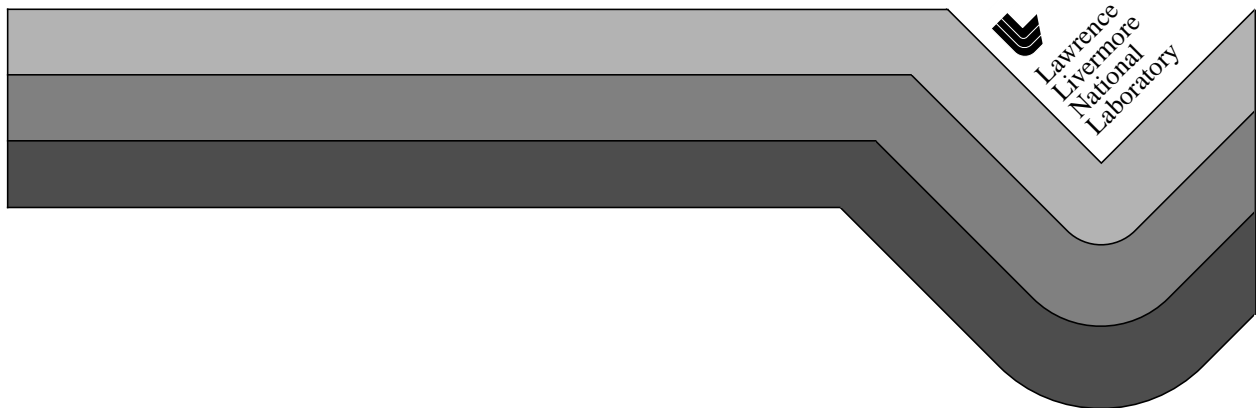


Getting Data Into VisIt

September 2006

Version 1.5.4



DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express 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 the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

Table of Contents

Introduction

Manual chapters	2
Manual conventions	2
Strategies	2
Picking a strategy	3
Definition of terms	4

Creating compatible files

Creating a conversion utility or extending a simulation	7
Survey of database reader plug-ins	9
BOV file format	9
X-Y Curve file format	11
Writing Silo files	12
Using the Silo library	12
Inspecting Silo files	14
Silo files and parallel codes	14
Creating a new Silo file	15
Dealing with time	16
Option lists	17
Writing a rectilinear mesh	18
Writing a curvilinear mesh	21
Writing a point mesh	25
Writing an unstructured mesh	28
Writing a scalar variable	33
Single precision vs. Double precision	45
Writing expressions	45
Creating a master file for parallel	46
Writing VTK files	52
Getting started with visit_writer	53
Regular meshes with data	54
Rectilinear meshes with data	56
Curvilinear meshes with data	58
Point meshes with data	61
Unstructured meshes with data	62
Creating a master file for parallel (.visit file)	64

Creating compatible files II Advanced topics

Writing vector data	67
Adding metadata for performance boosts	70

Writing data extents	70
Writing spatial extents	73
Ghost zones	74
Writing ghost zones to your files	76
Materials	81
Creating a database reader plug-in	
Structure of VisIt	87
plug-ins	89
Starting your plug-in	90
Picking a database reader plug-in interface	90
Using XMLEdit	90
Generating a plug-in code skeleton	94
Building your plug-in	95
Calling your plug-in for the first time	97
Implementing your plug-in	98
Required plug-in methods	98
Debugging your plug-in	100
Opening your file	102
Returning file metadata	103
Returning a mesh	109
Returning a scalar variable	122
Returning a vector variable	123
Using a VTK reader class	125
Advanced topics	126
Returning cycles and times	126
Auxiliary data	130
Returning ghost zones	136
Parallelizing your reader	137
Instrumenting a simulation code	
Architecture	139
Using libsim	142
Getting libsim	142
Building in libsim support	142
Initialization	143
Restructuring the main loop	147
Using libsim in a Fortran simulation	153
Using libsim in a parallel Fortran simulation	155
Running an instrumented simulation	161
Connecting to an instrumented simulation from VisIt	161
Writing data access code	162
The VisIt Data Interface	162
How data access functions are called	163
Compiler and platform issues	164
Making data access functions available	164
Data access function for metadata	166

Data access function for meshes	177
Data access function for scalars	194
Data access function for curves	199
Data access function for the domain list	203

1.0 Overview

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 plug-in architecture allows it to perform a wide variety of plotting and data processing operations, and also allows VisIt to import data from many different data formats.

