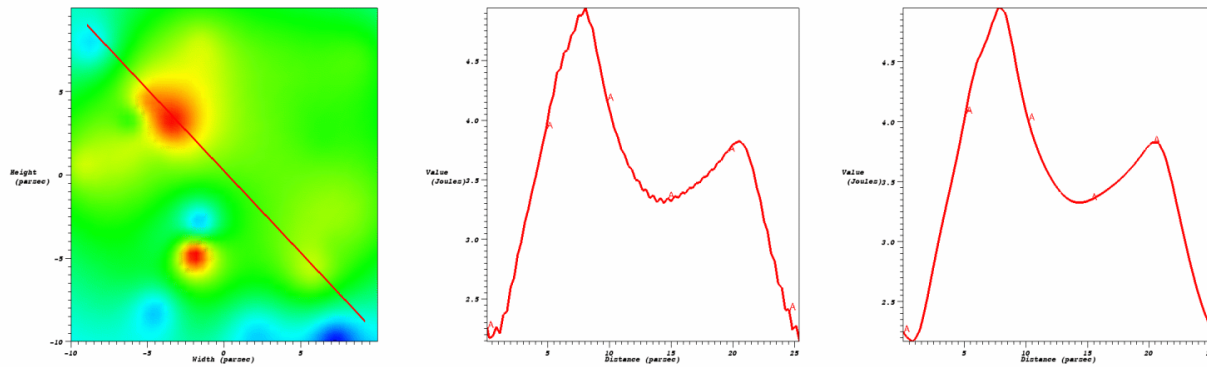
Fig. 1.210: Lineout via cell intersection and lineout via sampling

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.

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 1.212.

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.
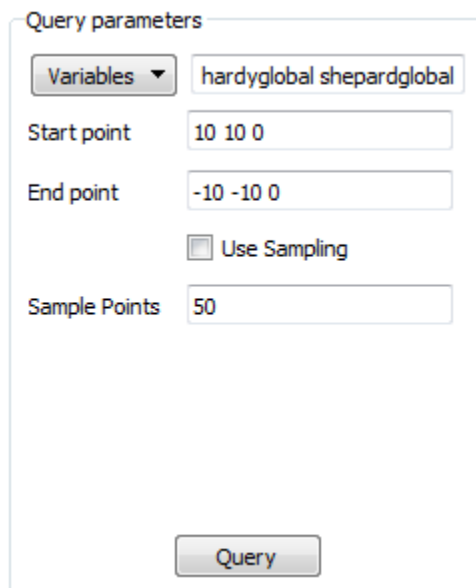
Fig. 1.211: Lineout query's parameters window
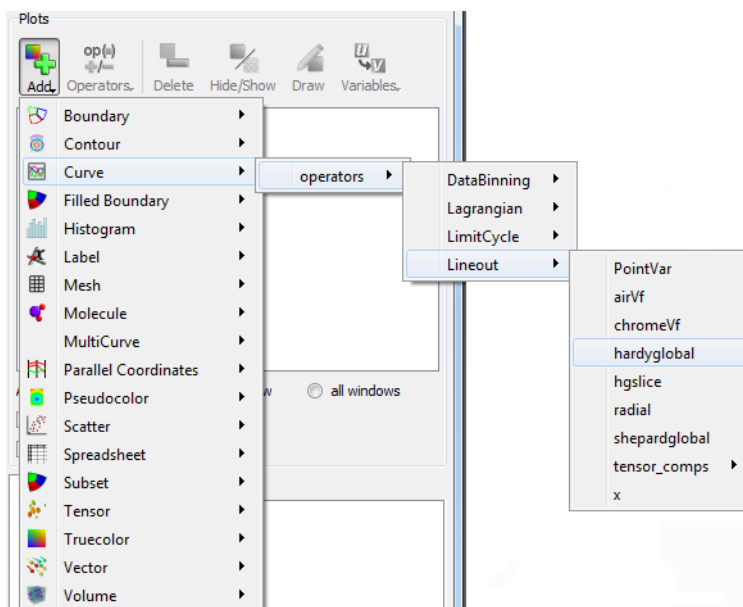
Fig. 1.212: Choosing lineout from the Curve plot's variable menu
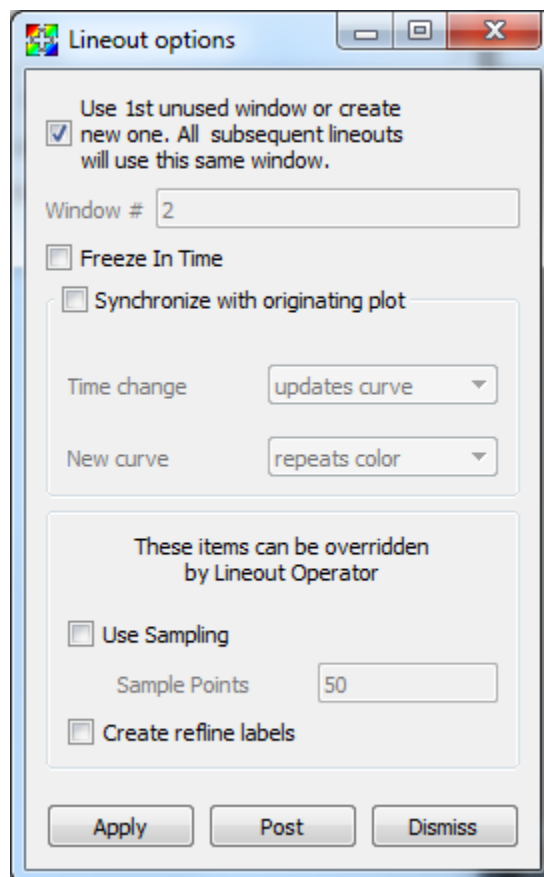
Fig. 1.213: Lineout Options Window

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 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 1.213).

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 1.214). 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.

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.

1.8.5 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
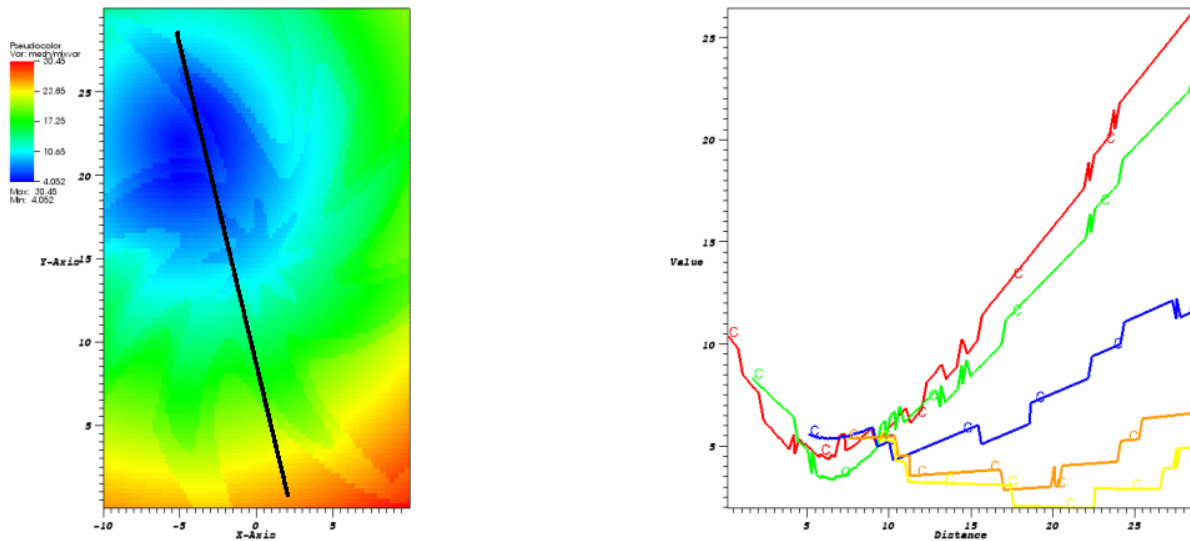
Fig. 1.214: Dynamic lineout can be used to create curves for multiple time states

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 1.215.

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 1.218

Once the database of the target mesh is specified, the target mesh within that database is specified with the **Target Mesh:** pull down list.
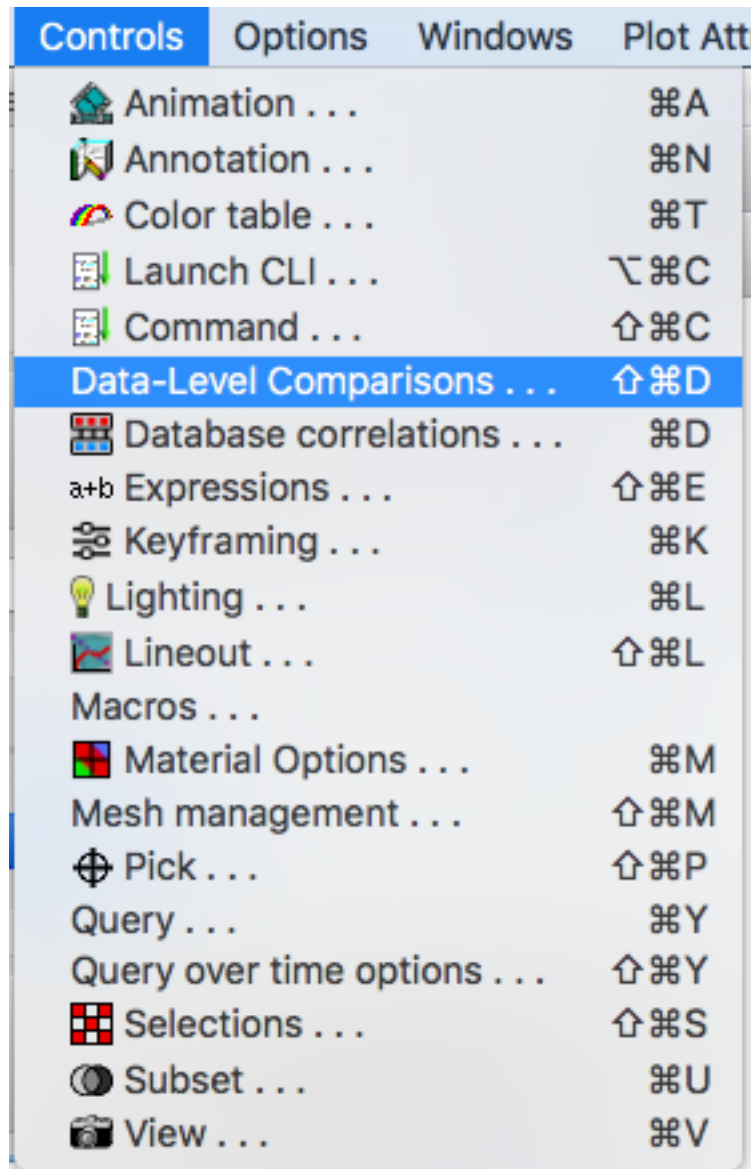
Fig. 1.215: Starting the Data-Level Comparisons Wizard
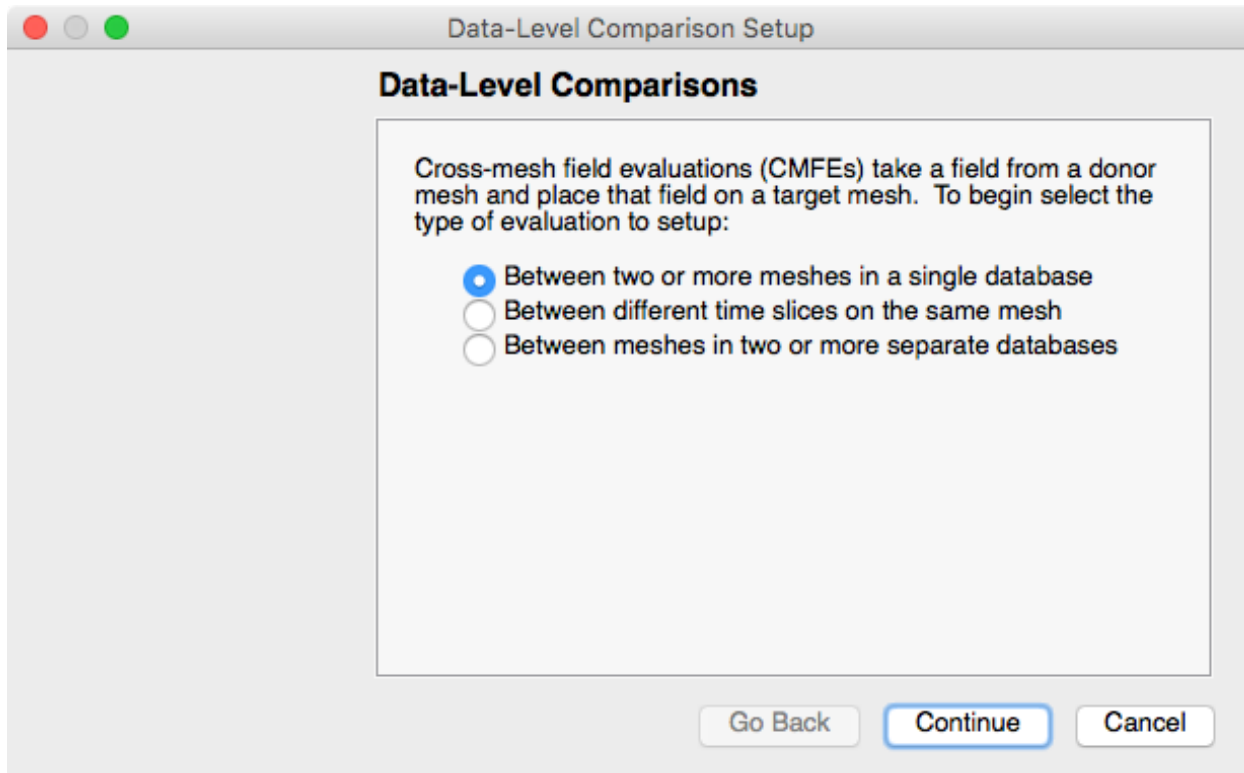
Fig. 1.216: Selecting among varieties of CMFE expressions
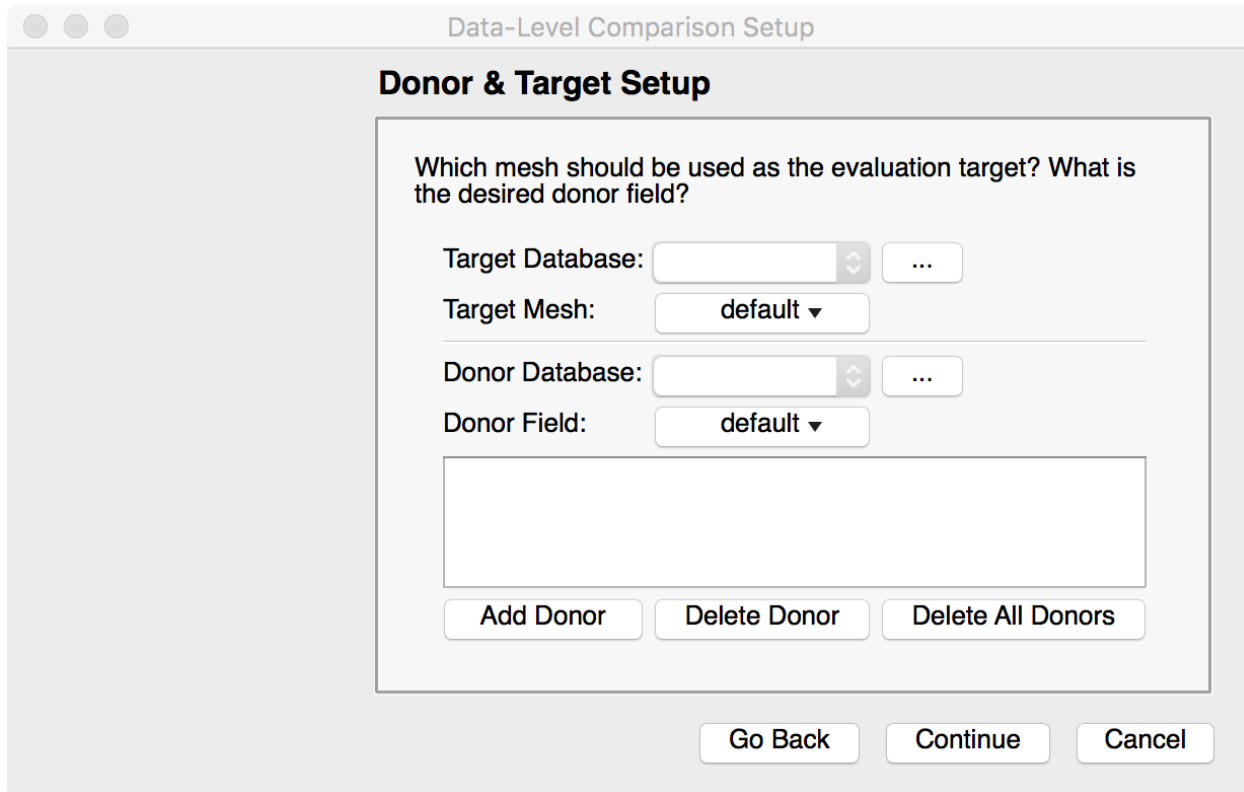
Fig. 1.217: Setting up the target mesh and donor variables
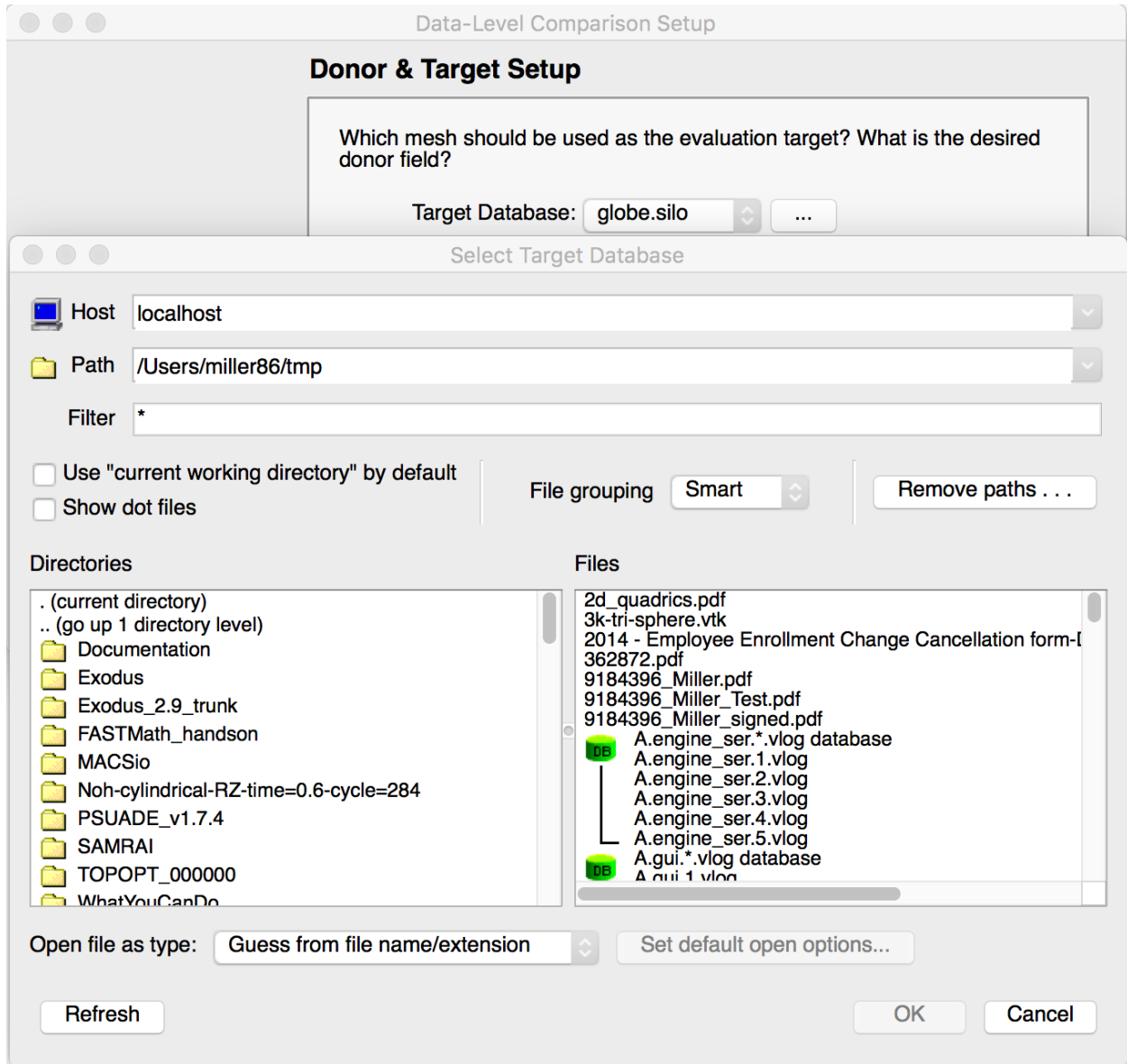
Fig. 1.218: Setting up the target mesh and donor variables

A similar sequence of steps is followed for specifying the donor variable. The example in Figure 1.219 demonstrates the selection of a specific donor variable from the donor database with the **Donor Variable:** pull down list.
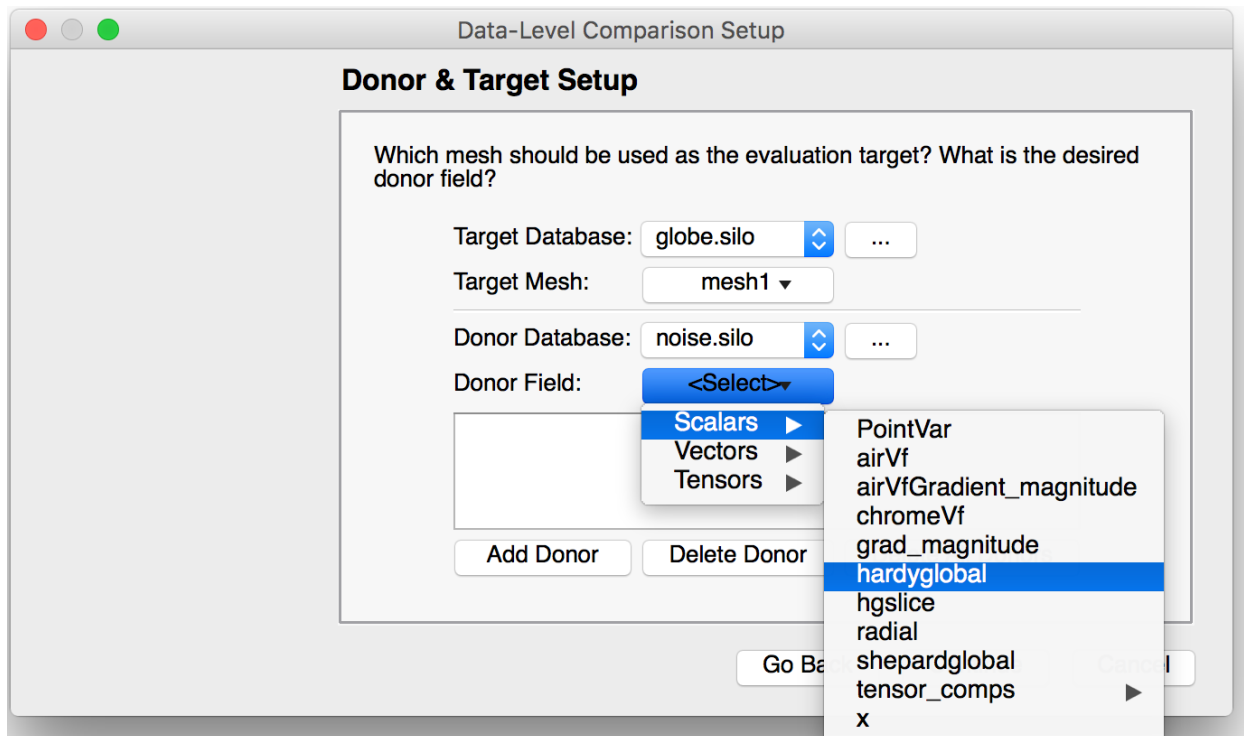
Fig. 1.219: Selecting a specific variable from a database

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 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 1.221, we have given the result variable the name *hardyglobal_onto_mesh1_from_globe*.

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
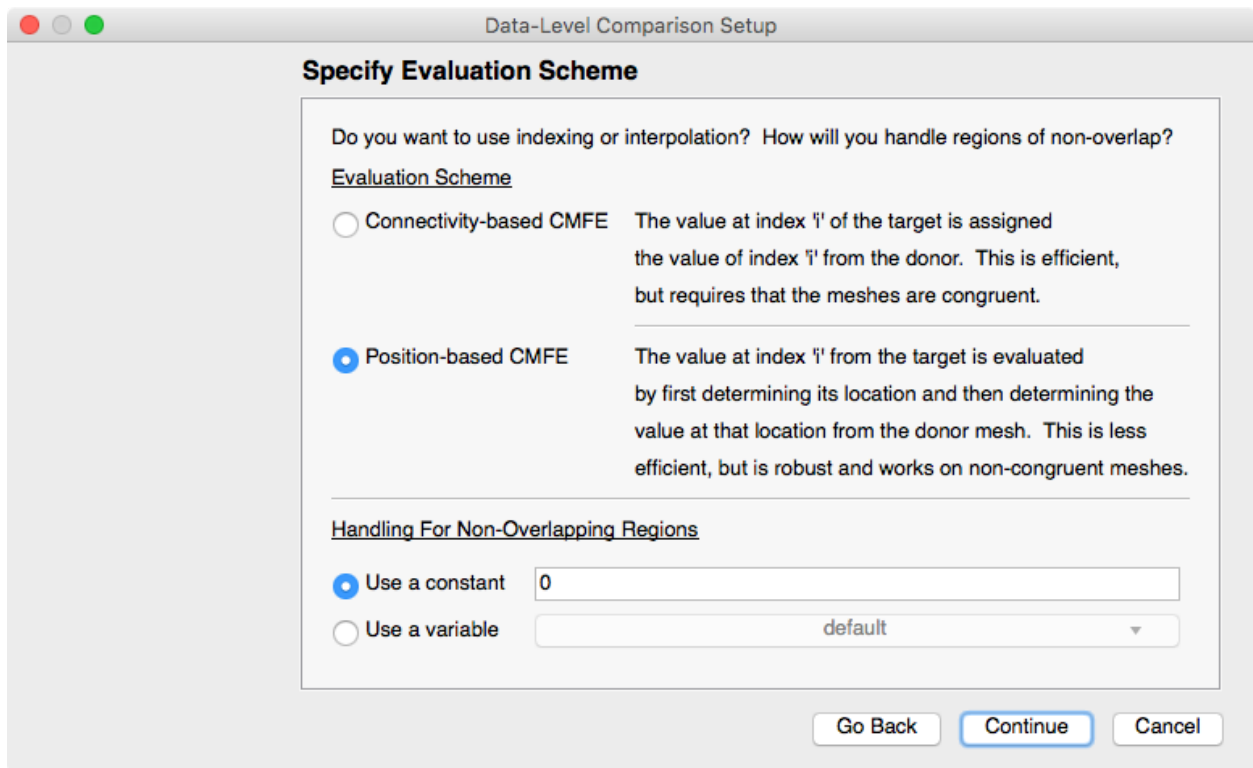
Fig. 1.220: Selecting the mode of evaluation
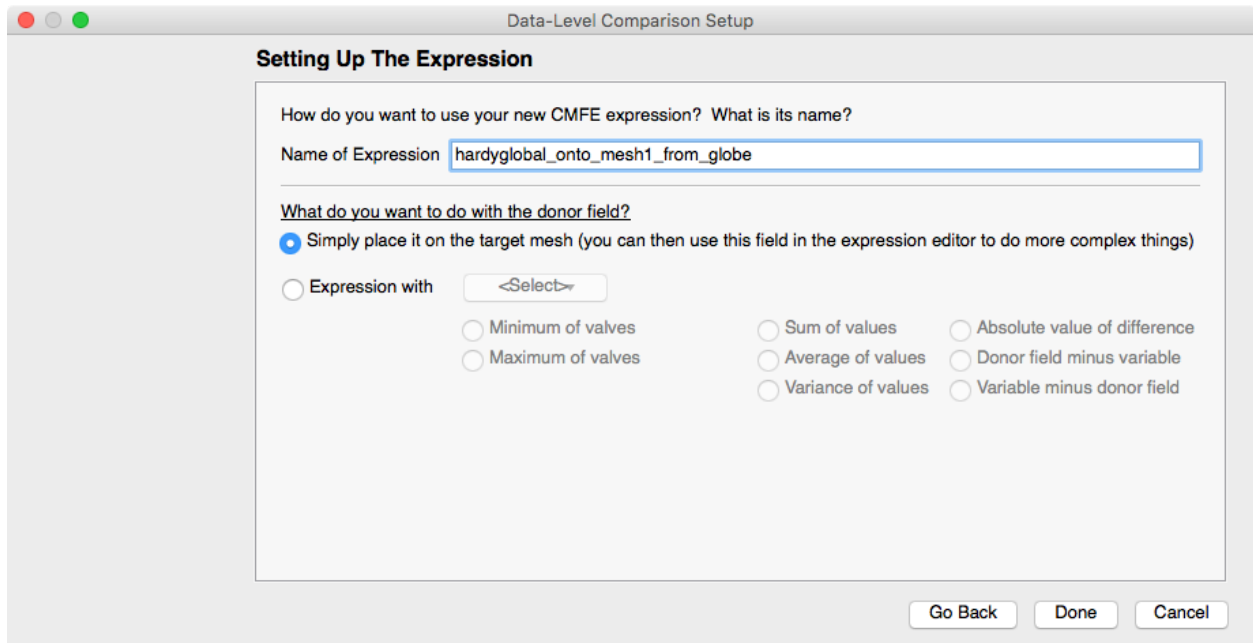
Fig. 1.221: Selecting result variable name and comparison method

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 1.222
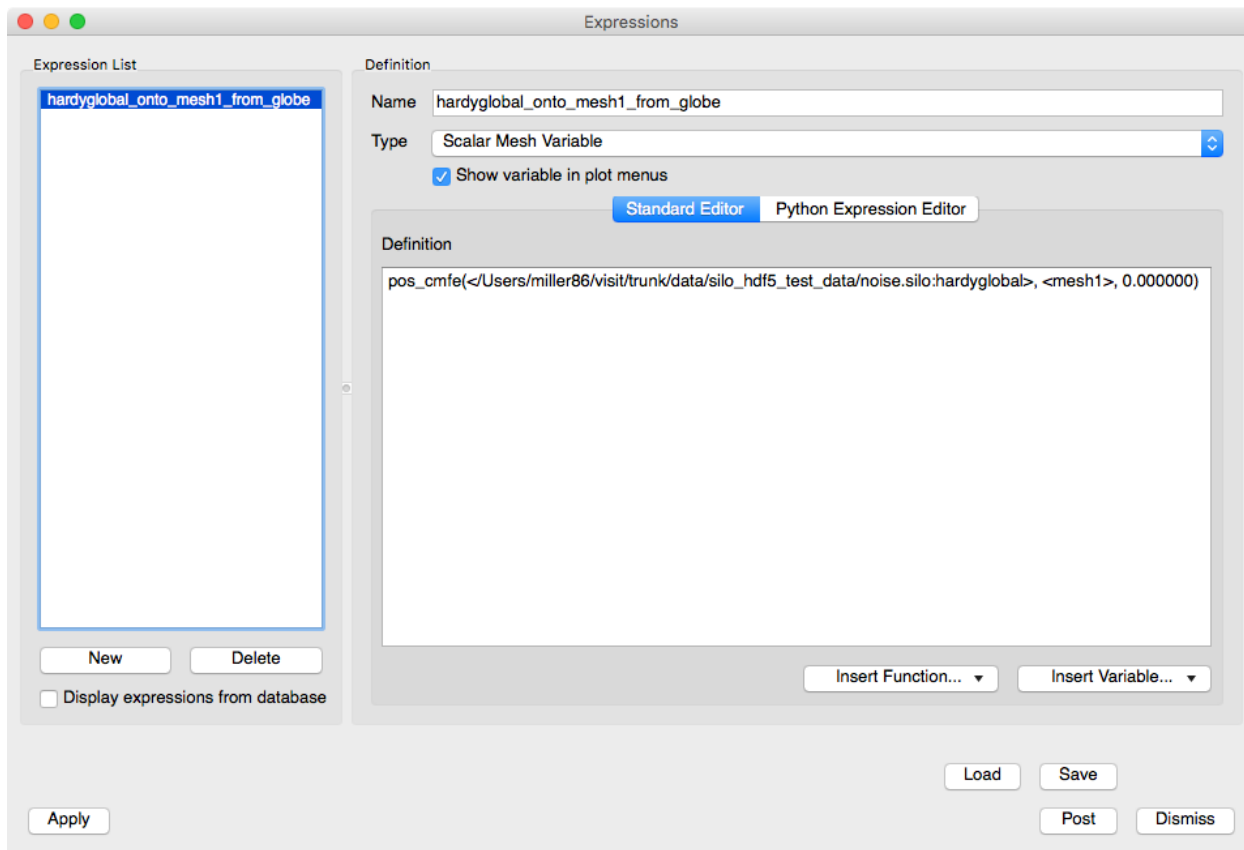
Fig. 1.222: New can be manipulated in the Expression window

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.

1.9 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.

1.9.1 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 1.223) 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 1.224 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.

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.
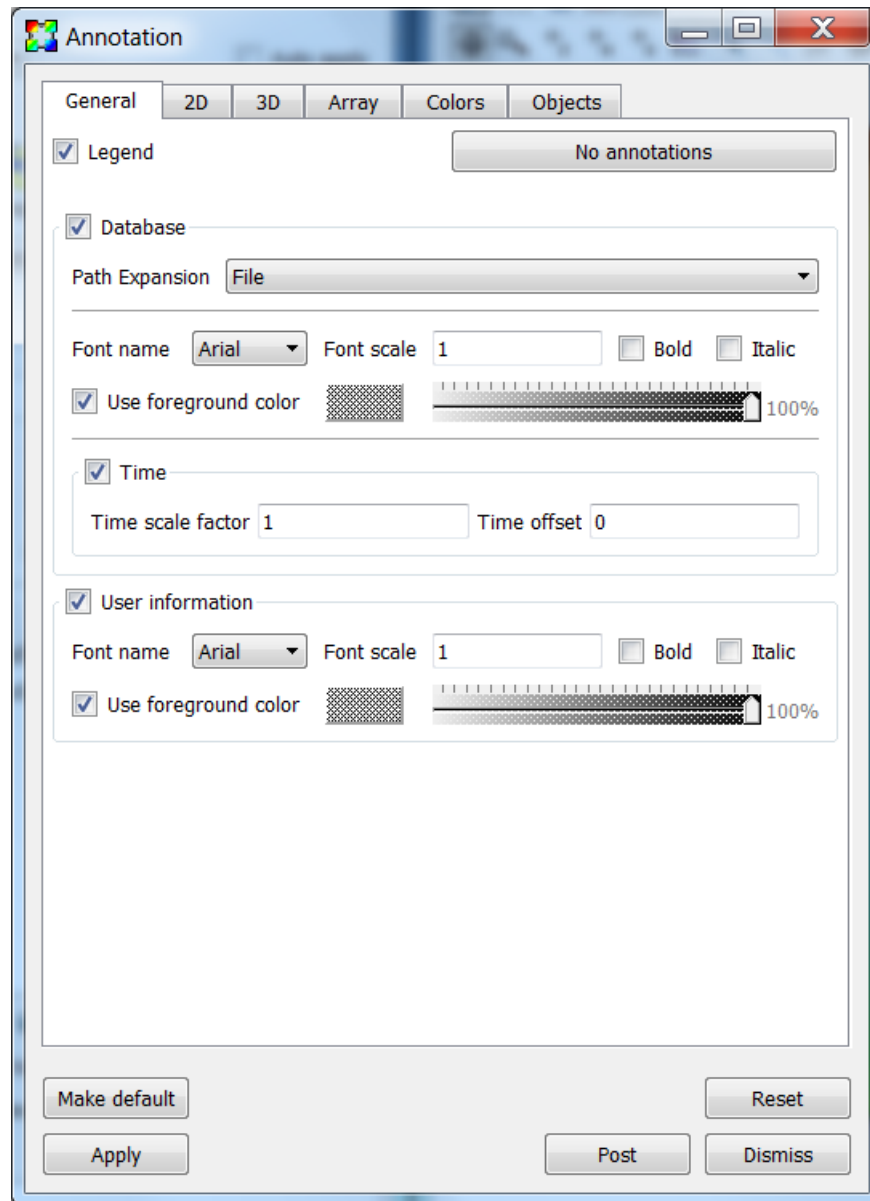
Fig. 1.223: The Annotation window
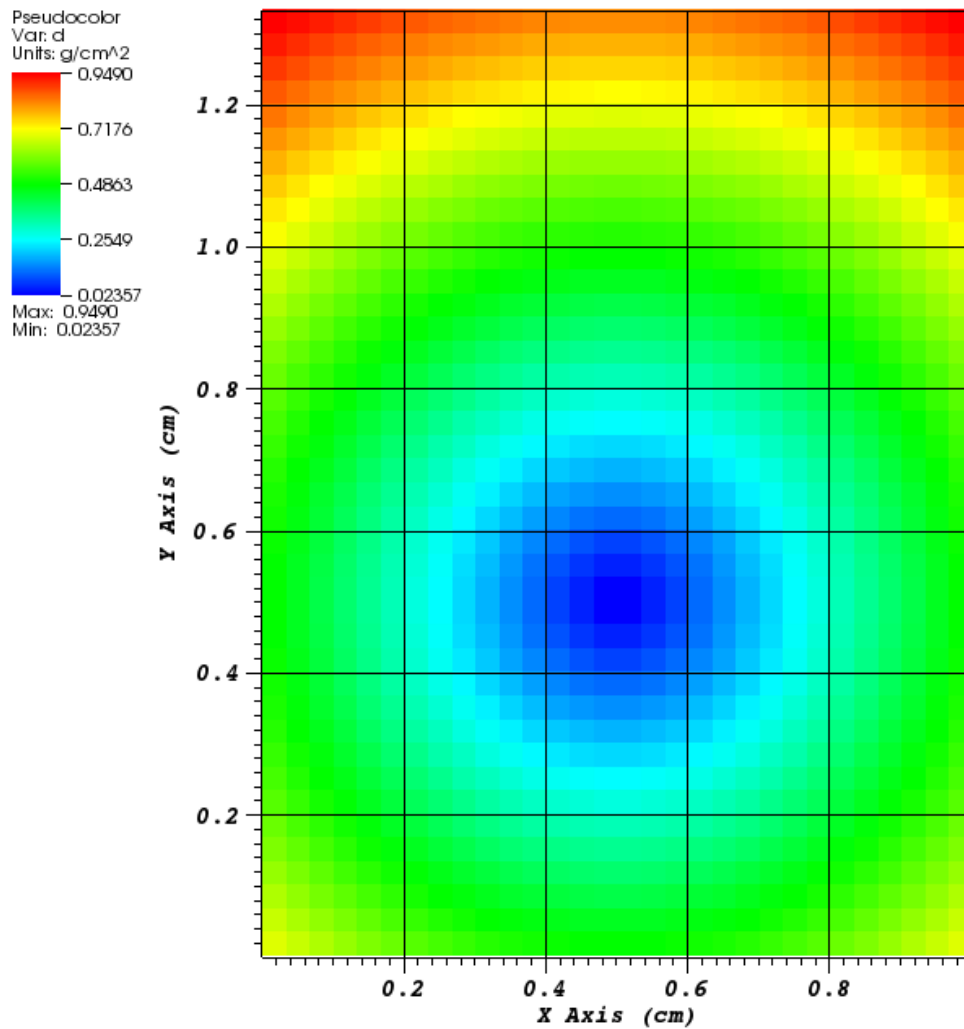
Fig. 1.224: 2D plot with annotations

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 1.225). The 2D annotation settings are primarily concerned with the appearance of the 2D axes that frame plots of 2D databases. Figure 1.224 shows a plot with various annotations.
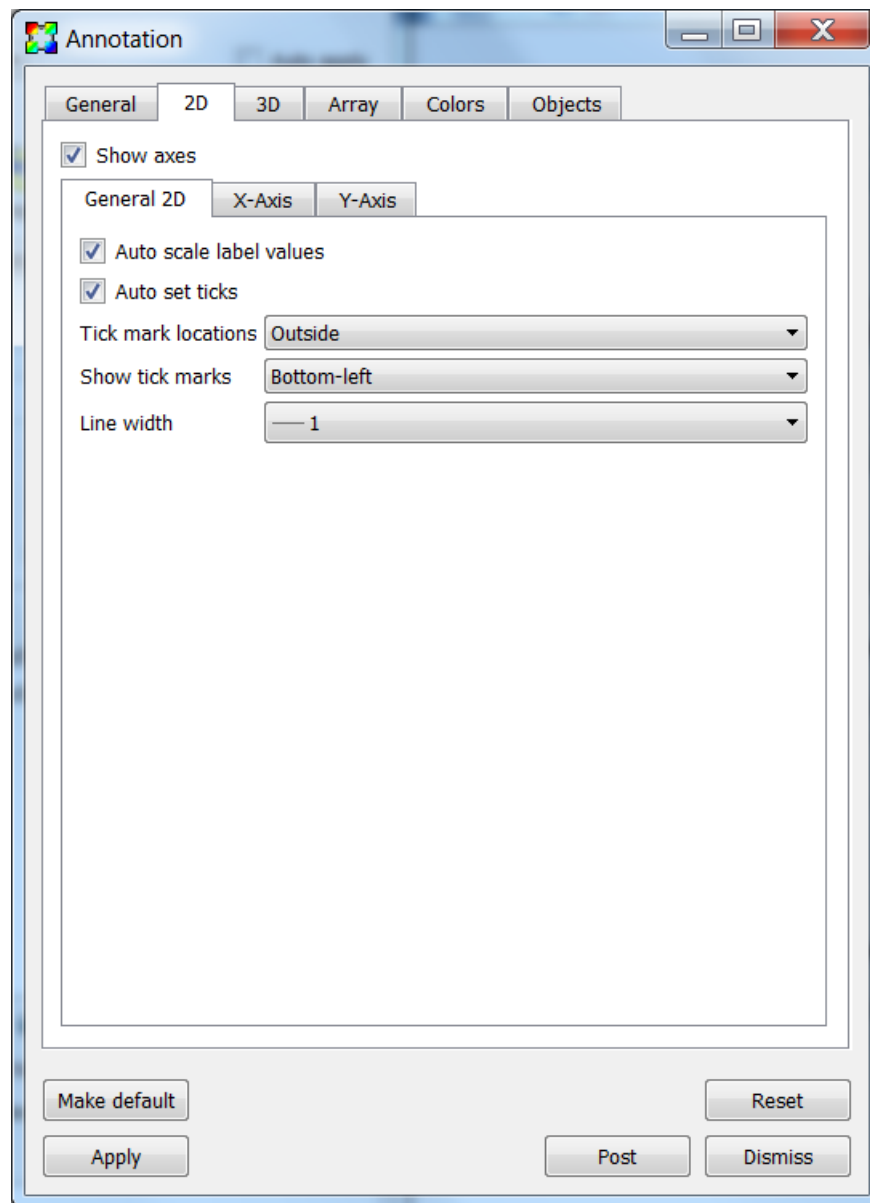
Fig. 1.225: The general 2D properties

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 1.226. 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.
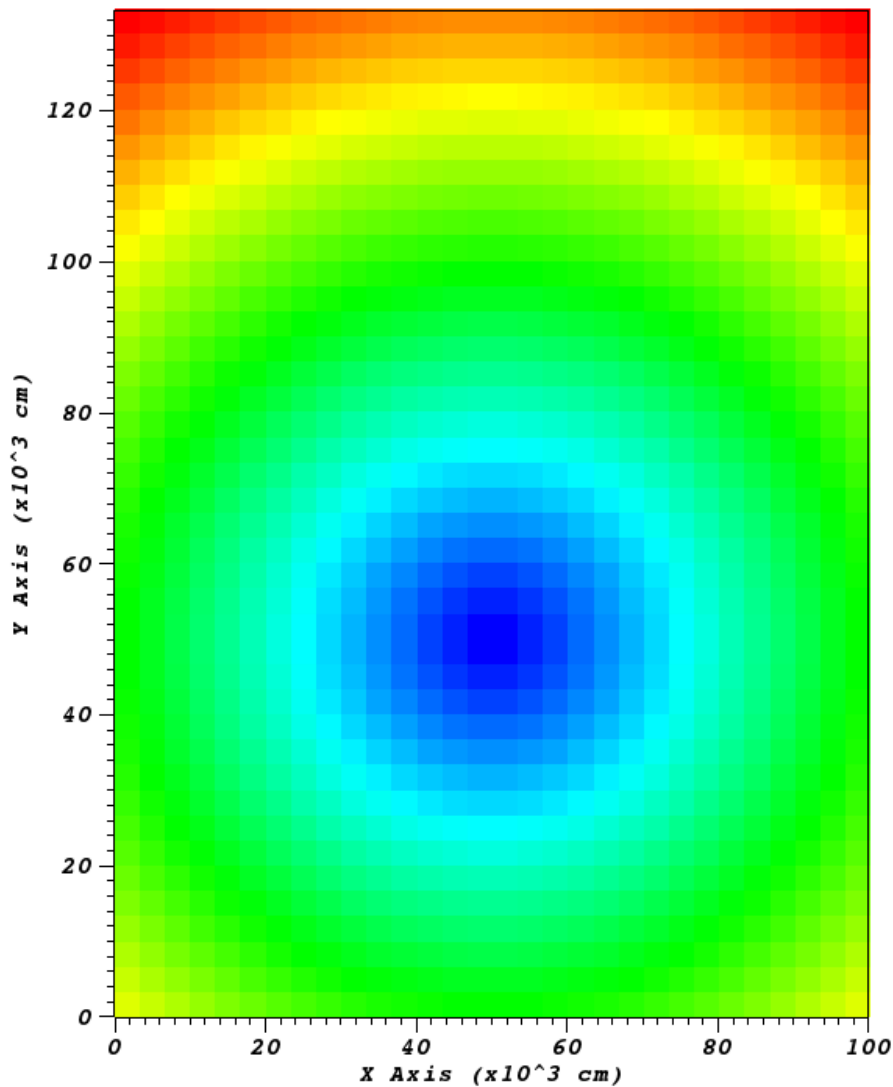
Fig. 1.226: 2D plot with axes labels being scaled by 10^3

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 1.227.
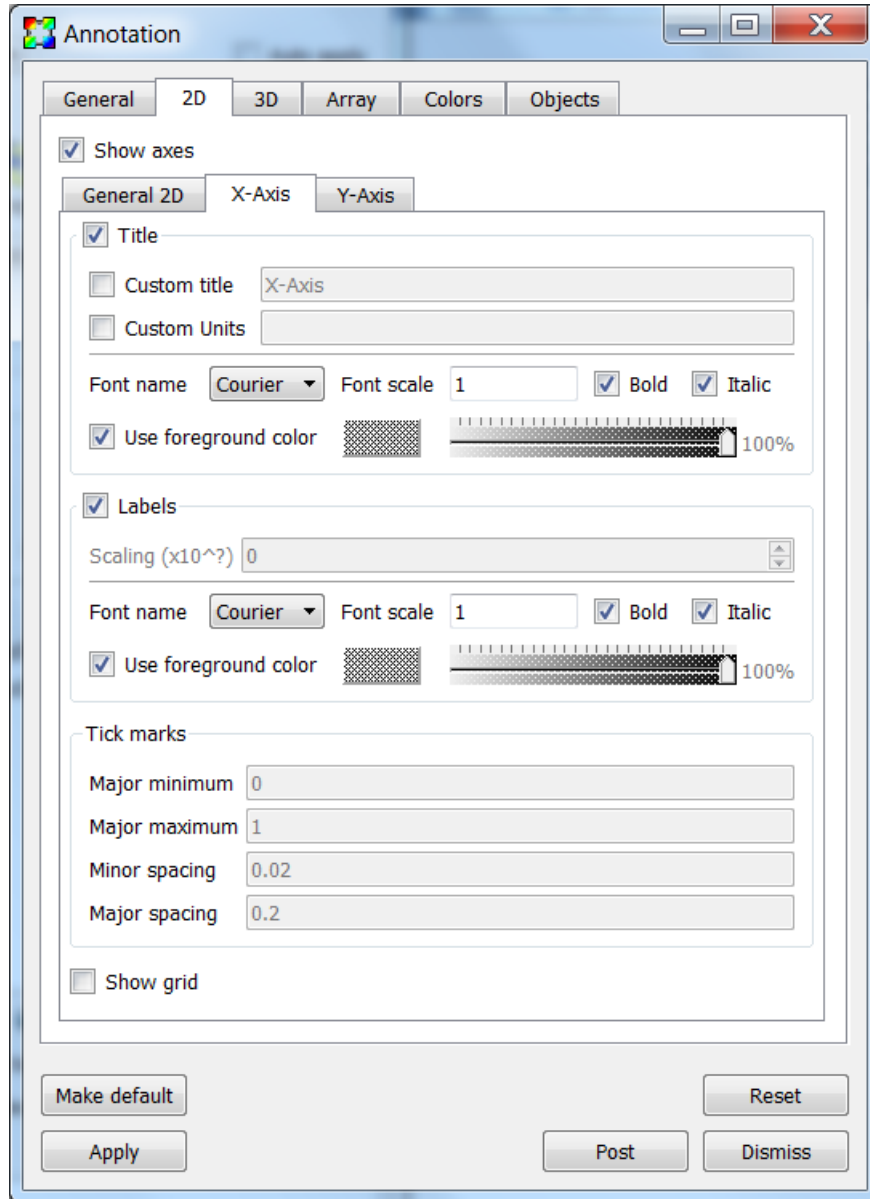
Fig. 1.227: The 2D axes properties