This manual explains in detail how to get your data into VisIt, concentrating on three main strategies: writing compatible files, writing a new plug-in for VisIt, or instrumenting a simulation code. In addition to providing the how-to's of getting your data into VisIt, this manual also presents reasons for why you might choose one strategy over another.

This manual is geared towards someone who wants to visualize and analyze data using VisIt. VisIt reads a large number of file formats so users of some existing simulation software will be able to use VisIt right away. This manual is for the user who has data files that VisIt does not read, or who wants to directly access data from a homegrown simulation code. Whichever the case, this manual assumes familiarity with computer programming since all of the covered approaches for getting data into VisIt require some programming. The examples in this manual are written primarily using the C and C++ programming languages, though relevant examples for the Fortran and Python languages are also included.

2.0 Manual chapters

This manual is broken down into the following chapters:

Chapter title	Chapter description
Introduction	This chapter.
Creating compatible files	Describes how to store data into file formats that VisIt already reads.
Creating compatible files II Advanced topics	Describes how to store metadata to boost VisIt's performance and also covers more exotic types of data that can be stored into file formats that VisIt already reads.
Creating a database reader plug-in	Describes how to create a new database reader plug-in for VisIt so it can read your own data file format.
Instrumenting simulation codes	Describes how to instrument your simulation code so VisIt can directly access its data without the need to write files.

3.0 Manual conventions

This manual uses the following conventions:

Element	All GUI elements, like windows, menus, and buttons will use bold helvetica .
Chapters	All references to other chapters will use Bold Times .
<i>Documents</i>	All document or file names will be <i>italicized</i> .

4.0 Strategies

Often, the first strategy to consider when trying to get your data into VisIt is creating data files using a data format that VisIt can already read. This is usually the simplest method for getting data into VisIt as it can be accomplished by adding a new I/O module to your simulation code or it can be achieved by creating an external data conversion utility.

Changing your simulation code to write out data that VisIt can read is sometimes not an option. For example, you might not have the simulation's source code or perhaps there is

too much risk involved in changing the source code. In addition, you might have gigabytes of archived data that you've written using your simulation's native data format and now you want to visualize that data in VisIt. If any of these cases apply to your situation then you might want to consider writing a database reader plug-in for VisIt so VisIt can natively understand your simulation code's data format.

If you want to maintain your current data format but you don't want to write a database reader plug-in for VisIt, you have another option: instrument the simulation code. VisIt provides a modestly sized library that contains C-Language functions that you can use to instrument your simulation code. When a simulation code is instrumented, VisIt can connect to it and access any of the arrays that you expose. This approach lets VisIt visualize the data from your simulation code directly without the need to write files.

5.0 Picking a strategy

The strategy you use to get your data into VisIt depends on your situation. The following table indicates reasons when you might pick one strategy over another.

Strategy	Reasons when to use
Create compatible files	<ul style="list-style-type: none"> • You have access to your simulation code's source code and one of VisIt's supported file formats can express your data. • You can write a conversion utility and don't mind using it to copy the existing data into a new data format.
Write a database reader plug-in	<ul style="list-style-type: none"> • You have written a lot of data files using your own data format or a format that VisIt does not read. • Changing the simulation's source code is not an option. • VisIt's supported file formats can't fully capture your data's structure or content. • Your data format is already supported in another visualization application.
Instrument simulation code	<ul style="list-style-type: none"> • You want to use VisIt to inspect your data as it is calculated. • You don't want to change your simulation code so it writes a different data format. • Your simulation code is written in the C, C++, or Fortran programming languages.

The following table indicates reasons why you would *not* pick one of the given strategies.

Strategy	Reason to <i>not</i> use
Create compatible files	<ul style="list-style-type: none"> •You don't want to change or are unable to change your simulation's source code •You don't want to replicate data in another data format, taking up more storage. •Your data format is already supported in another visualization application
Write a database reader plug-in	<ul style="list-style-type: none"> •Developing a VisIt database reader plug-in can be difficult, though this manual aims to lessen the difficulties. •You need to run VisIt on several platforms and you don't want to build the plug-in on all of those platforms. •You don't want to maintain a VisIt plug-in. Note that you could donate the plug-in to the VisIt development team.
Instrument simulation code	<ul style="list-style-type: none"> •You don't want to change or are unable to change your simulation's source code. •Your simulation code is not written in C, C++, or Fortran.

After examining the above tables, you probably have a pretty good idea of which strategy will work best for getting your data into VisIt. The following chapters will provide details on how best to get your data into VisIt using each of the recommended strategies.

6.0 Definition of terms

This section defines some of the terms that will be used to describe data structures that VisIt can visualize. These terms are defined here because many branches of science that might use VisIt to visualize and analyze data have their own terms. It is hoped that adding

the definition of terms here will reduce ambiguity when different types of data are covered in later chapters.

Term	Definition
Curvilinear mesh	A curvilinear mesh is a mesh composed entirely of quadrilateral or hexahedral cells. Furthermore, the mesh is constructed such that all zones exist in a logically contiguous brick having N_X zones in the X dimension, N_Y zones in the Y dimension, and in the case of 3-D: N_Z zones in the Z dimension. Each node in the mesh requires an explicitly provided coordinate value.
Domain	A domain is a unit of work that corresponds to a piece of the mesh that is handled by a given processor when running in parallel. Meshes are often split into multiple pieces, or domains, that can be assigned to different processors in order to handle larger simulations.
Ghost zone	A ghost zone is a zone on the boundaries of domains and it is used to ensure that each domain knows the data value on the other side of the domain boundary so operations requiring continuity do not give rise to discontinuities at domain boundaries.
Material	A physical material such as air or steel that is assigned to various zones in a mesh to indicate the types of materials that make up the simulated model. Zones that contain more than one material are said to be “mixed” since their compositions are determined by a set of volume fractions of various materials in the zone.
Mesh	A mesh is a structure composed of zones.
Node	A mathematical point. Nodes are used to describe the coordinates for zones that make up a mesh.
Node-centered	Node-centered is a term that applies to data stored on a mesh; it means that there is one data value for each node in the mesh and that values in the zone are created by interpolating data from the nodes.
Point mesh	A mesh consisting of a set of locations, or points, in space. These nodes are not connected.

Term	Definition
Rectilinear mesh	A rectilinear mesh is a mesh composed entirely of quadrilateral or hexahedral cells that are all the same shape. Furthermore, the mesh is constructed such that all zones exist in a contiguous brick having NX zones in the X dimension, NY zones in the Y dimension, and in the case of 3-D: NZ zones in the Z dimension. The coordinates for the nodes are supplied as lists of NX, NY, or NZ elements from which the full complement of nodes can be created.
Time step	Simulations proceed by calculating their state at the current time and then making adjustments that are needed to advance the state of the simulation to the next time. This is done in an iterative cycle. One iteration of the simulation is called a time step.
Unstructured mesh	An unstructured mesh consists of a set of nodes and a set of zones. The set of zones may consist of many different zone types such as triangles, quadrilaterals, tetrahedra, hexahedra, prisms, pyramids, or other polyhedra. Adjacent zones share the same nodes and the nodes are represented as a shape type identifier and a list of the nodes that comprise the zone.
Zone-centered	Zone-centered is a term that applies to data stored on a mesh; it means that there is one data value for each zone in the mesh.
Zone/Cell	Zone and Cell are used interchangeably in this document. A zone is a shape that unites one or more nodes into a connected structure where the nodes are the vertices of the connected structure. Point meshes can have nodes as zones. 1-D meshes contain zones that are lines that connect nodes. 2-D meshes contain 2-D shapes such as triangles and quadrilaterals that connect nodes together. 3-D meshes contain volumetric polyhedra such as: tetrahedrons, hexahedrons, prisms, pyramids, etc.

1.0 Overview

This chapter elaborates on how to create files that VisIt can read. The two main methods of creating files that VisIt can read are: creating a conversion utility and altering a simulation code to write out its data in a new file format. This chapter discusses the merits of each approach so you can decide which is best for your situation. Once you settle on an approach, you can elect to write out Silo files from C or Fortran, or you can write out VTK files from any programming language. If you decide to write out VTK files, this chapter presents examples for doing so in C and Python.

2.0 Creating a conversion utility or extending a simulation

Creating files using a data format that VisIt can read is often the easiest strategy for getting your data into VisIt. You can change your simulation code to natively write its data to a format that VisIt can read, such as Silo or VTK. Alternatively, you can create a conversion utility to post-process your data files into a format that VisIt can read. Both of these approaches have their pros and cons and, fortunately, the programming done to achieve either is essentially the same.

Approach	Pros	Cons
Modify simulation code	<ul style="list-style-type: none">•Data is in a format that can be immediately visualized	<ul style="list-style-type: none">•Depending on the simulation code's implementation language, there may not be a binding to a suitable I/O library.