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 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 1.228 shows an example 3D plot with the 3D annotations. Figure 1.229 and Figure 1.230 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 1.226. 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 1.230.
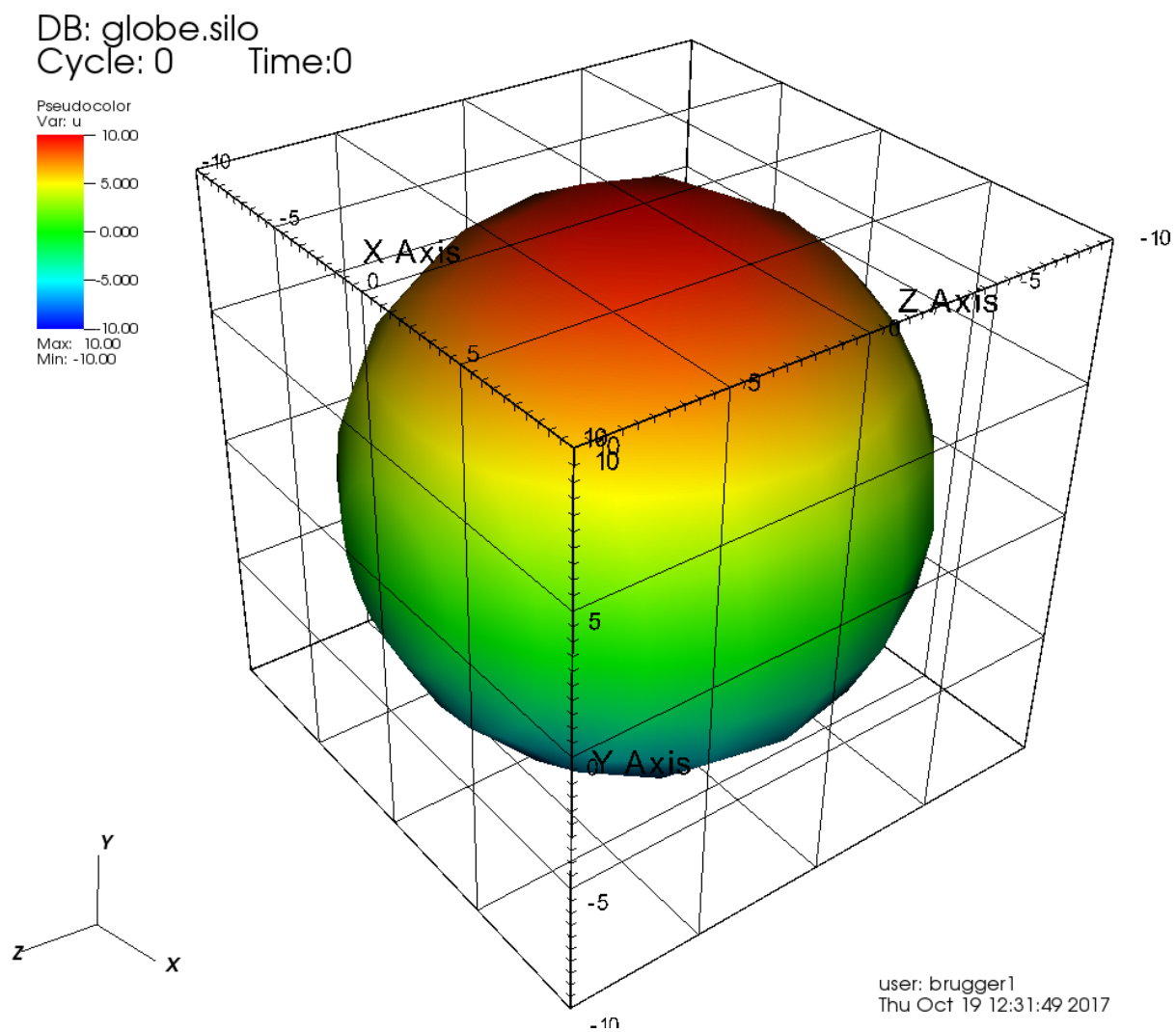
Fig. 1.228: 3D plot with annotations
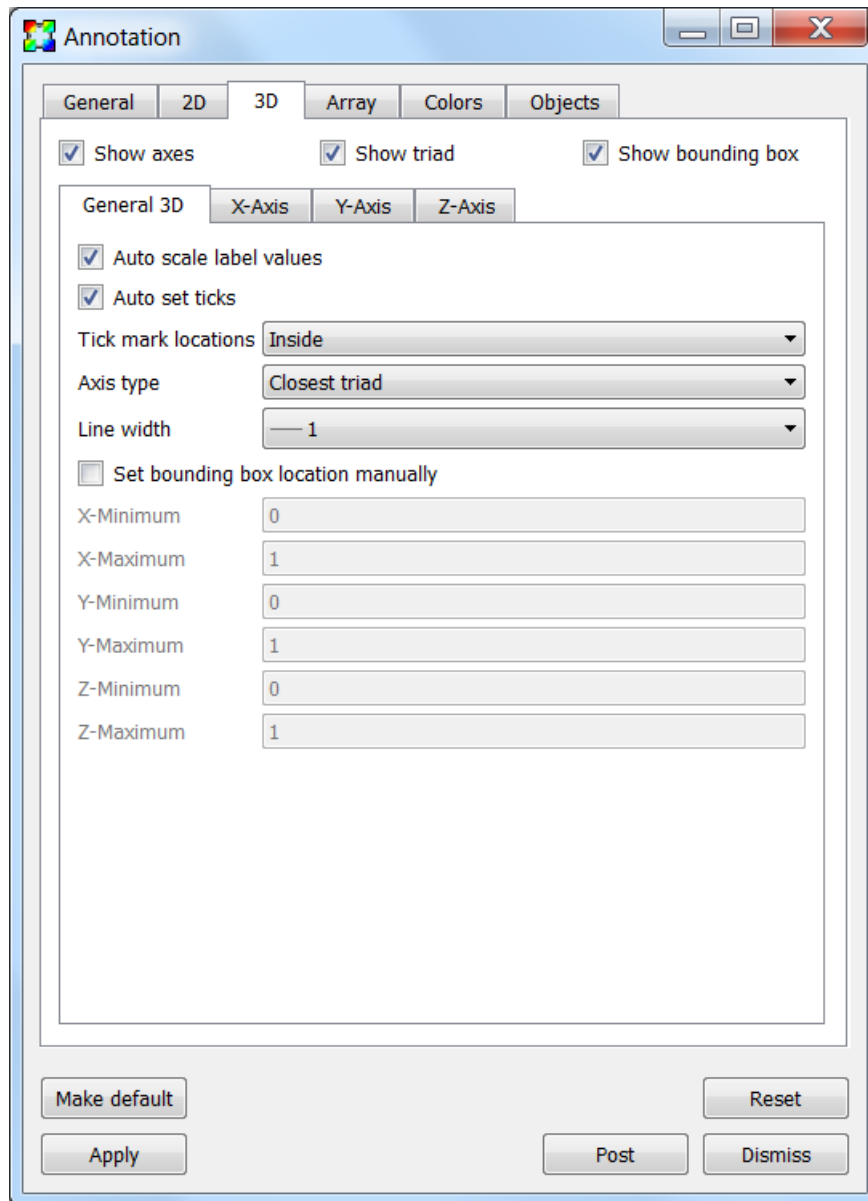
Fig. 1.229: The general 3D properties
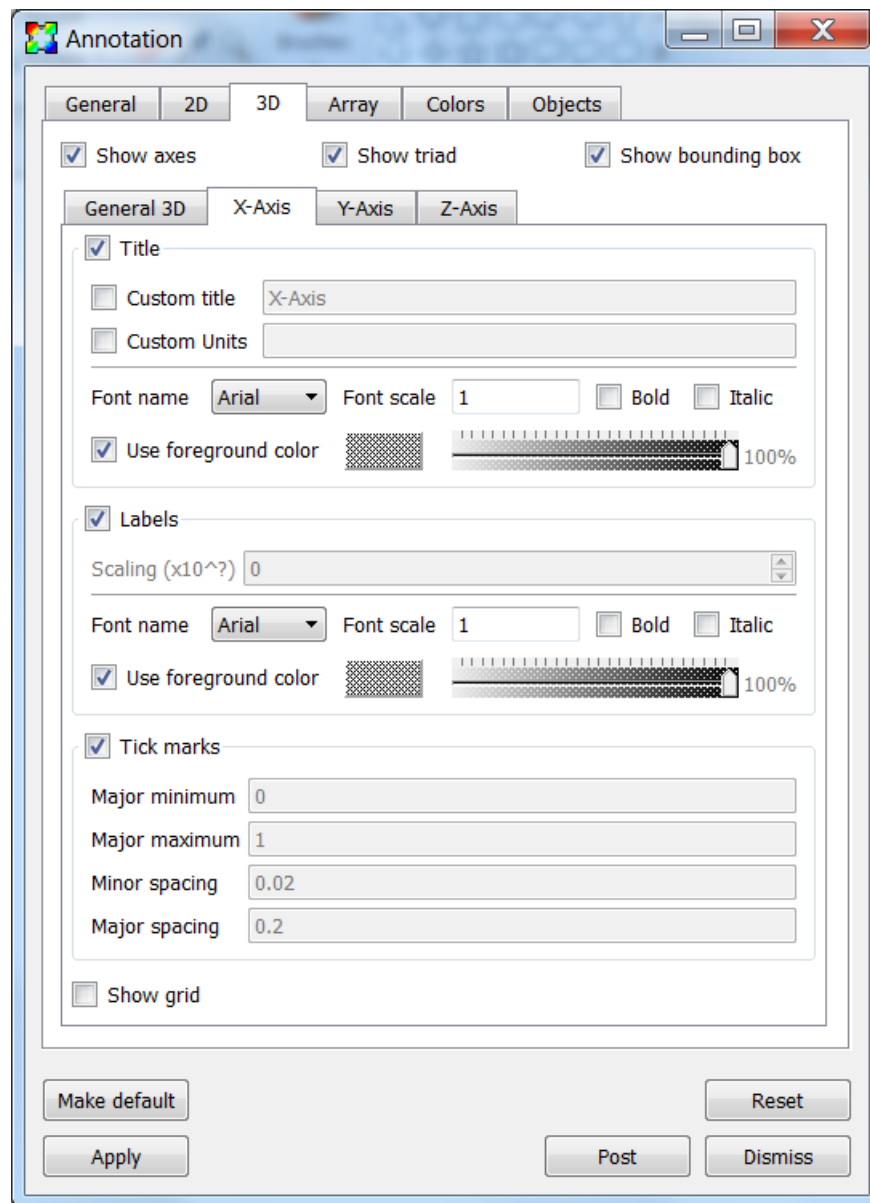
Fig. 1.230: The 3D axes properties

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 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 1.231, 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 1.232). 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 1.233.
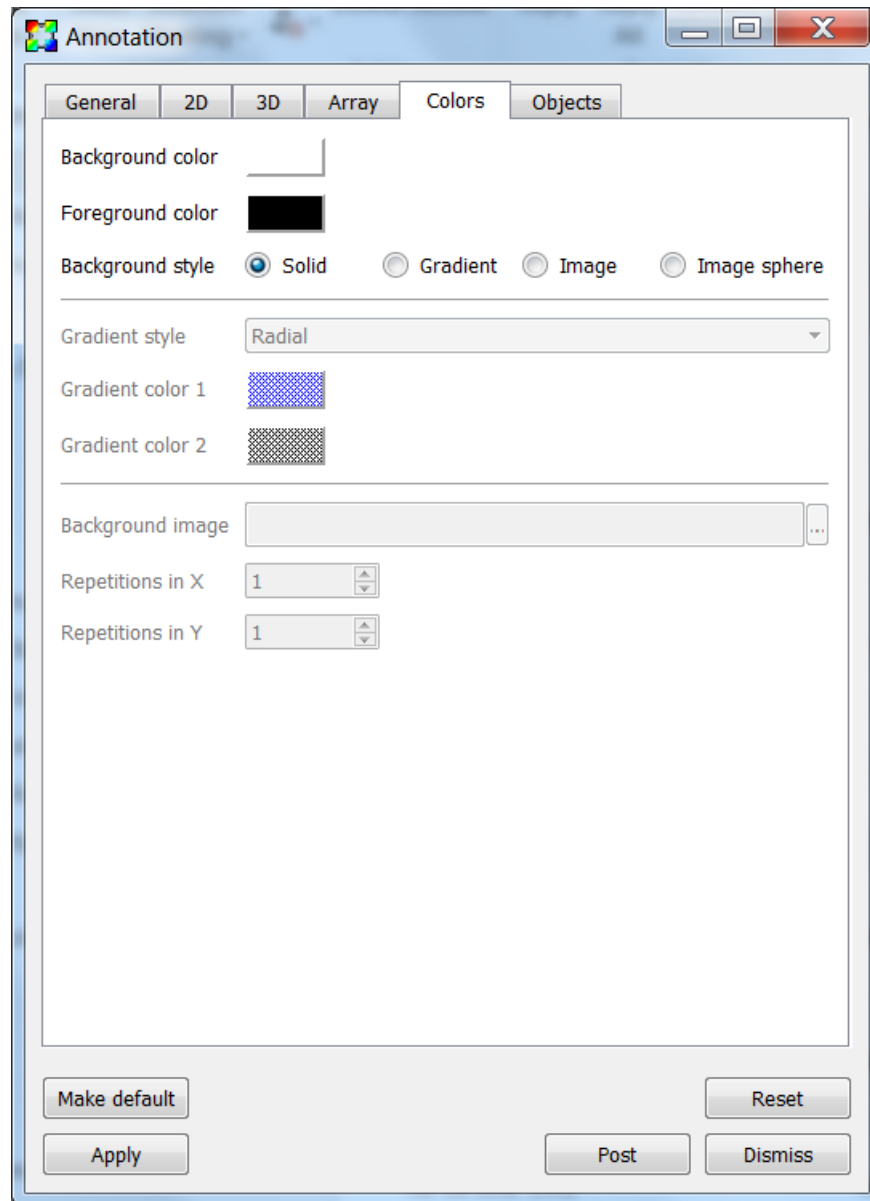
Fig. 1.231: The annotation colors tab
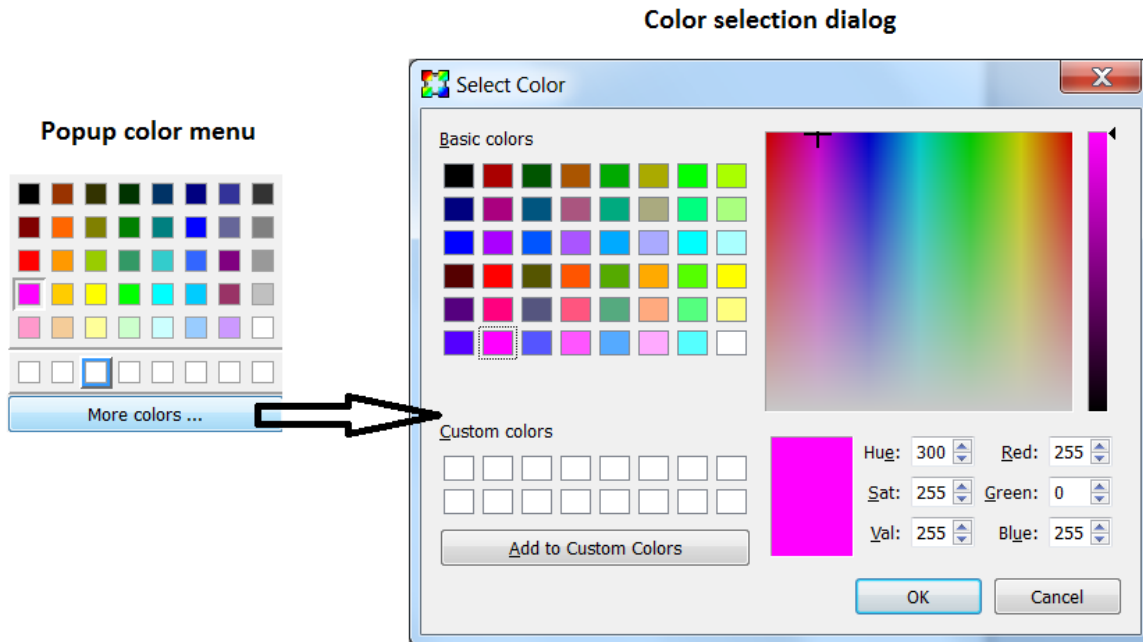
Fig. 1.232: The popup color menu and the color selection dialog

Fig. 1.233: The various gradient styles

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.

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 annotation objects: 2D text objects, 3D text objects, time slider objects, 2D line objects, 3D line objects and image objects. All of those types of annotation objects will be described herein. The **Objects** tab, in the **Annotation Window** (Figure 1.234) is devoted to managing the list of annotation objects and setting their properties.
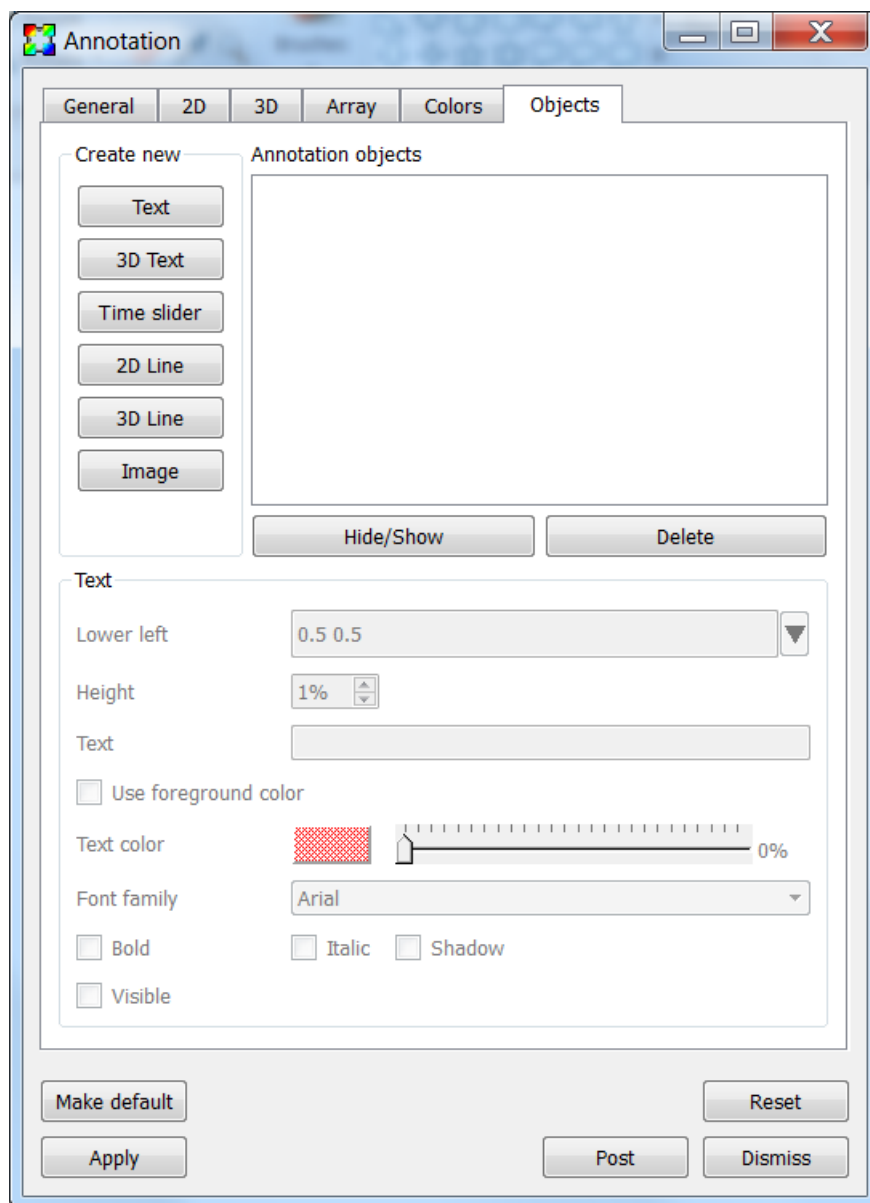
Fig. 1.234: The annotation objects tab

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 1.235, 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 1.236, 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

Fig. 1.235: Examples of text annotations
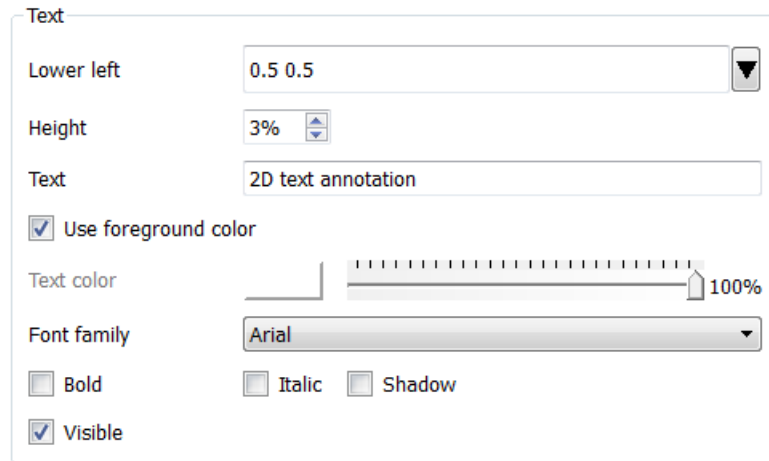
Fig. 1.236: The text annotation interface

positioning control, shown in Figure 1.237, 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.
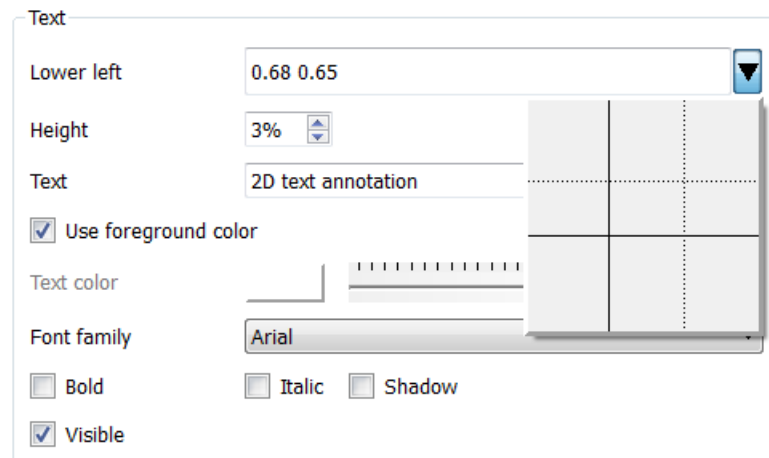
Fig. 1.237: Screen positioning control

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 but you can also use the $time wildcard string to make the text annotation object display the time for the current time state of the active database. A text string of the form: Time=$time will display Time=10 in the visualization window when the active database's time is 10. 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.

3D text annotation objects

3D text annotation objects, shown in Figure 1.238, 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.
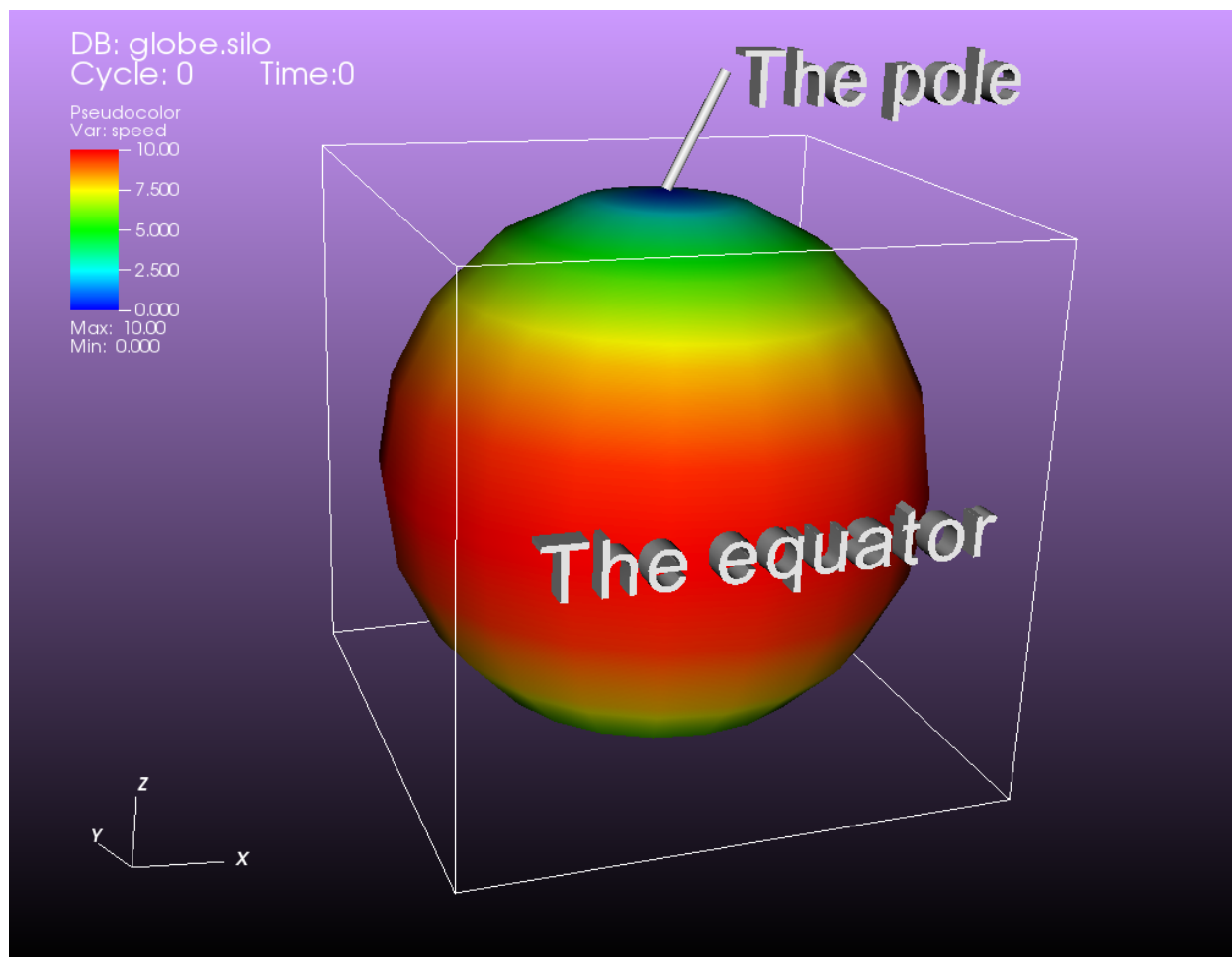
Fig. 1.238: Examples of 3d text annotations

The 3D text annotation object properties, shown in Figure 1.239, 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 the orientation relative to the screen, select the **Preserve orientation when view changes** radio button. To make the
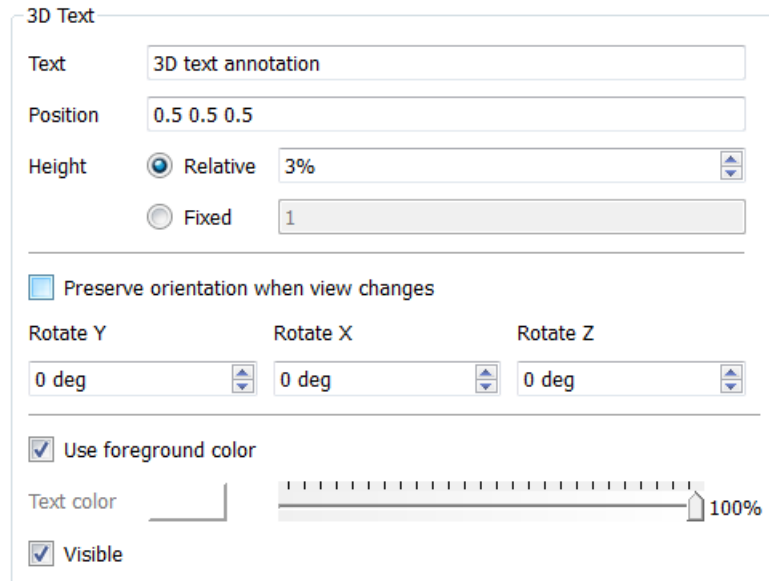
Fig. 1.239: The 3D text annotation interface

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 1.240, 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.

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.

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.
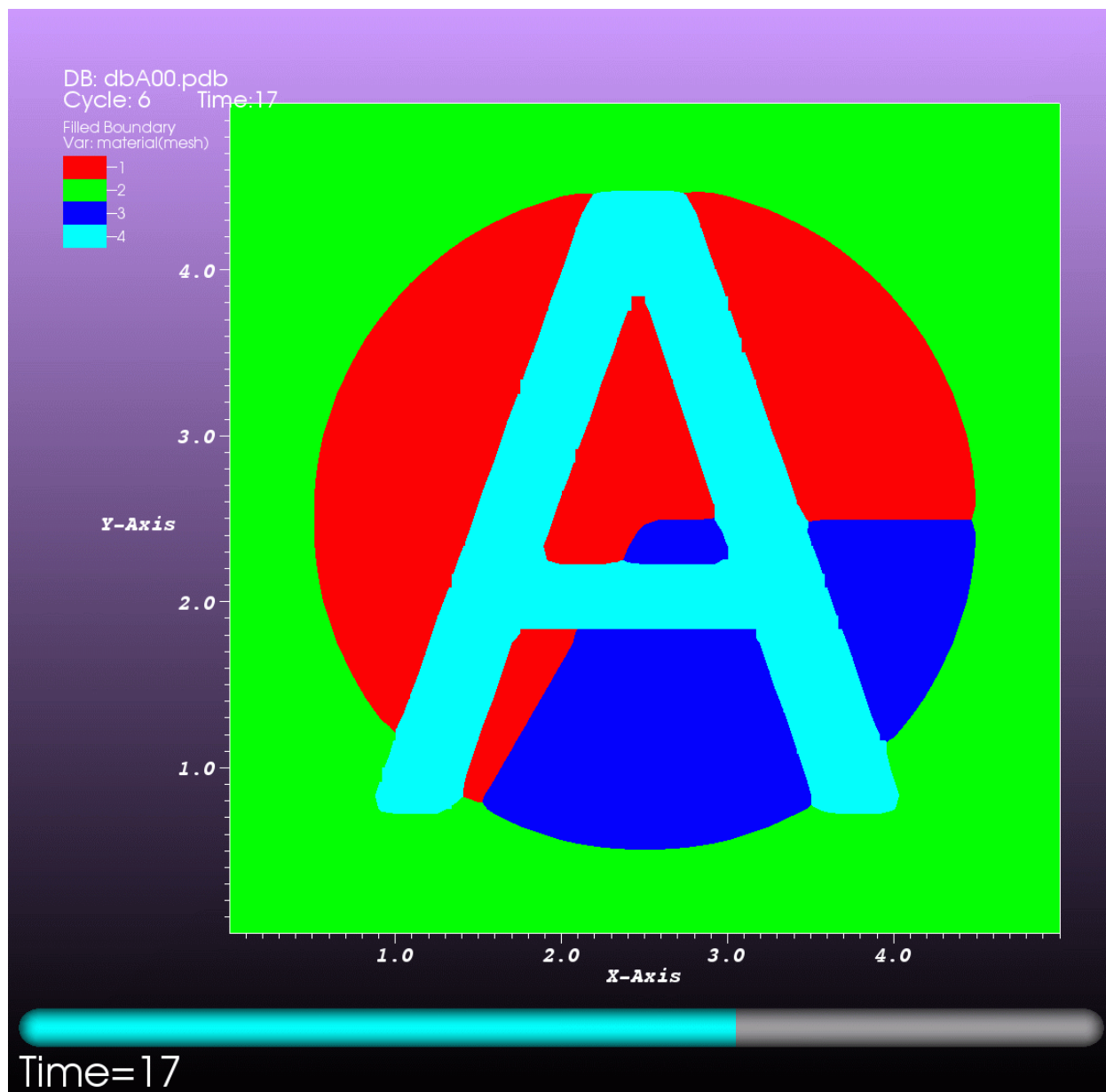
Fig. 1.240: An example of a time slider annotation object
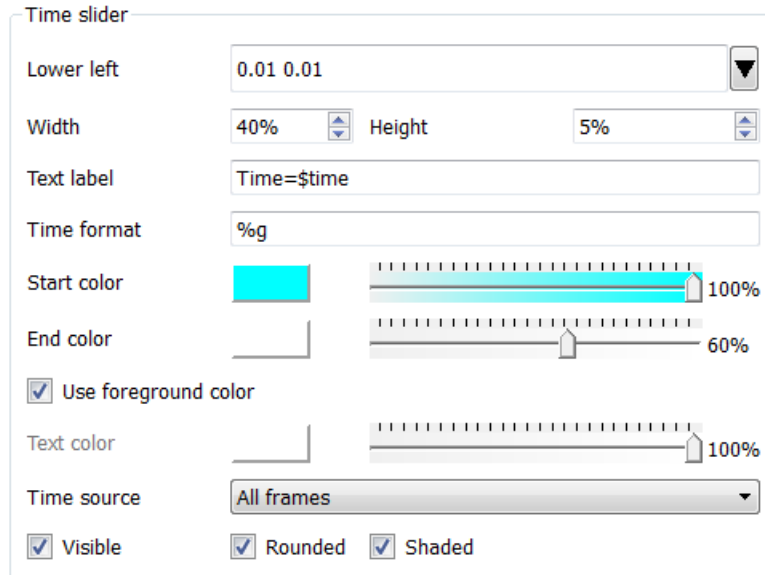
Fig. 1.241: The time slider interface

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 1.243, 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.

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.

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
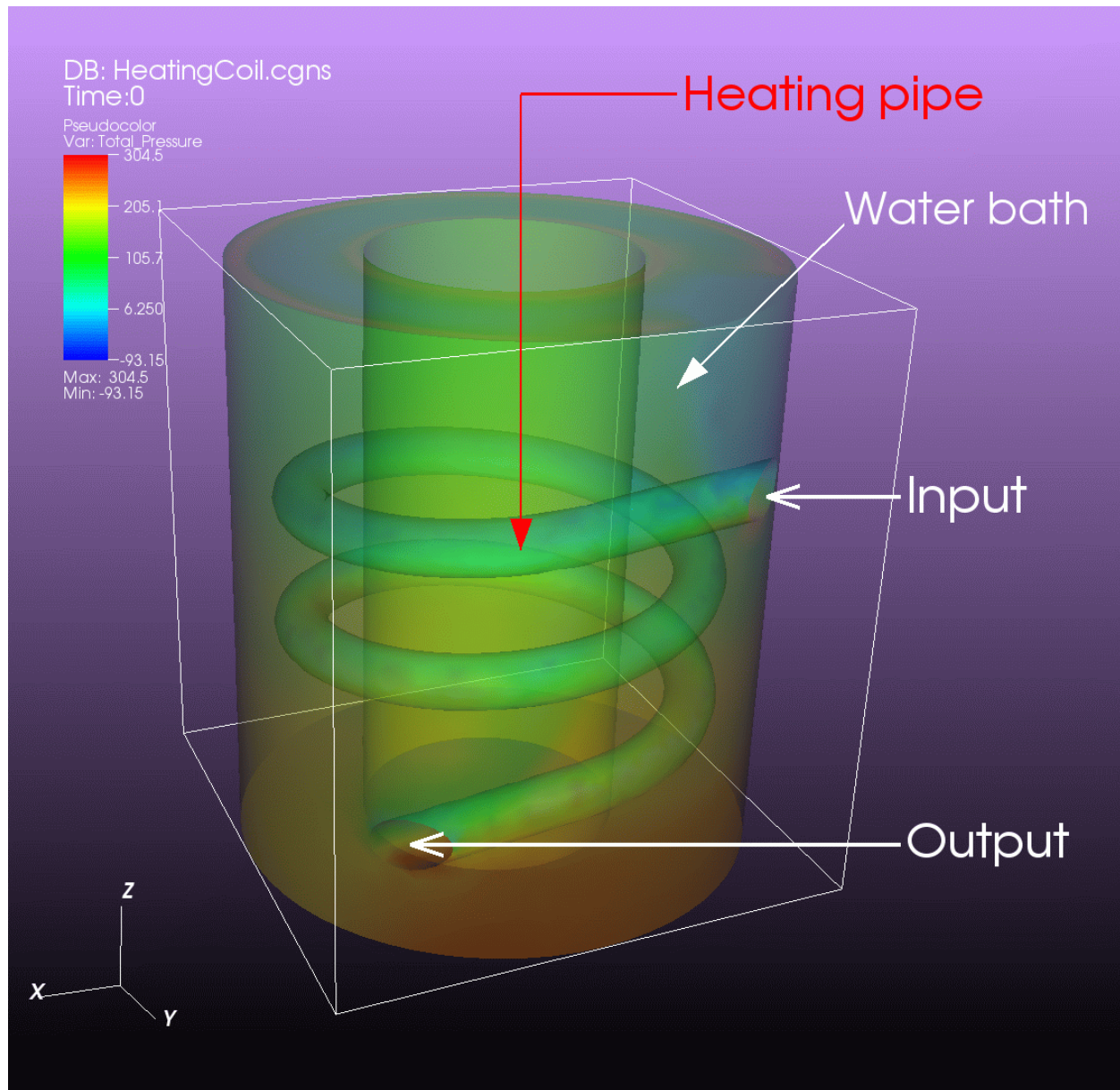
Fig. 1.242: Examples of 2D line annotations
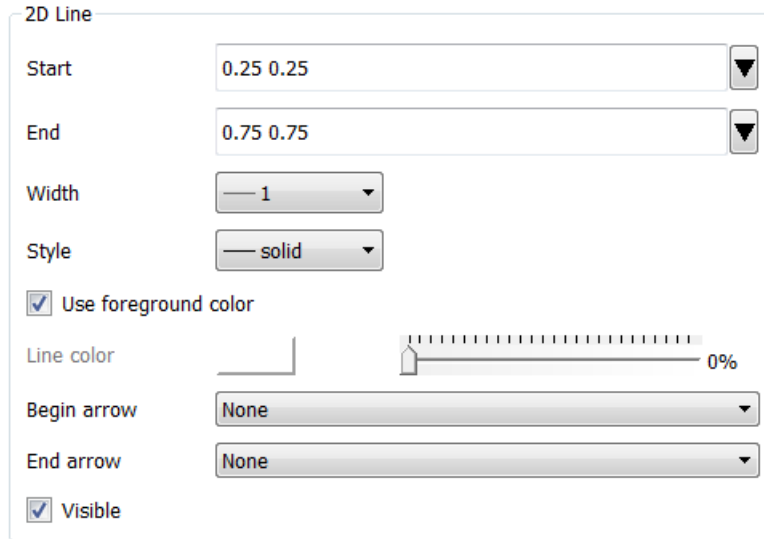
Fig. 1.243: The 2D line object interface

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 1.238, 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 1.244, 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 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

Fig. 1.244: The 3D line object interface

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 1.245, 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 1.246. Note that since image annotations are incorporated into a visualization inside of VisIt's viewer component, the image file must be located on the same computer that runs the viewer.

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 1.245 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
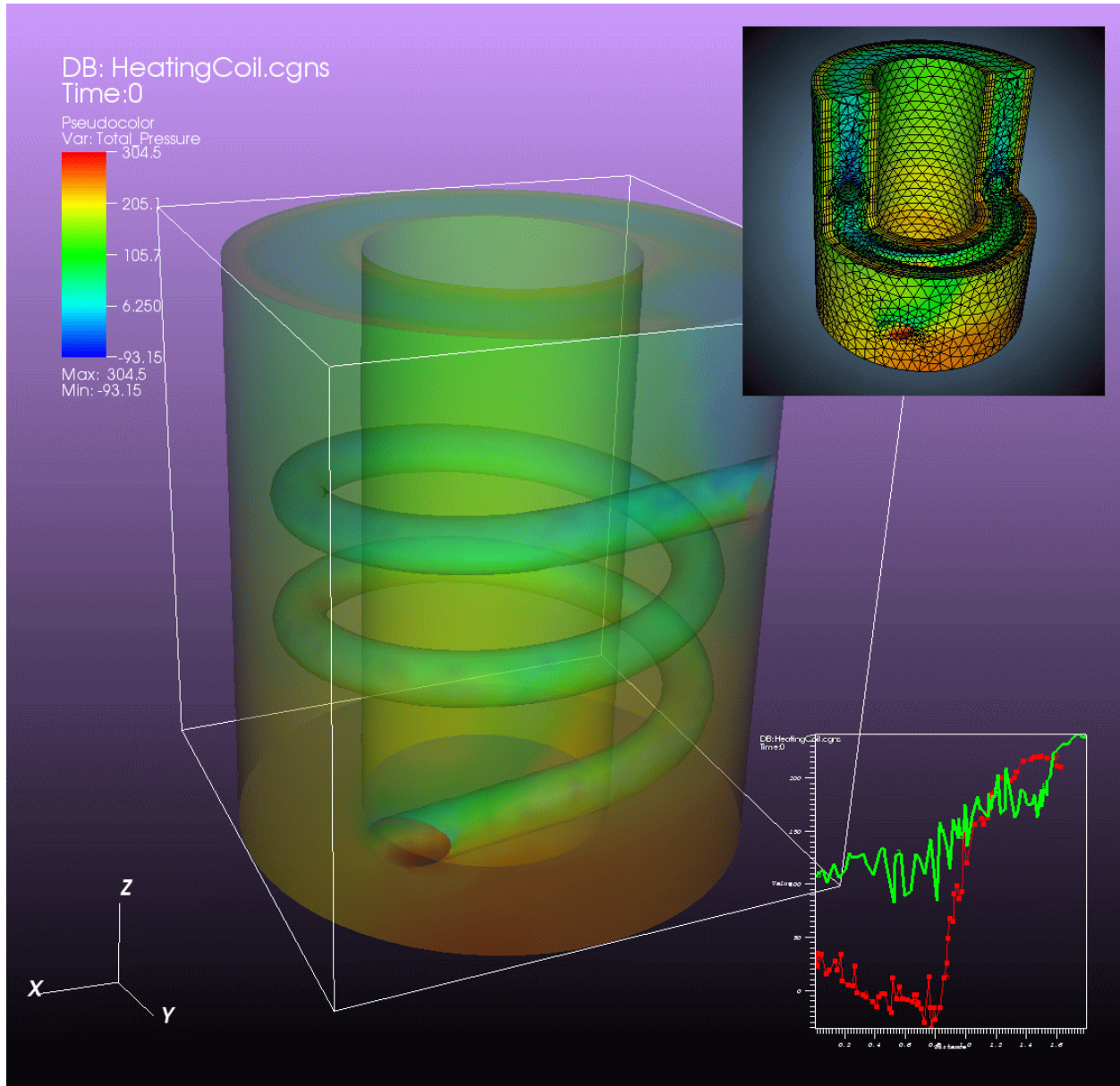
Fig. 1.245: An Example of a visualization with two overlaid image annotations
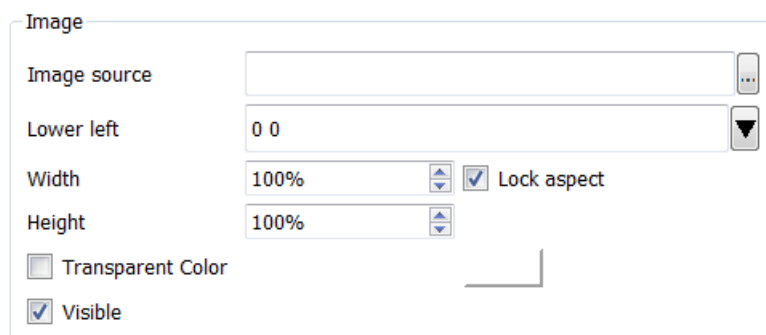
Fig. 1.246: The image object interface

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**.

1.9.2 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 designing custom color tables.

Color tables come in two types: continuous and discrete. A continuous color table is defined as a relatively few color control points defined at certain intervals in the color table and the gaps in between the color control points are filled by smoothly interpolating the 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 through 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. 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

You can open VisIt's **Color table window**, shown in Figure 1.247, by selecting **Color table** from the **Main Window's Controls** menu. The **Color table window** is vertically separated into three areas. The top area allows you to set the active color tables. The middle area, or manager portion of the window, allows you to create or delete new color tables, as well as export color tables. The bottom area, or editor portion of the window, allows you to edit color tables by adding, removing, moving, or changing the color of color control points. A color control point is a point with a color that influences how the color table will look.

Setting the active color table

VisIt has the concept of active color tables, which are the color tables used to color plots that do not specify a color table. There is both an active continuous color table (for plots that prefer to use continuous color tables) and an active discrete color table (for plots that prefer to use discrete color tables). The active color table can be different for each visualization window. To set the active continuous color table, select a new color table name from the **Continuous** menu in the **Active color table** area. To select a new active discrete color table, select a new color table name from the **Discrete** menu in the **Active color table** area.