Approach	Pros	Cons
Create conversion utility	<ul style="list-style-type: none"> •Simulation code does not have to be changed 	<ul style="list-style-type: none"> •Replicates data on disk •Extra step is required to visualize simulation data •Utility must be maintained •Utility must read data from file before it can be written to new data format.

The chief differences between the two approaches arise in where the new code is located. When changing a simulation code, you will most likely add a new I/O module that can dump out your simulation's data for the purpose of visualization. When creating a conversion utility, you are creating a stand-alone program that you have to run on the data after the simulation has completed.

A very simple simulation code's main loop might look like the example below. The purpose of the simple pseudocode listing is to point out where you might want to add additional routines that can write your data to files compatible with VisIt. You might want to provide a switch that tells your program to write data files that VisIt can read in addition to your regular data format. Alternatively, you might opt to just write files that are compatible with VisIt.

```

/* SIMPLE SIMULATION SKELETON */
void write_vis_dump()
{
    if(write_data_for_visit)
        /* Add your code to write VisIt data files here. */
    else
        write_vis_dump_using_regular_format();
}
int main(int argc, char **argv)
{
    read_input_deck();
    do
    {
        simulate_one_timestep();
        write_vis_dump();
    } while(!simulation_done());
    return 0;
}

```

If you choose to write a conversion utility, a pseudocode skeleton might look something like this:

```

/* SIMPLE CONVERSION UTILITY SKELETON */
void write_to_visit_format(const char *, MeshAndData *)
{
    /* Add your code to write a VisIt data file here. */
}

```



```

}
void convert_file(const char *filename)
{
    struct MeshAndData data;
    char newfilename[1024];
    read_data_from_regular_format(filename, &data);
    create_visit_filename(filename, newfilename);
    write_to_visit_format(newfilename, &data);
    free_data(&data);
}
int main(int argc, char *argv[])
{
    for(int i = 1; i < argc; ++i)
        convert_file(argv[i]);
    return 0;
}

```

3.0 Survey of database reader plug-ins

VisIt provides database reader plug-ins for over 5 dozen different file formats. This chapter will talk briefly about some specialized file formats before covering the Silo and VTK file formats. Silo and VTK will be covered much more extensively because they are two of the most general formats and they are capable of describing a wide variety of different data constructs.

Silo is a C-language library with a well-defined application programming interface (API) for writing out the types of objects in which most simulations are interested (e.g. meshes, variables). Silo files can be written to two different underlying file structures: HDF5 and PDB; both are self-describing, platform independent, binary file formats. If you write a file on one platform using the Silo library, it can be read by the Silo library on any other platform. Silo bindings also exist for the Fortran and Python programming languages. For more information, see the *Silo User's Guide*.

The VTK file format is written by various C++ classes in VTK (Visualization Tool Kit) and is most often stored in ASCII text files. The VTK file format does, more recently, support an XML-based file format, which includes support for binary data and compression. However, this manual will provide example code to write data into VTK's legacy ASCII format. The example code will use VisIt's `visit_writer` library to demonstrate creating VTK files without using the VTK library itself so the applications will be very lightweight.

3.1 BOV file format

As mentioned earlier, VisIt can read over 5 dozen file formats and this manual will mainly concentrate on two of them. There are other file formats that might be useful to you depending on how you have written your data files. For example, if you have written your data as a binary file consisting of 1 variable on a $NX \times NY \times NZ$ rectilinear mesh then it is

possible that you can use VisIt's BOV ("Brick of Values") database reader plug-in and not have to do any data conversion.

VisIt's BOV database reader plug-in is used to read data out of a binary file containing just the data values. If your data file was written using code resembling the following code fragments then you might be able to use VisIt's BOV database reader plug-in.

Listing 2-1: bov.c: C-Language example for creating data that the BOV plug-in can read.

```
/* Example C code */
float data[NZ][NY][NX];
FILE *fp = fopen("bov.values", "wb");
fwrite((void *)data, sizeof(float), NX*NY*NZ, fp);
fclose(fp);
```

Listing 2-2: fbov.f: Fortran language example for creating data that the BOV plug-in can read.

```
c Example Fortran code
    real values(NX, NY, NZ)
    open (unit=output, file='fbov.values', status='replace',
. form='unformatted')
    write(output) values
    close (output)
```

Files written in this manner typically have an auxiliary data header text file stored along side of the real data file to contain information such as the dimensions of the data and its type and endian representation. If this sounds like what you write from your simulation code then you should try using the BOV reader. Before trying to open the data using VisIt's BOV database reader plug-in, you will have to write a BOV-compatible header file to accompany your data files so VisIt knows how to read the binary data file.

Example BOV header file:

```
TIME: 1.23456
DATA_FILE: file0000.dat
# The data size corresponds to NX,NY,NZ in the above example code.
DATA_SIZE: 10 10 10
# Allowable values for DATA_FORMAT are: BYTE, INT, FLOAT, DOUBLE
DATA_FORMAT: FLOAT
VARIABLE: what_I_call_the_data
# Endian representation of the computer that created the data.
# Intel is LITTLE, many other processors are BIG.
DATA_ENDIAN: LITTLE
# Centering refers to how the data is distributed in a cell. If you
# give "zonal" then it's 1 data value per zone. Otherwise the data
# will be centered at the nodes.
CENTERING: zonal
# BRICK_ORIGIN lets you specify a new coordinate system origin for
# the mesh that will be created to suit your data.
```

```
BRICK_ORIGIN: 0. 0. 0.
# BRICK_SIZE lets you specify the size of the brick.
BRICK_SIZE: 10. 10. 10.
```

Additional BOV options:

```
# BYTE_OFFSET: is optional and lets you specify some number of
# bytes to skip at the front of the file. This can be useful for
# skipping the 4-byte header that Fortran tends to write to files.
# If your file does not have a header then DO NOT USE BYTE_OFFSET.
BYTE_OFFSET: 4

# DIVIDE_BRICK: is optional and can be set to "true" or "false".
# When DIVIDE_BRICK is true, the BOV reader uses the values stored
# in DATA_BRICKLETS to divide the data into chunks that can be
# processed in parallel.
DIVIDE_BRICK: true

# DATA_BRICKLETS: is optional and requires you to specify 3 integers
# that indicate the size of the bricklets to create when you have
# also specified the DIVIDE_BRICK option. The values chosen for
# DATA_BRICKLETS must be factors of the numbers used for DATA_SIZE.
DATA_BRICKLETS: 5 5 5

# DATA_COMPONENTS: is optional and tells the BOV reader how many
# components your data has. 1=scalar, 2=complex number, 3=vector,
# 4 and beyond indicate an array variable. You can use "COMPLEX"
# instead of "2" for complex numbers.
DATA_COMPONENTS: 1
```

Take the above example BOV header file template and save it to a new text file with a “.bov” file extension. Next, edit the file and change some of the values to make it relevant to the data file that you want to open. Once you’ve completed editing the “.bov” file, open it in VisIt. If you see that the **Plots menu** is enabled and the **Mesh** and **Pseudocolor** plot menus are enabled then you are halfway to success. If you can create a Pseudocolor plot, click the **Draw** button, and have VisIt process your data until there is a picture in the visualization window then this approach works for you and you can repeat it for your other data files. If the picture is not quite what you expected then you can fine-tune the values in the “.bov” file until you get the picture that you want to see. The most common cause of errors is failing to set the DATA_SIZE and DATA_FORMAT keywords to the right values for your data file.

3.2 X-Y Curve file format

VisIt is used to examine and analyze a wide variety of data in 2D and 3D on many different types of meshes. In addition to those capabilities, VisIt can also visualize and process 1D curves, sometimes known as X-Y plots. VisIt’s Lineout mode can extract data from a higher dimensional dataset and draw the resulting data as an X-Y plot, or a Curve plot as it is known in VisIt terms. VisIt can also import X-Y data and use it to create Curve

plots. The Curve file format, which is barely more than a list of X-Y pairs, is outlined below:

```
#curve1name
x0 y0
x1 y1
x2 y2
...
#curve2name
xn yn
xn1 yn1
...
```

As shown in the example Curve file, the Curve file format can contain data for more than 1 set of X-Y pairs. The name of each pair is indicated in a ‘#’ comment line. The X-Y pairs follow until the end of the file or until a new curve is declared using another ‘#’ comment line. If you write data to the Curve file format then the file extension should be “.curve” to ensure that VisIt recognizes it as a Curve file.