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 to fashion a new color table.
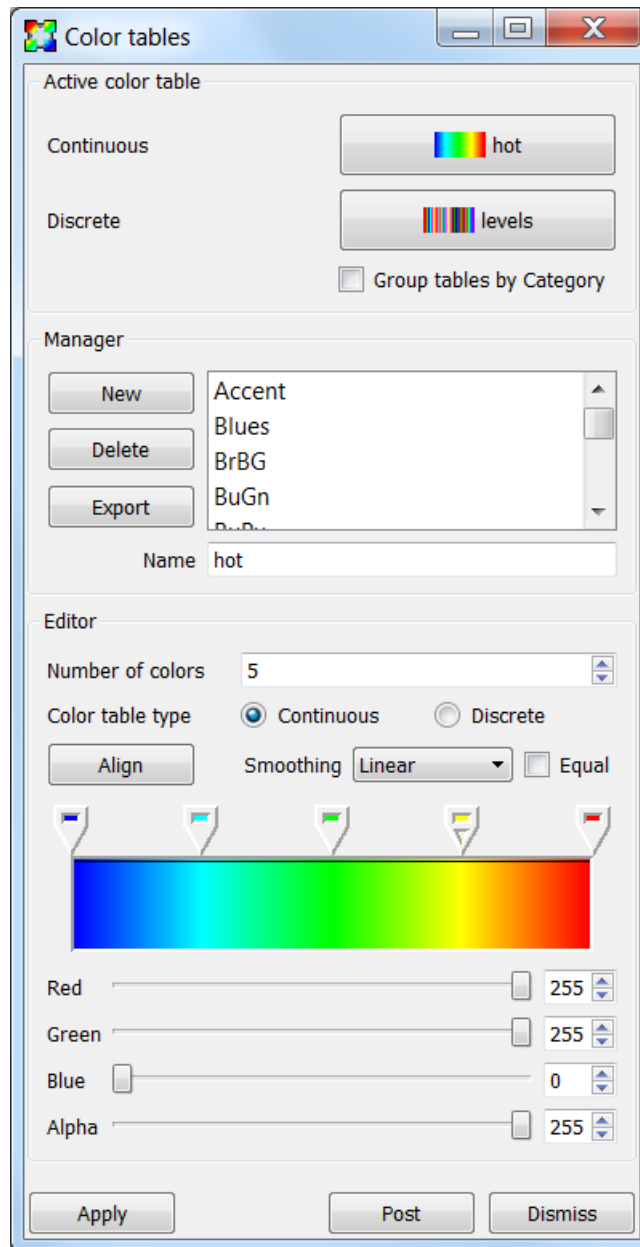
Fig. 1.247: The color table window

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 delete all color tables except for the last color table. VisIt makes no distinction between built-in color tables and user-defined color tables so any color table can be deleted. When you delete a color table, the active 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, 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
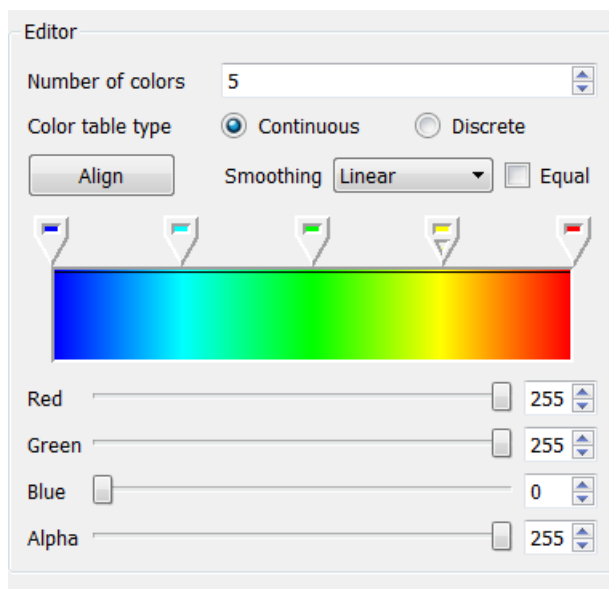
Fig. 1.248: The continuous color table editor

There are a handful of controls in the editor portion of the **Color table window**, shown in Figure 1.248, 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 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
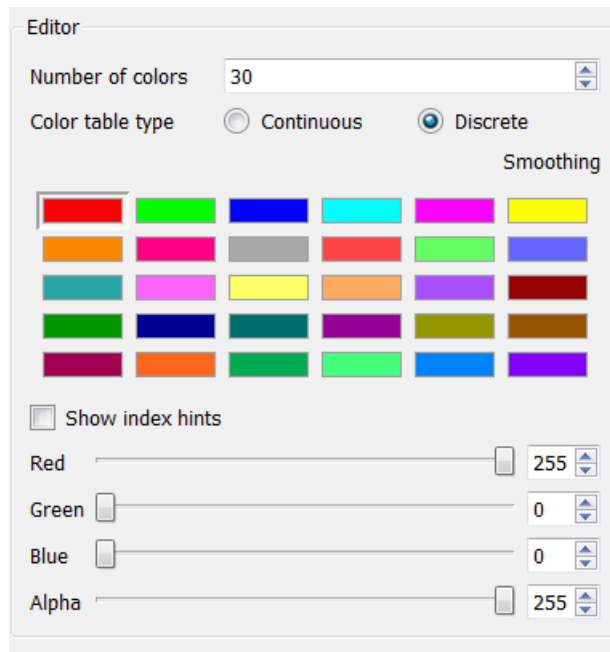
Fig. 1.249: The discrete color table editor

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 if 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.

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 the levels 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.

1.9.3 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 1.250) 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.
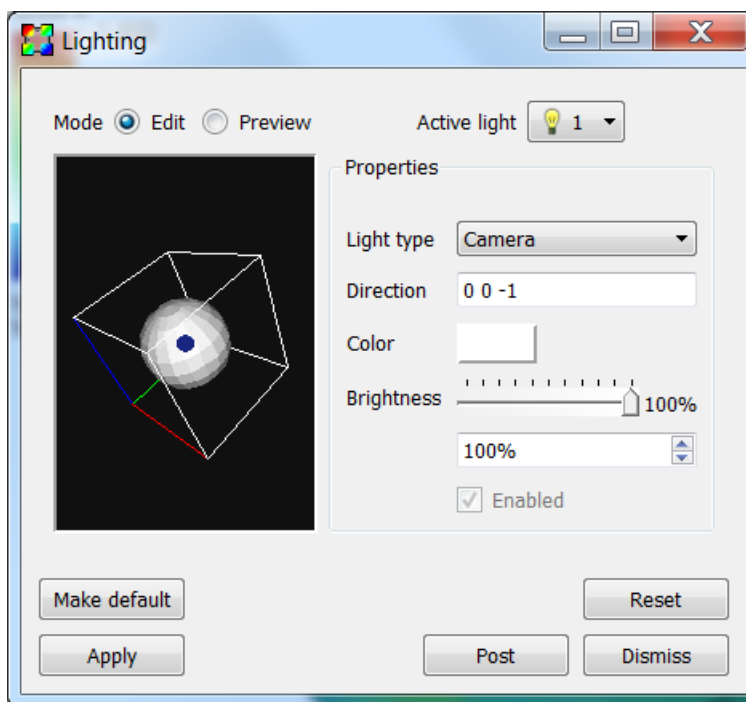
Fig. 1.250: The lighting Window

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. 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
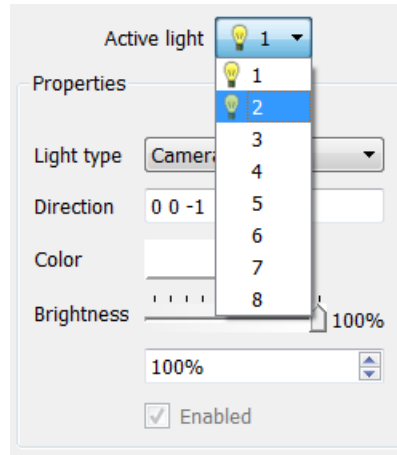
Fig. 1.251: The active light menu

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 1.251). 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

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,
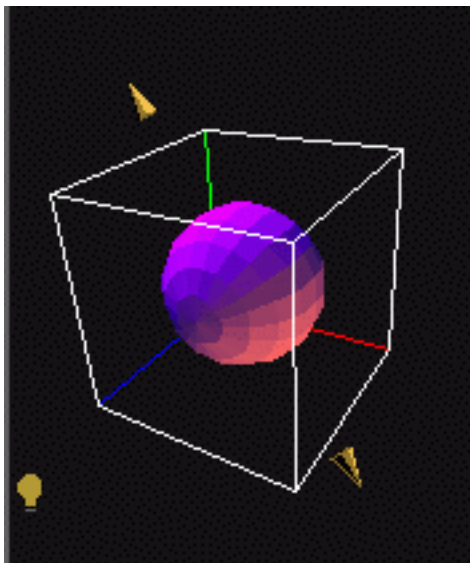
Fig. 1.252: The different kinds of lights

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 allows lights to have color as well as brightness. Colored lighting can produce interesting effects that may be desirable for presentations. 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**.

1.9.4 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 1.253) 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 1.254). 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 1.253). When antialiasing is enabled, all lines drawn in a visualization window are blended with the background image so that they look smoother.
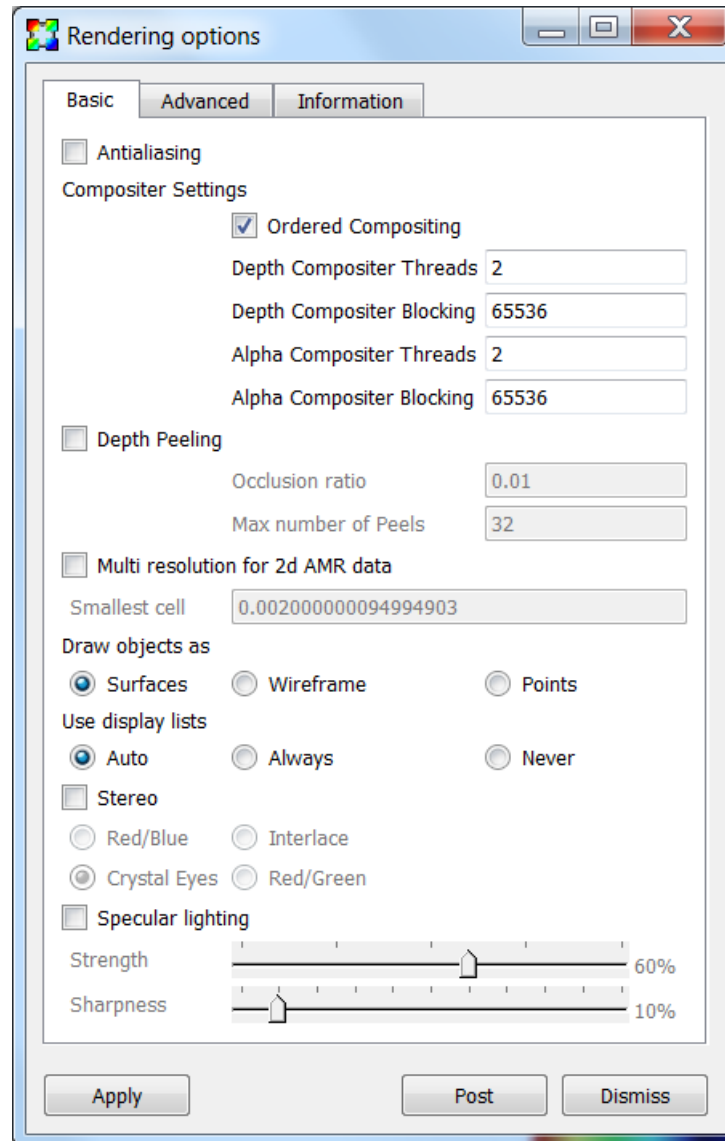
Fig. 1.253: The basic rendering options
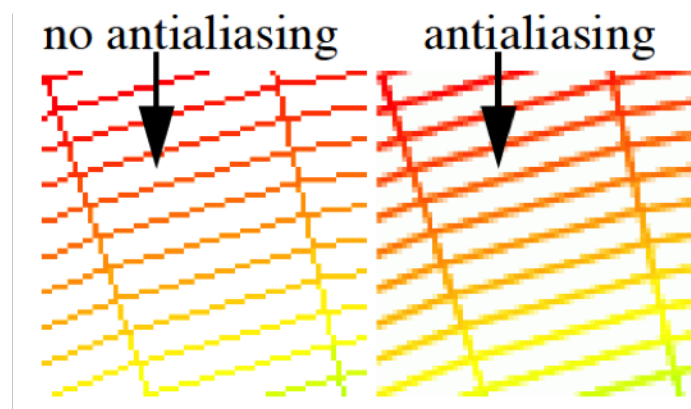
Fig. 1.254: An example of antialiasing

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 1.253).
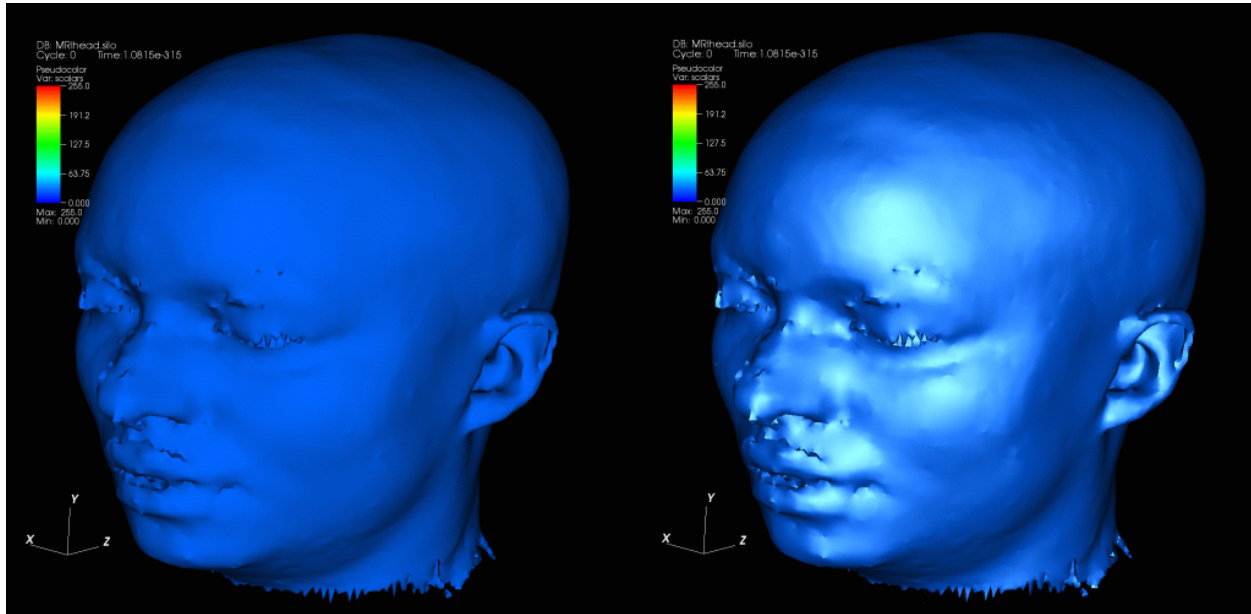
Fig. 1.255: The effects of specular lighting on plots

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 1.255. The plot on the left side has no specular highlights and the plot on the right side has specular highlights.

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 1.256). 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 1.257.

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
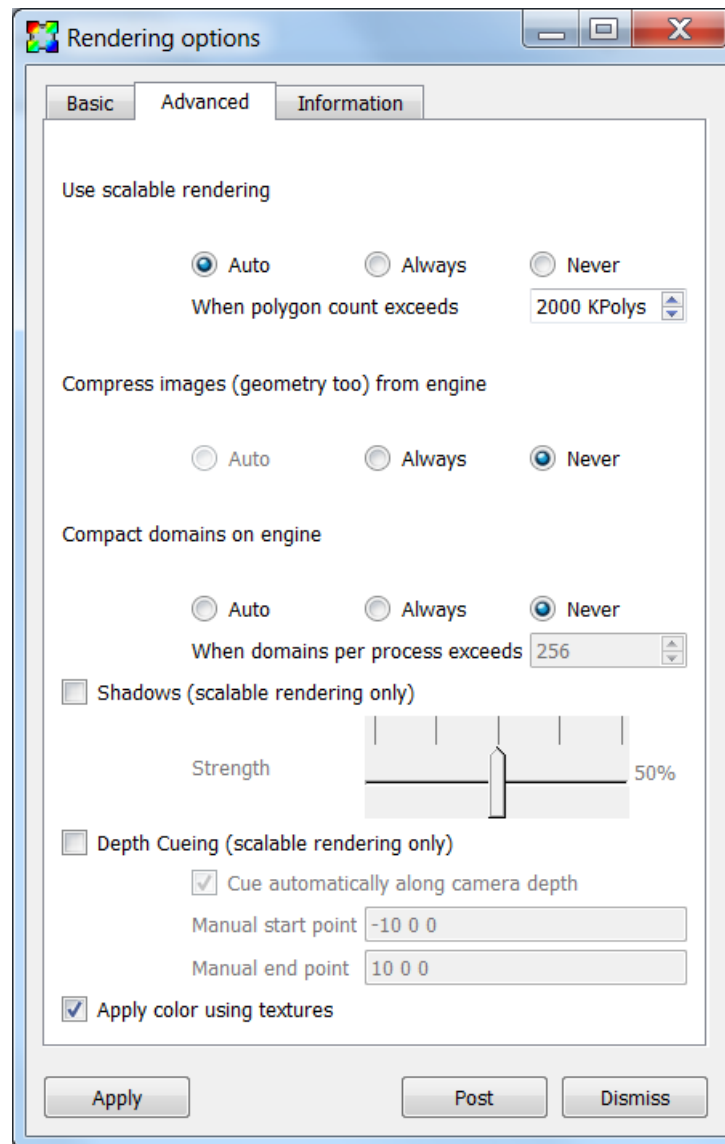
Fig. 1.256: The advanced rendering options
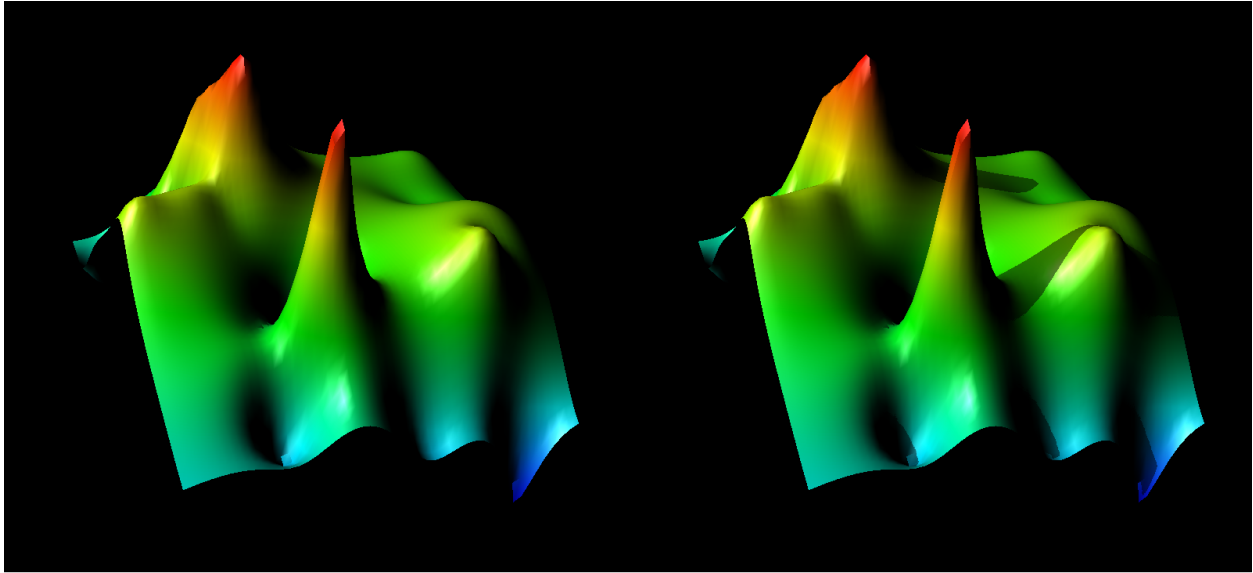
Fig. 1.257: The effects of shadows on plots

from the camera. The controls to turn on depth cueing can be found near the bottom of the **Advanced** tab (see Figure 1.256). 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 1.258.
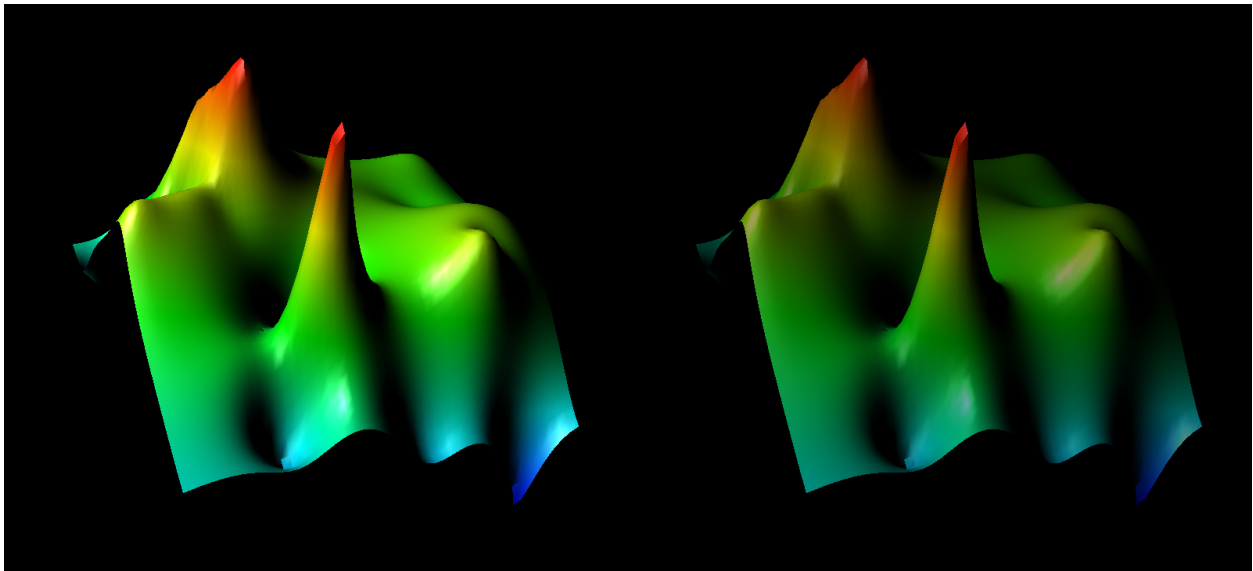
Fig. 1.258: The effects of depth cueing on plots

1.9.5 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 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 1.259). 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
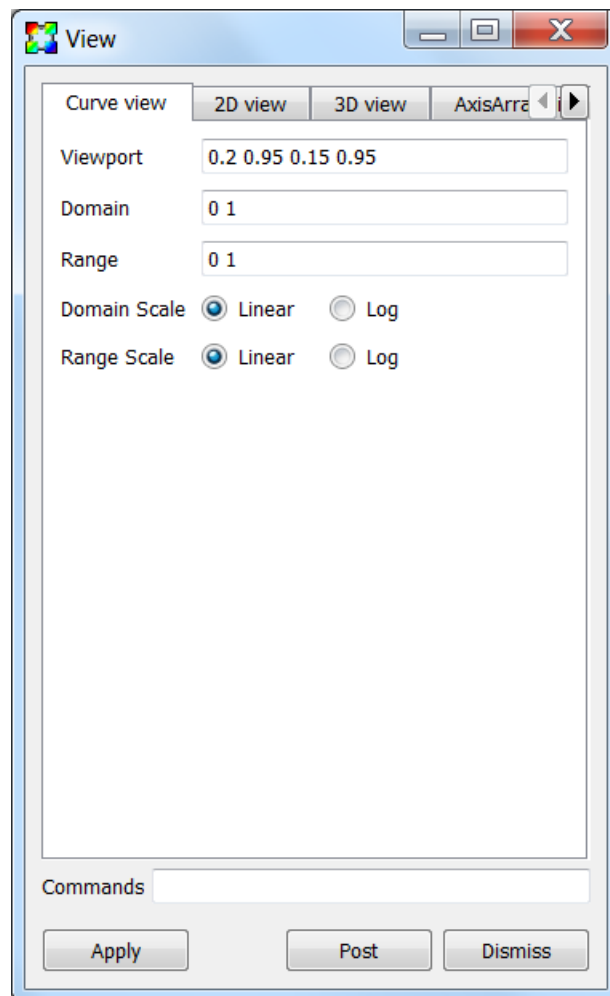
Fig. 1.259: The curve view options

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 1.260).
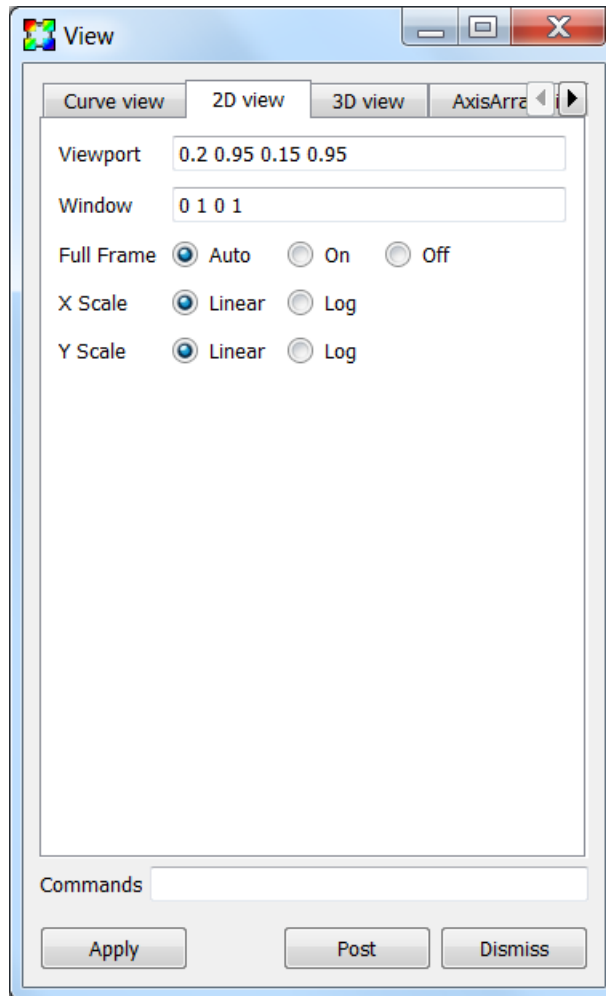
Fig. 1.260: The 2D view options

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. 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.

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.
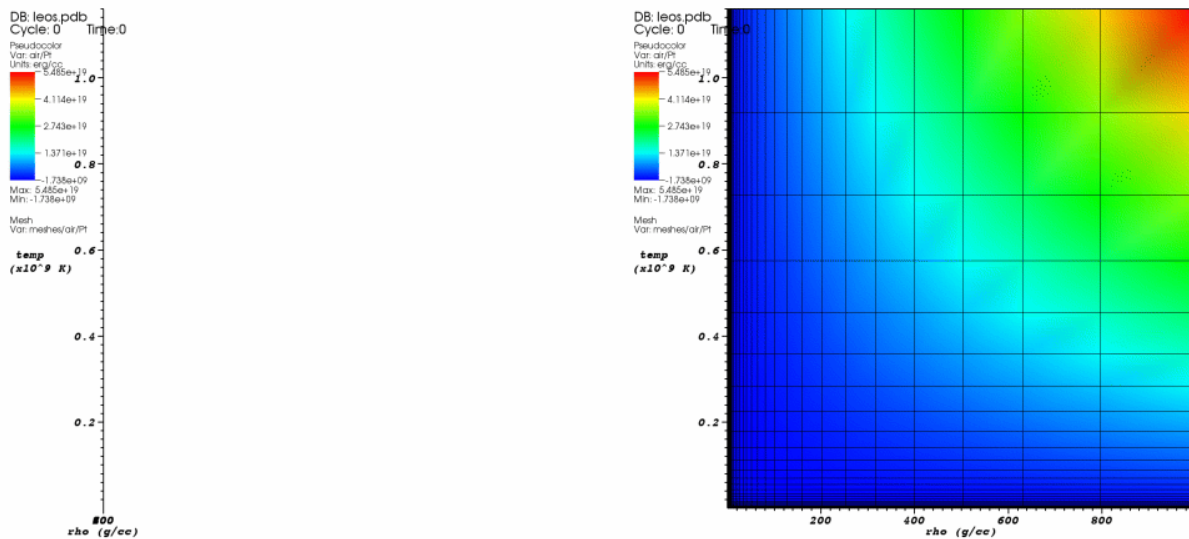
Fig. 1.261: The effect of full frame mode on an extremely skinny plot

Setting the 3D view

Setting the 3D view using controls in the **View Window's 3D view** tab (see Figure 1.262) 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 1.263 depicts the various components of a 3D view.

To set the 3D view, first decide on where you want to look from. Type a vector value into the **View normal** text field. Next, type the vector valued location of what you want to look at into the **Focus** text field. The **Up axis** vector is simply a vector that 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. The **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** are 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. **Parallel scale** acts as a zoom factor and larger values zoom the camera towards the focus. The **Perspective** check box 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**.
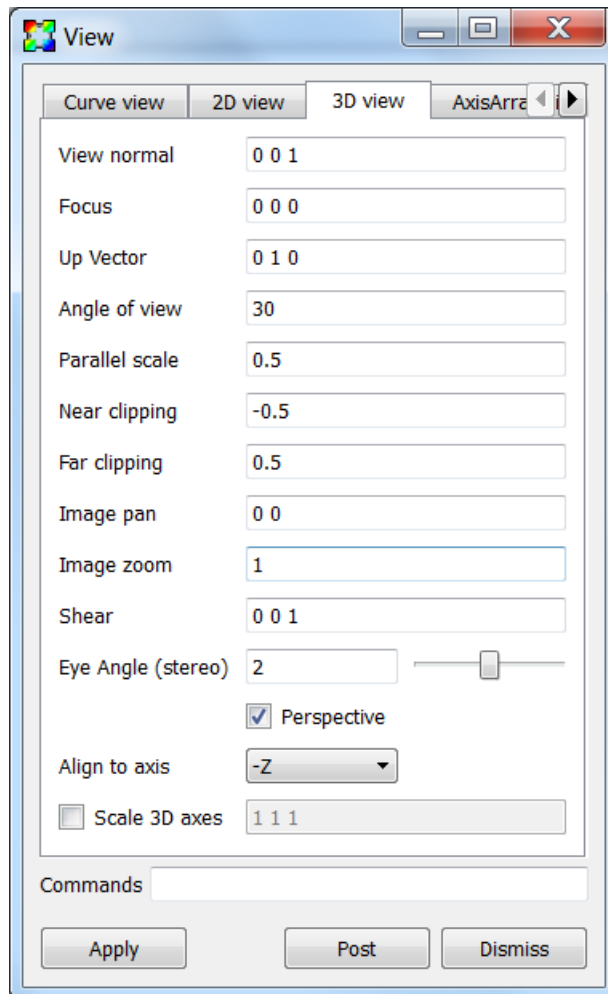
Fig. 1.262: The 3D view options
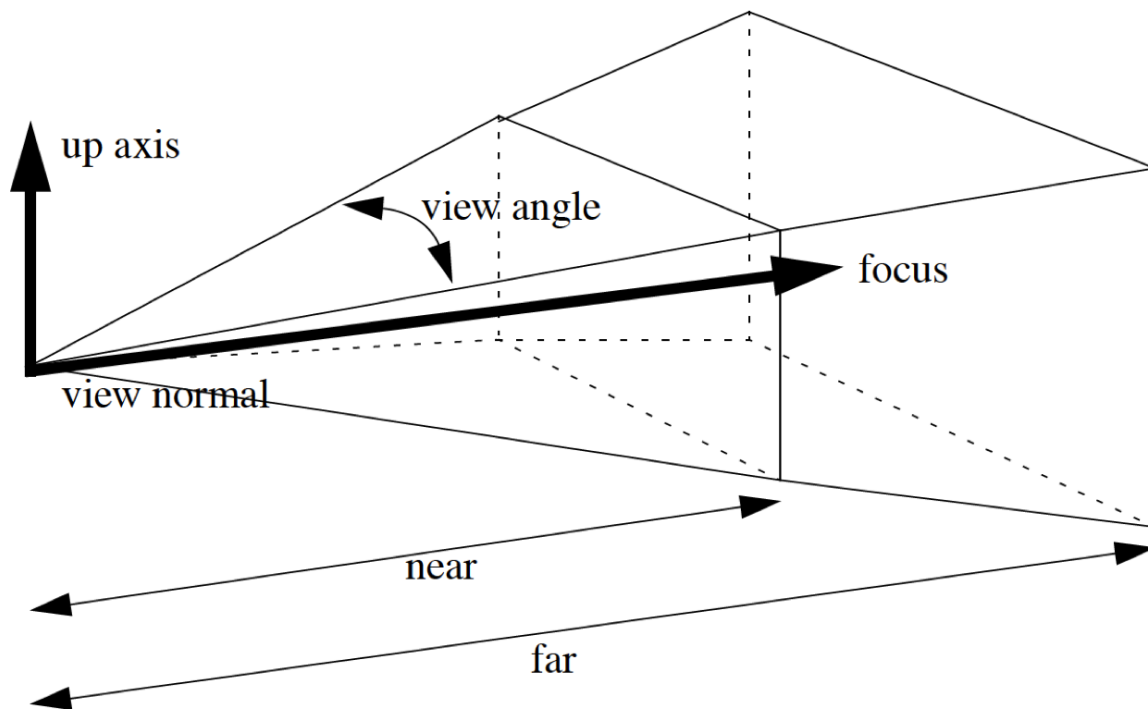
Fig. 1.263: The 3D perspective view volume

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 1.264). The minimum and maximum X values should come first, followed by the minimum and maximum Y values.

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 1.265, contains advanced features that are not needed by all users.
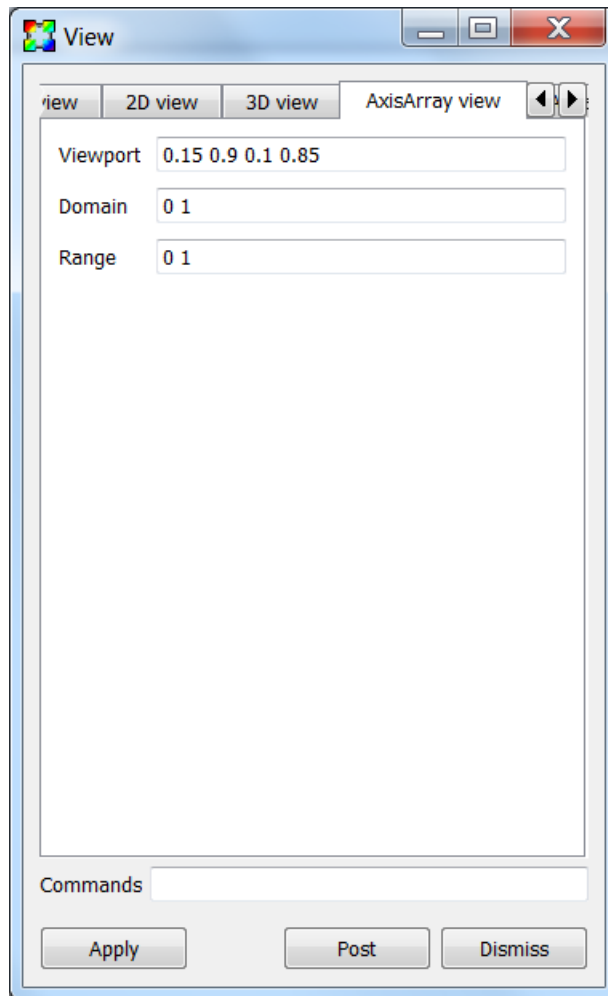
Fig. 1.264: The axis array view options
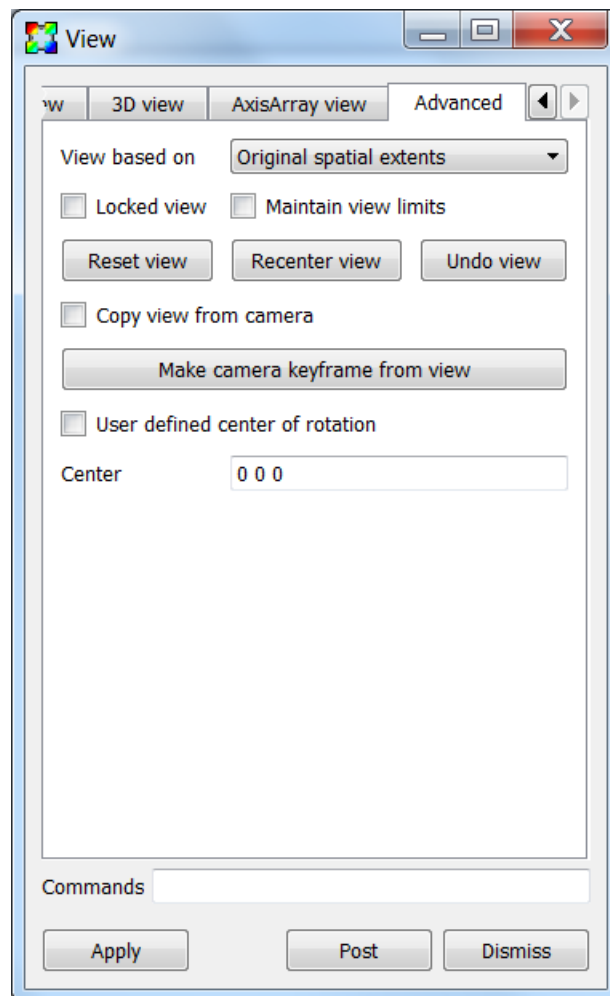
Fig. 1.265: The advanced view options

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 table has a description of the supported view commands:

Com- mand	Description
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.
re-set	Resets the 2D and 3D views
re-cen-ter	Recenters the 3D view
undo	Changes back to the previous view

1.10 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. 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.

1.10.1 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.