4.0 Writing Silo files

If you are writing a conversion utility or if you have a simulation code written in C, C++, or Fortran then writing out Silo files is a good choice for getting your data into VisIt. This section will illustrate how to use the Silo library to write out various types of scientific data. Since the Silo library provides bindings for multiple languages, including C, Fortran, and Python, the source code examples that demonstrate a particular topic will be given in more than 1 programming language, when appropriate. One goal of this section is to provide examples that are complete enough so that they can be readily adapted into working source code. In fact, most of the examples in this chapter are available as working programs in the accompanying “Getting your data into VisIt” code distribution. This section will not necessarily explain all of the various arguments to function calls in the Silo library. You can refer to the *Silo User’s Guide* for more information.

4.1 Using the Silo library

VisIt is always built with support for reading Silo databases so Silo can be a good file format in which to store your data. This subsection includes information about using Silo such as including the appropriate header files and linking with the Silo library.

4.1.1 Including Silo

When using any library in a program, you must tell the compiler about the symbols provided by the library. Here is what you need to include in your source code in order to use Silo:

C-Language:

```
#include <siloh.h>
```

Fortran language:

```
include "siloh.inc"
```

4.1.2 Linking with Silo

Before you can build a program that uses Silo, you must locate the Silo include files and the Silo library. Silo is not distributed as part of the VisIt source code or binary installations so you must obtain it separately unless you are developing on the Windows platform. A link to the most up-to-date version of the Silo library's source code can be found on the VisIt Web site at <http://www.llnl.gov/visit/source.html>.

Once you download the Silo source code, building and installing it is usually only a matter of running its *configure* script and running *make*. You can find information about configuring Silo with support for HDF5 in VisIt's *BUILD_NOTES* file, also available on the VisIt Web site.

After you've configured, built, and installed the Silo library, your program will have to be built against the Silo library. Building against the Silo library is usually accomplished by a simple adaptation of your *Makefile* and the inclusion of *siloh.h* in your C-language source code. If Silo has been installed in `/usr/local/silo` then you would add the following to your *Makefile*:

```
LDFLAGS=$(LDFLAGS) -L/usr/gapps/silo/lib -lsilo -lm
CPPFLAGS=$(CPPFLAGS) -I/usr/gapps/silo/include
```

If you discover that only `libsilo.a` exists in your Silo library directory then you may not be able to generate Silo files using the HDF5 file format. If you find that your Silo library directory contains a file called *libsiloh5.a* then you can use that version of the Silo library to create HDF5-style Silo files. You might still want to manually check your *libsilo.a* for HDF5 support using this command in your UNIX shell: `"nm libsilo.a | grep hdf5"`. If you see any output containing the word "hdf5" then you can use *libsilo.a* to create HDF5 files. If your Silo library does support HDF5 files then you must also locate your HDF5 installation directory so you can link HDF5 into your program to satisfy HDF5 calls for the Silo library. Your *Makefile* would look something like this:

```
HDF5DIR= Fill in the right path to your HDF5 installation
HDF5LIBS=$(HDF5DIR)/lib/libhdf5.a -lz
LDFLAGS=$(LDFLAGS) -L/usr/gapps/silo/lib -lsilo $(HDF5LIBS) -lm
CPPFLAGS=$(CPPFLAGS) -I/usr/gapps/silo/include
```

If your *Makefile* does not use CPPFLAGS then you might try adding the `-I` directive to CFLAGS, F77FLAGS, or whichever make variables are relevant for your *Makefile*.

4.1.3 Using Silo on Windows

When you build an application using the Silo library on Windows, you can use the precompiled Silo DLL and import library that comes with the VisIt source code distribution for Windows. The VisIt1.5.4 source code distribution for Windows is called `visitdev1.5.4.exe`. Other versions of VisIt would, of course, include a different version number in the filename. When you install the VisIt source code distribution for Windows, you get all of VisIt's project files, include files, and source code. In addition, certain precompiled libraries such as Silo are included.

If you want to build an application against the Silo library provided with VisIt, add the path to `silo.h` to your project file. If you build using a source code distribution for VisIt 1.5.4 that was installed in the default location, the path would be:
`C:\VisItDev1.5.4\include\silo.`

After setting the Silo include directory to your project file, make sure that the Silo's import library is in your linker path. You can add `C:\VisItDev1.5.4\lib\Release` or `C:\VisItDev1.5.4\lib\Debug` to your project to ensure that your linker can find Silo's import library. Next, add `silohdf5.lib` to the list of libraries that are linked with your program. That should be enough to get your program to build.