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
...
```

VisIt provides a **File grouping** combo box in the **File open** window (see Figure 1.266) 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.
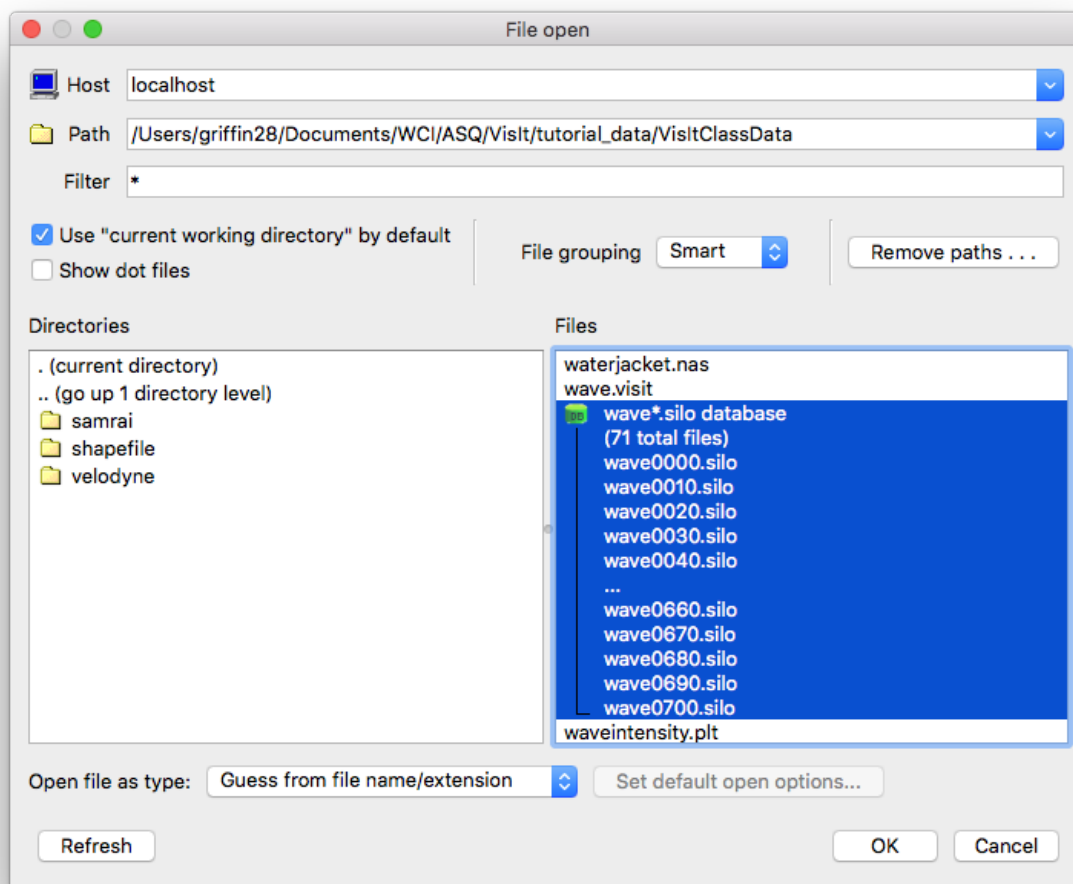
Fig. 1.266: File open window

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 1.267 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 1.267 , allow you to control how a flipbook animation plays. The animation controls are are also used for controlling keyframe
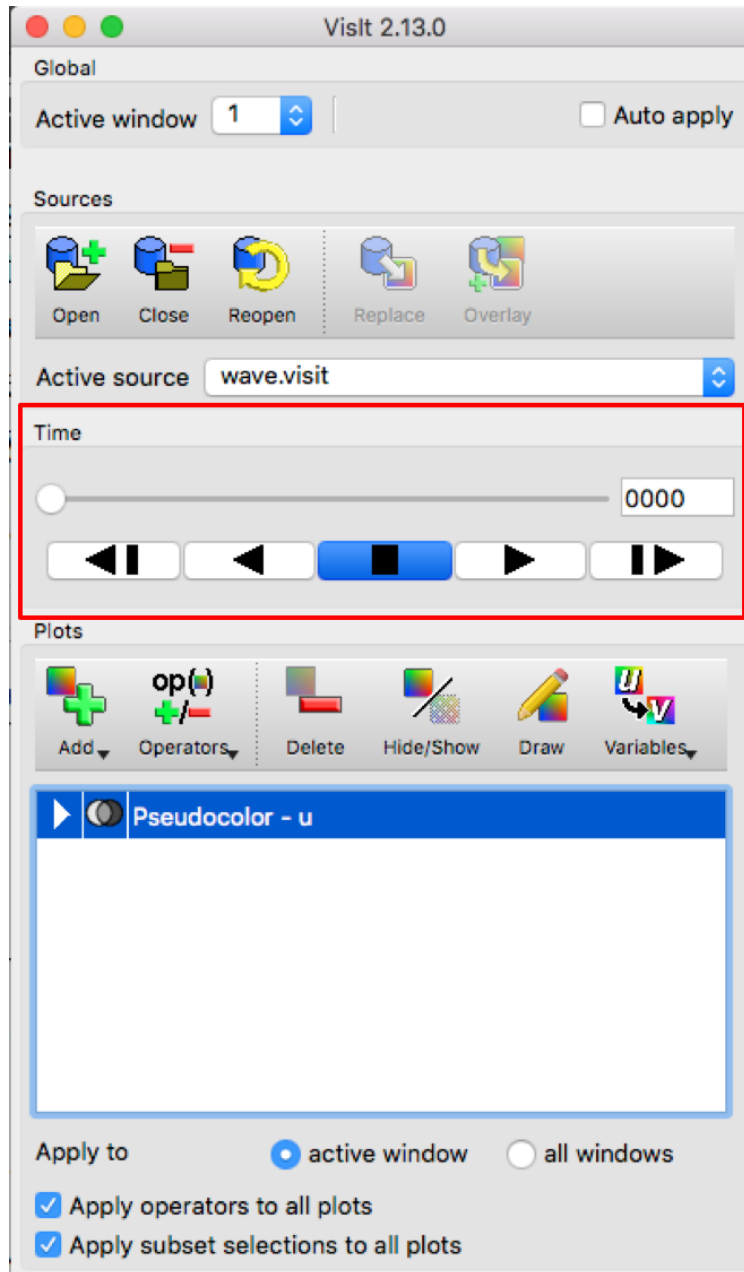
Fig. 1.267: Animation controls

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.

Animation Window

You can open the **Animation Window**, shown in Figure 1.268 , 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
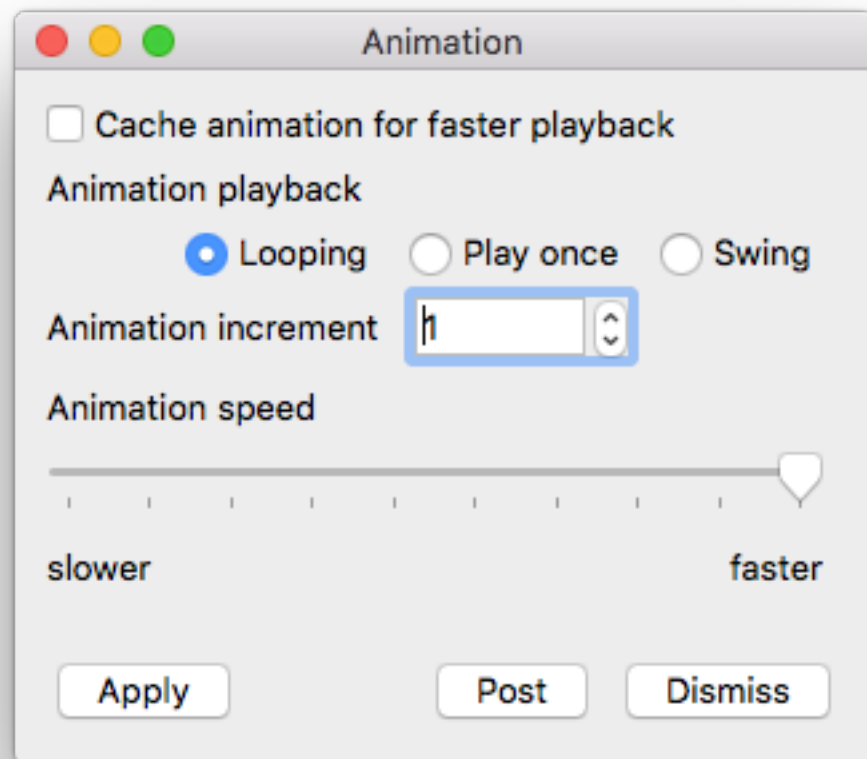
Fig. 1.268: Animation window

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** .

1.10.2 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.

Keyframing Window
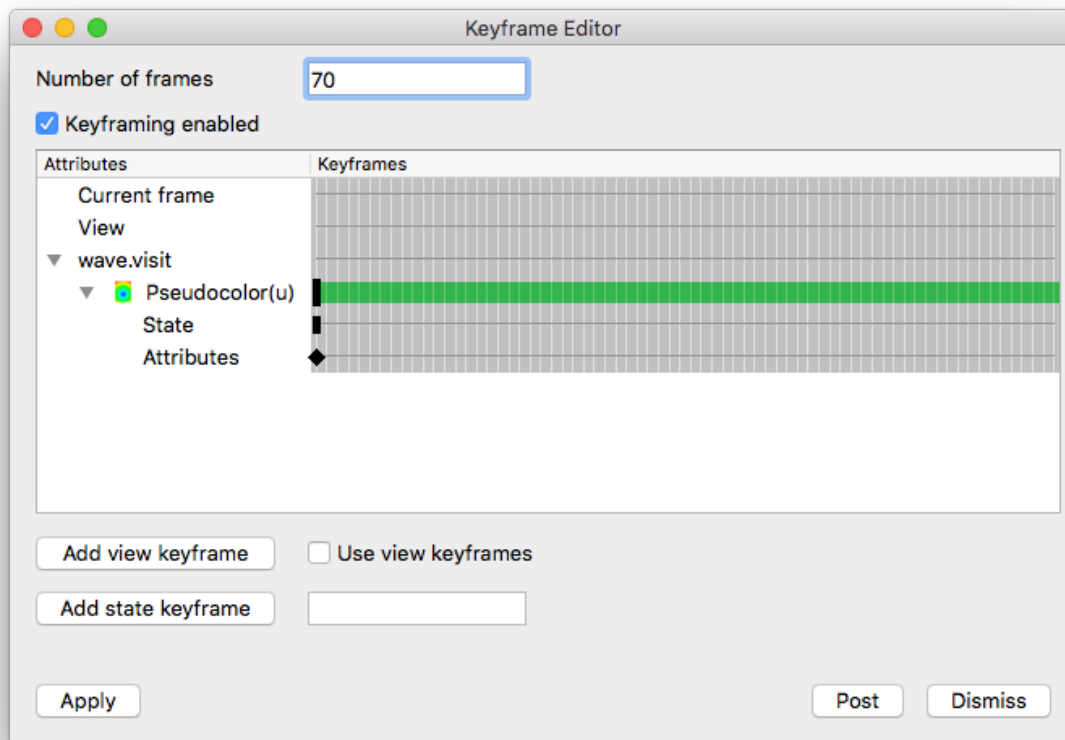
Fig. 1.269: Keyframing Window

Keyframe animations are designed using VisIt's **Keyframing Window** (see Figure 1.269), which you can open by selecting the **Keyframing** option from the **Controls** menu. The window is dominated by the **Keyframe area** , which 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
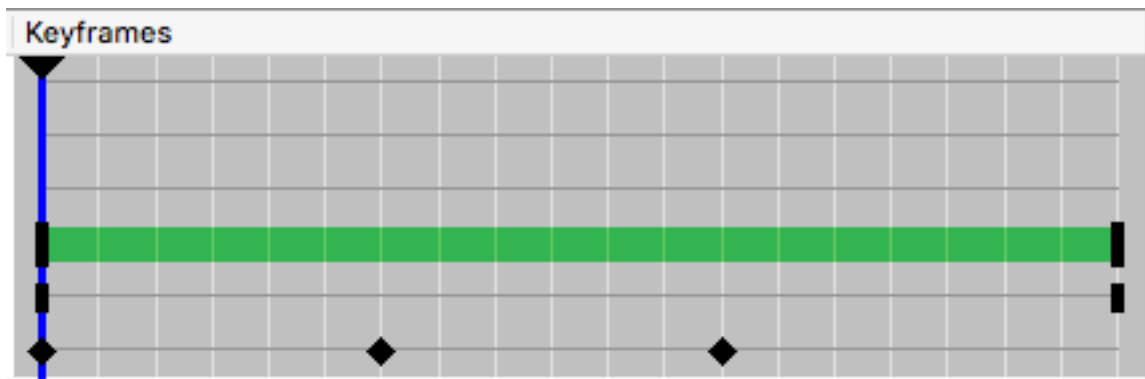
Fig. 1.270: Keyframe area

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 1.270), 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.

1.10.3 Scripting

Scripting is an alternate method of producing animations that can have the simplicity of flipbook animations with the flexibility of keyframing. Scripting animations is more difficult than other methods because you have to script each event by writing a Python or Java program to control VisIt's viewer. One clear strength of this method 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 *Visit Python Interface* manual.

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 1.271).

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.

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.
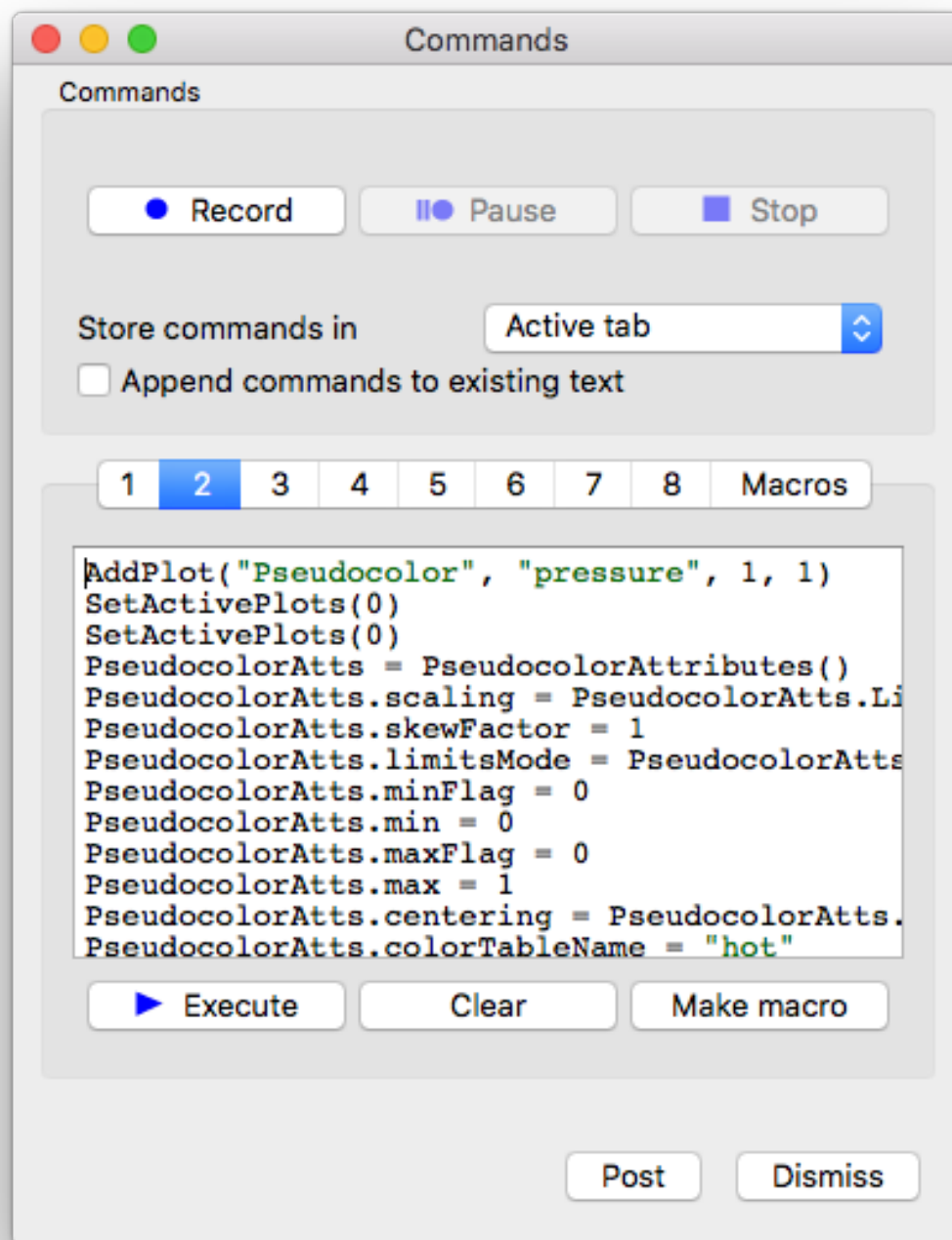
Fig. 1.271: Command Window

1.10.4 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 table lists `visit -movie` command line arguments:

Argument	Description
-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.
	The frame at which to start.
-end frame	The frame at which to end.
-output	The output option lets you set the name of your movie.
-fps number	Sets the frames per second at which the movie should be played.

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
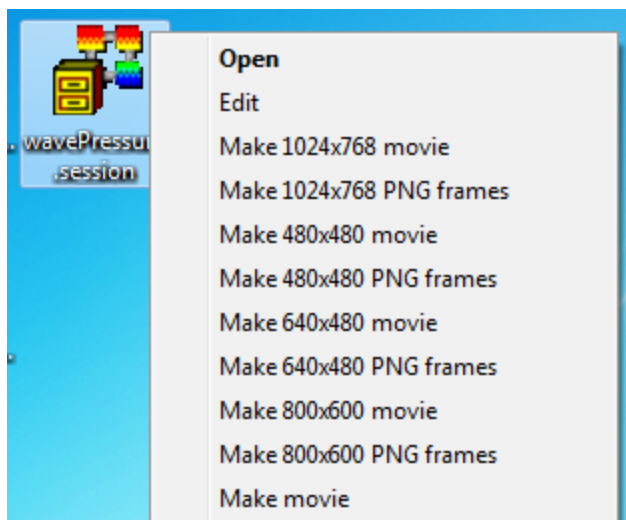
Fig. 1.272: Movie generation options for session files on Windows platform

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, 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 1.272.

1.11 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 1.273). 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.

1.11.1 Box Tool

The box tool, which is shown in Figure 1.274, 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 1.275).

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
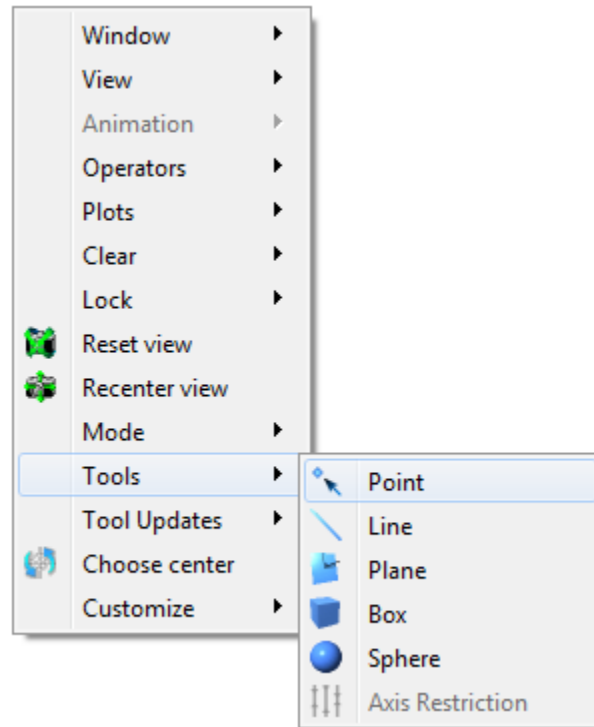
Fig. 1.273: Tools menu
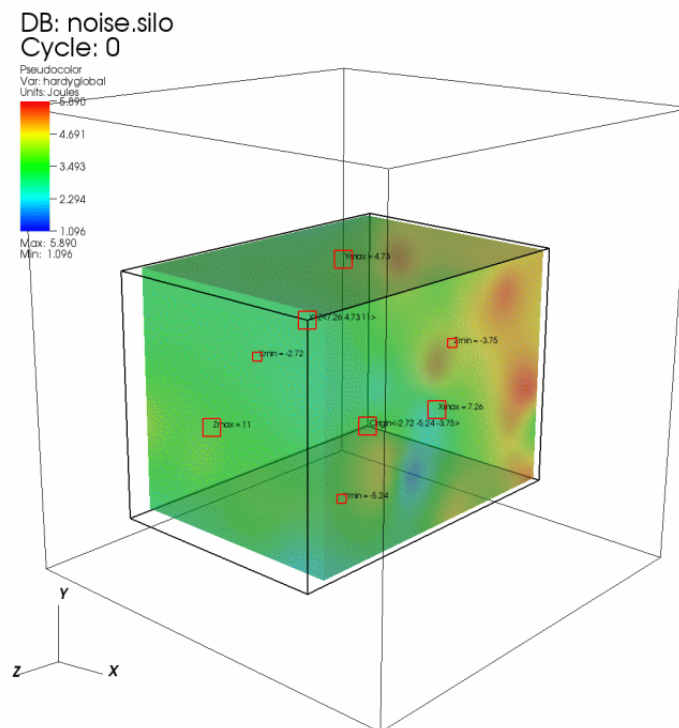
Fig. 1.274: Box tool with a plot restricted to the box

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 "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.
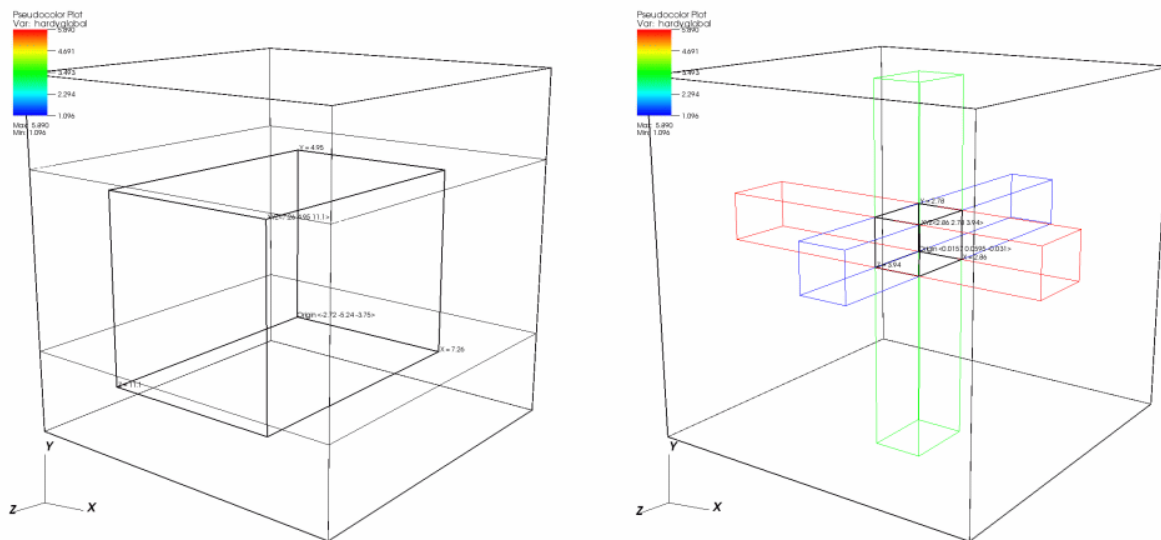
Fig. 1.275: Box tool while it is resized or moved

1.11.2 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 1.276). 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 1.277.

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, the line tool is initialized with that operator's endpoints when it is first enabled. As the
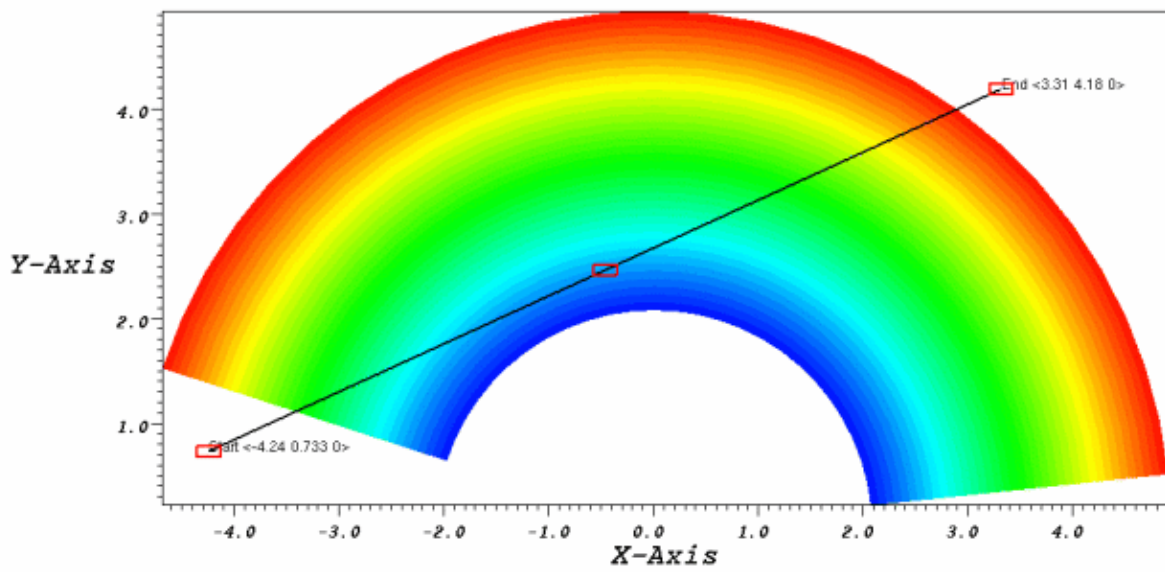
Fig. 1.276: Line tool with a 2D plot
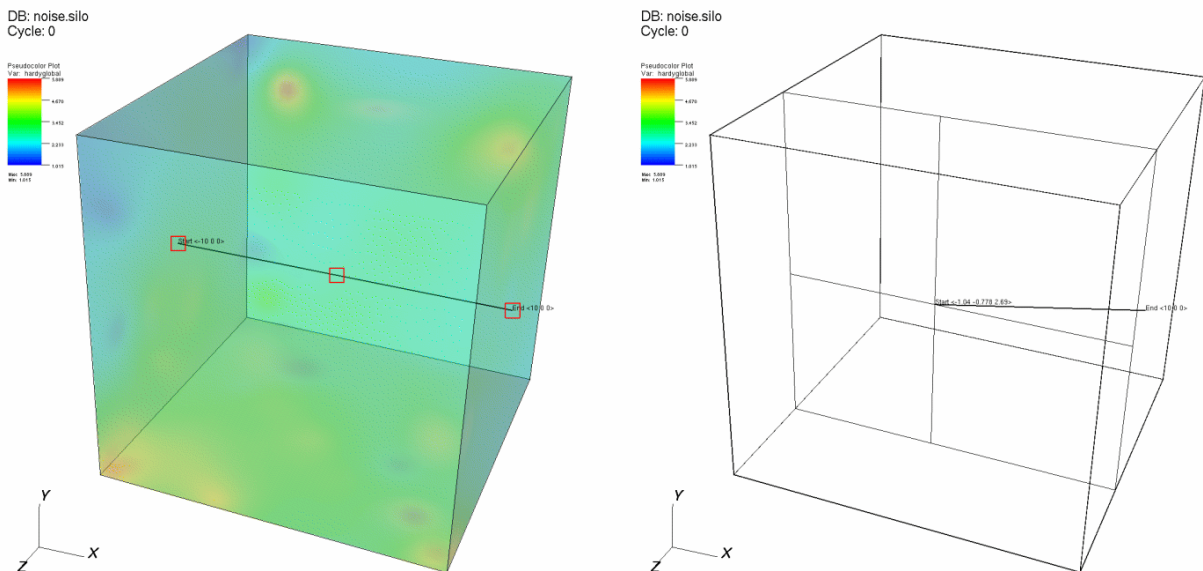
Fig. 1.277: Line tool in 3D

line tool is repositioned and reoriented, the line tool's line endpoints are given to the Lineout operator and and Curve plots that are fed by the Lineout operator are recalculated.

1.11.3 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 1.278, 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.

1.11.4 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 1.279 shows the point tool being used to set the origin for the *ThreeSlice operator*.

1.11.5 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 1.280, 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.

1.11.6 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 1.281, 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.
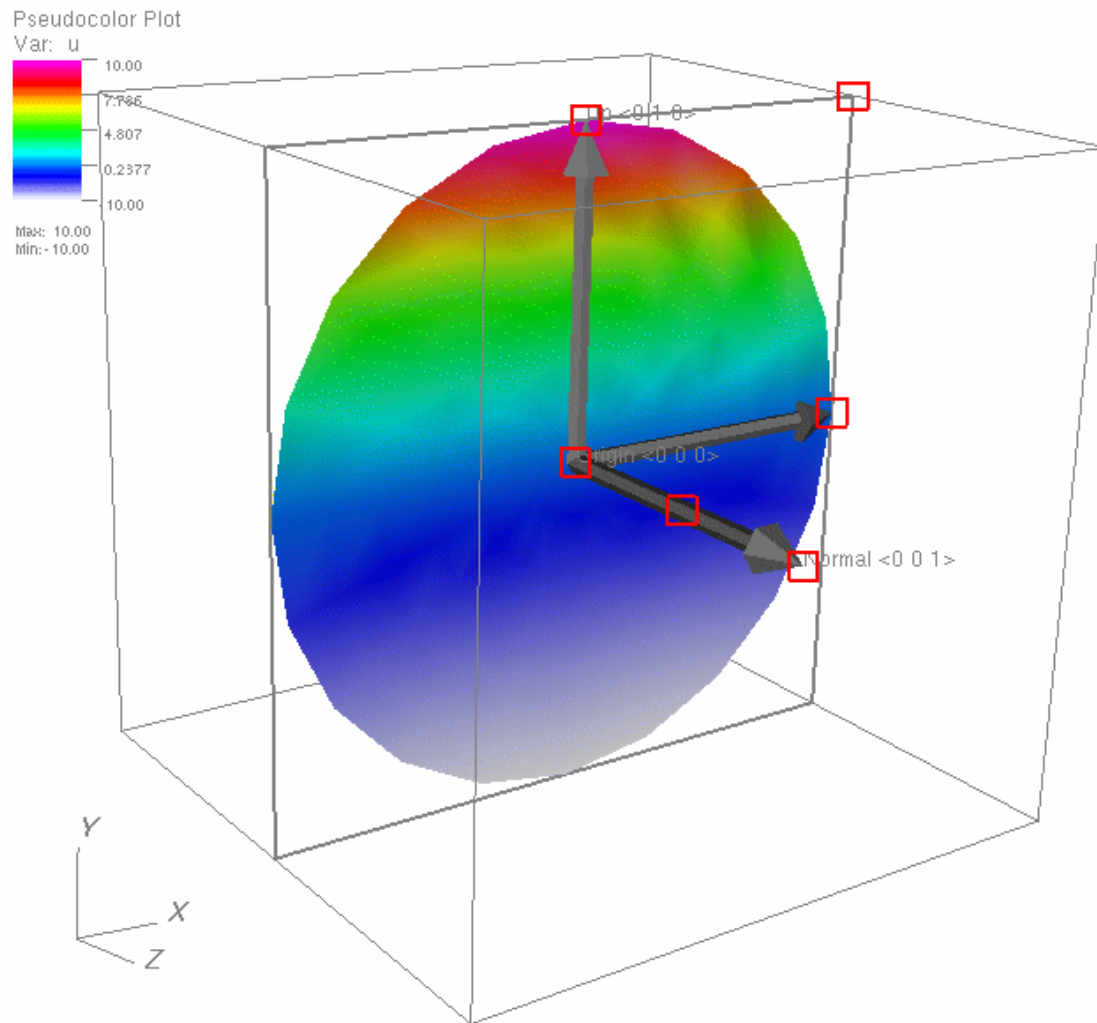
Fig. 1.278: Plane tool with sliced plot
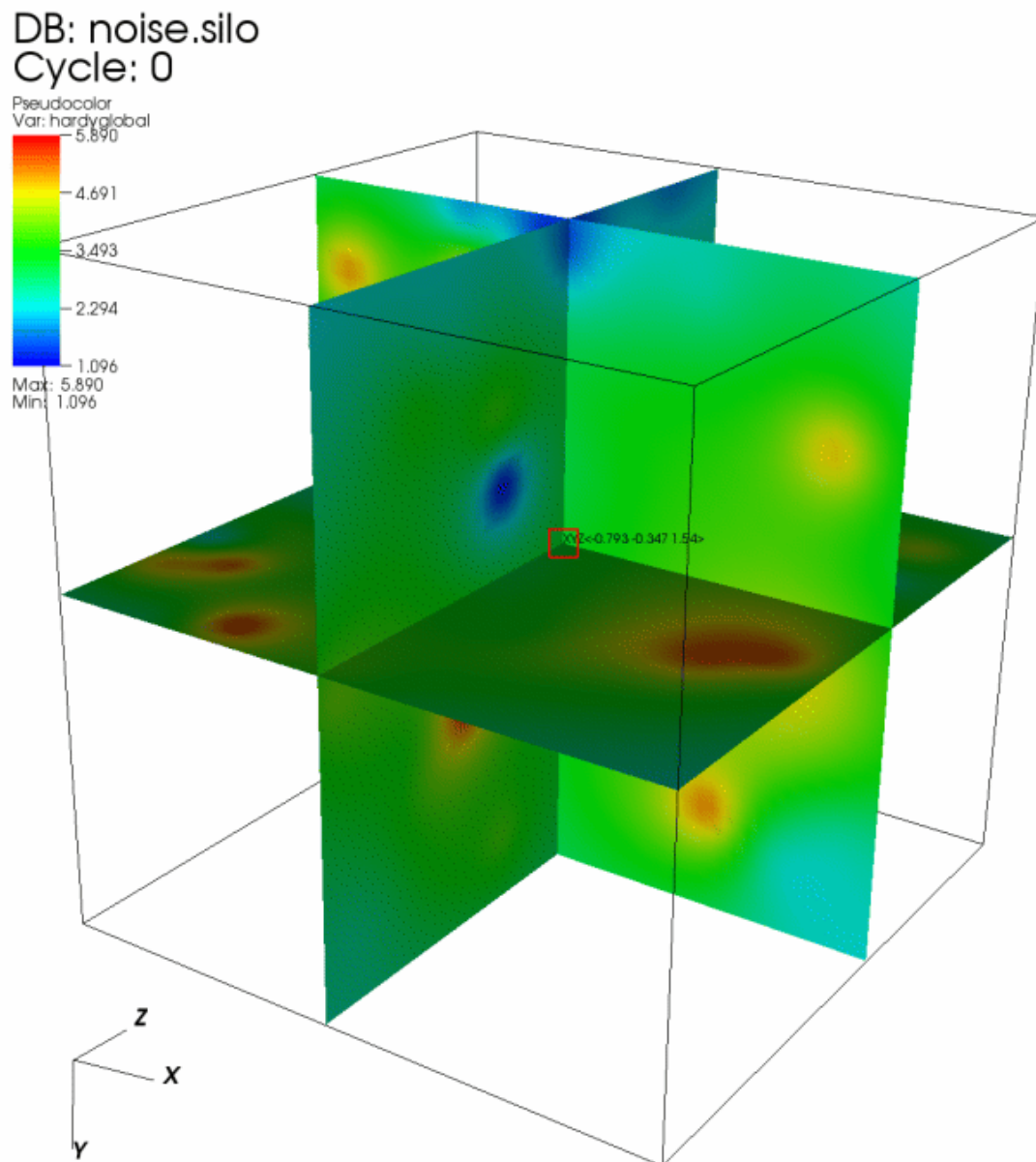
Fig. 1.279: Point tool
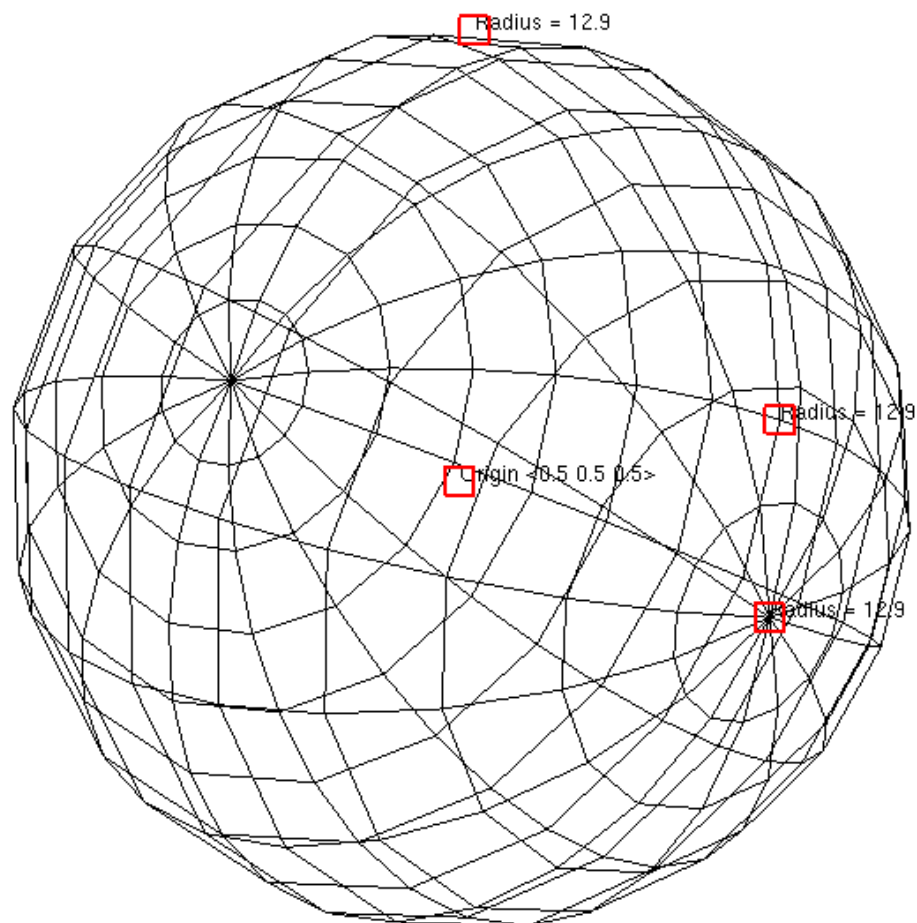
Fig. 1.280: Sphere tool
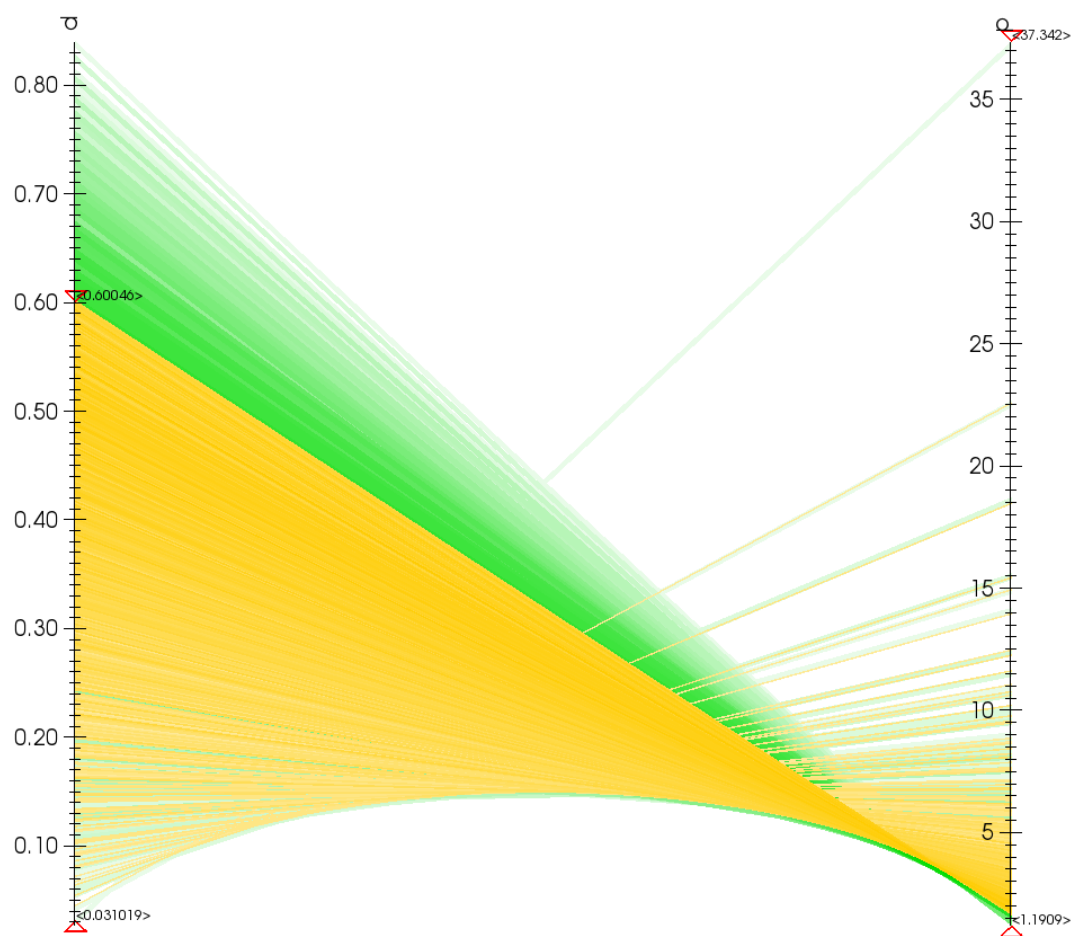
Fig. 1.281: Axis Restriction tool

1.12 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.

1.12.1 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 1.282.
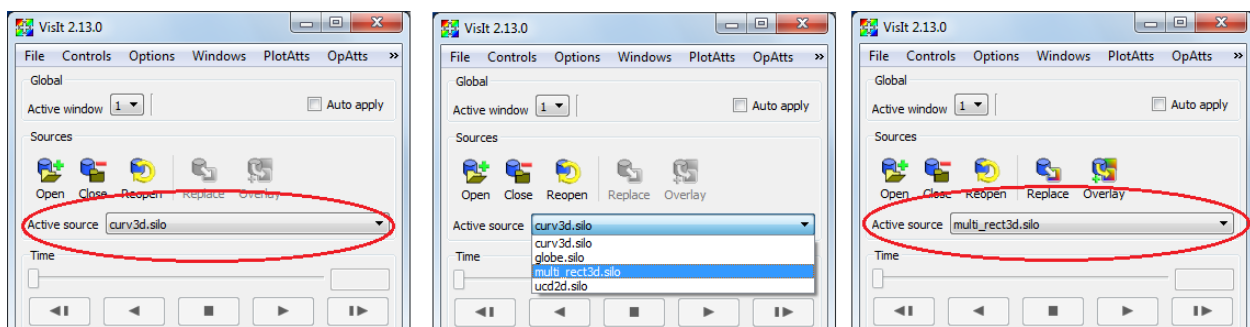
Fig. 1.282: Changing the active source.

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 1.283) that lets you choose which logical time slider to affect when you change time using the **Time slider**.
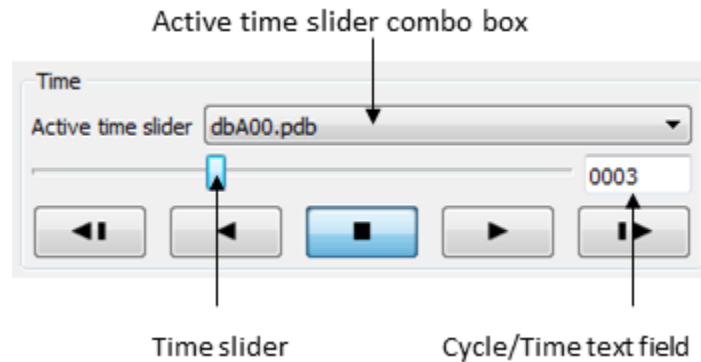
Fig. 1.283: Time slider and related controls

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.

1.12.2 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 1.285 shows a database correlation as the active time slider.
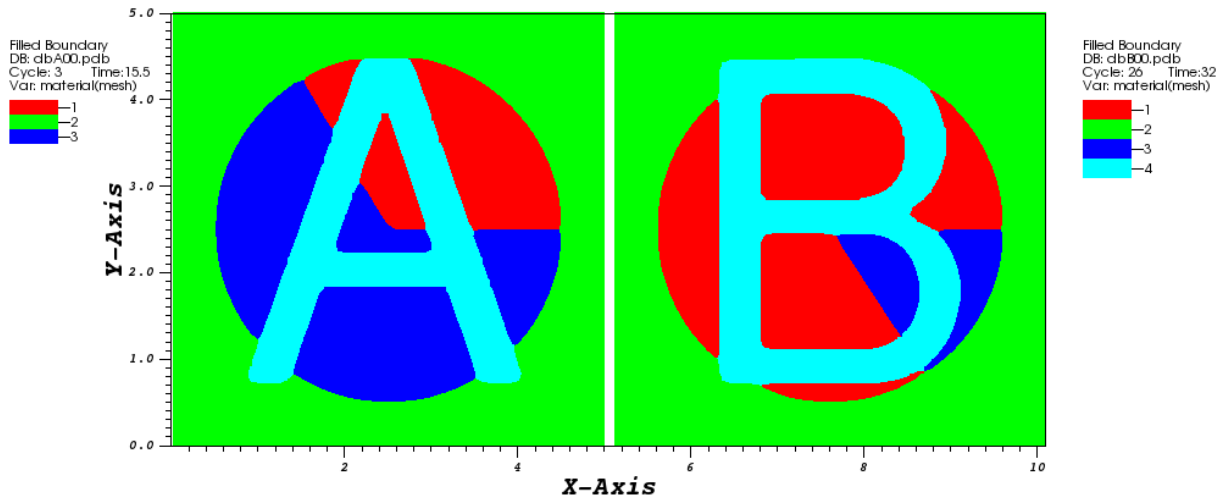
Fig. 1.284: Active time slider and time slider controls
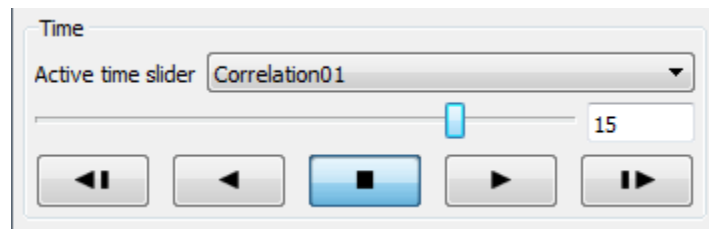
Fig. 1.285: Database correlation as the active time slider

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 1.2: 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 1.3: 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 1.4: 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 1.286.
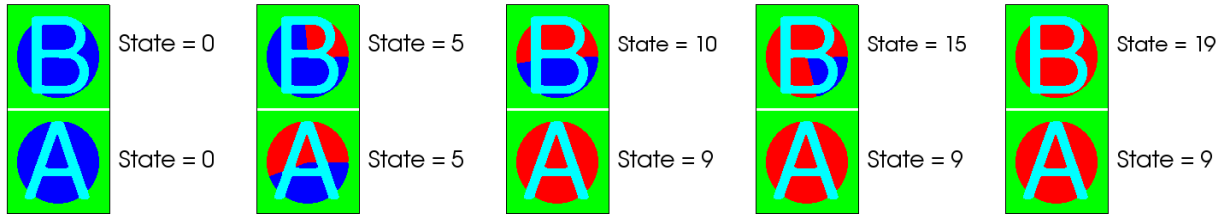
Fig. 1.286: Padded index database correlation of A and B (every 5th time state)

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 1.287 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.
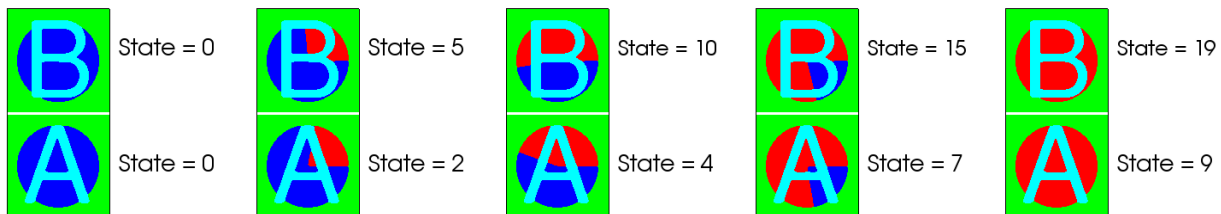
Fig. 1.287: Stretched index database correlation of A and B (every 5th time state)

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 1.288 shows the behavior of databases A and B when using a time database correlation.
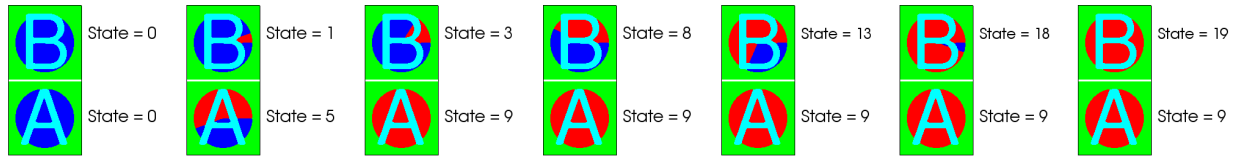
Fig. 1.288: Time database correlation of A and B (every 5th time state)

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 1.288 shows the behavior of databases A and B when using a cycle database correlation.
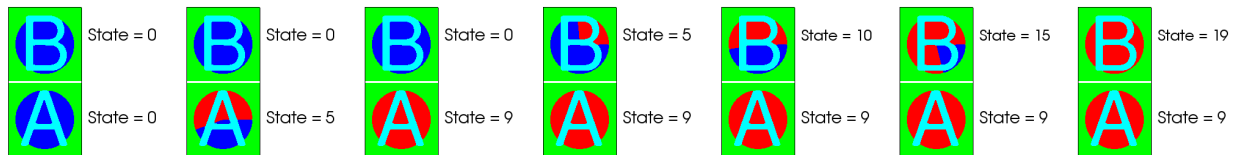
Fig. 1.289: Cycle database correlation of A and B (every 5th time state)

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 1.290, 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 1.291) 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 1.292, are: padded index, stretched index, time, and cycle.
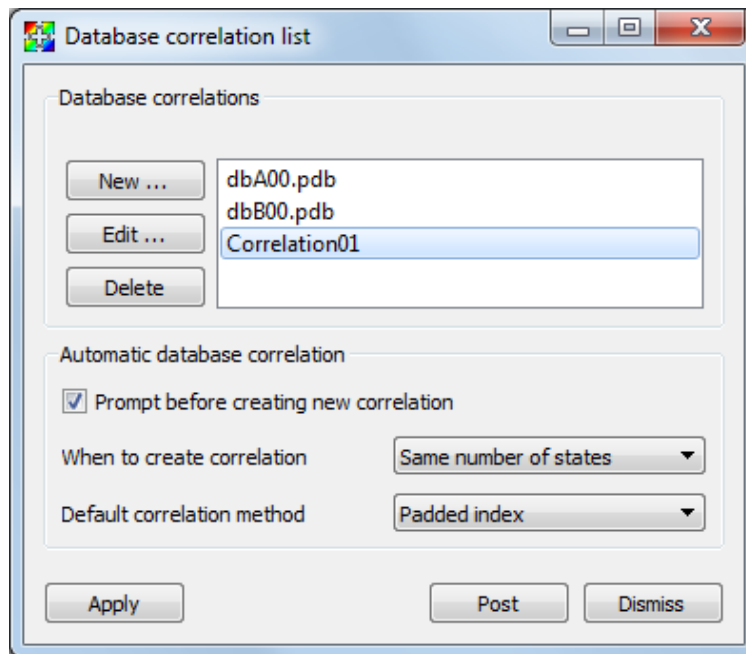
Fig. 1.290: Database Correlation Window
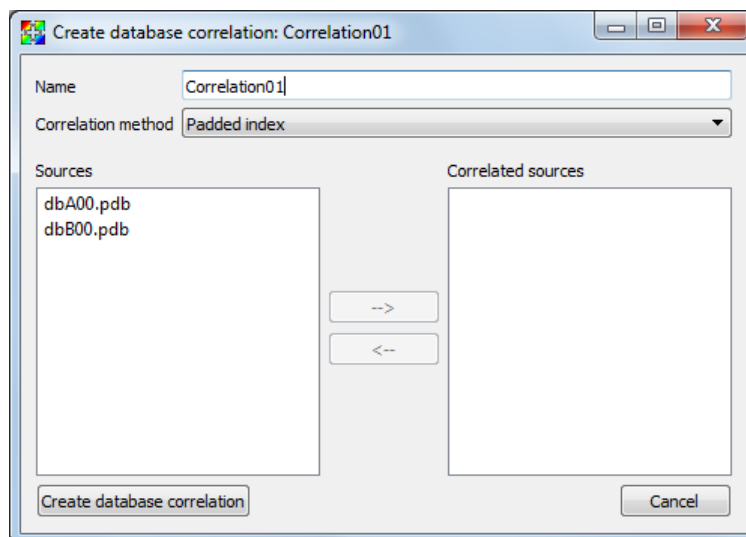
Fig. 1.291: Database Correlation Properties Window
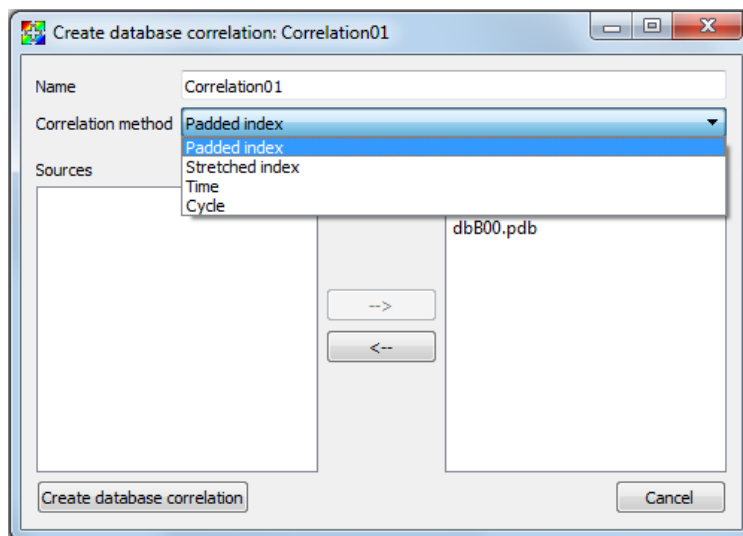
Fig. 1.292: Correlation methods

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 1.293). 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.
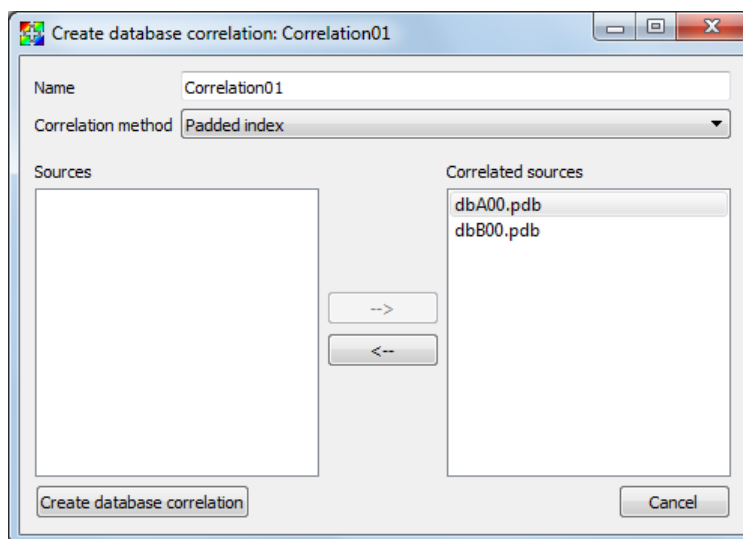
Fig. 1.293: Sources list and Correlated sources list

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
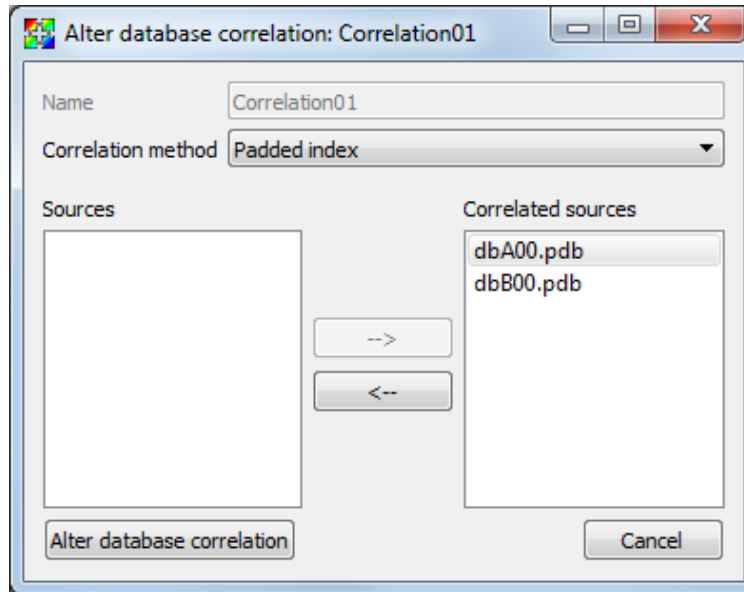
Fig. 1.294: Altering a database correlation

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 1.295). 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.
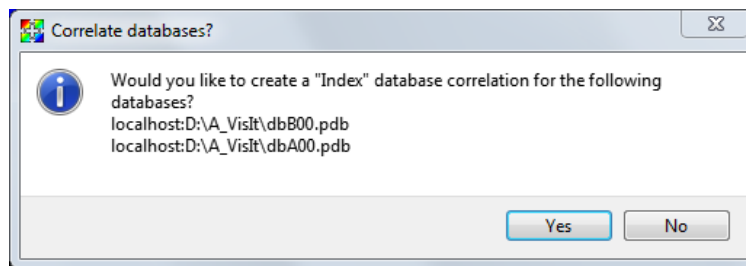
Fig. 1.295: Correlation question dialog

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.

1.12.3 Database comparison

Comparing the results of multiple related simulation databases is one of VisIt's main uses. You can put plots from multiple databases in the same window or you can put plots from each database in adjacent visualization windows, allowing you to compare plots visually. In addition to these visual comparison modes, VisIt supports more direct comparison of databases using database comparisons. Database comparison allows you to plot direct differences between two databases or between different time states in the same database (i.e. time derivatives). VisIt's expression language is used extensively to facilitate the database comparison.

The role of expressions

Database comparison is accomplished by creating new expressions that involve the contents of multiple databases using VisIt's expression language. 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 to another mesh The name "conn_cmfe" means *connectivity-based cross mesh field evaluation* (CMFE) and as the name implies, the expression takes fields from one mesh and maps the field onto another mesh by taking the cell or node-centered values on the donor mesh and mapping them onto the cells or nodes having the same indices in the new mesh. For

valid results, conn_cmfe requires the donor and target meshes be topologically congruent (e.g. size, connectivity, decomposition, etc.). If this is not the case, then one should use the position-based, pos_cmfe expression. The CMFE expressions can be used to map fields from a mesh in one database onto a mesh in the active database, which then allows you to create difference expressions involving the active database.

There is also a position-based CMFE (*pos_cmfe*), which will resample the field from one mesh onto another mesh by calculating the values of the field on the first mesh using the locations of the cells or nodes in a second mesh. More information on CMFE expressions are found in the *Cross-Mesh Field Evaluation (CMFE)* section of the *Exprssions* chapter.

Note that there is also a helpful *wizard*, the *Data Level Comparison Wizard*, that simplifies the process of defining comparison expressions. Here, we describe the *CMFE* expressions and demonstrate how to use them in comparisons.

Plotting the difference between two databases

Simulations are often run as parts of parameter studies, meaning the initial conditions are changed slightly to see what observable effects the changes produce. The ability to do direct numerical comparison of multiple simulation databases is required in order to observe the often miniscule differences between the fields in the two databases. VisIt provides the conn_cmfe expression, which allows you to map a field from one simulation database onto a mesh from another simulation database. Once the mapping has been done, you can then perform difference operations using an expression like this:

<mesh/ireg> - conn_cmfe(</usr/local/visit/data/dbB00.pdb:mesh/ireg>, mesh)

The expression above is a simple difference operation of database A minus database B. The assumption made by this expression is that database A is the active database and we're trying to map database B onto it so we can subtract it from database A's *mesh/ireg* variable. Note that the conn_cmfe expression takes two arguments. The first argument encodes the name of the file and the field that we're mapping onto a mesh from the active database, where the mesh name is given by the second argument. In this example, we're mapping database B's *mesh/ireg* field onto database A's mesh. Figure 1.296 shows a picture that illustrates the database differencing operation.