Before running your program, be sure to copy `silohdf5.dll`, `hdf5dll.dll`, `sziplib.dll`, and `zlib.dll` from `C:\VisItDev1.5.4\bin\Release` or `C:\VisItDev1.5.4\bin\Debug` (depending on whether your program is compiled with debugging information) into the directory where your program will execute. Note that you must configure your program to use a Multithreaded DLL version of the Microsoft runtime library or using the precompiled Silo library may result in fatal errors.

4.2 Inspecting Silo files

Silo includes a command line utility called `browser` that can access the contents of Silo files. To run `browser`, type "`browser`" into a terminal window followed by the name of a Silo file that you want to inspect. Once the browser application opens the Silo file, type "`ls`" to see the contents of the Silo file. From there, typing the name of any of the objects shown in the object listing will print information about that object to the console.

4.3 Silo files and parallel codes

Before we delve into examples about how to use the Silo library, let's first examine how parallel simulation codes process their data in a distributed-memory environment. Many parallel simulation codes will divide the entire simulated mesh into submeshes, called domains, which are assigned to processors that calculate the fields of interest on their domain. Often, the most efficient I/O strategy for the simulation code is to make each processor write its domain to a separate file. The examples that follow assume parallel simulations will write 1 file per processor. It is possible for multiple processors to append

their data to a single Silo file but it requires synchronization and that technique is beyond the scope of the examples that will be presented.

4.4 Creating a new Silo file

The first step to saving data to a Silo file is to create the file and obtain a handle that will be used to reference the file. The handle will be passed to other Silo function calls in order to add new objects to the file. Silo creates new files using the `DBCreate` function, which takes the name of the new file, access modes, a descriptive comment, and the underlying file type as arguments.

In addition to being a library, Silo is a self-describing data model, which can be implemented on top of many different underlying file formats. Silo includes drivers that allow it to read data from several different file formats, the most important of which are: PDB (A legacy LLNL file format) format, and HDF5 format. Silo files stored in HDF5 format often provide performance advantages so the following code to open a Silo file will create HDF5-based Silo files. You tell Silo to create HDF5-based Silo files by passing the `DB_HDF5` argument to the `DBCreate` function. If your Silo library does not have built-in HDF5 support then you can pass `DB_PDB` instead to create PDB-based Silo files.

Listing 2-3: basic.c: C-Language example for creating a new Silo file.

```
#include <siloh>
#include <stdio.h>
int
main(int argc, char *argv[])
{
    DBfile *dbfile = NULL;
    /* Open the Silo file */
    dbfile = DBCreate("basic.silo", DB_CLOBBER, DB_LOCAL,
        "Comment about the data", DB_HDF5);
    if(dbfile == NULL)
    {
        fprintf(stderr, "Could not create Silo file!\n");
        return -1;
    }
    /* Add other Silo calls here. */
    /* Close the Silo file. */
    DBClose(dbfile);
    return 0;
}
```

Listing 2-4: fbasic.f: Fortran language example for creating a new Silo file..

```
program main
  implicit none
  include "silo.inc"
  integer dbfile, ierr
  c The 11 and 22 arguments represent the lengths of strings
```

```

        ierr = dbcreate("fbasic.silo", 11, DB_CLOBBER, DB_LOCAL,
        . "Comment about the data", 22, DB_HDF5, dbfile)
        if(dbfile.eq.-1) then
            write (6,*) 'Could not create Silo file!\n'
            goto 10000
        endif
        c Add other Silo calls here.
        c Close the Silo file.
        ierr = dbclose(dbfile)
10000 stop
        end

```

In addition to using the `DBCcreate` function, the previous examples also use the `DBCclose` function. The `DBCclose` function ensures that all data is written to the file and then closes the Silo file. You must call the `DBCclose` function when you want to close a Silo file or your file may not be complete.

4.5 Dealing with time

Silo files are a flexible container for storing many types of data. Silo's ability to store data hierarchically in directories can allow you to store multiple time states of your simulation data within a single data file. However, since Silo is primarily an I/O library for storing files that contain a single time step's worth of data, VisIt only recognizes one time state per Silo file. Consequently, when writing out data, programs that use Silo will write a new Silo file for each time step. By convention, the new file will contain an index indicating either the simulation cycle or a simple integer counter.