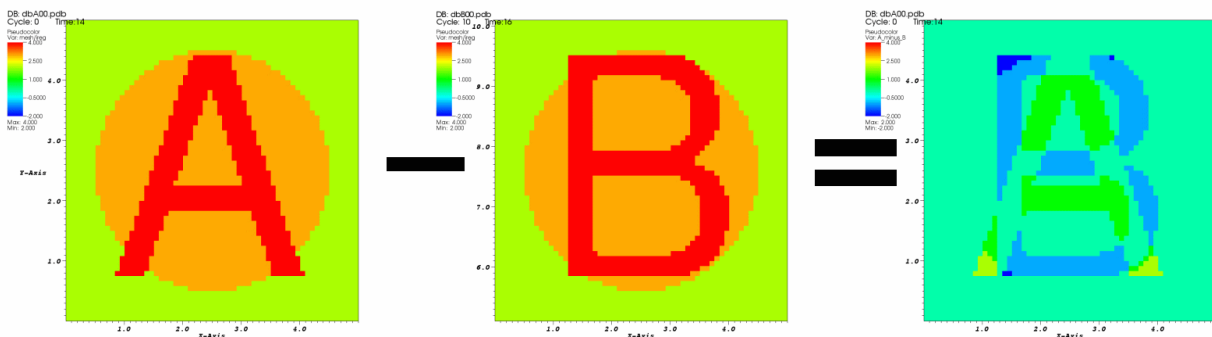
Fig. 1.296: Database B subtracted from database A

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 from the value of *mesh/ireg* at the current time state. The interesting feature about the above expression is its use of the expression language's *[]* operator for specifying a database time state. The expression uses *"[0]i"* , which means use time state zero because the *"i"* suffix indicates that the number inside of the square brackets is to be used as an absolute time state index. As you change the time slider, the values for the current time state 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 you to create expressions that compare the current time state to a fixed time state. You can, of course, substitute different time state indices into the conn_cmfe expression so you don't have to always compare the current time state to time state zero.
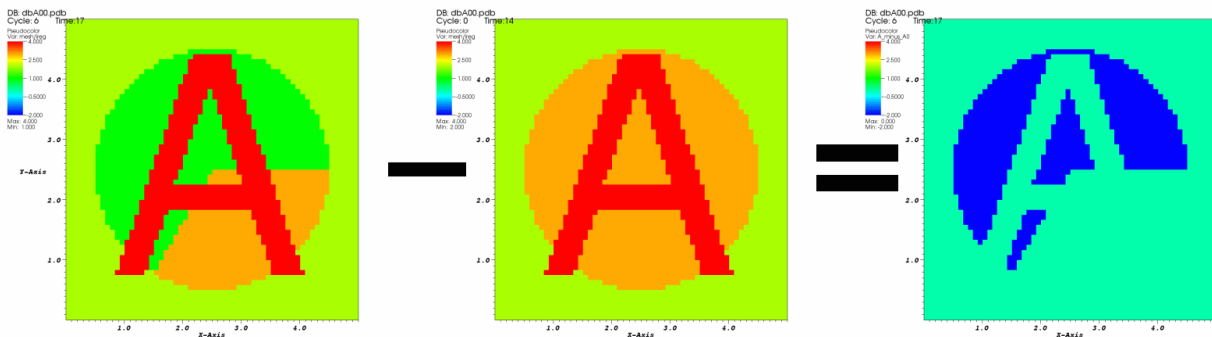
Fig. 1.297: Time state 6 minus time state 0

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 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 1.298 shows an example plot of a time derivative.
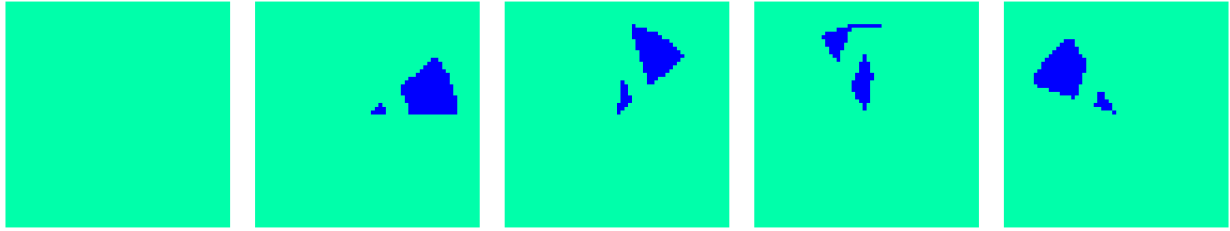
Fig. 1.298: Plot of a variable and its time derivative plot

1.12.4 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.
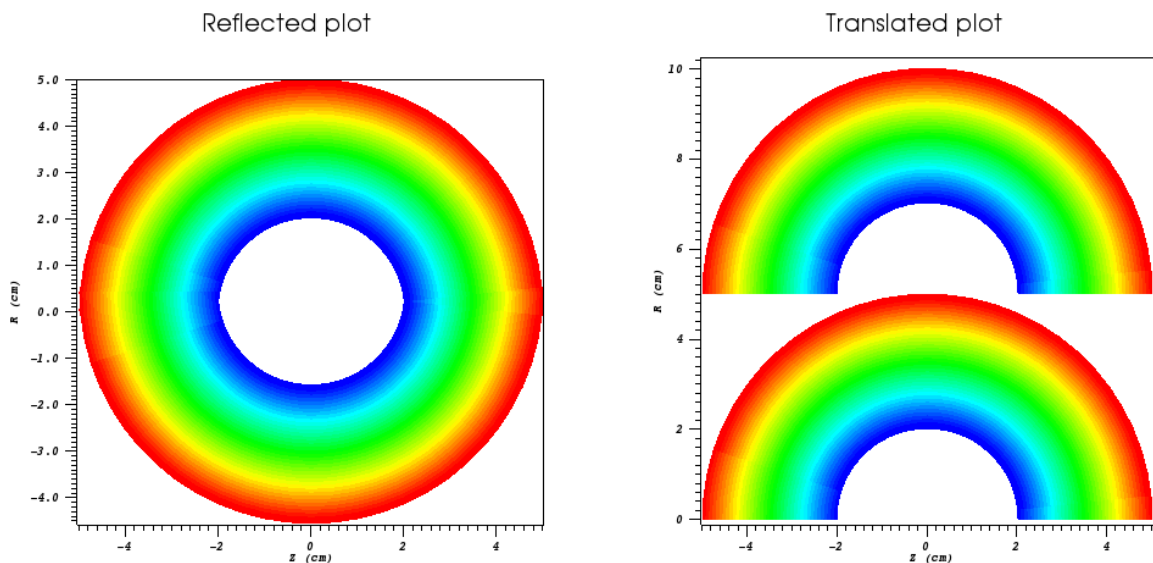
Fig. 1.299: Plots side by side using the Reflect or Transform operator

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 1.300.
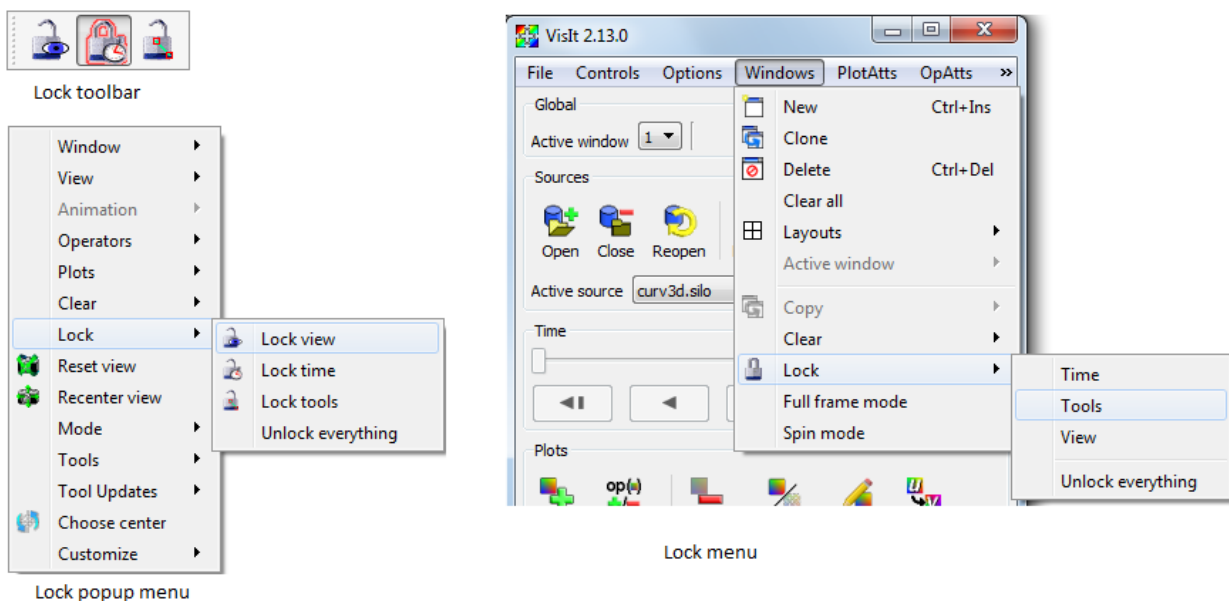
Fig. 1.300: Mechanisms for locking windows

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 1.301.
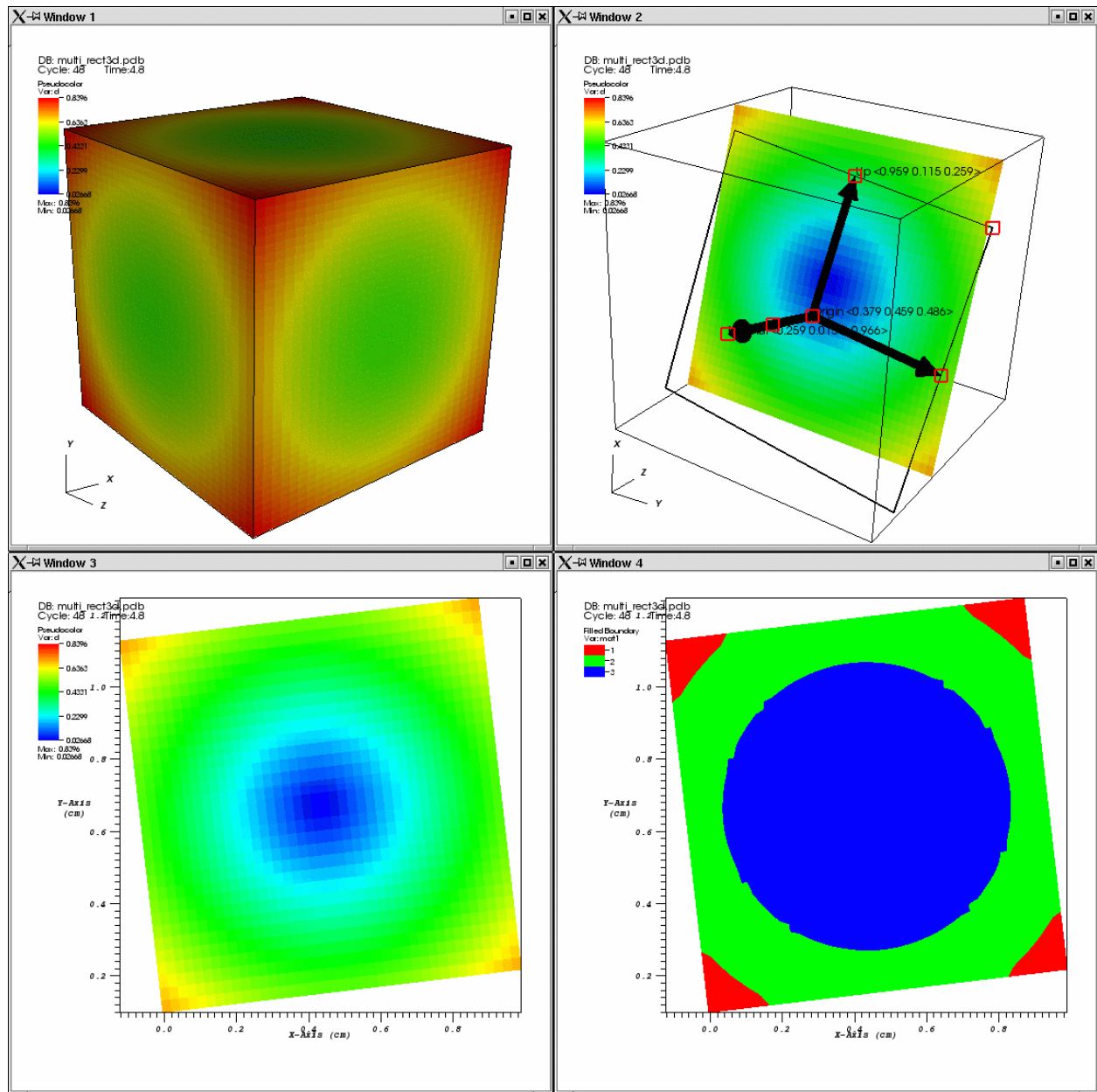
Fig. 1.301: Multiple visualization windows with locked tools

1.13 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.

1.13.1 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 1.302). Once you are 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.
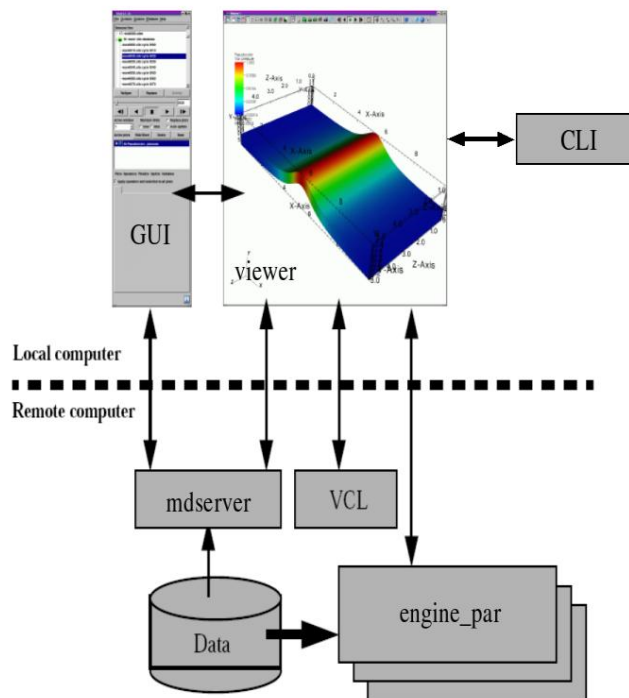
Fig. 1.302: VisIt's Architecture

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 1.303). 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.
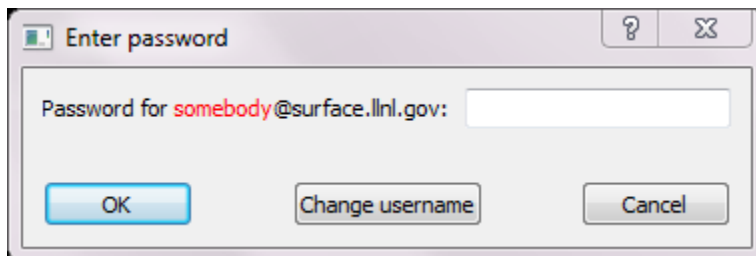
Fig. 1.303: Password Window

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 1.304). Enter the correct username in the text field provided and click **Confirm username**. Proceed with entering the password in the **Password Window**.
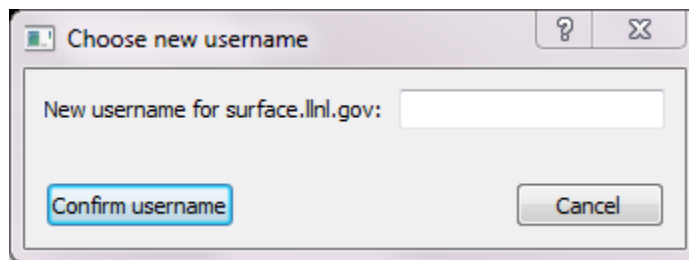
Fig. 1.304: Change Username Window

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 OSX. 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.
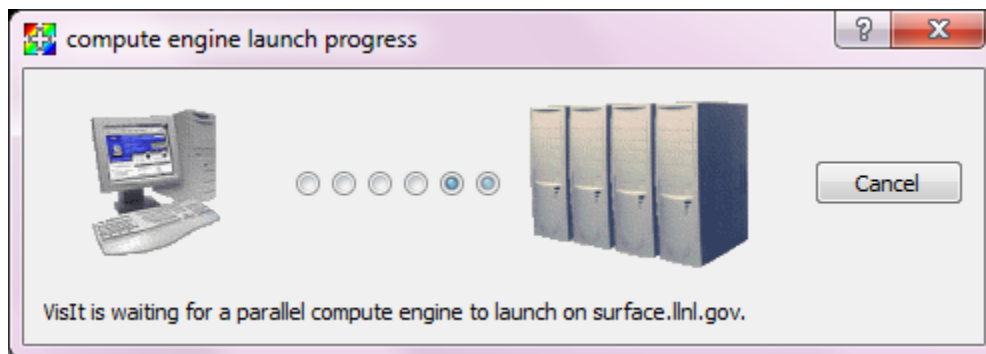
Fig. 1.305: Launch Progress Window

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 1.305. 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 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.

1.13.2 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: remote username, number of processors, 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 Profile Window**, shown in Figure 1.306, that you can use to manage your host profiles. You can open the **Host Profile 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
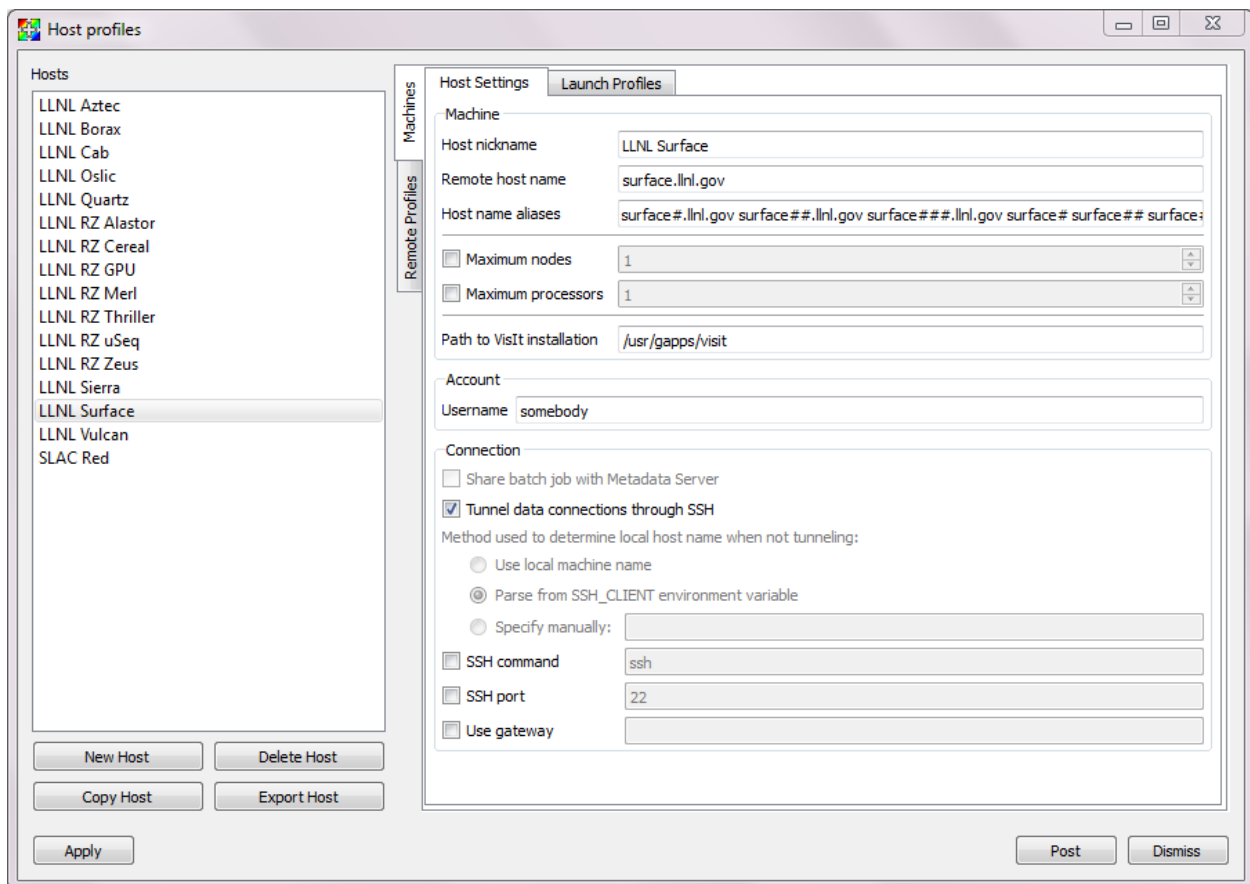
Fig. 1.306: Host Profiles Window

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 nickname, full host name, aliases, 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: Using the *Install Profiles Configuration Window* window, or using the *Remote Profiles Tab* in the **Host Profiles Window**.

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.

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 checkbox. The exported host will be saved to your user VisIt directory (~/.visit/hosts on Linux).

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)*.

Fig. 1.307: Host Settings tab

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: **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 computes. 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.

VisIt's ports

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 1.308) 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.
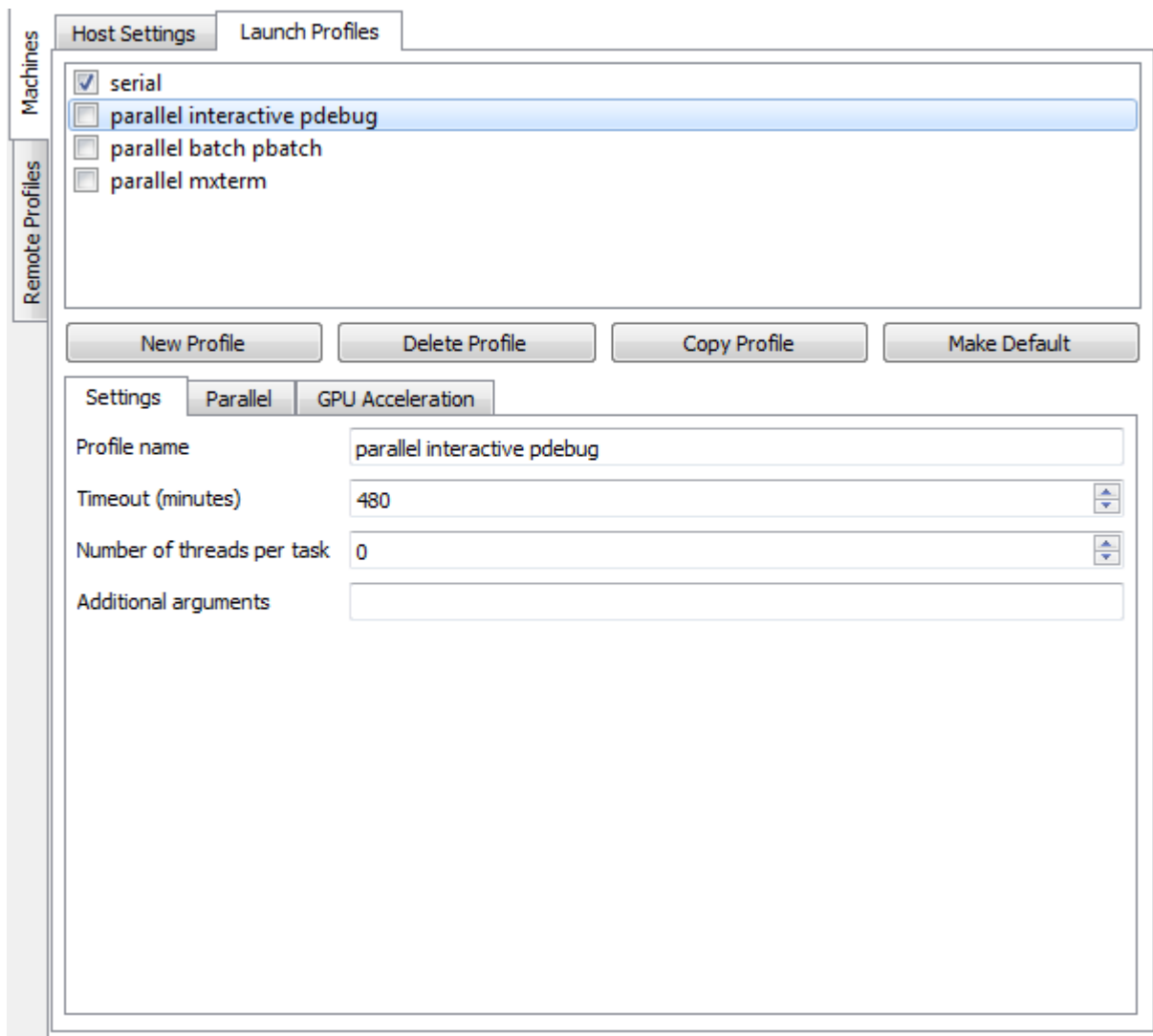
Fig. 1.308: Launch Profiles tab

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 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 is checked in the list. To activate a different launch profile, select it from the list and click the **Make Default** 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
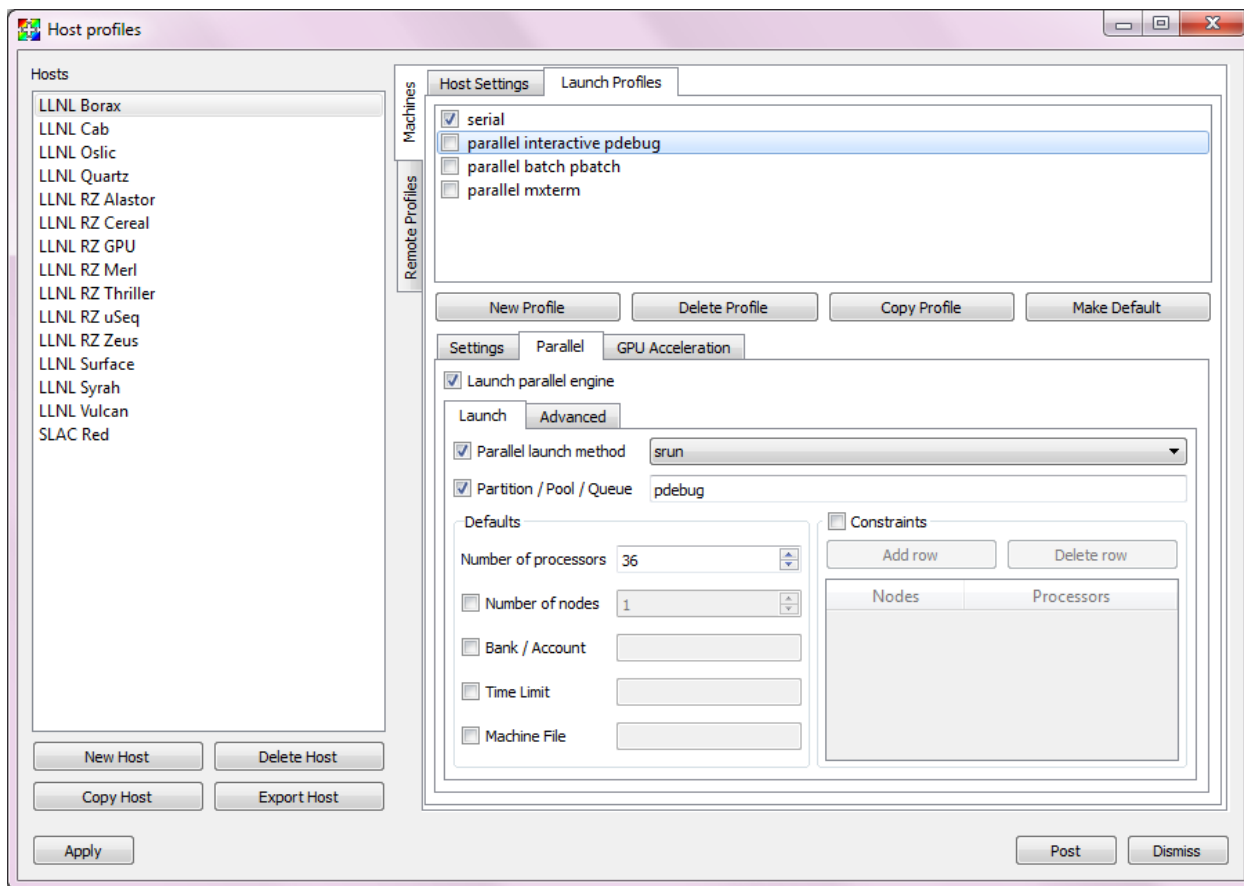
Fig. 1.309: Parallel options

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 computer is configured. Some computers have multiple launch programs depending on which part of the parallel machine you want to use. Figure 1.310 shows some common parallel-launch options that VisIt currently supports.
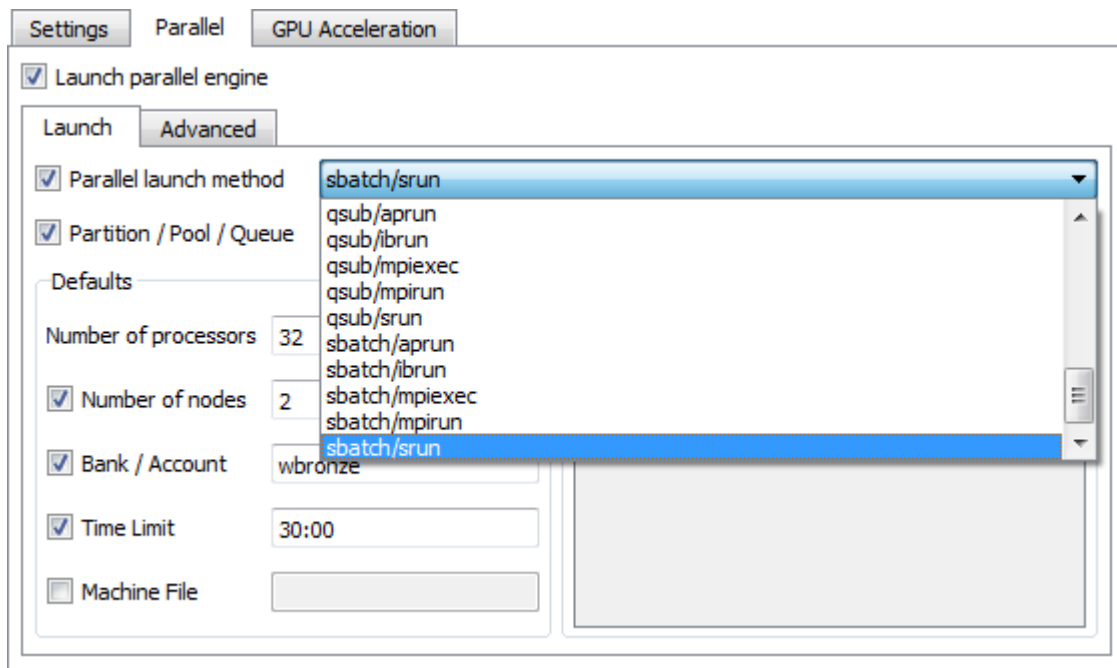
Fig. 1.310: Parallel launch method options

Setting the partition/pool

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.

Fig. 1.311: Parallel launch constraints

Fig. 1.312: Advanced options tab

Advanced host profile options

The **Advanced** tab (see Figure 1.312) 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.

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.

Install Profiles Configuration Window

The **Host Profiles And Configuration Setup** window is available from the **Options** dropdown. It will list all the host profiles currently available for 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 1.313)
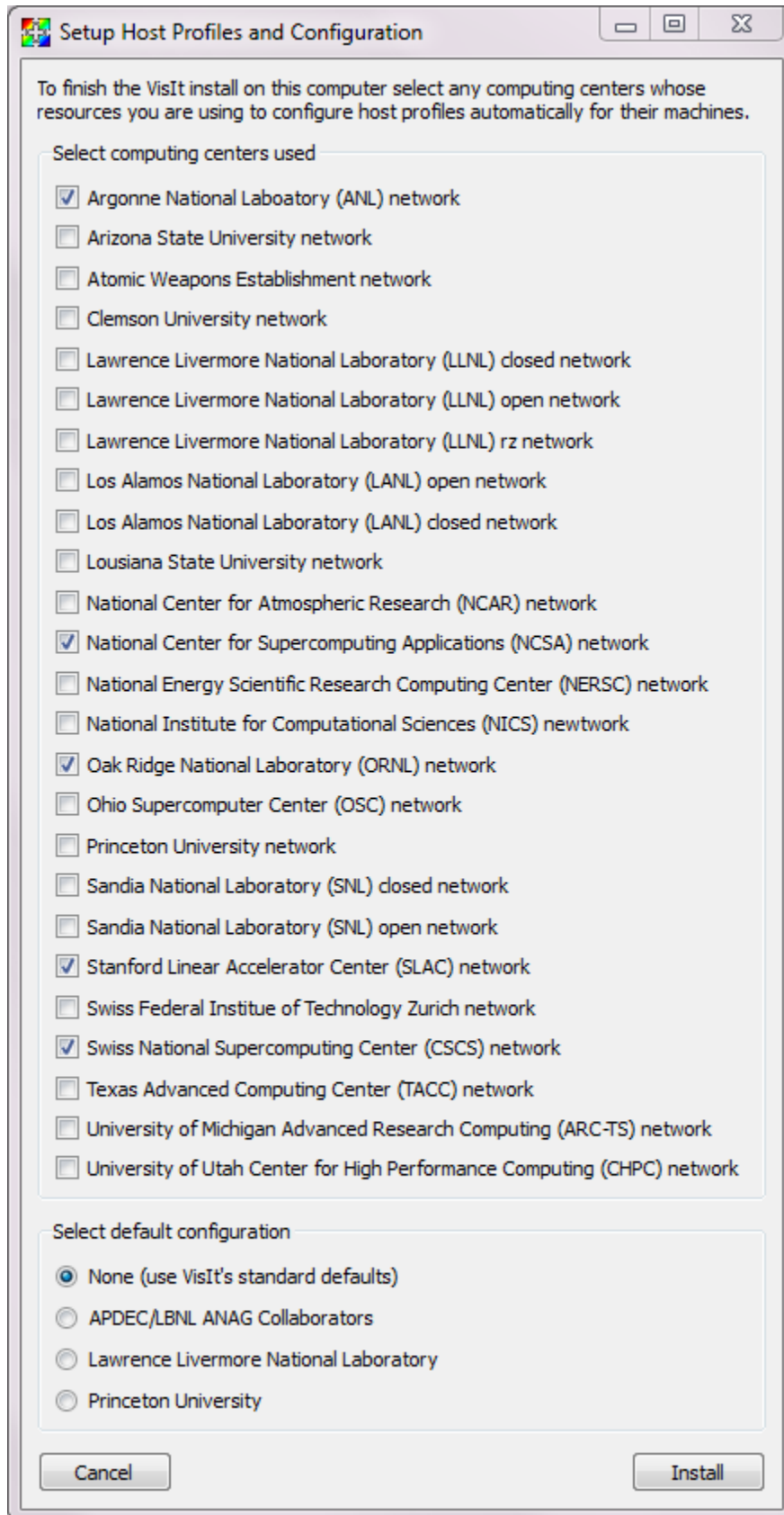
Fig. 1.313: Host Profile Configuration Window

Remote Profiles Tab

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 1.314)
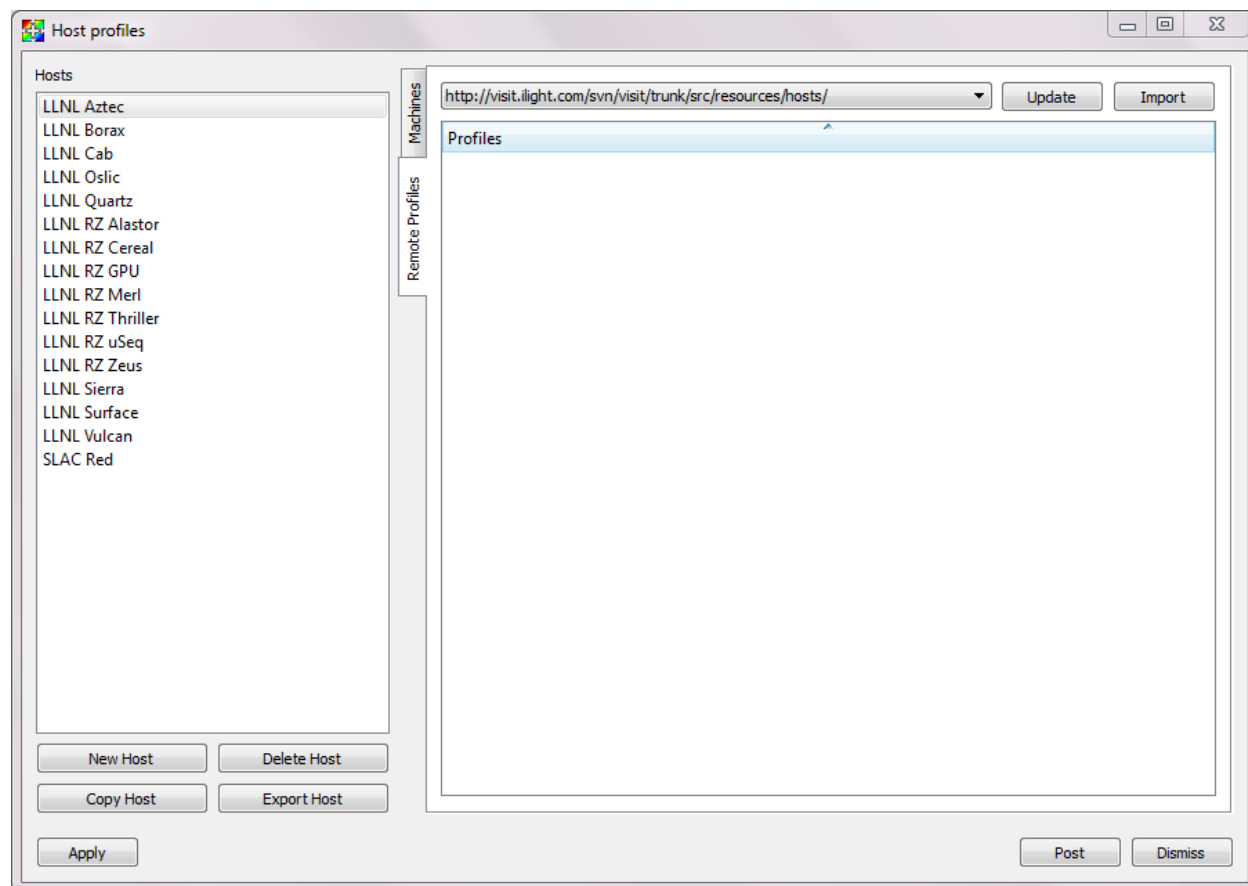
Fig. 1.314: Remote Profiles tab

If you click the **Update** button, the list of host profiles available from the remote location will be displayed. (Figure 1.315)

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 1.316) 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 profile for future sessions.

Engine Options Window

You can use **Engine Options Window**, shown in (Figure 1.317), 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 when VisIt launches a 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.
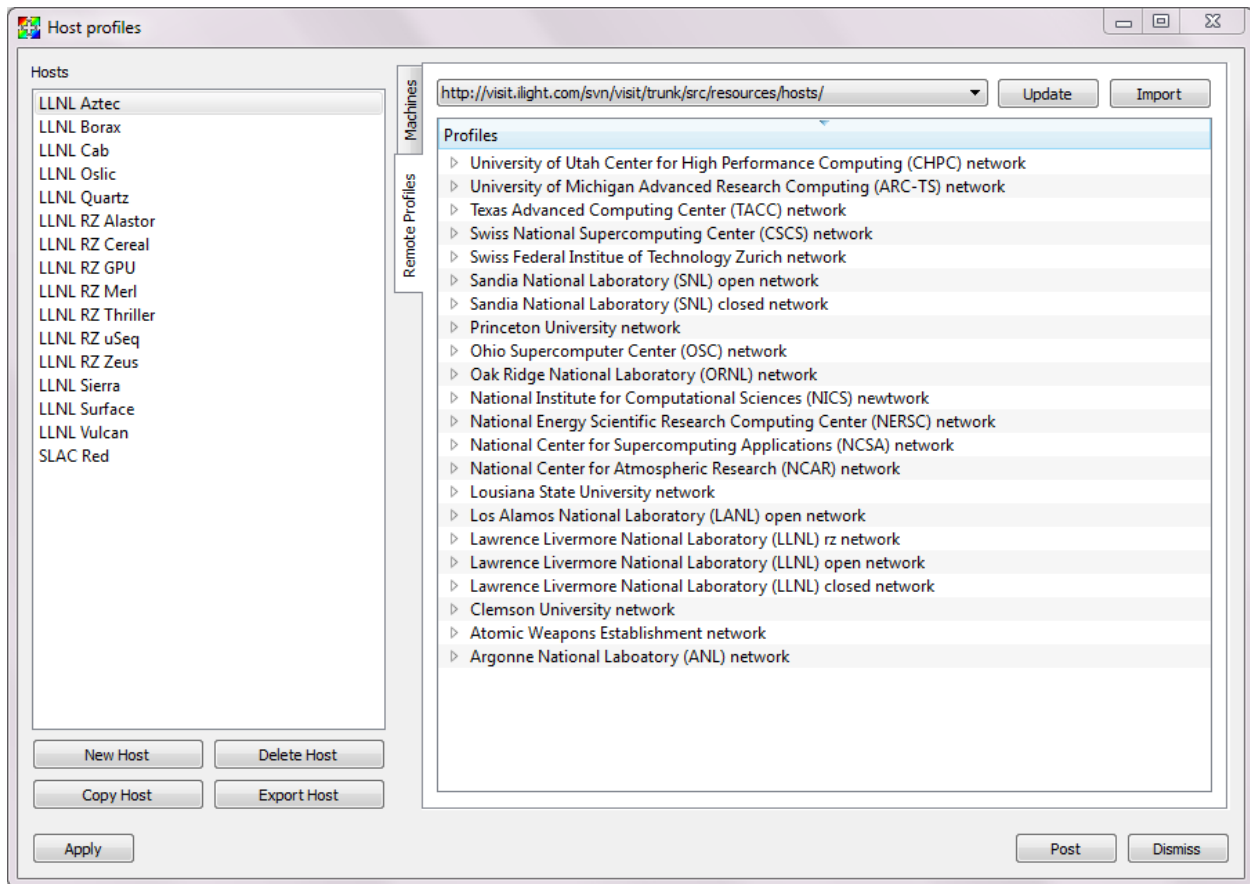
Fig. 1.315: Remote Profiles tab with Updated content
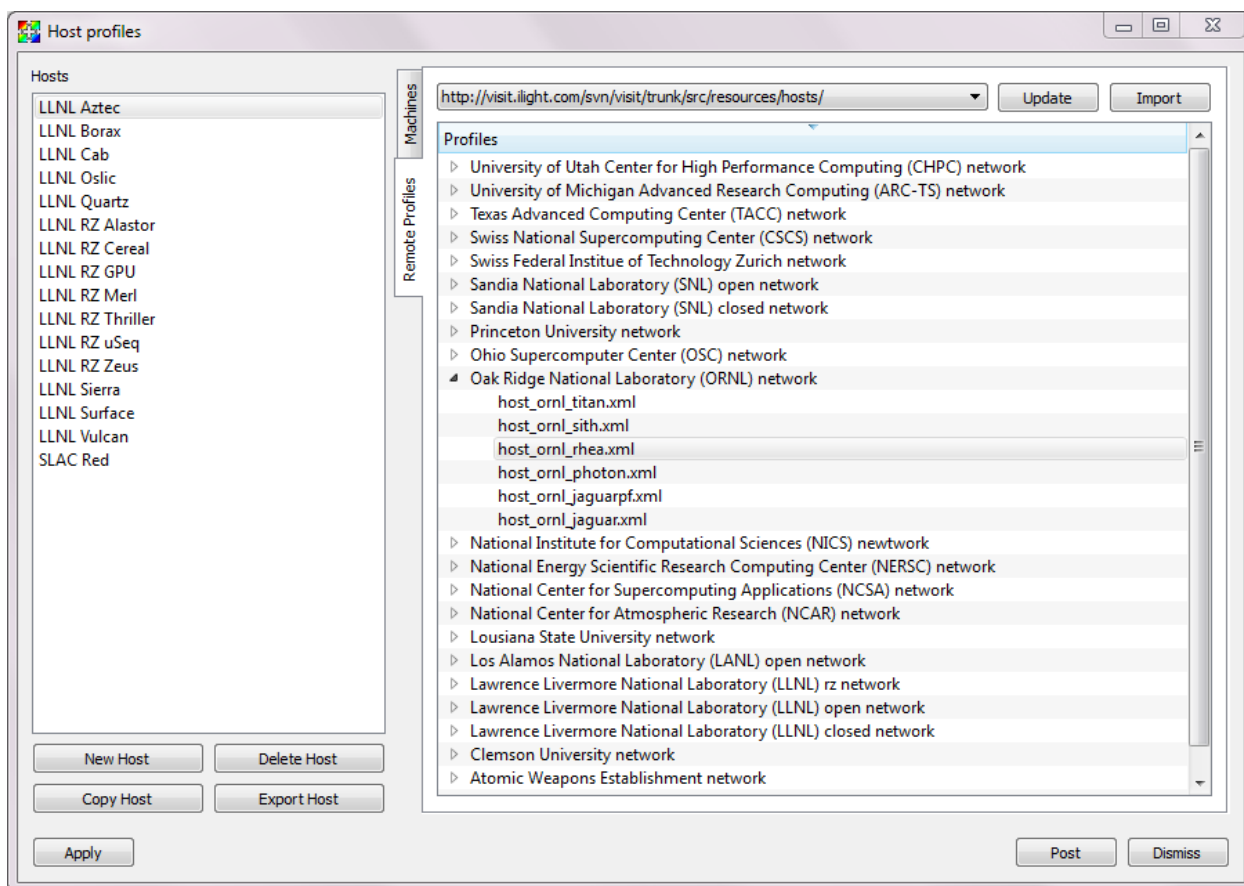
Fig. 1.316: Remote Profiles tab with host selected for import

The **Engine 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.
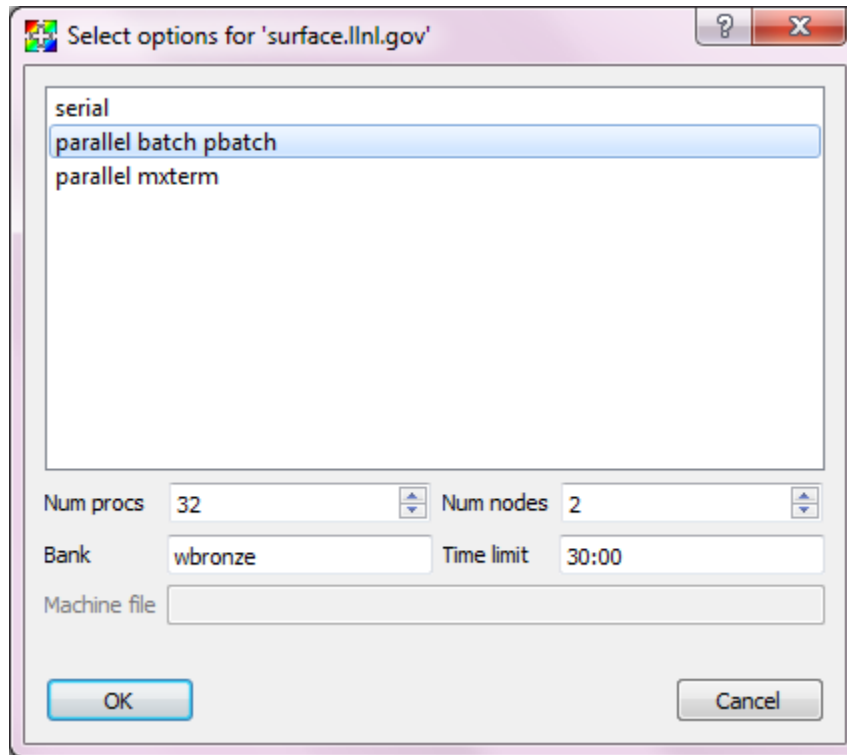
Fig. 1.317: Engine options window

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 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 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. The **Machine file** text field is only enabled when the **Default Machine File** check box is enabled in the **Host Profile Window's** parallel options.

1.14 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.

1.14.1 Compute Engines Window

You can open the **Compute Engines Window**, shown in Figure 1.318, 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.

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.
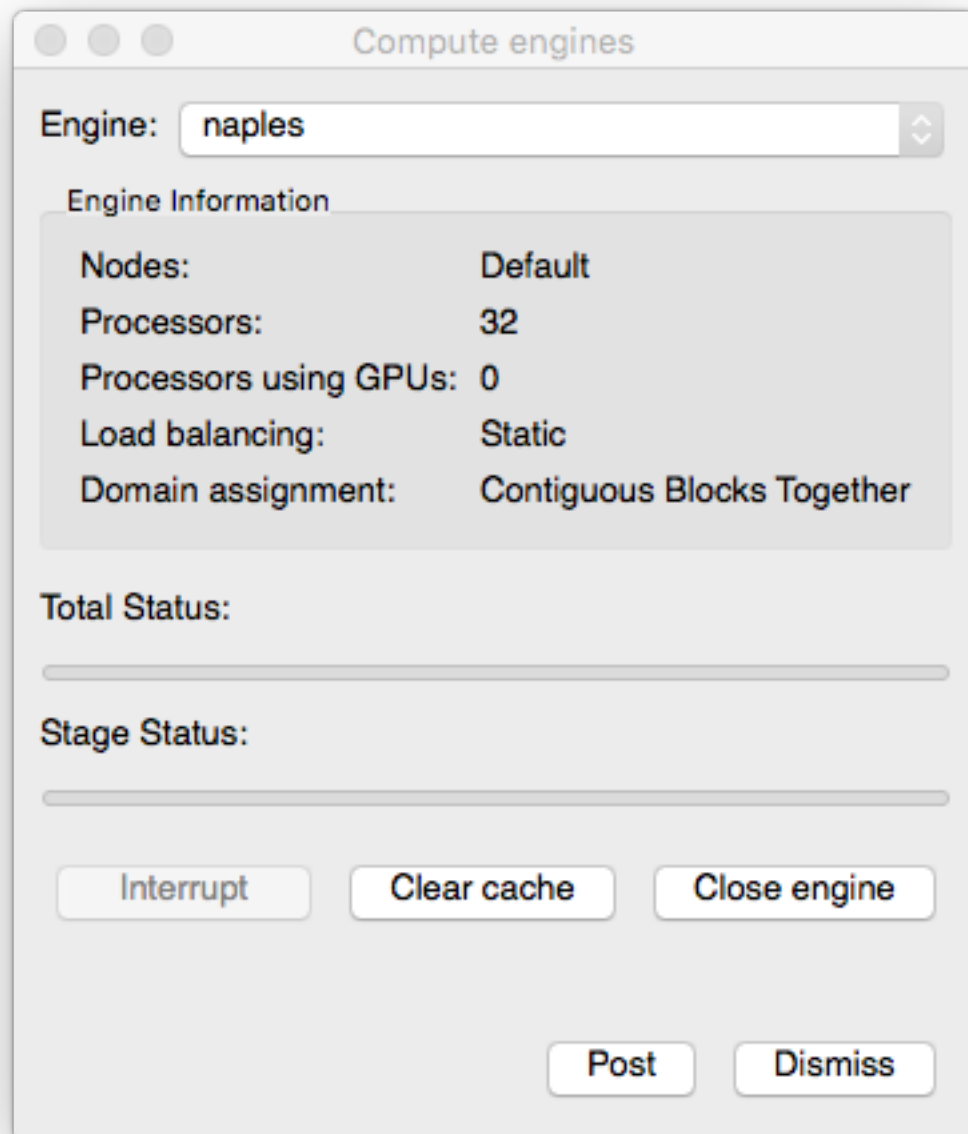
Fig. 1.318: Compute Engines Window

Closing a compute engine
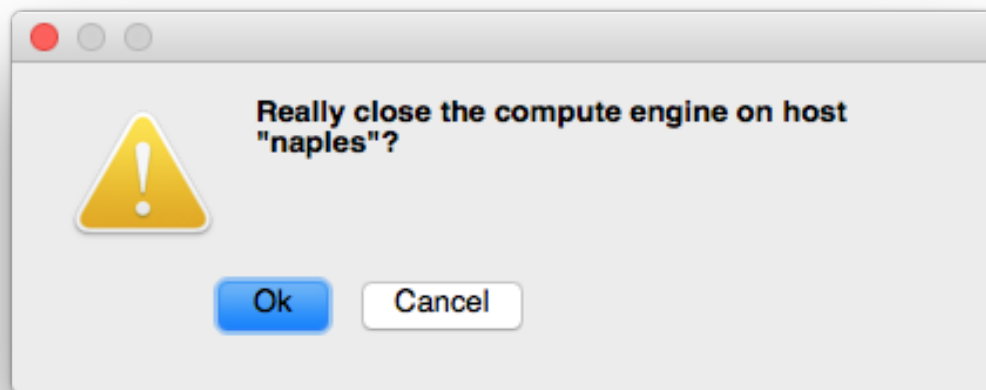
Fig. 1.319: Close compute engine confirmation dialog

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**.

1.14.2 Simulation Window

You can open the **Simulation Window**, shown in Figure 1.320, 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 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.
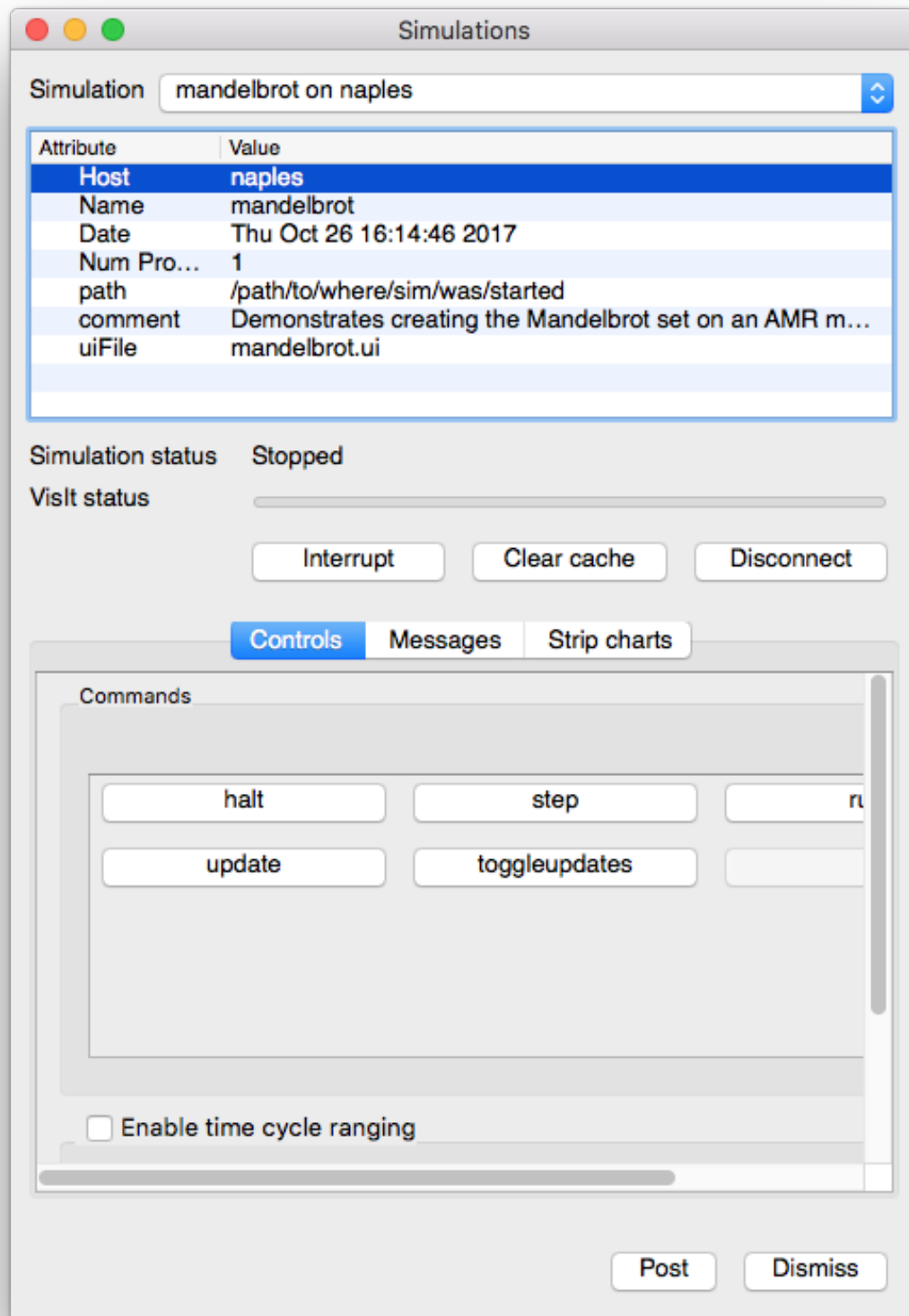
Fig. 1.320: Simulation Window

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.

1.15 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.

1.15.1 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.

1.15.2 Setting Default Values
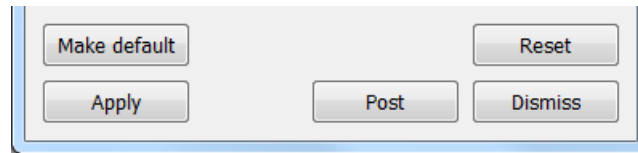
Fig. 1.321: The make default button

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 1.321.

1.15.3 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.

1.15.4 Appearance Window

The **Appearance** window is responsible for setting preferences for the appearance of the GUI windows. The **Appearance** window shown in Figure 1.322 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.
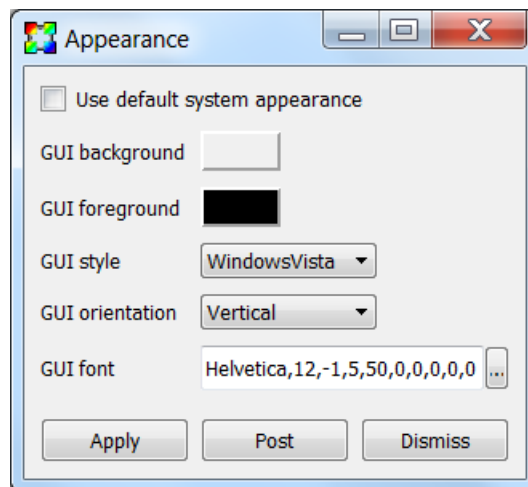
Fig. 1.322: The appearance window

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.

1.15.5 Plugin Manager Window

The **Plugin Manager** window , shown in Figure 1.323, 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.

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.

1.15.6 Rendering Options Window

The **Rendering options** window (shown in Figure 1.324) 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.
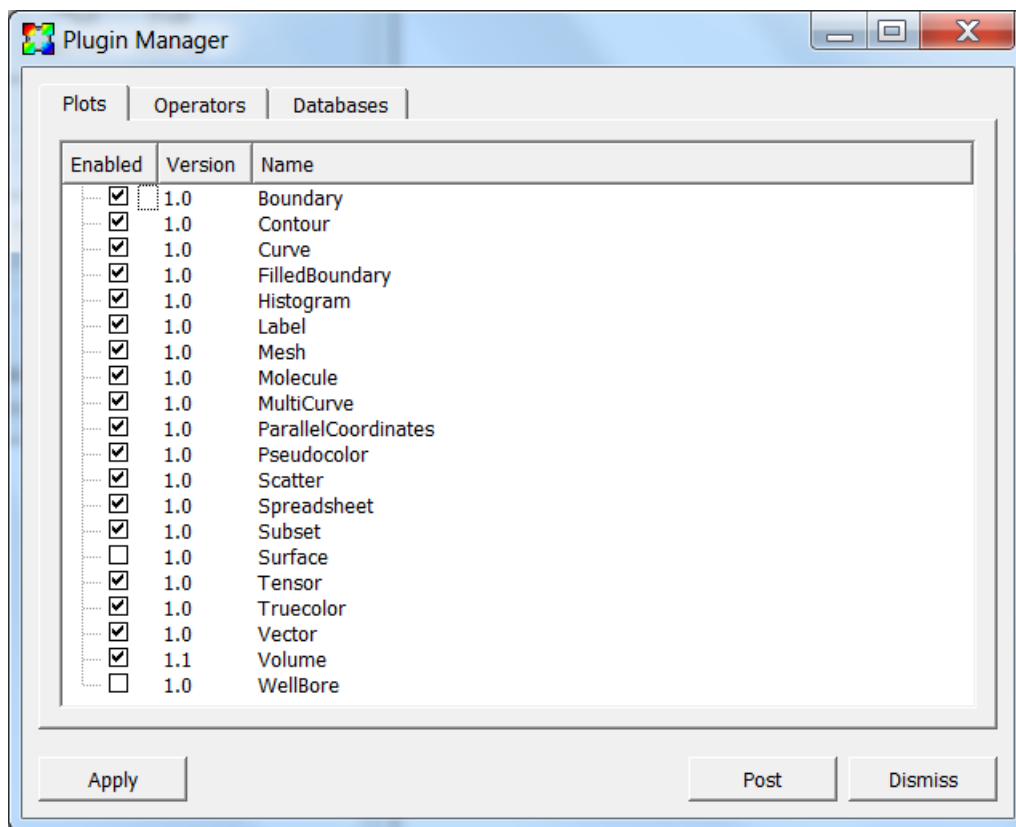
Fig. 1.323: The plugin manager window
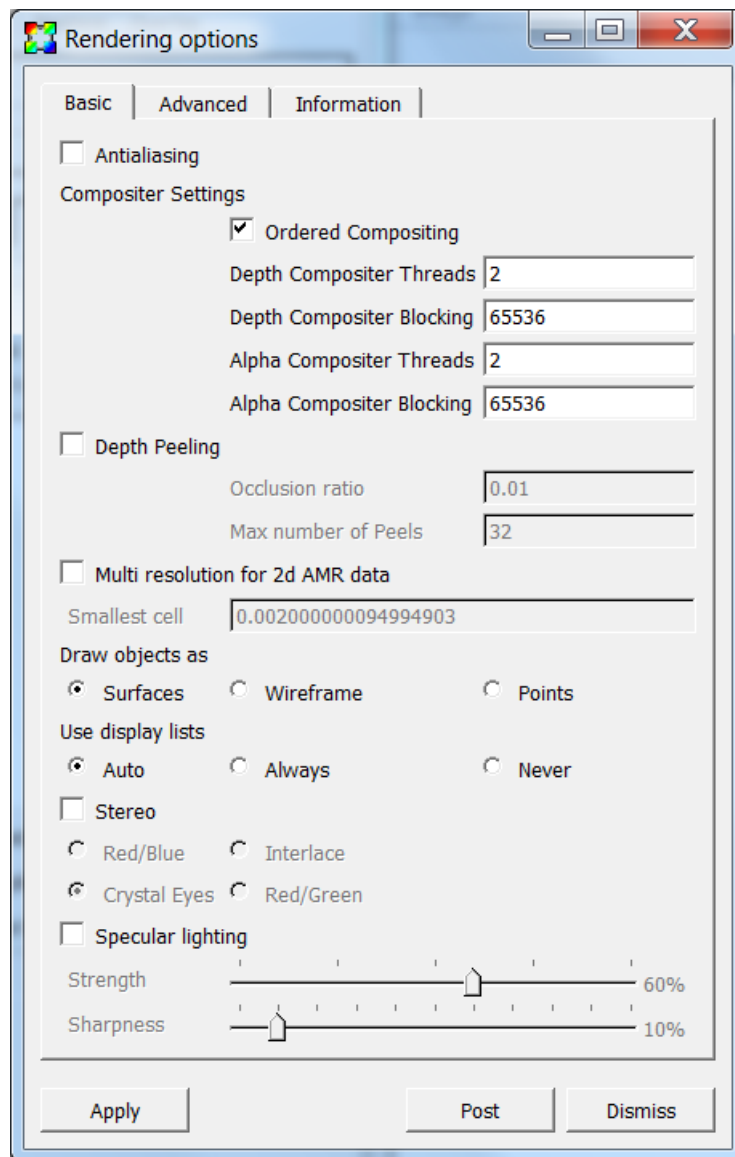
Fig. 1.324: The basic rendering options

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 1.325). 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.
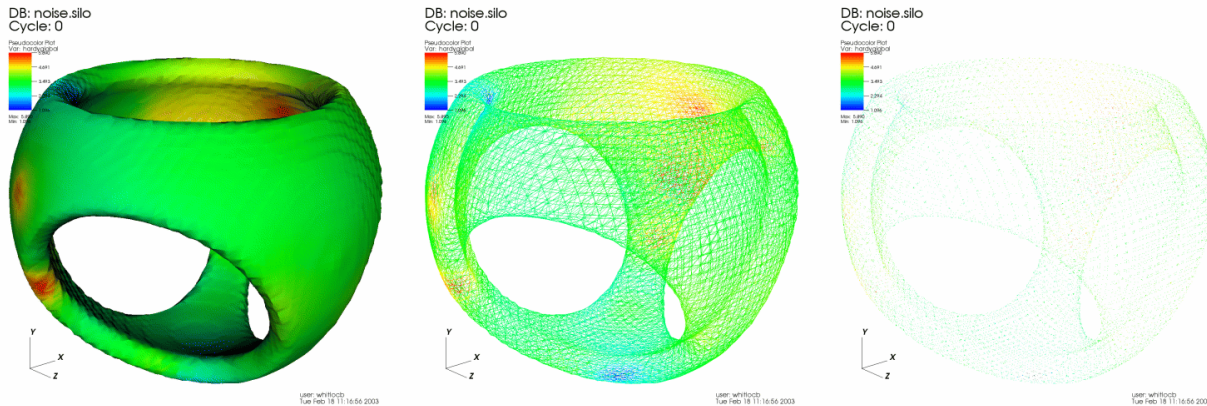
Fig. 1.325: The different surface representations

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 1.326) 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.
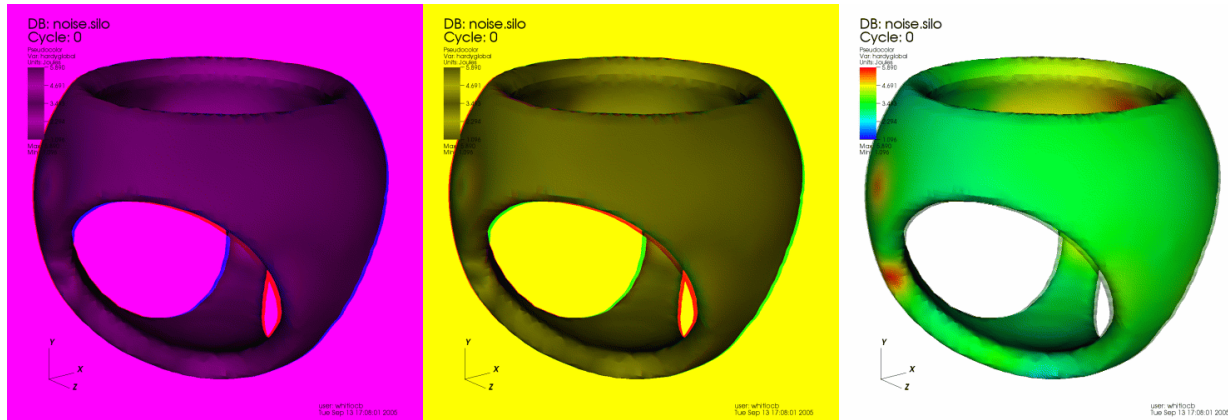
Fig. 1.326: Some various stereo image types

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.
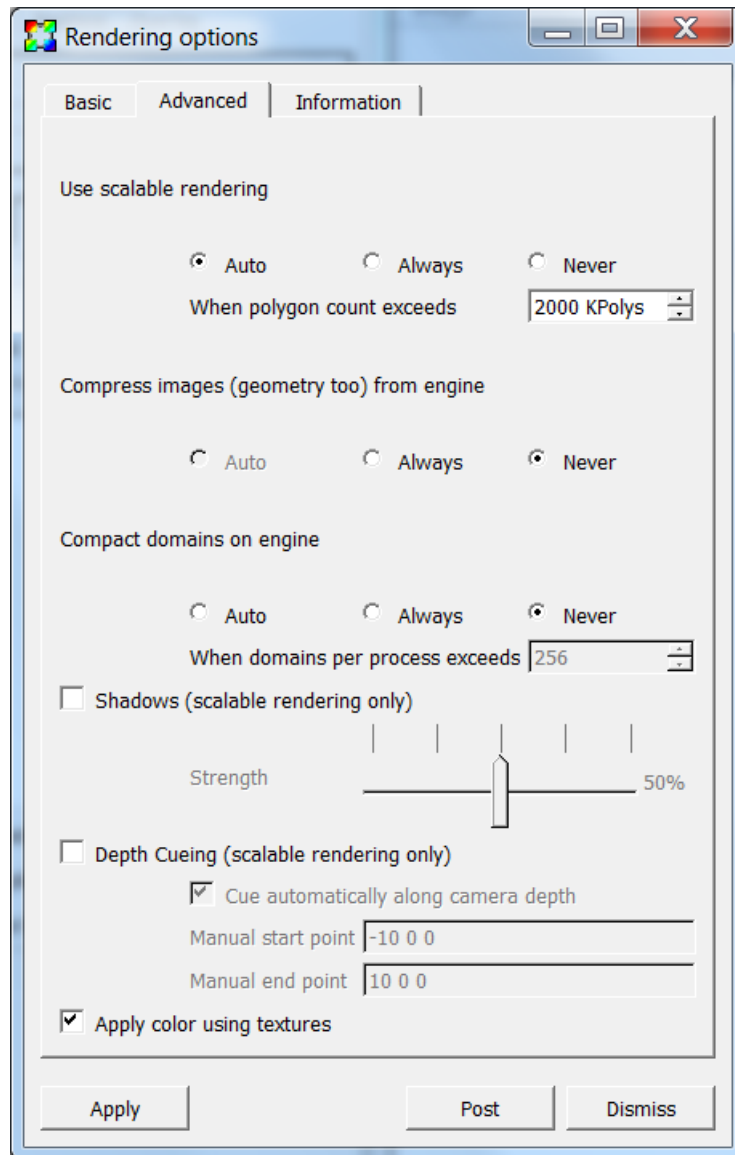
Fig. 1.327: The advanced rendering options

Rendering Information
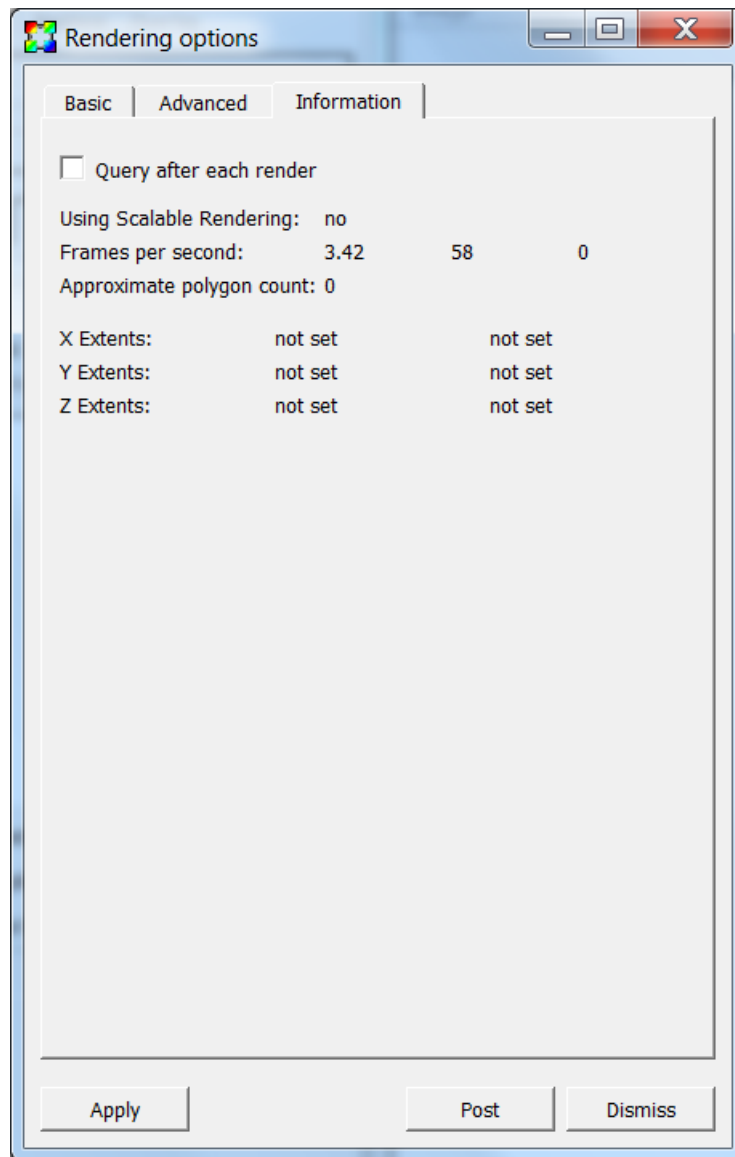
Fig. 1.328: The rendering information

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. .

1.15.7 Preferences Window

The **Preferences** window, shown in Figure 1.329, contains controls that allow setting global options that influence VisIt's behavior. The top portion of the window contains a collection of miscellaneous options. This is followed by collections of options that are grouped by functionality. The groups are contained in the **Floating point precision**, **Databases**, **Session files** and **File panel properties** areas.

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.

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.
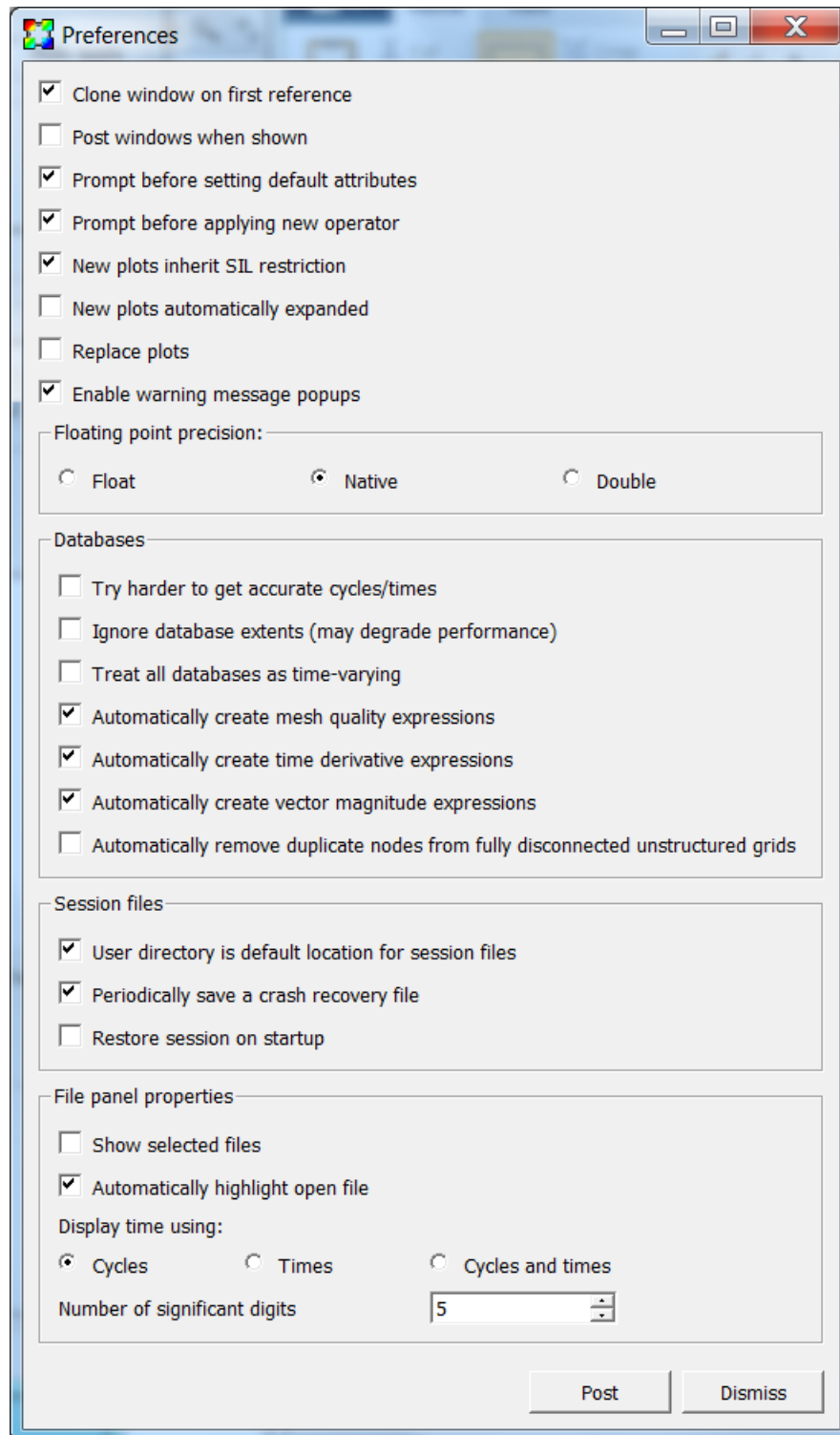
Fig. 1.329: The preferences window

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.

1.16 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.

1.16.1 About VisIt

VisIt provides a **Splash screen** (Figure 1.330) 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.

1.16.2 Help Window

VisIt's **Help Window**, shown in Figure 1.331, 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**.

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.

Fig. 1.330: Splash screen
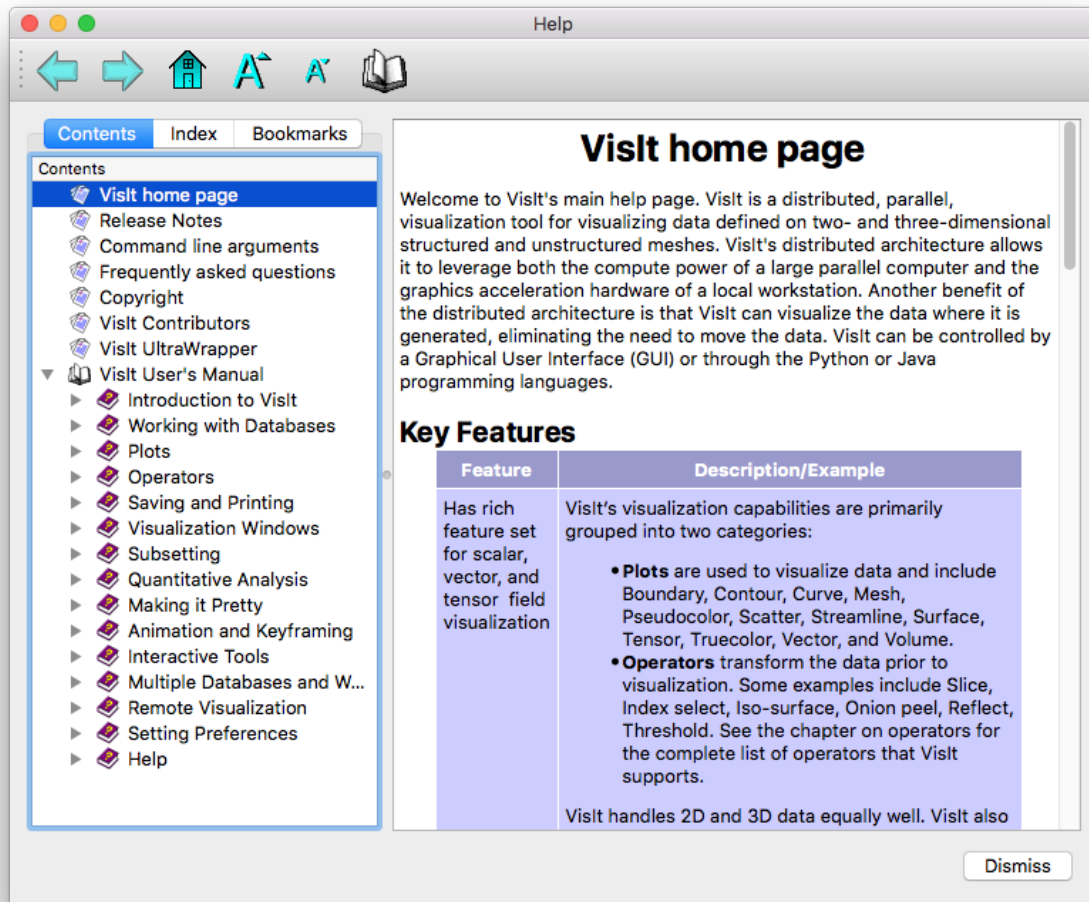
Fig. 1.331: Help window

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 1.332, 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. 1.332: 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.

1.17 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. Note: This
                        argument only takes effect with -cli or -movie.
    -sessionfile <filename>  Open the specified session file at startup
                        Note that this argument only takes effect with
```

(continues on next page)

```
                           -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
                           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.llnl.gov
    email:               visit-users@ornl.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.
--------------------------------------------------------------------------
    -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.

    See the VisIt wiki for more information:

      http://visitusers.org/index.php?title=Parallel_Hardware_Acceleration
```

```
    -------------------------------------------------------------------------
    -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
                           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
                          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
                        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
-----------------------------------------------------------------------
    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.
    -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.

   -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.
```

1.18 Building

In this chapter, we will discuss how to build visit. 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 moving then moving to more complex use cases.

1.18.1 Basic Usage

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_0b
```

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_0b --makeflags -j4
```

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_0b --thirdparty-path /usr/gapps/visit/third_party
```

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_0b --hdf5 --silo
```

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_0b --optional
```

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 3.0.0b release of VisIt.

```
./build_visit3_0_0b --optional --tarball visit3.0.0b.tar.gz
```

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.

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.

```
visit3.0.0b/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 visit3.0.0b/build
make package
```

if you built using a tar file.

1.18.2 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.

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_0b --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_0b --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_0b --mpich --osmesa
```

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_0b --mesagl
```

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.

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_0b --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/3.0.0b/third_party/Imaging-1.1.7.
↪tar.gz
wget http://portal.nersc.gov/project/visit/releases/3.0.0b/third_party/setuptools-28.
↪0.0.tar.gz
wget http://portal.nersc.gov/project/visit/releases/3.0.0b/third_party/Cython-0.25.2.
↪tar.gz
wget http://portal.nersc.gov/project/visit/releases/3.0.0b/third_party/numpy-1.14.1.
↪zip
wget http://portal.nersc.gov/project/visit/releases/3.0.0b/third_party/pyparsing-1.5.
↪2.tar.gz
wget http://portal.nersc.gov/project/visit/releases/3.0.0b/third_party/requests-2.5.1.
↪tar.gz
wget http://portal.nersc.gov/project/visit/releases/3.0.0b/third_party/seedme-python-
↪client-v1.2.4.zip
```

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_0b --no-thirdparty --no-visit --python
```

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_0b` 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.

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_0b` then the log file will be named `build_visit3_0_0b_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.

1.18.3 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.

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.

```
./build_visit3_0_0b --required --optional --no-visit \
--thirdparty-path /usr/gapps/visit/thirdparty_shared/3.0.0b --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/3.0.0b)
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 visit3.0.0b.tar.gz
cp kickit.cmake visit3.0.0b/src/config-site
cd visit3.0.0b
mkdir build
cd build
/usr/gapps/visit/thirdparty_shared/3.0.0b/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
```

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/mvapich2-2.2-gcc-4.9.3/bin/mpicc \
    PAR_COMPILER_CXX=/usr/tce/packages/mvapich2/mvapich2-2.2-gcc-4.9.3/bin/mpicxx \
    PAR_INCLUDE=-I/usr/tce/packages/mvapich2/mvapich2-2.2-gcc-4.9.3/include \
    PAR_LIBS=-lmpl \
./build_visit3_0_0b --required --optional --mesagl --uintah --parallel \
--no-visit --thirdparty-path /usr/workspace/wsa/visit/visit/thirdparty_shared/3.0.0b/
→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/3.0.0b)
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/mvapich2-2.2-
→gcc-4.9.3/include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_C_FLAGS   -I/usr/tce/packages/mvapich2/mvapich2-2.2-
→gcc-4.9.3/include TYPE STRING)
VISIT_OPTION_DEFAULT(VISIT_MPI_LD_FLAGS  "-L/usr/tce/packages/mvapich2/mvapich2-2.2-
→gcc-4.9.3/lib -Wl,-rpath=/usr/tce/packages/mvapich2/mvapich2-2.2-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/mvapich2-2.2-
→gcc-4.9.3/lib")
```

VisIt was then manually built with the following steps.

```
tar zxf visit3.0.0b.tar.gz
cp kickit.cmake visit3.0.0b/src/config-site
cd visit3.0.0b
mkdir build
cd build
/usr/workspace/wsa/visit/visit/thirdparty_shared/3.0.0b/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
```

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_0b \
 --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 --openssl --p7zip --pidx \
 --silo --szip --vtkm --vtkh --xdmf --zlib \
 --mesagl --uintah --parallel \
 --thirdparty-path /usr/workspace/wsa/visit/visit/thirdparty_shared/3.0.0b/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/3.0.0b/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 visit3.0.0b.tar.gz
cp kickit.cmake visit3.0.0b/src/config-site
```

```
cd visit3.0.0b
mkdir build
cd build
/usr/workspace/wsa/visit/visit/thirdparty_shared/3.0.0b/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
```

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
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.
##
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
```

1.19 Building on Windows

In this chapter, we will discuss how to build visit on Windows.

1.19.1 Prerequisites

VisIt's Source Code

For a released version

If you want to build a released version of VisIt, you can download a windows installer that contains all that is necessary from the source code downloads page. Look for the *VisIt Windows sources* link for the particular version you want.

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 3.8 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

   (if you want to use our pre-built thirdparty libraries).

3. NSIS *Optional*

   For creating an installer for VisIt. NSIS 2 is known to work. NSIS 3 hasn't been tested.

4. 7zip *Optional*

   Used to untar testdata files.

5. 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.

1.19.2 Configuring With CMake GUI

Run cmake-gui.exe, which will display this window. Figure 1.333

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 your src directory.

The Browse buttons come in handy here.
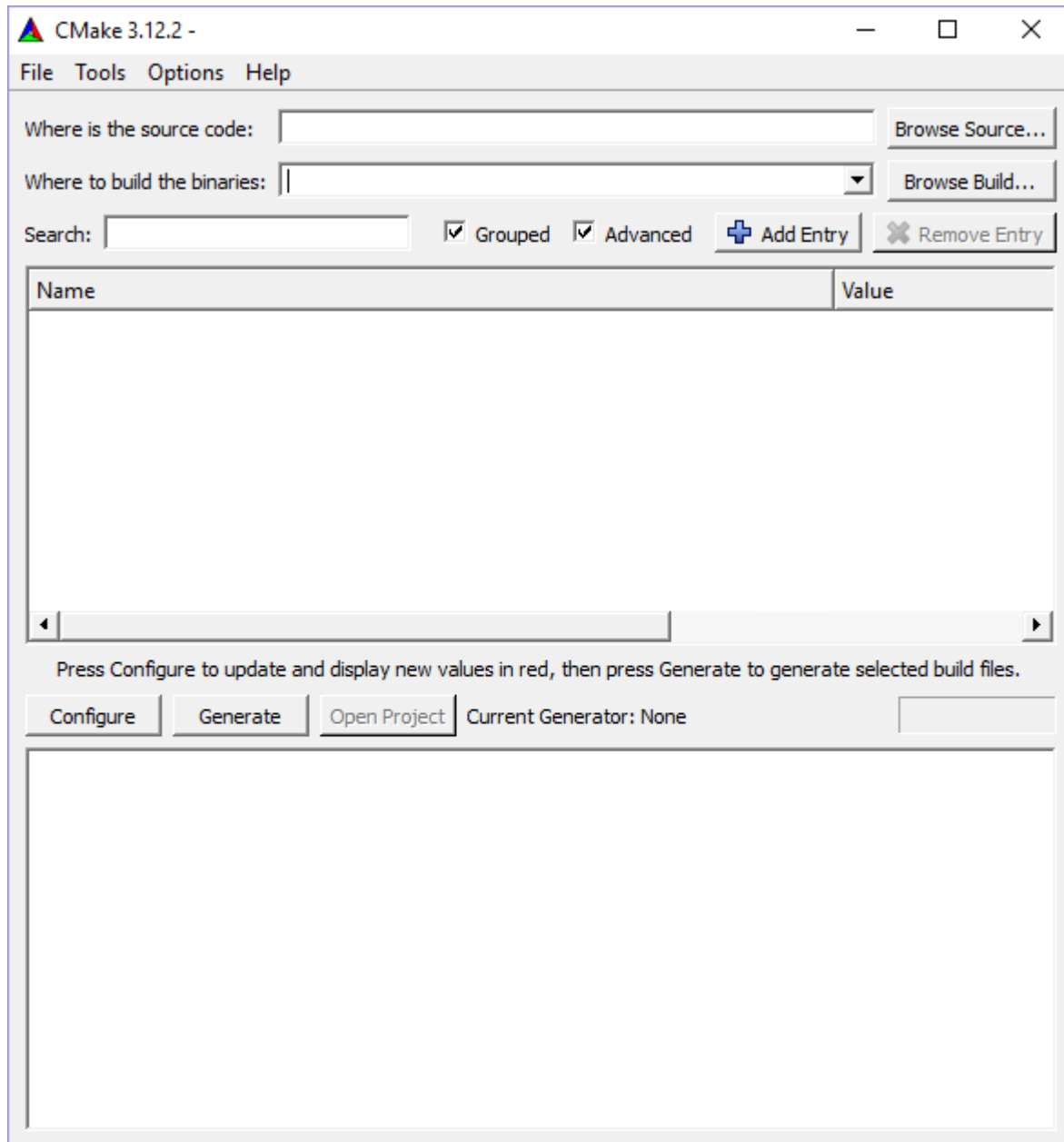
Fig. 1.333: CMake-gui

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 1.334
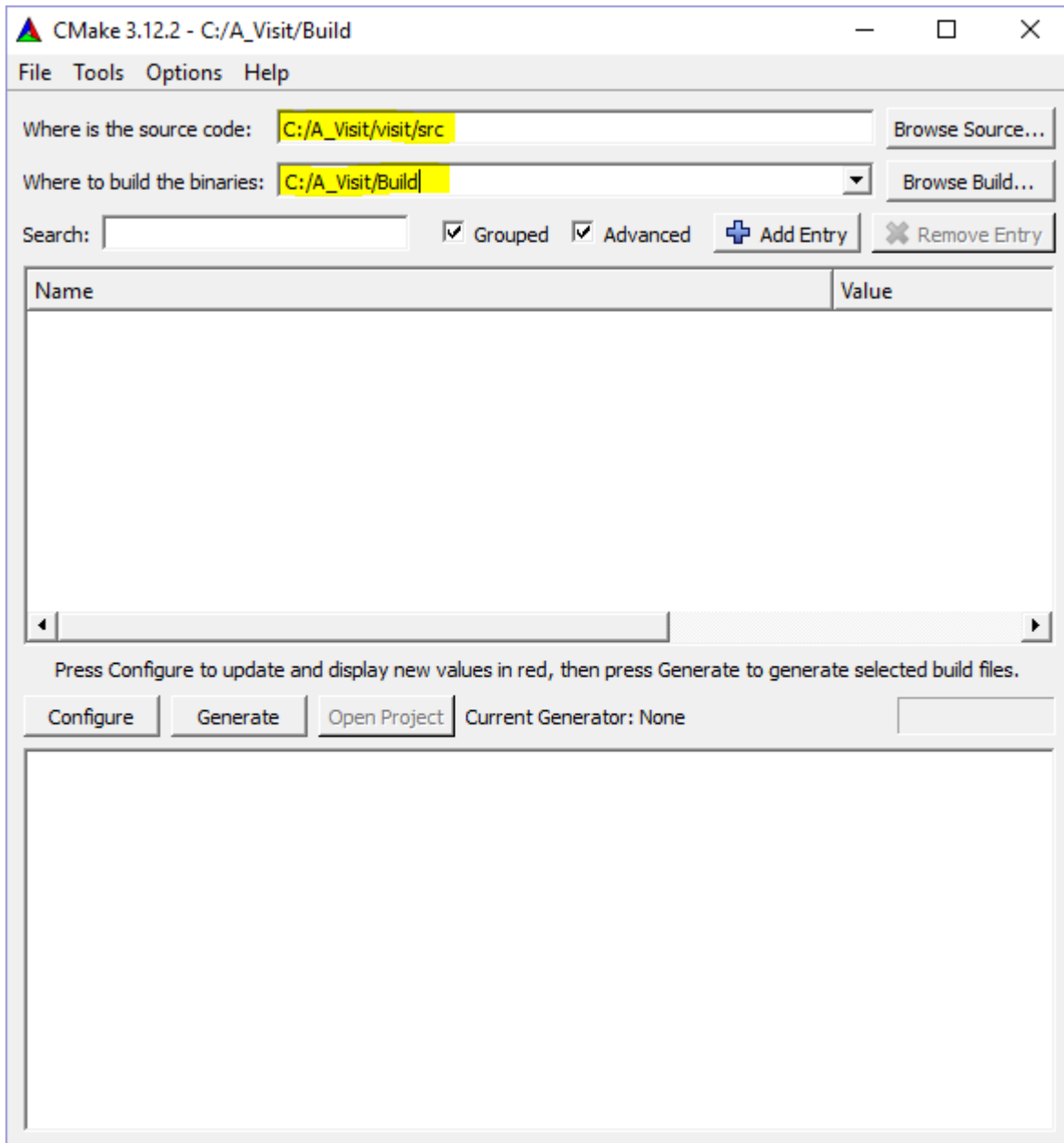
Fig. 1.334: Setting source and build directories

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 1.335

For developement build cloned or downloaded from the github repositories, in order for CMake to locate the directory automatically, visit-deps should be peer to visit. Figure 1.336
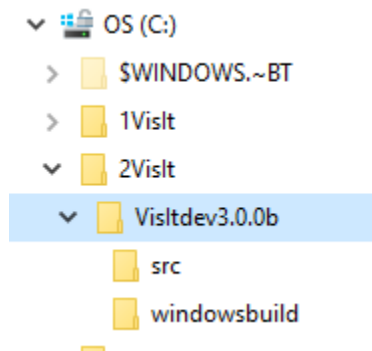
Fig. 1.335: Directory structure with source from a released version
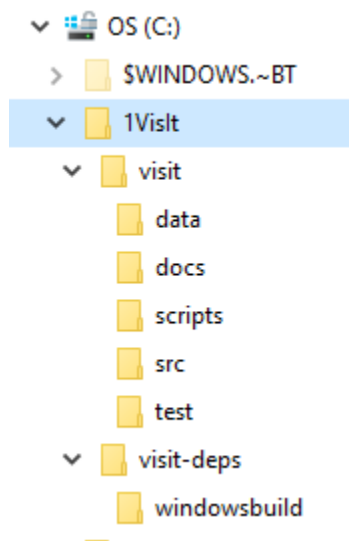
Fig. 1.336: Expected directory structure with source from GitHub repo

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 1.337
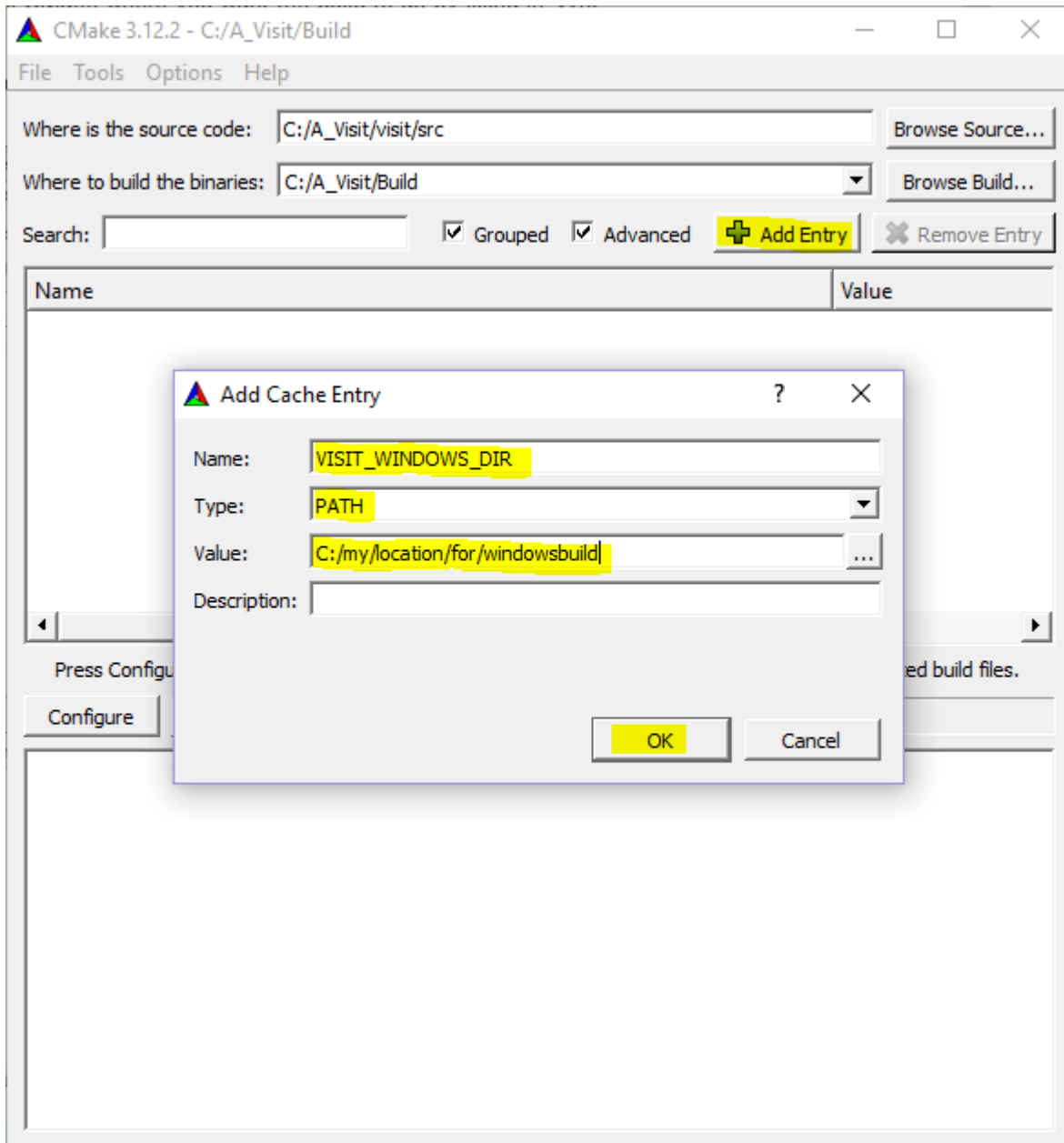
Fig. 1.337: Setting VISIT_WINDOWS_DIR

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 1.338

To limit the database plugins to a specific set of plugins, set the **Name:** to *VISIT_SELECTED_DATABASE_PLUGINS*.
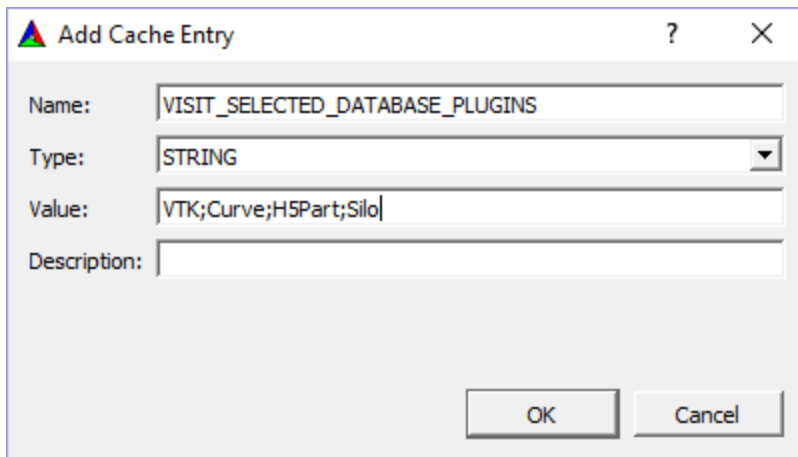
Fig. 1.338: Selecting a limited number of database plugins

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*.

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 *Supress Warnings* section. Click **OK**. Figure 1.339
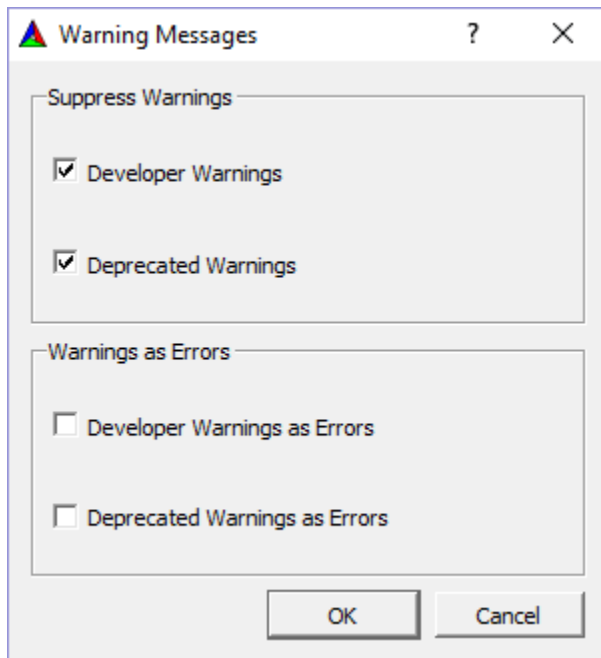
Fig. 1.339: Suppress CMake warnings

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 dropdown and add *host=x64* to use the full 64-bit toolset. Figure 1.340
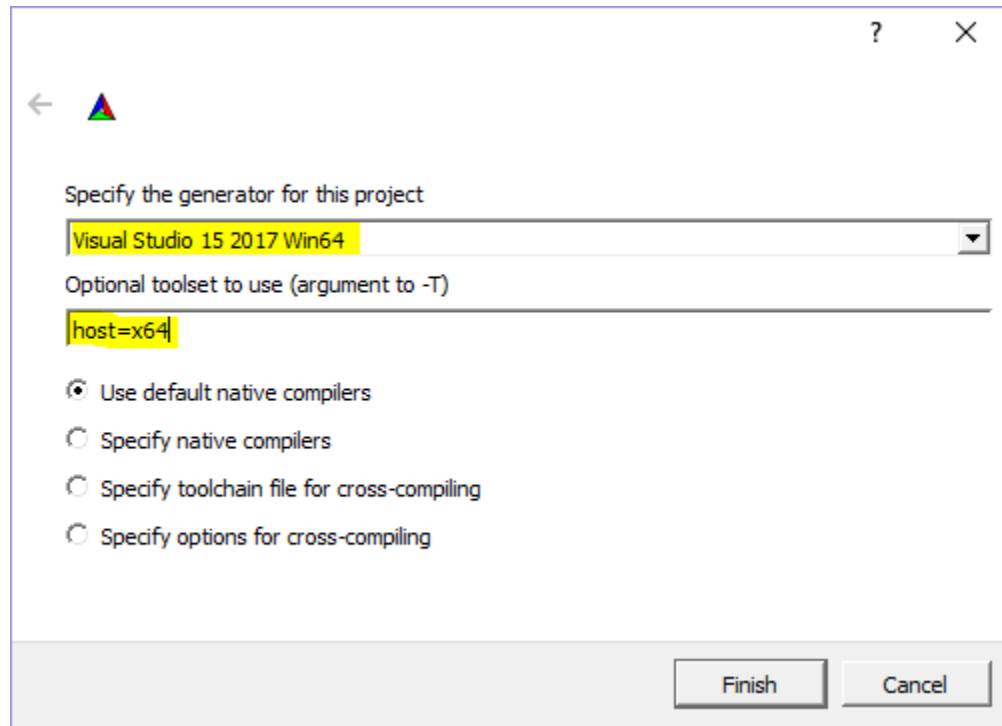
Fig. 1.340: Choosing the generator

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 1.341

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.

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 by hand.
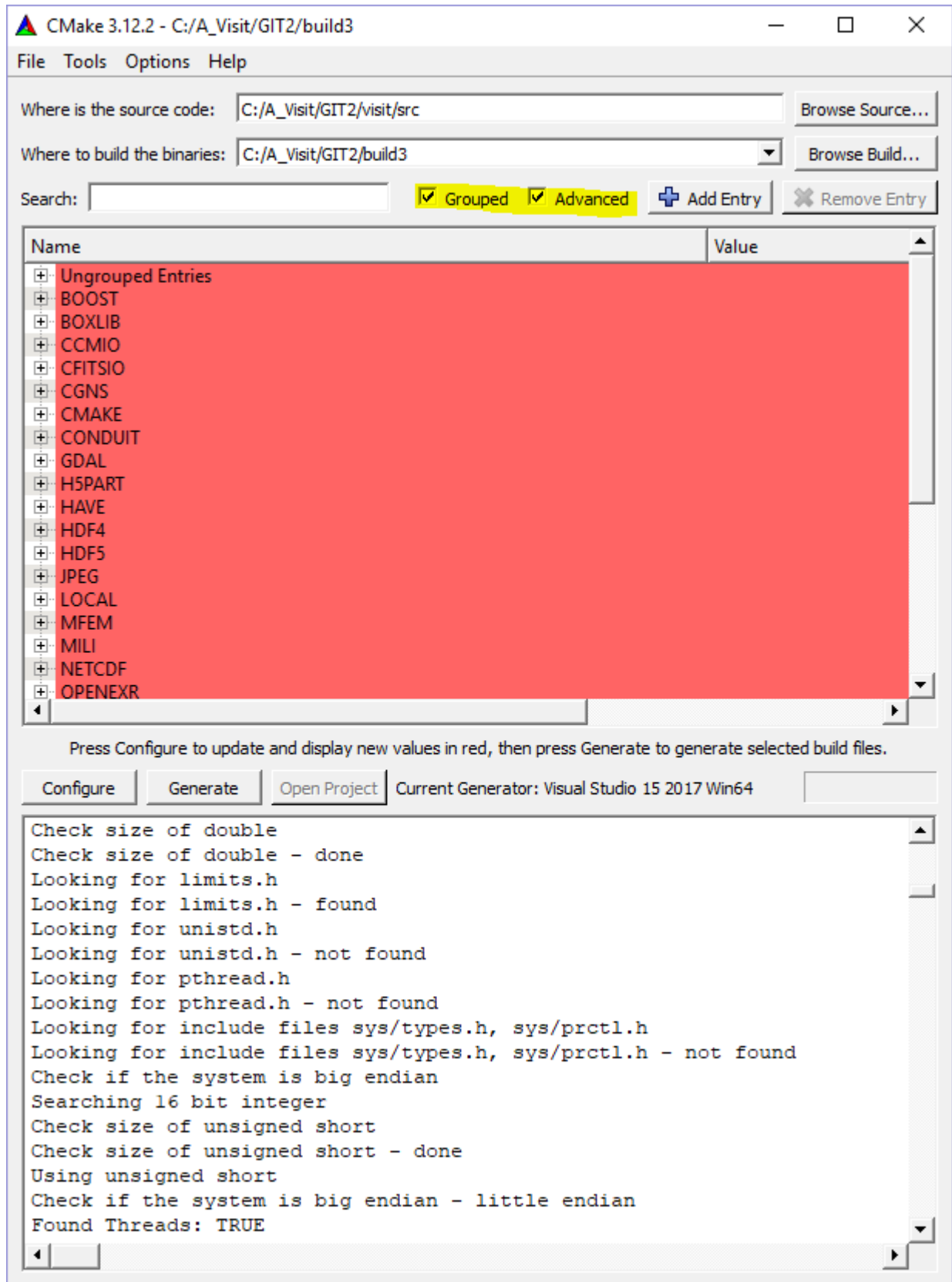
Fig. 1.341: After first configure

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_SUPRESS_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 file). Select desired Configuration and Build solution.

1.20 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.

1.21 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.

Node

Point

Vertex These terms refer to the *corners* or *ends* of mesh elements.

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.

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.

1.22 Contributing

This is a short contributing guide on the VisIt project's use of Sphinx for documentation.

You can check out the Sphinx manual with:

```
svn co svn+ssh://<USERNAME>@edison.nersc.gov/project/projectdirs/visit/svn/visit/
↪trunk/docs/SphynxDocs
```

If you have Sphinx You can build the html manual locally using the command:

```
sphinx-build -b html . _build -a
```

You can then browse the root of the manual by pointing your browser to `./_build/index.html`. The `-a` forces a re-build of everything. Remove it when you are constantly revising and rebuilding.

Your changes to any files in `trunk/docs/SphinxDocs` will go live here at approximately 50 minutes passed every even numbered hour.

1.22.1 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. . .

20. First Level Heading

This is an example of some text under the heading...

20.1. Second Level Heading

This is an example of some text under the heading...

20.1.1. Third Level Heading

This is an example of some text under the heading...

20.1.1.1. Fourth level heading

This is an example of some text under the heading...

- 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.

- Wherever possible, keep lines in `.rst` files to 80 columns or less.

- 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 (MPI, VTK)

- Use case conventions of product names (QuickTime, TotalView, Valgrind).

- Bracket word(s) with two stars (`**some words**`) for **bold**.

- Bracket word(s) with one star (`*word*`) for *italics*.

- Bracket word(s) with two backticks (` ``some words`` `) for `literal`.

- Bracketed word(s) should not span line breaks.

- Use **bold** to refer to VisIt **Widget** names, **Operator** or **Plot** names and other named objects part of VisIt's interface.

- Use the following terminology when referring to widget names.

- Avoid use of **bold** for other purposes. Instead use *italics*.

- Use `literals` for code, commands, arguments, file names, etc.

- 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.

- Begin a line with `..` followed by space for single line comments:

button	New	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	☐ Group tables by Category	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	Continuous ▮▮▮ hot	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	difference gray hot hot_and_cold	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	Smoothing Linear ▾	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	Linear ▾ None Linear Cubic Spline	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.
panel	Manager New difference Delete gray hot Export hot_and_cold Name hot	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	Color table type ◉ Continuous ○ Discrete	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	Number of colors 5 ▲▼	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	Curve view | 2D view | 3D view | AxisArra ◀▶ View normal 0 0 1 Focus 0 0 0 Up Vector 0 1 0	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	Name hot	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.
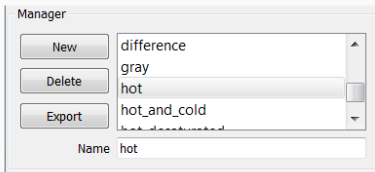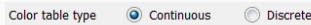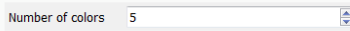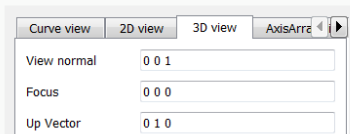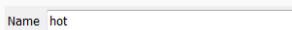
```
.. this is a single line comment

..

    This is a multi-line
    comment
```

- Define anchors ahead of sections or paragraphs you want to cross reference:

```
.. _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. 1.342:

```
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 <../Quantitative/VerdictManual-revA.pdf>`
```

- 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 an on-line, real-time reStructuredText Renderer to quickly try different things and see how they work.

1.22.2 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
```
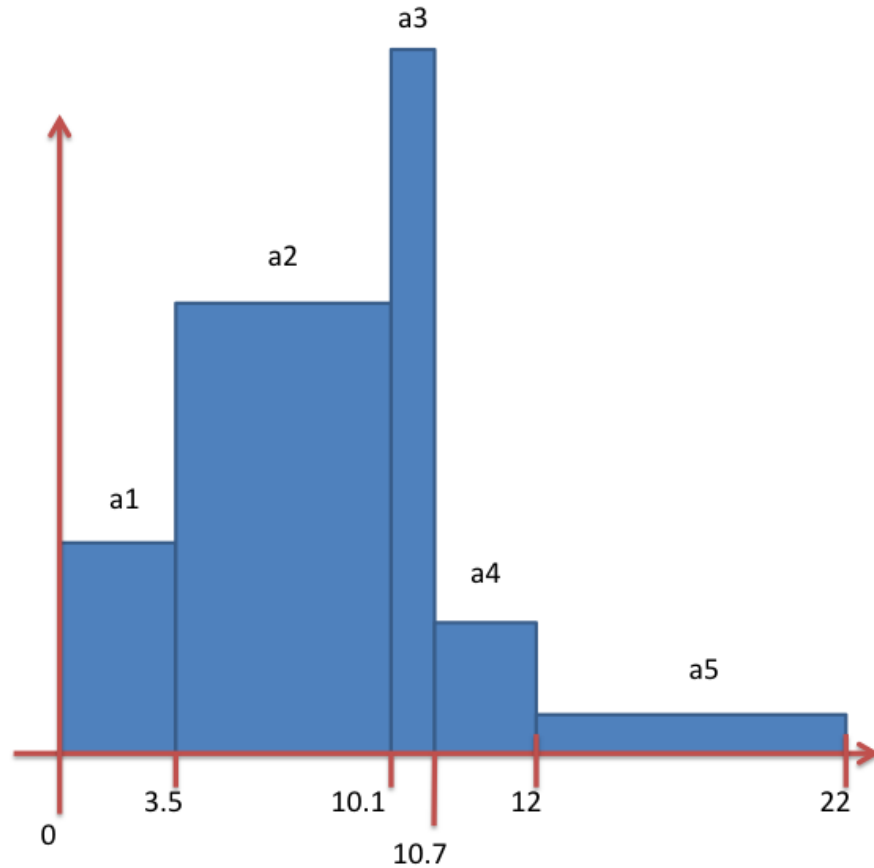

Same image with :scale: 50% option

```
.. 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%
```

which can now be cross referenced using an inline Fig. **??** 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.
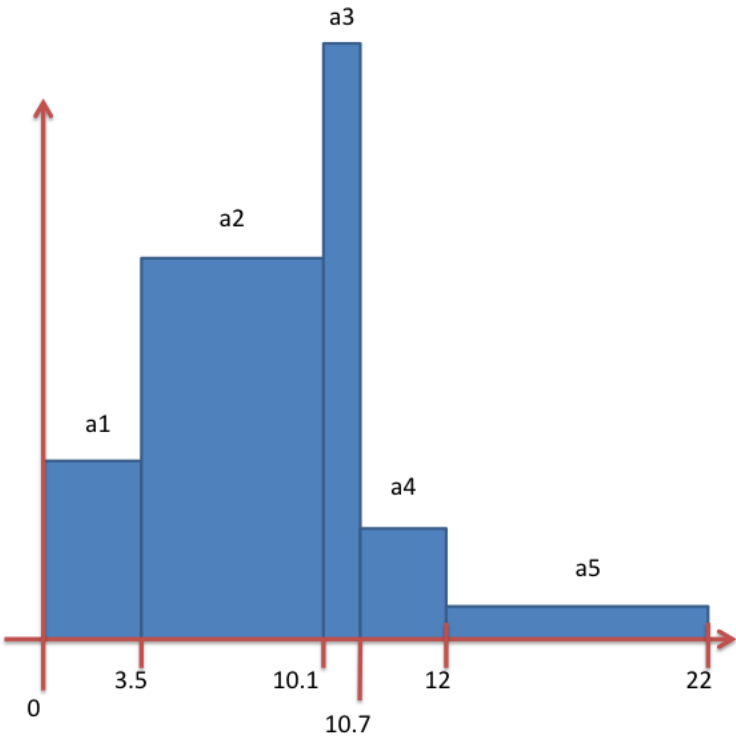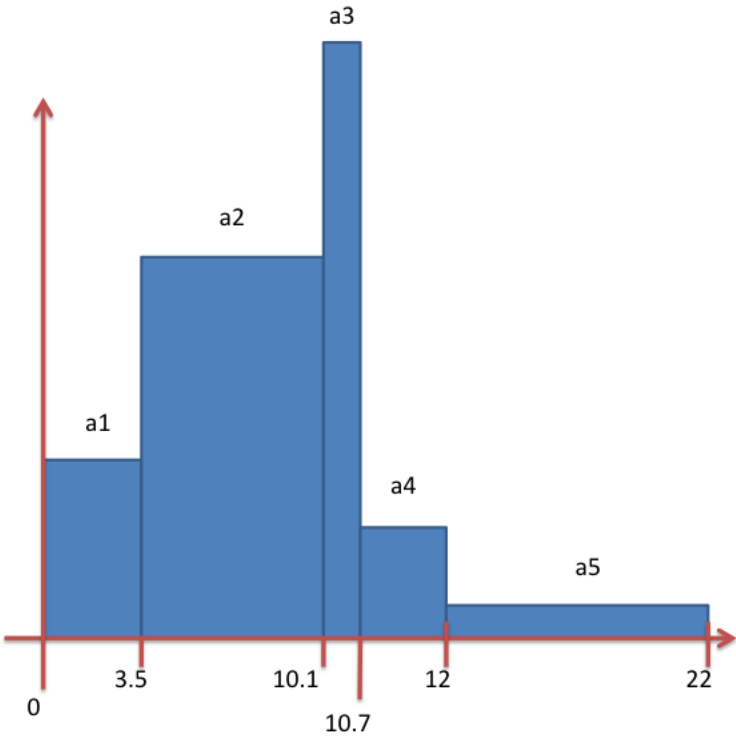
```
.. _my_figure2:

.. figure:: images/array_compose_with_bins.png
   :scale: 50%

   Here is a caption for the figure.
```
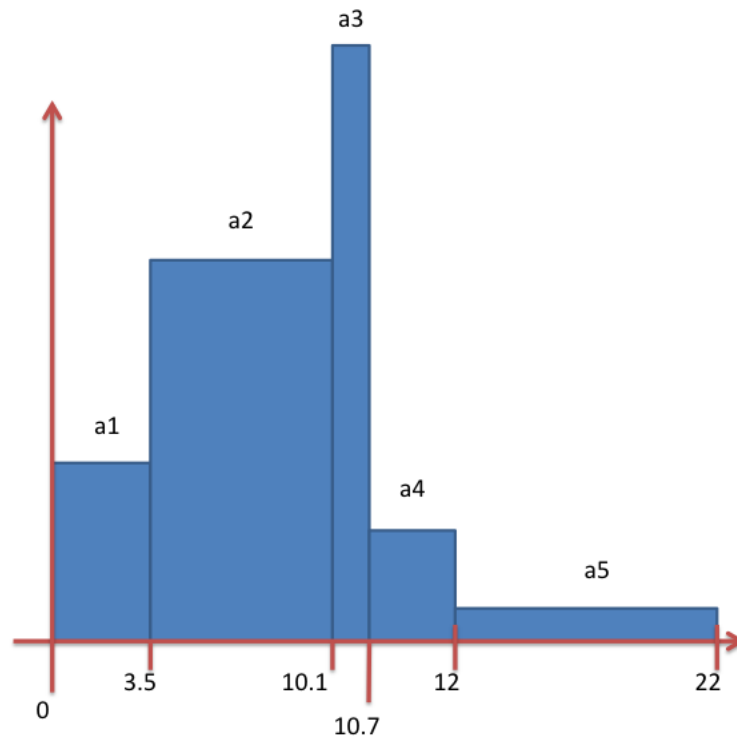

Fig. 1.342: Here is a caption for the figure.

Note that the figure label (e.g. Fig 20.2) will not appear if there is no caption.

1.22.3 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.

1.22.4 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.

1.22.5 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.

1.22.6 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.

- Change name of docs dir to `Sphinx` and not `Sphynx`.

- Add at least another LLNL person to RTD project so we have coverage to fix issues as they come up.

- 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

  - Substitutions for names of products and projects we refer to frequently such as [VTK](#) or [VisIt](#) (as is used throughout this section) or for frequently used text such as **Viewer Window**:

    ```
    Substitutions for names of products and projects we refer to frequently
    such as VTK_ or VisIt_ (as is used throughout this section) or for
    frequently used text such as |viswin|.
    ```

    with the following substitutions defined:

    ```
    .. _VisIt: https://visit.llnl.gov
    .. _VTK: https://www.vtk.org
    .. |viswin| replace:: **Viewer Window**
    ```

    Note that the `.. _VisIt: ...` substitution is already defined for the whole doctree in the `rst_prolog` variable in `conf.py`.

- 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.

```python
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.CurrentPlot
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')
```

VisIt Python (CLI) Interface Manual
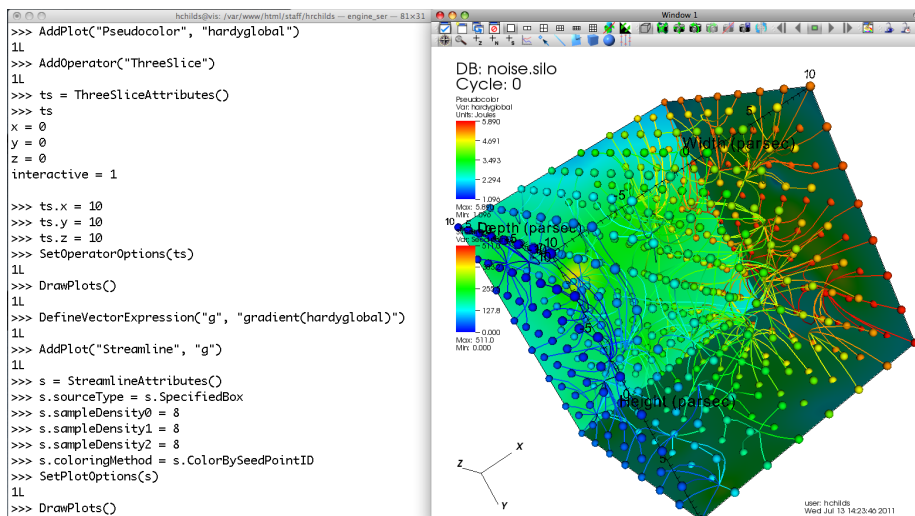
2.1 Introduction to VisIt

2.1.1 Overview

VisIt is a 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. Another benefit of the distributed architecture is that VisIt can visualize the data where it is generated, eliminating the need to move data. VisIt can be controlled by a Graphical User Interface (GUI) or through the Python scripting language. More information about VisIt's Graphical User Interface can be found in the *VisIt User's Manual*.

2.1.2 Manual chapters

This manual is broken down into the following chapters:

Chapter title	Chapter description
Introduction to VisIt	This chapter.
Python	Describes the basic features of the
	Python programming language.
Quick Recipes	Describes common patterns for scripting
	using the VisIt Python Interface.
Functions	Describes functions in the VisIt Python
	Interface.
Attributes References	Describes attributes for setting common
	operations, as well as for VisIt's plugins
CLI Events	Describes possible events for callbacks.

2.1.3 Understanding how VisIt works

VisIt visualizes data by creating one or more plots in a visualization window, also known as a vis window. Examples of plots include Mesh plots, Contour plots and Pseudocolor plots. Plots take as input one or more mesh, material, scalar, or tensor variables. It is possible to modify the variables by applying one or more operators to the variables before passing them to a plot. Examples of operators include arithmetic operations or taking slices through the mesh. It is also possible to restrict the visualization of the data to subsets of the mesh. VisIt provides Python bindings to all of its plots and operators so they may be controlled through scripting. Each plot or operator plugin provides a function, which is added to the VisIt namespace, to create the right type of plot or operator attributes. The attribute object can then be modified by setting its fields and then it can be passed to a general-purpose function to set the plot or operator attributes. To display a complete list of functions in the VisIt Python Interface, you can type dir() at the Python prompt. Similarly, to inspect the contents of any object, you can type its name at the Python prompt. VisIt supports up to 16 visualization windows, also called vis windows. Each vis window is independent of the other vis windows and VisIt Python functions generally apply only to the currently active vis window. This manual explains how to use the VisIt Python Interface which is a Python extension module that controls VisIt's viewer. In that way, the VisIt Python Interface fulfills the same role as VisIt's GUI. The difference is that the viewer is totally controlled through Python scripting, which makes it easy to write scripts to create visualizations and even movies. Since the VisIt module controls VisIt's viewer, the Python interpreter currently has no direct mechanism for passing data to the compute engine (see Figure *[fig:architecture]*). If you want to write a script that generates simulation data and have that script pass data to the compute engine, you must pass the data through a file on disk. The VisIt Python Interface comes packaged in two varieties: the extension module and the Command Line Interface (CLI). The extension module version of the VisIt Python Interface is imported into a standard Python interpreter using the import directive. VisIt's command line interface (CLI) is essentially a Python interpreter where the VisIt Python Interface is built-in. The CLI is provided to simplify the process of running VisIt Python scripts.

2.1.4 Starting VisIt

You can invoke VisIt's command line 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 "visit" front-end module 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.
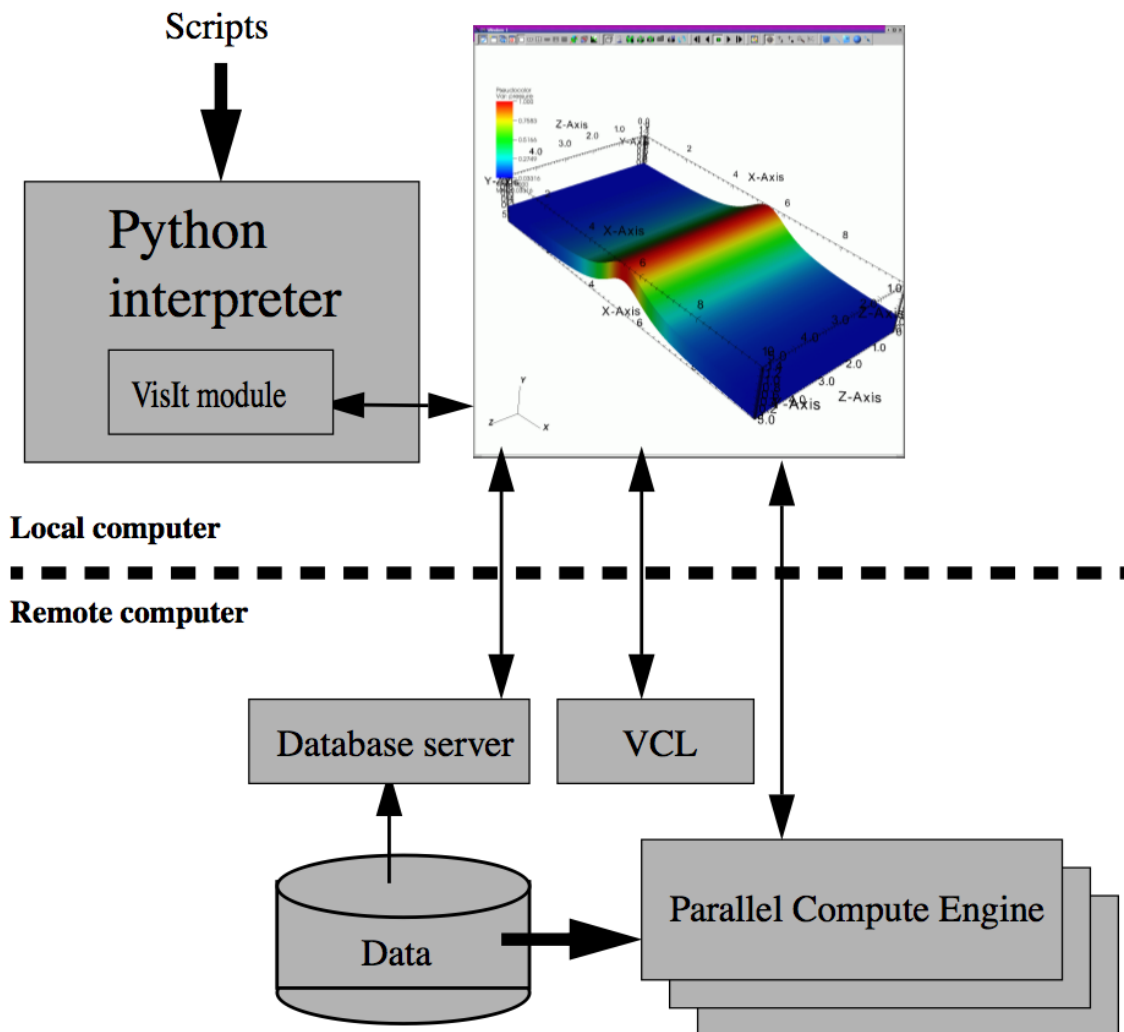
Fig. 2.1: VisIt's architecture

If you do not install the visit.so module as a Python site package, you can tell the Python interpreter where it is located by appending a new path to the sys.path variable. Be sure to substitute the correct path to visit.so on your system.

```python
import sys
sys.path.append("/path/to/visit/<version>/<architecture>/lib/site-packages")
```

Here is how to import all functions into the global Python namespace:

```python
from visit import *
Launch()
```

Here is how to import all functions into a "visit" module namespace:

```python
import visit
visit.Launch()
```

2.1.5 Getting started

VisIt is a tool for visualizing 2D and 3D scientific databases. The first thing to do when running VisIt is select databases to visualize. To select a database, you must first open the database using the OpenDatabase function. After a window has an open database, any number of plots and operators can be added. To create a plot, use the AddPlot function. After adding a plot, call the DrawPlots function to make sure that all of the new plots are drawn.

Example:

```python
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 VisIt Python 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:

```python
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.

2.2 Python

2.2.1 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.

2.2.2 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
```

2.2.3 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
```

2.2.4 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.

2.2.5 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.

```python
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])
```

2.2.6 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.

```python
# 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.

```python
# Iterating through the characters of a string
for c in "characters":
    print c
```

```python
# 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
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.

```python
token = get_next_token()
while token != "":
    do_something(token)
    token = get_next_token()
```

2.2.7 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.

```python
# 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
```

2.3 Quick Recipes

2.3.1 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.

2.3.2 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.

```python
# Import a session file from the current working directory.
RestoreSesssion("my_visualization.session", 0)
# Now that VisIt has restored the session, animate through time.
for states in range(TimeSliderGetNStates()):
  SetTimeSliderState(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.

```python
# Step 1: Open a database
OpenDatabase("/usr/local/visit/data/wave.visit")

# Step 2: Add plots
AddPlot("Pseudocolor", "pressure")
AddPlot("Mesh", "quadmesh")

# Step 3: Draw the plots
DrawPlots()

# Step 4: Animate through time and save images
for states in range(TimeSliderGetNStates()):
  SetTimeSliderState(state)
  SaveWindow()
```

2.3.3 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.

```python
# 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)
```

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.

```python
# 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
```

2.3.4 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 at time state 0
OpenDatabase("/usr/local/visit/data/allinone00.pdb")
```

Opening a database at late 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 later time state to pick up transient variables
OpenDatabase("/usr/local/visit/data/wave.visit", 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 first file in series wave0000.silo, wave0010.silo, ...
OpenDatabase("/usr/local/visit/data/wave0000.silo")

# Opening a virtual database representing all wave*.silo files.
OpenDatabase("/usr/local/visit/data/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:/usr/local/visit/data/wave.visit", 17)
```

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 remote, parallel compute engine before opening a database
OpenComputeEngine("mcr", ("-np", "4", "-nn", "2"))
OpenDatabase("mcr:/usr/local/visit/data/multi_ucd3d.silo")
```

2.3.5 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
print PlotPlugins()
# Create plots
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()
# Look in the object
print p
# Set the min/max values
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", "density")

# List the plots. The second plot should be active.
ListPlots()

# Draw the plots
DrawPlots()

# Hide the first plot
SetActivePlots(0)
HideActivePlots()

# Set both plots' color table to "hot"
p = PseudocolorAttributes()
p.colorTableName = "hot"
SetActivePlots((0,1))
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.

```
# Save an image and take care of plots that entered the error state.
drawThePlots = 0
for state in range(TimeSliderGetNStates()):
  if SetTimeSliderState(state) == 0:
    drawThePlots = 1
  if drawThePlots == 1:
    if DrawPlots() == 0:
      print "VisIt could not draw plots for state: %d" % state
    else:
      drawThePlots = 0
  SaveWindow()
```

2.3.6 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

Adding an operator is similar to adding a plot in that you call a function with the name of the operator to be added. The list of available operators is returned by the OperatorPlugins function. Any of the names returned in that plugin can be used to add an operator using the AddOperator function. Operators are added to the active plots by default but you can also force VisIt to add them to all plots in the plot list.

```
# Print available operators
print OperatorPlugins()
# Create a plot
AddPlot("Pseudocolor")
# Add an Isovolume operator and a Slice operator
AddOperator("Isovolume")
AddOperator("Slice")
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("/usr/local/visit/data/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()
```

2.3.7 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("/usr/local/visit/data/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("/usr/local/visit/data/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
```

Lineout

VisIt allows you to extract data along a line, called a lineout, and plot the data using a Curve plot.

```
OpenDatabase("/usr/local/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()
Lineout((-5,-3), (5,8))
# Specify a number of sample points
Lineout((-5,-4), (5,7))
```

Query

VisIt can perform a number of different queries based on values calculated about plots or their originating database.

```
OpenDatabase("/usr/local/visit/data/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 the id's of the MinMax query.
def GetMinMaxIds():
  Query("MinMax")
  import string
  s = string.split(GetQueryOutputString(), " ")
  retval = []
  nextGood = 0
  idType = 0
  for token in s:
    if token == "(zone" or token == "(cell":
      idType = 1
      nextGood = 1
      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("/usr/local/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()

# Do picks on the ids that were returned by MinMax.
for ids in GetMinMaxIds():
  idType = ids[0]
  id = ids[1]
  if idType == 0:
    PickByNode(id)
  else:
    PickByZone(id)
```

2.3.8 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("/usr/local/visit/data/multi_rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Turning off all but the last domain
d = GetDomains()
for dom in d[:-1]:
  TurnDomainsOff(dom)
# 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("/usr/local/visit/data/multi_rect2d.silo")
AddPlot("FilledBoundary", "mat1")
DrawPlots()
# Print the materials are:
GetMaterials()
# Turn off material 2
TurnMaterialsOff("2")
```

2.3.9 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("/usr/local/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
AddPlot("Mesh", "Mesh2D")
AddPlot("Label", "hgslice")
DrawPlots()
print "The current view is:", GetView2D()
# Get an initialized 2D view object.
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's 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("/usr/local/visit/data/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
AddPlot("Mesh", "Mesh")
DrawPlots()
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.

```python
# Do a pseudocolor plot of u.
OpenDatabase("/usr/local/visit/data/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
```

```python
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
# 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)
```

2.3.10 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 *[fig:annotations1]*.

```python
# 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)
```
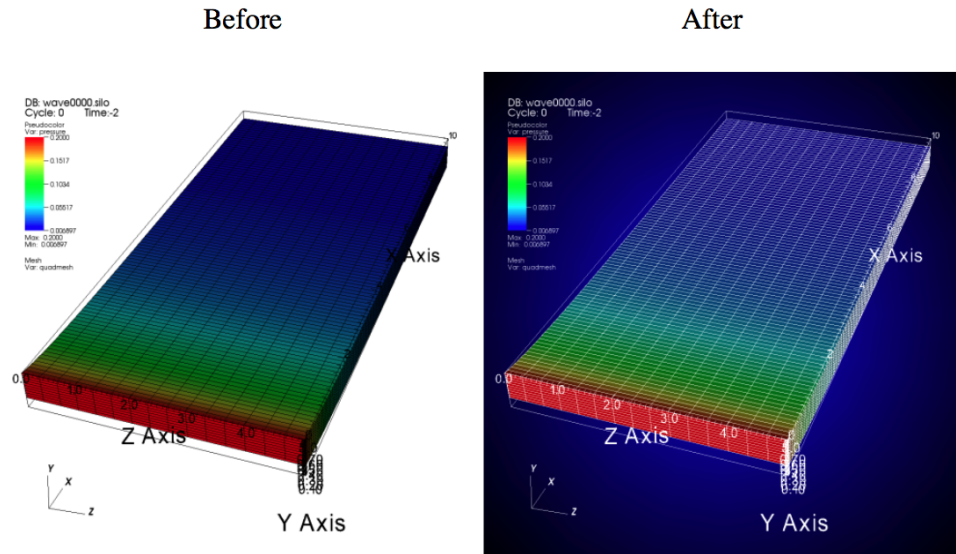
Fig. 2.2: Before and after image of adding a gradient background.

Adding   a   banner

Banners are useful for providing titles for a visualization or for marking its content (see Figure *[fig:annotations2]*). To add an "Unclassified" banner to a visualization, use the following bit of Python code:

```python
# Create a text object that we'll use to indicate that our
# visualization is unclassified.
banner = CreateAnnotationObject("Text2D")
banner.text = "Unclassified"
banner.position = (0.37, 0.95)
banner.fontBold = 1
# print the attributes that you can set in the banner object.
print banner
```

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 *[fig:annotations3]*.

```python
# Add a time slider in the lower left corner
slider = CreateAnnotationObject("TimeSlider")
slider.height = 0.07
# Print the options that are available in the time slider object
print slider
```

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
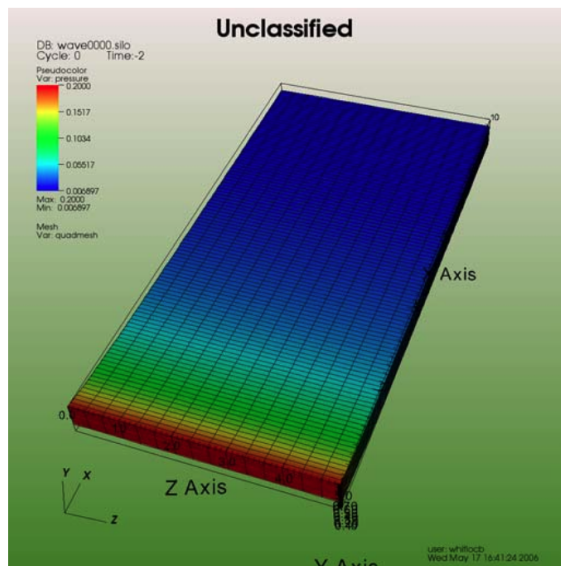
Fig. 2.3: Adding a banner
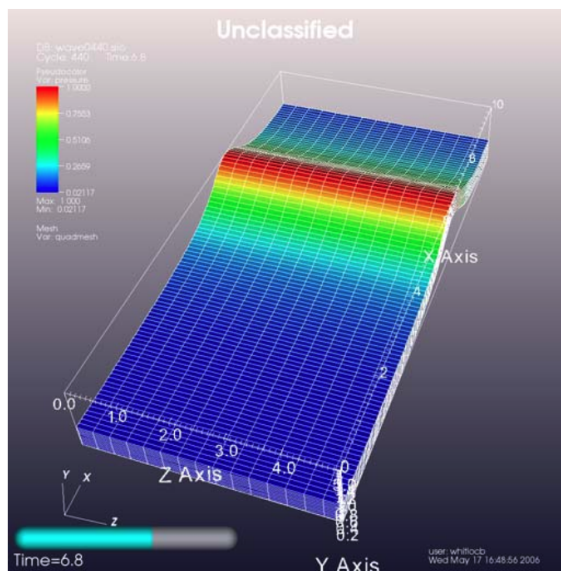
Fig. 2.4: Time slider annotation in the lower left corner

VisIt as an image annotation (see Figure *[fig:annotations4]*). 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.

```
# 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
```
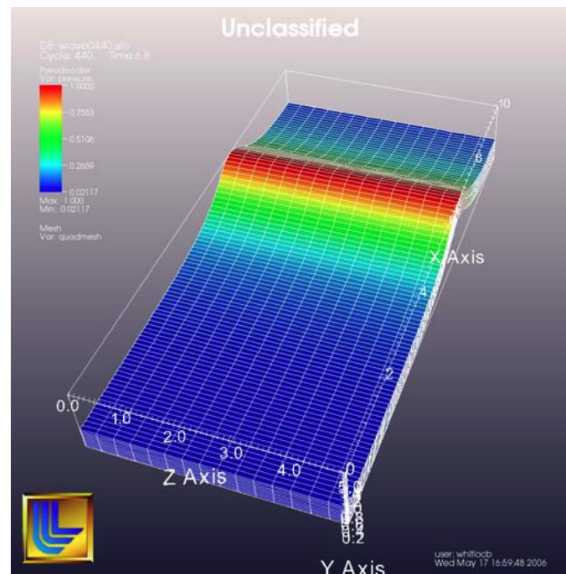

Fig. 2.5: Image annotation used to incorporate LLNL logo

2.4 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.

2.4.1 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:

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()
```

2.4.2 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.
```

2.4.3 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

2.4.4 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()
```

2.4.5 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()
```

2.4.6 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
```

2.4.7 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)
```

2.4.8 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")
```

2.4.9 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")
```

2.4.10 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)
```

2.4.11 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:", GetView3D().centerOfRotation
```

2.4.12 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.
```

2.4.13 ClearCache

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#%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()
```

2.4.14 ClearCacheForAllEngines

<div align="center">

Synopsis:

</div>

```
ClearCacheForAllEngines() -> integer
```

return type [CLI_return_t] 1 on success and 0 on failure.

<div align="center">

Description:

</div>

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()
```

2.4.15 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()
```

2.4.16 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()
```

2.4.17 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()
```

2.4.18 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()
```

2.4.19 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.
```

2.4.20 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
```

2.4.21 Close

Synopsis:

```
Close()
```

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```python
import visit
visit.Launch()
visit.Close() # Close the viewer
```

2.4.22 CloseComputeEngine

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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 host-Name 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.

<div align="center">

Example:

</div>

```python
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo") # Launches an engine
AddPlot("Pseudocolor", "u")
DrawPlots()
CloseComputeEngine() # Close the compute engine
```

2.4.23 CloseDatabase

<div align="center">

Synopsis:

</div>

```
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)
```

2.4.24 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)
```

2.4.25 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
```

2.4.26 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.
```

2.4.27 CopyLightingToWindow

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

The Copy functions copy attributes from one visualization window to another visualization window. The CopyLightingAttributes function copies lighting.

<div align="center">

Example:

</div>

```
#% 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.
```

2.4.28 CopyPlotsToWindow

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
AddWindow()
```

<div align="right">

(continues on next page)

</div>

```
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.
```

2.4.29 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.
```

2.4.30 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)
```

2.4.31 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)
```

2.4.32 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")
```

2.4.33 DatabasePlugins

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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".

<div align="center">

Example:

</div>

```
#% visit -cli
dbp = DatabasePlugins("localhost")
print dbp["host"]
print dbp["plugins"]
```

2.4.34 DeIconifyAllWindows

<div align="center">

Synopsis:

</div>

```
DeIconifyAllWindows()
```

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% visit -cli
SetWindowLayout(4) # Have 4 windows
```

<div align="right">(continues on next page)</div>

```
IconifyAllWindows()
DeIconifyAllWindows()
```

2.4.35 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 DefineExpression functions return 1 on success and 0 on failure.

Description:

DefineArrayExpression creates new array variables. Expression variables can be plotted like any other variable. 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 math operators and pre-existing variable names Reference the VisIt User's Manual if you want more information on creating expressions, such as expression syntax, or a list of built-in expression functions.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
# Plot the scalar expression variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()
# Plot a vector expression variable.
DefineVectorExpression("myvec", "{u,v,w}")
AddPlot("Vector", "myvec")
DrawPlots()
```

2.4.36 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 DefineExpression functions return 1 on success and 0 on failure.

Description:

DefineCurveExpression creates new curve variables. Expression variables can be plotted like any other variable. 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 math operators and pre-existing variable names Reference the VisIt User's Manual if you want more information on creating expressions, such as expression syntax, or a list of built-in expression functions.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
# Plot the scalar expression variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()
# Plot a vector expression variable.
DefineVectorExpression("myvec", "{u,v,w}")
AddPlot("Vector", "myvec")
DrawPlots()
```

2.4.37 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 DefineExpression functions return 1 on success and 0 on failure.

Description:

The DefineMaterialExpression function creates new material variables. Expression variables can be plotted like any other variable. 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 math operators and pre-existing variable names Reference the VisIt User's Manual if you want more information on creating expressions, such as expression syntax, or a list of built-in expression functions.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
# Plot the scalar expression variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()
# Plot a vector expression variable.
DefineVectorExpression("myvec", "{u,v,w}")
AddPlot("Vector", "myvec")
DrawPlots()
```

2.4.38 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 DefineExpression functions return 1 on success and 0 on failure.

<div align="center">

Description:

</div>

The DefineMeshExpression creates new mesh variables. Expression variables can be plotted like any other variable. 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 math operators and pre-existing variable names Reference the VisIt User's Manual if you want more information on creating expressions, such as expression syntax, or a list of built-in expression functions.

<div align="center">

Example:

</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
# Plot the scalar expression variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()
# Plot a vector expression variable.
DefineVectorExpression("myvec", "{u,v,w}")
AddPlot("Vector", "myvec")
DrawPlots()
```

2.4.39 DefinePythonExpression

<div align="center">

Synopsis:

</div>

```
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'.

<div align="center">

Description:

</div>

Used to define a Python Filter Expression.

2.4.40 DefineScalarExpression

<div align="center">

Synopsis:

</div>

```
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 DefineExpression functions return 1 on success and 0 on failure.

Description:

The DefineScalarExpression function creates a new scalar variable based on other variables from the open database. Expression variables can be plotted like any other variable. 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 math operators and pre-existing variable names Reference the VisIt User's Manual if you want more information on creating expressions, such as expression syntax, or a list of built-in expression functions.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
# Plot the scalar expression variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()
# Plot a vector expression variable.
DefineVectorExpression("myvec", "{u,v,w}")
AddPlot("Vector", "myvec")
DrawPlots()
```

2.4.41 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 DefineExpression functions return 1 on success and 0 on failure.

Description:

The DefineSpeciesExpression creates new species variables. Expression variables can be plotted like any other variable. 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 math operators and pre-existing variable names Reference the VisIt User's Manual if you want more information on creating expressions, such as expression syntax, or a list of built-in expression functions.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
# Plot the scalar expression variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()
# Plot a vector expression variable.
DefineVectorExpression("myvec", "{u,v,w}")
```

```
AddPlot("Vector", "myvec")
DrawPlots()
```

2.4.42 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 DefineExpression functions return 1 on success and 0 on failure.

Description:

The DefineTensorExpression creates new tensor variables. Expression variables can be plotted like any other variable. 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 math operators and pre-existing variable names Reference the VisIt User's Manual if you want more information on creating expressions, such as expression syntax, or a list of built-in expression functions.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
# Plot the scalar expression variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()
# Plot a vector expression variable.
DefineVectorExpression("myvec", "{u,v,w}")
AddPlot("Vector", "myvec")
DrawPlots()
```

2.4.43 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 DefineExpression functions return 1 on success and 0 on failure.

Description:

The DefineVectorExpression creates new vector variables Expression variables can be plotted like any other variable. 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 math operators

and pre-existing variable names Reference the VisIt User's Manual if you want more information on creating expressions, such as expression syntax, or a list of built-in expression functions.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")
# Plot the scalar expression variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()
# Plot a vector expression variable.
DefineVectorExpression("myvec", "{u,v,w}")
AddPlot("Vector", "myvec")
DrawPlots()
```

2.4.44 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.
```

2.4.45 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.
```

2.4.46 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])
```

2.4.47 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.

<div align="center">

Example:

</div>

```
#% 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.
```

2.4.48 DeleteNamedSelection

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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")
```

2.4.49 DeletePlotDatabaseKeyframe

<div align="center">

Synopsis:

</div>

```
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()
```

2.4.50 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()
```

(continues on next page)

```
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)
```

2.4.51 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)
```

2.4.52 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.
```

2.4.53 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()
```

2.4.54 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()
```

2.4.55 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.
```

2.4.56 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.
```

2.4.57 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)
```

2.4.58 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.

2.4.59 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)
```

2.4.60 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)
```

2.4.61 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')
```

2.4.62 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"
ExportDatabase(e)
```

2.4.63 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()
```

2.4.64 GetActiveContinuousColorTable

Synopsis:

```
GetActiveContinuousColorTable() -> string
```

return type [string] Both functions return 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 GetActiveContinuousColorTable function returns the name of the default continuous color table. The GetActiveDiscreteColorTable function returns the name of the default discrete color table.

Example:

```
#% visit -cli
print "Default continuous color table: %s" % GetActiveContinuousColorTable()
print "Default discrete color table: %s" % GetActiveDiscreteColorTable()
```

2.4.65 GetActiveDiscreteColorTable

Synopsis:

```
GetActiveDiscreteColorTable() -> string
```

return type [string] Both functions return 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 GetActiveContinuousColorTable function returns the name of the default continuous color table. The GetActiveDiscreteColorTable function returns the name of the default discrete color table.

Example:

```
#% visit -cli
print "Default continuous color table: %s" % \
GetActiveContinuousColorTable()
print "Default discrete color table: %s" % \
GetActiveDiscreteColorTable()
```

2.4.66 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()
```

2.4.67 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
```

2.4.68 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()
```

2.4.69 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)
```

2.4.70 GetAnnotationObject

Synopsis:

```
GetAnnotationObject(string) -> Annotation object
```

string [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.

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. Also note that although this function will apparently also accept an integer index, that mode of access is not reliably and should be avoided.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
a = CreateAnnotationObject("TimeSlider")
GetAnnotationObjectNames()
["plot0000", "TimeSlider1"]
ref = GetAnnotationObject("TimeSlider1")
print ref
```

2.4.71 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"]
```

2.4.72 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)
```

2.4.73 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()
```

2.4.74 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()
```

2.4.75 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()
```

2.4.76 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")
```

2.4.77 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)
```

2.4.78 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])
```

2.4.79 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])
```

2.4.80 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
```

2.4.81 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"))
```

2.4.82 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)
```

2.4.83 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)
```

2.4.84 GetLastError

Synopsis:

```
GetLastError() -> string
```

return type [string] GetLastError returns a string containing the last error message that VisIt issued.

Description:

The GetLastError function returns a string containing the last error message that VisIt issued.

Example:

```
#% visit -cli
OpenDatabase("/this/database/does/not/exist")
print "VisIt Error: %s" % GetLastError()
```

2.4.85 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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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)
```

2.4.86 GetLocalHostName

<div align="center">

Synopsis:

</div>

```
GetLocalHostName() -> string
```

return type [string] Both functions return a string.

<div align="center">

Description:

</div>

The GetLocalHostName function returns a string that contains the name of the local computer.

<div align="center">

Example:

</div>

```
#% visit -cli
print "Local machine name is: %s" % GetLocalHostName()
print "My username: %s" % GetLocalUserName()
```

2.4.87 GetLocalUserName

<div align="center">

Synopsis:

</div>

```
GetLocalUserName() -> string
```

return type [string] Both functions return a string.

<div align="center">

Description:

</div>

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()
```

2.4.88 GetMachineProfile

Synopsis:

```
GetMachineProfile(hostname) -> MachineProfile object
```

hostname : string

return type [MachineProfile object] MachineProfile for hostname.

Description:

Gets the MachineProfile for a given hostname

2.4.89 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

2.4.90 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()
```

(continues on next page)

```
# 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()
```

2.4.91 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)
```

2.4.92 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()
```

2.4.93 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 xrange(md.GetNumScalars()):
AddPlot('Pseudocolor', md.GetScalars(i).name)
DrawPlots()
```

2.4.94 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()
```

2.4.95 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
```

2.4.96 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.
```

2.4.97 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
```

2.4.98 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
```

2.4.99 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.

<div align="center">**Example:**</div>

```
#%visit -cli
offon = ("off", "on")
print "Pipeline caching is %s" % offon[GetPipelineCachingMode()]
```

2.4.100 GetPlotInformation

<div align="center">**Synopsis:**</div>

```
GetPlotInformation() -> dictionary
```

return type [dictionary] GetPlotInformation returns a dictionary.

<div align="center">**Description:**</div>

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>.

<div align="center">**Example:**</div>

```
#% 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]
```

2.4.101 GetPlotList

<div align="center">**Synopsis:**</div>

```
GetPlotList() -> PlotList object
```

return type [PlotList object] The GetPlotList function returns a PlotList object.

<div align="center">**Description:**</div>

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

<div align="center">**Example:**</div>

```
# Copy plots (without operators to window 2)
pL = GetPlotList()
AddWindow()
for i in xrange(pL.GetNumPlots()):
AddPlot(PlotPlugins()[pL.GetPlots(i).plotType], pL.GetPlots(i).plotVar)
DrawPlots()
```

2.4.102 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)
```

2.4.103 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',)
```

2.4.104 GetQueryOutputObject

Synopsis:

```
GetQueryOutputObject() -> dictionary or value
```

return type [dictionary or value] GetQueryOutputObject returns an xml string produced by the last query.

Description:

Both the GetQueryOutputString and GetQueryOutputValue functions return information about the last query to be executed but the type of information returns differs. GetQueryOutputString returns a string containing the output of the last query. GetQueryOutputValue returns a single number or tuple of numbers, depending on the nature of the last query to be executed. GetQueryOutputXML and GetQueryOutputObject expose more complex query output.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
Query("MinMax")
print GetQueryOutputString()
print "The min is: %g and the max is: %g" % GetQueryOutputValue()
```

2.4.105 GetQueryOutputString

Synopsis:

```
GetQueryOutputString() -> string
```

return type [string] GetQueryOutputString returns a string.

Description:

Both the GetQueryOutputString and GetQueryOutputValue functions return information about the last query to be executed but the type of information returns differs. GetQueryOutputString returns a string containing the output of the last query. GetQueryOutputValue returns a single number or tuple of numbers, depending on the nature of the last query to be executed. GetQueryOutputXML and GetQueryOutputObject expose more complex query output.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
Query("MinMax")
print GetQueryOutputString()
print "The min is: %g and the max is: %g" % GetQueryOutputValue()
```

2.4.106 GetQueryOutputValue

Synopsis:

```
GetQueryOutputValue() -> double, tuple of doubles
```

return type [double, tuple of doubles] GetQueryOutputValue returns a single double precision number or a tuple of double precision numbers.

Description:

Both the GetQueryOutputString and GetQueryOutputValue functions return information about the last query to be executed but the type of information returns differs. GetQueryOutputString returns a string containing the output of the last query. GetQueryOutputValue returns a single number or tuple of numbers, depending on the nature of the last query to be executed. GetQueryOutputXML and GetQueryOutputObject expose more complex query output.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
Query("MinMax")
print GetQueryOutputString()
print "The min is: %g and the max is: %g" % GetQueryOutputValue()
```

2.4.107 GetQueryOutputXML

Synopsis:

```
GetQueryOutputXML() -> string
```

return type [string] GetQueryOutputXML returns an xml string produced by the last query.

Description:

Both the GetQueryOutputString and GetQueryOutputValue functions return information about the last query to be executed but the type of information returns differs. GetQueryOutputString returns a string containing the output of the last query. GetQueryOutputValue returns a single number or tuple of numbers, depending on the nature of the last query to be executed. GetQueryOutputXML and GetQueryOutputObject expose more complex query output.

Example:

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
Query("MinMax")
print GetQueryOutputString()
print "The min is: %g and the max is: %g" % GetQueryOutputValue()
```

2.4.108 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")
```

(continues on next page)

```
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()
```

2.4.109 GetQueryParameters

Synopsis:

```
GetQueryParameters(name) -> dictionary
```

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.)
xrayInput["image_size"]=(300,300)
xrayInput["vars"]=("p", "d")
Query("XRay Image", xrayInput)
```

2.4.110 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)
```

2.4.111 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
```

2.4.112 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.

<div align="center">**Example:**</div>

```
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)
```

2.4.113 GetSelectionList

<div align="center">**Synopsis:**</div>

```
GetSelectionList() -> SelectionList object
```

return type [SelectionList object] The GetSelectionList function returns a SelectionList object.

<div align="center">**Description:**</div>

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.

<div align="center">**Example:**</div>

```
s = GetSelectionList()
```

2.4.114 GetSelectionSummary

<div align="center">**Synopsis:**</div>

```
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.

<div align="center">**Description:**</div>

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.

<div align="center">**Example:**</div>

```
print GetSelectionSummary('selection1')
```

2.4.115 GetTimeSliders

<div align="center">**Synopsis:**</div>

```
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()
```

2.4.116 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.

2.4.117 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()
```

(continues on next page)

```
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.
```

2.4.118 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.
```

2.4.119 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.
```

2.4.120 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.
```

2.4.121 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()
```

2.4.122 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()
AddPlot("FilledBoundary", "mat1")
DrawPlots()
```

2.4.123 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.

<div align="center">**Example:**</div>

```
#% visit -cli
SetWindowLayout(4)
HideToolbars()
ShowToolbars()
# Hide the toolbars for all windows.
HideToolbars(1)
```

2.4.124 IconifyAllWindows

<div align="center">**Synopsis:**</div>

```
IconifyAllWindows()
```

<div align="center">**Description:**</div>

The IconifyAllWindows function minimizes all of the hidden visualization windows to get them out of the way.

<div align="center">**Example:**</div>

```
#% visit -cli
SetWindowLayout(4) # Have 4 windows
IconifyAllWindows()
DeIconifyAllWindows()
```

2.4.125 InitializeNamedSelectionVariables

<div align="center">**Synopsis:**</div>

```
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.

<div align="center">**Description:**</div>

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.

<div align="center">**Example:**</div>

```
InitializeNamedSelectionVariables('selection1')
```

2.4.126 InvertBackgroundColor

<div align="center">**Synopsis:**</div>

```
InvertBackgroundColor()
```

<div align="center">**Description:**</div>

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()
```

2.4.127 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()
```

2.4.128 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()
```

2.4.129 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)
```

2.4.130 ListDomains

Synopsis:

```
ListDomains()
```

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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
```

2.4.131 ListMaterials

<div align="center">

Synopsis:

</div>

```
ListMaterials()
```

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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
```

2.4.132 ListPlots

<div align="center">

Synopsis:

</div>

```
ListPlots() -> string
ListPlots(stringOnly) -> string
```

return type [string] The ListPlots function returns a string containing a representation of the. plot list.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "u")
```

<div align="right">

(continues on next page)

</div>

```
AddPlot("Contour", "d")
DrawPlots()
ListPlots()
```

2.4.133 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)
```

2.4.134 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")
```

2.4.135 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
>>>
```

2.4.136 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()
```

2.4.137 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.

<div align="center">

Description:

</div>

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.

2.4.138 MoveAndResizeWindow

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

MoveAndResizeWindow moves and resizes a visualization window.

<div align="center">

Example:

</div>

```
#% visit -cli
MoveAndResizeWindow(1, 100, 100, 300, 600)
```

2.4.139 MovePlotDatabaseKeyframe

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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)
```

2.4.140 MovePlotKeyframe

Synopsis:

```
MovePlotKeyframe(index, oldFrame, newFrame)
```

index [integer] An integer representing the index of the plof 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
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()
```

2.4.141 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)
```

2.4.142 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)
```

2.4.143 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)
```

2.4.144 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)
```

2.4.145 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.

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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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))
```

2.4.146 NumColorTableNames

<div align="center">

Synopsis:

</div>

```
NumColorTableNames() -> integer
```

return type [CLI_return_t] The NumColorTableNames function return an integer.

<div align="center">

Description:

</div>

The NumColorTableNames function returns the number of color tables that have been defined.

<div align="center">

Example:

</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
p = PseudocolorAttributes()
p.colorTableName = "default"
SetPlotOptions(p)
```

<div align="right">

(continues on next page)

</div>

```
DrawPlots()
print "There are %d color tables." % NumColorTableNames()
for ct in ColorTableNames():
SetActiveContinuousColorTable(ct)
SaveWindow()
```

2.4.147 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()
```

2.4.148 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()
```

2.4.149 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

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()
```

2.4.150 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")
```

(continues on next page)

```
OpenDatabase("file.visit")
OpenDatabase("file.visit", 4)
OpenDatabase("mcr:/usr/gapps/visit/data/multi_ucd3d.silo",0,"Silo_1.0")
```

2.4.151 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:

```
-assume_format PDB
% visit -cli
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()
```

2.4.152 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
```

2.4.153 OverlayDatabase

Synopsis:

```
OverlayDatabase(databaseName) -> integer
OverlayDatabase(databaseName, state) -> integer
```

databaseName [string] The name of the new plot database.

state 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")
```

2.4.154 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.

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()
```

2.4.155 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.

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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
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()
```

2.4.156 PickByNode

<div align="center">

Synopsis:

</div>

```
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.

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.

<div align="center">

Description:

</div>

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()
```

2.4.157 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.

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()
```

2.4.158 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.

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 PickBy-Zone 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()
```

2.4.159 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.

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()
```

2.4.160 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
```

2.4.161 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.

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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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))
```

2.4.162 PrintWindow

<div align="center">

Synopsis:

</div>

```
PrintWindow() -> integer
```

return type [CLI_return_t] The PrintWindow function returns an integer value of 1 for success and 0 for failure.

<div align="center">

Description:

</div>

The PrintWindow function tells the viewer to print the image in the active visualization window using the current printer settings.

<div align="center">

Example:

</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Pseudocolor", "d")
AddPlot("Contour", "u")
DrawPlots()
PrintWindow()
```

2.4.163 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()
```

2.4.164 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.

2.4.165 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()
```

2.4.166 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)
```

2.4.167 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)
```

2.4.168 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 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 QueryOverTime method, or named arguments can be used instead.

<div align="center">

Example:

</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()
for q in QueriesOverTime():
QueryOverTime(q)
ResetView()
```

2.4.169 ReOpenDatabase

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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()
```

2.4.170 ReadHostProfilesFromDirectory

<div align="center">

Synopsis:

</div>

```
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)
```

2.4.171 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()
```

2.4.172 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")
```

(continues on next page)

```
DrawPlots()
v = GetView2D()
v.windowCoords = (-2.3,2.4,0.2,4.9)
SetView2D(v)
UndoView()
RedoView()
```

2.4.173 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()
```

2.4.174 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:

```python
import visit
def print_sliceatts(atts):
print "SLICEATTS=", atts
visit.RegisterCallback("SliceAttributes", print_sliceatts)
```

2.4.175 RegisterMacro

<div align="center">

Synopsis:

</div>

```
RegisterMacro(name, callable)
```

name [string] The name of the macro.

callable [python function] A Python function that will be associated with the macro name.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```python
def SetupMyPlots():
OpenDatabase('noise.silo')
AddPlot('Pseudocolor', 'hardyglobal')
DrawPlots()
RegisterMacro('Setup My Plots', SetupMyPlots)
```

2.4.176 RemoveAllOperators

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```python
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddOperator("Threshold")
```

<div align="right">(continues on next page)</div>

```
AddOperator("Slice")
AddOperator("SphereSlice")
DrawPlots()
RemoveLastOperator() # Remove SphereSlice
RemoveOperator(0) # Remove Threshold
RemoveAllOperators() # Remove the rest of the operators
```

2.4.177 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
```

2.4.178 RemoveMachineProfile

Synopsis:

```
RemoveMachineProfile(hostname) -> integer
```

hostname : string

Description:

Removes machine profile with hostname from HostProfileList

2.4.179 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
```

2.4.180 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')
```

2.4.181 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')
```

2.4.182 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)
```

2.4.183 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.

2.4.184 ResetOperatorOptions

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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")
```

2.4.185 ResetPickLetter

<div align="center">

Synopsis:

</div>

```
ResetPickLetter() -> integer
```

return type [CLI_return_t] ResetPickLetter returns 1 on success and 0 on failure.

<div align="center">

Description:

</div>

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.

2.4.186 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")
```

2.4.187 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()
```

2.4.188 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)
```

2.4.189 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()
```

2.4.190 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()
```

2.4.191 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)

```
SaveAttribute('mesh.xml', a)
b = MeshPlotAttributes()
LoadAttribute('mesh.xml', b)
```

2.4.192 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")
```

2.4.193 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")
```

2.4.194 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
```

2.4.195 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.

<div align="center">

Description:
</div>

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.

2.4.196 SetActiveContinuousColorTable

<div align="center">

Synopsis:
</div>

```
SetActiveContinuousColorTable(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.

<div align="center">

Description:
</div>

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).

<div align="center">

Example:
</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Contour", "hgslice")
DrawPlots()
SetActiveDiscreteColorTable("levels")
```

2.4.197 SetActiveDiscreteColorTable

<div align="center">

Synopsis:
</div>

```
SetActiveDiscreteColorTable(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.

<div align="center">Description:</div>

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).

<div align="center">Example:</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Contour", "hgslice")
DrawPlots()
SetActiveDiscreteColorTable("levels")
```

2.4.198 SetActivePlots

<div align="center">Synopsis:</div>

```
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.

<div align="center">Description:</div>

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.

<div align="center">Example:</div>

```
#% 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
```

2.4.199 SetActiveTimeSlider

<div align="center">Synopsis:</div>

```
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.

<div align="center">

Description:

</div>

Sets the active time slider, which is the time slider that is used to change time states.

<div align="center">

Example:

</div>

```
#% 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)
```

2.4.200 SetActiveWindow

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% visit -cli
SetWindowLayout(2)
SetActiveWindow(2)
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Mesh", "mesh1")
DrawPlots()
```

2.4.201 SetAnimationTimeout

<div align="center">

Synopsis:

</div>

```
SetAnimationTimeout(milliseconds) -> integer
```

return type [CLI_return_t] 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
```

2.4.202 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)
```

2.4.203 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()
```

2.4.204 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.
```

2.4.205 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)
```

2.4.206 SetCreateMeshQualityExpressions

Synopsis:

```
SetCreateMeshQualityExpressions(val) -> integer
```

val [integer] Either a zero (false) or non-zero (true) integer value to indicate ifMesh 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
```

2.4.207 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
```

2.4.208 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
```

2.4.209 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)
```

2.4.210 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()
```

2.4.211 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)
```

2.4.212 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")
```

2.4.213 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)
```

2.4.214 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)

```
m.forceMIR = 1
SetMaterialAttributes(m)
ClearWindow()
# Redraw the plot forcing VisIt to use the mixed variable information.
DrawPlots()
```

2.4.215 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()
```

2.4.216 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()
```

2.4.217 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))
```

2.4.218 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()
```

2.4.219 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))
```

2.4.220 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)
```

2.4.221 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)
```

2.4.222 SetLight

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

>   The SetLight function sets the attributes for a specific light.

<div align="center">

Example:

</div>

```
#% 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)
```

2.4.223 SetMachineProfile

<div align="center">

Synopsis:

</div>

```
SetMachineProfile(MachineProfile) -> integer
```

MachineProfile [MachineProfile object] A MachineProfile object containing the new settings.

<div align="center">

Description:

</div>

>   Sets the input machine profile in the HostProfileList, replaces if one already exists Otherwise adds to the
>   list

2.4.224 SetMaterialAttributes

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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()
```

2.4.225 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()
```

2.4.226 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)
```

2.4.227 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")
```

```
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()
```

2.4.228 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))
```

2.4.229 SetPipelineCachingMode

Synopsis:

```
SetPipelineCachingMode(mode) -> integer
```

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)
```

2.4.230 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)
```

2.4.231 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')
```

2.4.232 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()
```

2.4.233 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
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
```

(continues on next page)

```
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()
```

2.4.234 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()
```

2.4.235 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.

<div align="center">**Description:**</div>

Move the i'th plot in the plot list to the start of the plot list.

<div align="center">**Example:**</div>

```
AddPlot('Mesh', 'mesh')
AddPlot('Pseudocolor', 'pressure')
# Make the Pseudocolor plot first in the plot list
SetPlotOrderToFirst(1)
```

2.4.236 SetPlotOrderToLast

<div align="center">**Synopsis:**</div>

```
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.

<div align="center">**Description:**</div>

Move the i'th plot in the plot list to the end of the plot list.

<div align="center">**Example:**</div>

```
AddPlot('Mesh', 'mesh')
AddPlot('Pseudocolor', 'pressure')
# Make the Mesh plot last in the plot list
SetPlotOrderToLast(0)
```

2.4.237 SetPlotSILRestriction

<div align="center">**Synopsis:**</div>

```
SetPlotSILRestriction(silr) -> integer
SetPlotSILRestriction(silr, all) -> integer
```

silr [SIL restriction object] A SIL restriction object.

all An optional argument that tells the function if the SIL restriction should be applied to all plots in the plot list.

return type [CLI_return_t] The SetPlotSILRestriction function returns an integer value of 1 for success and 0 for failure.

<div align="center">**Description:**</div>

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.

<div align="center">**Example:**</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/multi_curv2d.silo")
AddPlot("Subset", "mat1")
silr = SILRestriction()
silr.TurnOffSet(silr.SetsInCategory('mat1')[1])
SetPlotSILRestriction(silr)
DrawPlots()
```

2.4.238 SetPrecisionType

<div align="center">

Synopsis:

</div>

```
SetPrecisionType(typeAsInt)
SetPrecisionType(typeAsString)
```

typeAsInt [double] Precision type specified as an integer. 0 = float 1 = native 2 = double

typeAsString [string] Precision type specified as a string. Options are 'float', 'native', and 'double'.

<div align="center">

Description:

</div>

> The SetPrecisionType function sets the floating point pecision 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"

<div align="center">

Example:

</div>

```
SetPrecisionType("double")
SetPrecisionType(2)
```

2.4.239 SetPreferredFileFormats

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

> 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.

<div align="center">

Example:

</div>

```
SetPreferredFileFormats('Silo_1.0')
SetPreferredFileFormats(('Silo_1.0','PDB_1.0'))
```

2.4.240 SetPrinterAttributes

<div align="center">

Synopsis:

</div>

```
SetPrinterAttributes(atts)
```

atts [PrinterAttributes object] A PrinterAttributes object.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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()
```

2.4.241 SetQueryFloatFormat

<div align="center">

Synopsis:

</div>

```
SetQueryFloatFormat(format_string)
```

format_string [string] A string object that provides a printf style floating point format.

<div align="center">

Description:

</div>

The SetQueryFloatFormat method sets a printf style format string that isused by VisIt's querys to produce textual output.

<div align="center">

Example:

</div>

```
#% 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")
```

2.4.242 SetQueryOutputToObject

<div align="center">

Synopsis:

</div>

```
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
```

2.4.243 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>)
'
```

2.4.244 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)
```

2.4.245 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()
```

2.4.246 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
```

2.4.247 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)
```

2.4.248 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()
```

2.4.249 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)

```
tsNames = GetWindowInformation().timeSliders
for ts in tsNames:
SetActiveTimeSlider(ts)
for state in list(range(TimeSliderGetNStates())) + [0]:
SetTimeSliderState(state)
```

2.4.250 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
```

2.4.251 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
```

2.4.252 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.

2.4.253 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.
```

2.4.254 SetView3D

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
% 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.
```

2.4.255 SetViewAxisArray

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.
```

2.4.256 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.
```

2.4.257 SetViewExtentsType

<div align="center">Synopsis:</div>

```
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.

<div align="center">Description:</div>

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.

<div align="center">Example:</div>

```
#% 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")
```

2.4.258 SetViewKeyframe

<div align="center">Synopsis:</div>

```
SetViewKeyframe() -> integer
```

return type [CLI_return_t] The SetViewKeyframe function returns 1 on success and 0 on failure.

<div align="center">Description:</div>

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.

<div align="center">Example:</div>

```
#% 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()
```

2.4.259 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)
```

2.4.260 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
```

2.4.261 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
```

2.4.262 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()
```

2.4.263 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)
```

2.4.264 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")
```

2.4.265 SuppressMessages

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

The SuppressMessage function sets the supression 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.

<div align="center">

Example:

</div>

```
#% visit -cli
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
# Turn off Warning and Message messages.
SuppressMessages(2)
SaveWindow()
```

2.4.266 SuppressQueryOutputOff

<div align="center">

Synopsis:

</div>

```
SuppressQueryOutputOff() -> integer
```

return type [CLI_return_t] The SuppressQueryOutput function returns 1 on success and 0 on failure.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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")
```

2.4.267 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")
```

2.4.268 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()
```

2.4.269 TimeSliderNextState

Synopsis:

```
TimeSliderNextState() -> integer
```

return type [CLI_return_t] The TimeSliderNextState function returns 1 on success and 0 on failure.

<div align="center">

Description:

</div>

The TimeSliderNextState function advances the active time slider to the next time slider state.

<div align="center">

Example:

</div>

```
# 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()
```

2.4.270 TimeSliderPreviousState

<div align="center">

Synopsis:

</div>

```
TimeSliderPreviousState() -> integer
```

return type [CLI_return_t] The TimeSliderPreviousState function returns 1 on success and 0 on failure.

<div align="center">

Description:

</div>

The TimeSliderPreviousState function moves the active time slider to the previous time slider state.

<div align="center">

Example:

</div>

```
# 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()
```

2.4.271 TimeSliderSetState

<div align="center">

Synopsis:

</div>

```
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.

<div align="center">

Description:

</div>

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.

<div align="center">

Example:

</div>

```
#% 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)
```

2.4.272 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()
```

2.4.273 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.

<div align="center">Example:</div>

```
#% 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()
```

2.4.274 ToggleFullFrameMode

<div align="center">Synopsis:</div>

```
ToggleFullFrameMode() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

<div align="center">Description:</div>

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 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.

<div align="center">Example:</div>

```
#% 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()
```

2.4.275 ToggleLockTime

<div align="center">Synopsis:</div>

```
ToggleLockTime() -> integer
```

return type [CLI_return_t] All functions return 1 on success and 0 on failure.

<div align="center">Description:</div>

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()
```

2.4.276 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()
```

2.4.277 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()
```

2.4.278 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()
```

2.4.279 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()
```

2.4.280 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 Turn functions are provided to simplify the removal of material or domain subsets. Instead of creating a SILRestriction object, you can use the Turn functions to turn materials or domains on or off. The

TurnDomainsOff function turns domains off. All of the Turn functions have three possible argument lists. When you do not provide any arguments, the function applies to all subsets in the SIL so if you called the TurnDomainsOff function with no arguments, all domains would be turned off. All functions can also take a string argument, which is the name of the set to modify. For example, you could turn off domain 0 by calling the TurnDomainsOff with a single argument of "domain0" (or the appropriate set name). All of the Turn functions can also be used to modify more than one set if you provide a tuple of set names. After you use the Turn functions to change the SIL restriction, you might want to call the ListMaterials or ListDomains functions to make sure that the SIL restriction was actually modified.

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
```

2.4.281 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 Turn functions are provided to simplify the removal of material or domain subsets. Instead of creating a SILRestriction object, you can use the Turn functions to turn materials or domains on or off. The TurnDomainsOn function turns domains on. All of the Turn functions have three possible argument lists. When you do not provide any arguments, the function applies to all subsets in the SIL so if you called the TurnDomainsOn function with no arguments, all domains would be turned on. All functions can also take a string argument, which is the name of the set to modify. For example, you could turn on domain 0 by calling the TurnDomainsOn with a single argument of "domain0" (or the appropriate set name). All of the Turn functions can also be used to modify more than one set if you provide a tuple of set names. After you use the Turn functions to change the SIL restriction, you might want to call the ListMaterials or ListDomains functions to make sure that the SIL restriction was actually modified.

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
```

2.4.282 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 Turn functions are provided to simplify the removal of material or domain subsets. Instead of creating a SILRestriction object, you can use the Turn functions to turn materials or domains on or off. The TurnMaterialsOff function turns materials off. All of the Turn functions have three possible argument lists. When you do not provide any arguments, the function applies to all subsets in the SIL so if you called the TurnMaterialsOff function with no arguments, all materials would be turned off. All functions can also take a string argument, which is the name of the set to modify. For example, you could turn off material 0 by calling TurnMaterialsOff with a single argument of "material0" (or the appropriate set name). All of the Turn functions can also be used to modify more than one set if you provide a tuple of set names. After you use the Turn functions to change the SIL restriction, you might want to call the ListMaterials or ListDomains functions to make sure that the SIL restriction was actually modified.

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
```

2.4.283 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 Turn functions are provided to simplify the removal of material or domain subsets. Instead of creating a SILRestriction object, you can use the Turn functions to turn materials or domains on or off. The TurnMaterialsOn function turns materials on. All of the Turn functions have three possible argument lists. When you do not provide any arguments, the function applies to all subsets in the SIL so if you called the TurnMaterialsOn function with no arguments, all materials would be turned off. All functions can also take a string argument, which is the name of the set to modify. For example, you could turn on

material 0 by calling the TurnMaterialsOn with a single argument of "material0" (or the appropriate set name). All of the Turn functions can also be used to modify more than one set if you provide a tuple of set names. After you use the Turn functions to change the SIL restriction, you might want to call the ListMaterials or ListDomains functions to make sure that the SIL restriction was actually modified.

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
```

2.4.284 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()
```

2.4.285 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)
```

2.4.286 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()
```

2.4.287 WriteConfigFile

Synopsis:

```
WriteConfigFile()
```

Description:

The viewer maintains internal settings which determine the default values for objects like plots and op-
erators. 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()
```

2.4.288 WriteScript

Example:

```
f = open('script.py', 'wt')
WriteScript(f)
f.close()
```

2.4.289 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.

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)
```

(continues on next page)

```
# Perform zone pick in world space.
pick_out = ZonePick(coord = (-5.0, 5.0, 0))
```

2.5 Attribute   Reference

This chapter shows all the attributes that can be set to control the behavior of VisIt. The attributes themselves are not documented, but their names are usually quite explanatory. When a member of an attribute can take values from a given list of options, the default option is printed first in italic followed by a comma separated list of the other available options.

The listing is ordered in alphabetical ordering of the name of the attribute set. For each set the function that will provide you with these attributes is printed in italic.

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. | |

2.5.1 AMRStitchCell: *AMRStitchCellAttributes()*

Attribute	**Default**/Allowed Values
CreateCellsOfType	**DualGridAndStitchCells**, DualGrid, StitchCells

2.5.2 Animation: *AnimationAttributes()*

Attribute	**Default**/Allowed Values
animationMode	**StopMode**, ReversePlayMode, PlayMode
pipelineCachingMode	0
frameIncrement	1
timeout	1
playbackMode	**PlayOnce**, Looping, Swing

2.5.3 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"
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	""

Continued on next page

Table 2.1 – continued from previous page

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
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

Table 2.1 – continued from previous page

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
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)

Continued on next page

Table 2.1 – continued from previous page

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
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

Table 2.1 – continued from previous page

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

2.5.4 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

2.5.5 AxisAlignedSlice4D: *AxisAlignedSlice4DAttributes()*

Attribute	**Default**/Allowed Values
I	()
J	()
K	()
L	()

2.5.6 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

2.5.7 BoundaryOp: *BoundaryOpAttributes()*

Attribute	**Default**/Allowed Values
smoothingLevel	0

2.5.8 Box: *BoxAttributes()*

Attribute	**Default**/Allowed Values
amount	**Some**, All
minx	0
maxx	1
miny	0
maxy	1
minz	0
maxz	1
inverse	0

2.5.9 CartographicProjection: *CartographicProjectionAttributes()*

Attribute	**Default**/Allowed Values
projectionID	**aitoff**, eck4, eqdc, hammer, laea, lcc, merc, mill, moll, ortho, wink2
centralMeridian	0

2.5.10 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

2.5.11 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

2.5.12 ConnectedComponents: *ConnectedComponentsAttributes()*

Attribute	**Default**/Allowed Values
EnableGhostNeighborsOptimization	1

2.5.13 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

2.5.14 Contour: *ContourAttributes()*

Attribute	**Default**/Allowed Values
defaultPalette.GetControlPoints(0).colors	(255, 0, 0, 255)
defaultPalette.GetControlPoints(0).position	0

Table 2.2 – continued from previous page

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)
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

Table 2.2 – continued from previous page

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.categoryName	"Standard"
changedColors	()
colorType	**ColorByMultipleColors**, ColorBySingleColor, ColorByColorTable
colorTableName	"Default"
invertColorTable	0
legendFlag	1
lineWidth	0
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))
	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))
contourNLevels	10
contourValue	()
contourPercent	()
contourMethod	**Level**, Value, Percent
minFlag	0
maxFlag	0
min	0
max	1
scaling	**Linear**, Log
wireframe	0

2.5.15 CoordSwap: *CoordSwapAttributes()*

Attribute	**Default**/Allowed Values
newCoord1	**Coord1**, Coord2, Coord3
newCoord2	**Coord2**, Coord1, Coord3
newCoord3	**Coord3**, Coord1, Coord2

2.5.16 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)

2.5.17 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

2.5.18 Cylinder: *CylinderAttributes()*

Attribute	**Default**/Allowed Values
point1	(0, 0, 0)
point2	(1, 0, 0)
radius	1
inverse	0

2.5.19 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

2.5.20 DeferExpression: *DeferExpressionAttributes()*

Attribute	**Default**/Allowed Values
exprs	()

2.5.21 Displace: *DisplaceAttributes()*

Attribute	Default/Allowed Values
factor	1
variable	"default"

2.5.22 DualMesh: *DualMeshAttributes()*

Attribute	Default/Allowed Values
mode	**Auto**, NodesToZones, ZonesToNodes

2.5.23 Edge: *EdgeAttributes()*

Attribute	Default/Allowed Values
dummy	1

2.5.24 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"

2.5.25 EllipsoidSlice: *EllipsoidSliceAttributes()*

Attribute	**Default**/Allowed Values
origin	(0, 0, 0)
radii	(1, 1, 1)
rotationAngle	(0, 0, 0)

2.5.26 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.

2.5.27 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
opts.types	()
opts.help	""

2.5.28 ExternalSurface: *ExternalSurfaceAttributes()*

Attribute	Default/Allowed Values
removeGhosts	0
edgesIn2D	1

2.5.29 Extrude: *ExtrudeAttributes()*

Attribute	Default/Allowed Values
axis	(0, 0, 1)
byVariable	0
variable	"default"
length	1
steps	1
preserveOriginalCellNumbers	1

2.5.30 FFT: *FFTAttributes()*

Attribute	Default/Allowed Values
dummy	0

2.5.31 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

2.5.32 Flux: *FluxAttributes()*

Attribute	**Default**/Allowed Values
flowField	"default"
weight	0
weightField	"default"

2.5.33 Font: *FontAttributes()*

Attribute	**Default**/Allowed Values
font	**Arial**, Courier, Times
scale	1
useForegroundColor	1
color	(0, 0, 0, 255)
bold	0
italic	0

2.5.34 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

2.5.35 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

2.5.36 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"

2.5.37 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, 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, NIMR
fieldConstant	1
velocitySource	(0, 0, 0)
integrationType	**DormandPrince**, Euler, Leapfrog, AdamsBashforth, RK4, M3DC12DIntegrator
parallelizationAlgorithmType	**VisItSelects**, LoadOnDemand, ParallelStaticDomains, MasterSlave
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
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	""

2.5.38 InverseGhostZone: *InverseGhostZoneAttributes()*

Attribute	**Default**/Allowed Values
requestGhostZones	1
showDuplicated	1
showEnhancedConnectivity	1
showReducedConnectivity	1
showAMRRefined	1
showExterior	1
showNotApplicable	1

2.5.39 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"

2.5.40 Isovolume: *IsovolumeAttributes()*

Attribute	**Default**/Allowed Values
lbound	-1e+37
ubound	1e+37
variable	"default"

2.5.41 Keyframe: *KeyframeAttributes()*

Attribute	**Default**/Allowed Values
enabled	0
nFrames	1
nFramesWasUserSet	0

2.5.42 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
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, NIMRO
fieldConstant	1
velocitySource	(0, 0, 0)
integrationType	**DormandPrince**, Euler, Leapfrog, AdamsBashforth, RK4, M3DC12DIntegrator
clampLogValues	0
parallelizationAlgorithmType	**VisItSelects**, LoadOnDemand, ParallelStaticDomains, MasterSlave
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

Table 2.4 – continued from previous page

boundaryLimit	0.1
seedLimit	10
issueAdvectionWarnings	1
issueBoundaryWarnings	1
issueTerminationWarnings	1
issueStepsizeWarnings	1
issueStiffnessWarnings	1
issueCriticalPointsWarnings	1
criticalPointThreshold	0.001

2.5.43 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"

2.5.44 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"

2.5.45 Light: *LightAttributes()*

Attribute	Default/Allowed Values
enabledFlag	1
type	**Camera**, Ambient, Object
direction	(0, 0, -1)
color	(255, 255, 255, 255)
brightness	1

2.5.46 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

Table 2.5 – continued from previous page

fieldType	**Default**, FlashField, M3DC12DField, M3DC13DField, Nek5000Field, NektarPPField, NIMR
fieldConstant	1
velocitySource	(0, 0, 0)
integrationType	**DormandPrince**, Euler, Leapfrog, AdamsBashforth, RK4, M3DC12DIntegrator
parallelizationAlgorithmType	**VisItSelects**, LoadOnDemand, ParallelStaticDomains, MasterSlave
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
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

2.5.47 Lineout: *LineoutAttributes()*

Attribute	**Default**/Allowed Values
point1	(0, 0, 0)
point2	(1, 1, 0)
interactive	0
ignoreGlobal	0
samplingOn	0
numberOfSamplePoints	50
reflineLabels	0

2.5.48 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

2.5.49 Mesh: *MeshAttributes()*

Attribute	**Default**/Allowed Values
legendFlag	1
lineWidth	0
meshColor	(0, 0, 0, 255)
meshColorSource	**Foreground**, MeshCustom
opaqueColorSource	**Background**, OpaqueCustom
opaqueMode	**Auto**, On, Off
pointSize	0.05
opaqueColor	(255, 255, 255, 255)
smoothingLevel	**None**, Fast, High
pointSizeVarEn-abled	0
pointSizeVar	"default"
pointType	**Point**, Box, Axis, Icosahedron, Octahedron, Tetrahedron, SphereGeometry, Sphere
showInternal	0
pointSizePixels	2
opacity	1

2.5.50 MeshManagement: *MeshManagementAttributes()*

Attribute	**Default**/Allowed Values
discretizationTolerance	(0.02, 0.025, 0.05)
discretizationToleranceX	()
discretizationToleranceY	()
discretizationToleranceZ	()
discretizationMode	**Uniform**, Adaptive, MultiPass
discretizeBoundaryOnly	0
passNativeCSG	0

2.5.51 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

2.5.52 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

Table 2.6 – continued from previous page

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)
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.categoryName	"Standard"

Continued on next page

Table 2.6 – continued from previous page

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))
	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))
lineWidth	0
yAxisTitleFormat	"%g"
useYAxisTickSpacing	0
yAxisTickSpacing	1
displayMarkers	1
markerScale	1
markerLineWidth	0
markerVariable	"default"
displayIds	0
idVariable	"default"
legendFlag	1

2.5.53 MultiresControl: *MultiresControlAttributes()*

Attribute	**Default**/Allowed Values
resolution	0
maxResolution	1
info	""

2.5.54 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

2.5.55 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

2.5.56 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"

2.5.57 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, NIMRC
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)

Table 2.7

colorTableName	"Default"
dataValue	**SafetyFactorQ**, Solid, SafetyFactorP, SafetyFactorQ_NotP, SafetyFactorP_NotQ, ToroidalWin
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, MasterSlave
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

2.5.58 Printer: *PrinterAttributes()*

Attribute	**Default**/Allowed Values
printerName	""
printProgram	"lpr"
documentName	"untitled"
creator	""
numCopies	1
portrait	1
printColor	1
outputToFile	0
outputToFileName	"untitled"
pageSize	2

2.5.59 Process: *ProcessAttributes()*

Attribute	**Default**/Allowed Values
pids	()
ppids	()
hosts	()
isParallel	0
memory	()
times	()

2.5.60 Project: *ProjectAttributes()*

Attribute	**Default**/Allowed Values
projectionType	**XYCartesian**, ZYCartesian, XZCartesian, XRCylindrical, YRCylindrical, ZRCylindrical
vectorTransform-Method	**AsDirection**, None, AsPoint, AsDisplacement

2.5.61 Pseudocolor: *PseudocolorAttributes()*

Attribute	**Default**/Allowed Values
scaling	**Linear**, Log, Skew
skewFactor	1
limitsMode	**OriginalData**, CurrentPlot
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	"hot"
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
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

Table 2.8 – continued from previous page

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)

2.5.62 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

2.5.63 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

2.5.64 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

2.5.65 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
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

2.5.66 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)

2.5.67 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

2.5.68 Revolve: *RevolveAttributes()*

Attribute	**Default**/Allowed Values
meshType	**Auto**, XY, RZ, ZR
autoAxis	1
axis	(1, 0, 0)
startAngle	0
stopAngle	360
steps	30

2.5.69 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

2.5.70 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
subWindowAtts.win6.omitWindow	0
subWindowAtts.win7.position	(0, 0)

Table 2.10 – continued from previous page

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
subWindowAtts.win16.omitWindow	0
opts.types	()

Continu

Table 2.10 – continued from previous page

| opts.help | "" |

2.5.71 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

Continued on next page

Table 2.11 – continued from previous page

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
legendFlag	1

2.5.72 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

2.5.73 SmoothOperator: *SmoothOperatorAttributes()*

Attribute	**Default**/Allowed Values
numIterations	20
relaxationFactor	0.01
convergence	0
maintainFeatures	1
featureAngle	45
edgeAngle	15
smoothBoundaries	0

2.5.74 SphereSlice: *SphereSliceAttributes()*

Attribute	**Default**/Allowed Values
origin	(0, 0, 0)
radius	1

2.5.75 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	()

2.5.76 Stagger: *StaggerAttributes()*

Attribute	**Default**/Allowed Values
offsetX	0
offsetY	0
offsetZ	0

2.5.77 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

2.5.78 SubdivideQuads: *SubdivideQuadsAttributes()*

Attribute	**Default**/Allowed Values
threshold	0.500002
maxSubdivs	4
fanOutPoints	1
doTriangles	0
variable	"default"

2.5.79 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
pointSizeVarEn-abled	0
pointSizeVar	"default"
pointSizePixels	2

2.5.80 SurfaceNormal: *SurfaceNormalAttributes()*

Attribute	**Default**/Allowed Values
centering	**Point**, Cell

2.5.81 Tensor: *TensorAttributes()*

Attribute	**Default**/Allowed Values
useStride	0
stride	1
nTensors	400
scale	0.25
scaleByMagnitude	1
autoScale	1
colorByEigenvalues	1
useLegend	1
tensorColor	(0, 0, 0, 255)
colorTableName	"Default"
invertColorTable	0

2.5.82 ThreeSlice: *ThreeSliceAttributes()*

Attribute	**Default**/Allowed Values
x	0
y	0
z	0
interactive	1

2.5.83 Threshold: *ThresholdAttributes()*

Attribute	**Default**/Allowed Values
outputMeshType	0
boundsInputType	0
listedVarNames	("default")
zonePortions	()
lowerBounds	()
upperBounds	()
defaultVarName	"default"
defaultVarIsScalar	0
boundsRange	()

2.5.84 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
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

Continued on next page

<div align="center">Table 2.12 – continued from previous page</div>

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

2.5.85 TriangulateRegularPoints: *TriangulateRegularPointsAttributes()*

Attribute	**Default**/Allowed Values
useXGridSpacing	0
xGridSpacing	1
useYGridSpacing	0
yGridSpacing	1

2.5.86 Truecolor: *TruecolorAttributes()*

Attribute	**Default**/Allowed Values
opacity	1
lightingFlag	1

2.5.87 Tube: *TubeAttributes()*

Attribute	**Default**/Allowed Values
scaleByVarFlag	0
tubeRadiusType	**FractionOfBBox**, Absolute
radiusFractionBBox	0.01
radiusAbsolute	1
scaleVariable	"default"
fineness	5
capping	0

2.5.88 Vector: *VectorAttributes()*

Attribute	**Default**/Allowed Values
glyphLocation	**AdaptsToMeshResolution**, UniformInSpace
useStride	0
stride	1
nVectors	400
lineWidth	0
scale	0.25
scaleByMagnitude	1
autoScale	1
headSize	0.25
headOn	1
colorByMag	1
useLegend	1
vectorColor	(0, 0, 0, 255)
colorTableName	"Default"
invertColorTable	0
vectorOrigin	**Tail**, Head, Middle
minFlag	0
maxFlag	0
limitsMode	**OriginalData**, CurrentPlot
min	0
max	1
lineStem	**Line**, Cylinder
geometryQuality	**Fast**, High
stemWidth	0.08
origOnly	1
glyphType	**Arrow**, Ellipsoid
animationStep	0

2.5.89 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

2.5.90 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

2.5.91 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

2.5.92 ViewAxisArray: *ViewAxisArrayAttributes()*

Attribute	**Default**/Allowed Values
domainCoords	(0, 1)
rangeCoords	(0, 1)
viewportCoords	(0.15, 0.9, 0.1, 0.85)

2.5.93 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

2.5.94 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.categoryName	""
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)

2.6 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

Table 2.14 – continued from previous page

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
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

Table 2.14 – continued from previous page

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
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

Continued on next page

Table 2.14 – continued from previous page

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
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

Continued on next page

Table 2.14 – continued from previous page

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
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

Continued on next page

Table 2.14 – continued from previous page

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
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

Table 2.14 – continued from previous page

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
StaggerAttributes	1
StartPlotAnimationRPC	1
StatisticalTrendsAttributes	1
StopPlotAnimationRPC	1
SubdivideQuadsAttributes	1
SubsetAttributes	1
SuppressQueryOutputRPC	1
SurfaceNormalAttributes	1
TensorAttributes	1
ThreeSliceAttributes	1
ThresholdAttributes	1
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

Table 2.14 – continued from previous page

UpdateNamedSelectionRPC	1
VectorAttributes	1
View2DAttributes	1
View3DAttributes	1
ViewCurveAttributes	1
VolumeAttributes	1
WindowInformation	1
WriteConfigFileRPC	0

2.7 Contributing To VisIt CLI Documentation

Note: This procedure is planned for change in the future.

At present, all VisIt Python CLI documentation is actually composed directly as Python strings in the source C++ file in `../../visitpy/common/MethodDoc.C`.

We recognize this isn't the most convenient way to write documentation and are planning on changing it in the future. However, it does permit us to have a single source file for documentation which is then used to provide `help(func)` at the Python prompt as well as generate the restructured text used here.

In the future, we will swap this arrangement and write documentation in restructured text and then generate the contents of `../../visitpy/common/MethodDoc.C` from the restructured text.

The documentation here is then generated from the `MethodDoc.C` file using the script `../sphinx_cli_extractor.py`. That script produces `attributes.rst`, `events.rst` and `functions.rst` files. The other `.rst` files here are manually managed and can be modified normally as needed.

2.7.1 Steps to update the CLI Manual

1. Modify `../../visitpy/common/MethodDoc.C` as needed

2. Build and run the VisIt cli and assure yourself `help(<your-new-func-doc>)` produces the desired output

3. Run the `sphinx_cli_extractor.py` tool producing new `attributes.rst`, `events.rst` and `functions.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 `PTHONHOME` 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 OSX 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=../../build/third_party/python/2.7.14/i386-apple-darwin17_clang/bin:../..
↪/build/visit/build/bin:$PATH \
PYTHONPATH=../../build/visit/build/lib/site-packages/ \
./sphinx_cli_extractor.py
```

The whole process only takes a few seconds.

4. Assuming you succesfully run the above command, producing new `attributes.rts`, `events.rst` and `functions.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
```

5. Then open the file, `_build/index.html`, in your favorite browser to view.

6. Add all the changed files to a commit and push to GitHub

7. 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.

2.8 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.

VisIt Developer Manual

3.1 Creating a Pull Request

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.1.4 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.

3.1.5 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.

3.2 Reviewing   a   Pull   Request

3.2.1 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.

3.2.2 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

- New and existing unit tests pass

- If necessary, the developer added any new baselines to the repository

3.2.3 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.

3.2.4 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.

3.2.5 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.

Index