Listing 2-5: `time.c`: C-Language example for dealing with time.

```

/* SIMPLE SIMULATION SKELETON */
void write_vis_dump(int cycle)
{
    DBfile *dbfile = NULL;
    /* Create a unique filename for the new Silo file*/
    char filename[100];
    sprintf(filename, "output%04d.silo", cycle);
    /* Open the Silo file */
    dbfile = DBCreate(filename, DB_CLOBBER, DB_LOCAL,
        "simulation time step", DB_HDF5);
    /* Add other Silo calls to write data here. */
    /* Close the Silo file. */
    DBClose(dbfile);
}
int main(int, char **)
{
    int cycle = 0;
    read_input_deck();
    do

```



```

    {
        simulate_one_timestep();
        write_vis_dump(cycle);
        cycle = cycle + 1;
    } while(!simulation_done());
    return 0;
}

```

The above code listing will write out Silo files with names such as: *output0000.silo*, *output0001.silo*, *output0002.silo*, ... Each file contains the data from a particular simulation time state. It may seem like the data are less related because they are stored in different files but the fact that the files are related in time is subtly encoded in the name of each of the files. When VisIt recognizes a pattern in the names of the files such as “output????.silo”, in this case, VisIt automatically groups the files into a time-varying database. If you choose names for your Silo files that cannot be grouped by recognizing a numeric pattern in the trailing part of the file name then you must use a *.visit* file to tell VisIt that your files are related in time. For more information about *.visit* files, consult the *VisIt User’s Manual*.

4.6 Option lists

Many of Silo’s more complex functions accept an auxiliary argument called an option list. An option list is a list of option/value pairs and it is used to specify additional metadata about the data being stored. Each Silo function that accepts an option list has its options enumerated in the *Silo User’s Manual*. This manual will cover only a subset of available options. Option lists need not be passed to the Silo functions that do support them. In fact, most of the source code examples in this manual will pass `NULL` instead of passing a pointer to an option list. Omitting the option list from the Silo function call in this way is not harmful; it only means that certain pieces of additional metadata will not be stored with the data.

Option lists are created using the `DBMakeOptlist` function. Once an option list object is created, you can add options to it using the `DBAddOption` function. Option lists are freed using the `DBFreeOptlist` function.

4.6.1 Cycle and time

We’ve established that a notion of time can be encoded into filenames using ranges of numbers in each filename. VisIt can use the numbers in the names of related files to guess cycle number, a metric for how many times a simulation has iterated. It is possible to use Silo’s option list feature to directly encode the cycle number and the simulation time into the stored data.

Listing 2-6: `optlist.c`: C-Language example for saving cycle and time using an option list..

```

/* Create an option list to save cycle and time values. */
int cycle = 100;
double dtime = 1.23456789;
DBOptlist *optlist = DBMakeOptlist(2);
DBAddOption(optlist, DBOPT_DTIME, &dtime);
DBAddOption(optlist, DBOPT_CYCLE, &cycle);
/* Write a mesh using the option list. */
DBPutQuadmesh(dbfile, "quadmesh", coordnames, coords, dims, ndims,
    DB_FLOAT, DB_COLLINEAR, optlist);
/* Free the option list. */
DBFreeOptlist(optlist);

```

Listing 2-7: foptlist.f: Fortran language example for saving cycle and time using an option list..

```

c Create an option list to save cycle and time values.
    integer cycle /100/
    double precision dtime /1.23456789/
    integer err, ierr, optlistid
    err = dbmkoptlist(2, optlistid)
    err = dbaddiopt(optlistid, DBOPT_CYCLE, cycle)
    err = dbadddopt(optlistid, DBOPT_DTIME, dtime)
c Write a mesh using the option list.
    err = dbputqm (dbfile, "quadmesh", 8, "xc", 2,
    . "yc", 2, "zc", 2, x, y, DB_F77NULL, dims, ndims,
    . DB_FLOAT, DB_COLLINEAR, optlistid, ierr)
c Free the option list.
    err = dbfreeoptlist(optlistid)

```

4.7 Writing a rectilinear mesh

A rectilinear mesh is a 2D or 3D mesh where all coordinates are aligned with the axes. Each axis of the rectilinear mesh can have different, non-uniform spacing, allowing for details to be concentrated in certain regions of the mesh. Rectilinear meshes are specified by lists of coordinate values for each axis. Since the mesh is aligned to the axes, it is only necessary to specify one set of X and Y values to generate all of the coordinates for the entire mesh. Figure 2-8 contains an example of a 2D rectilinear mesh. The Silo function call to write a rectilinear mesh is called `DBPutQuadmesh`.

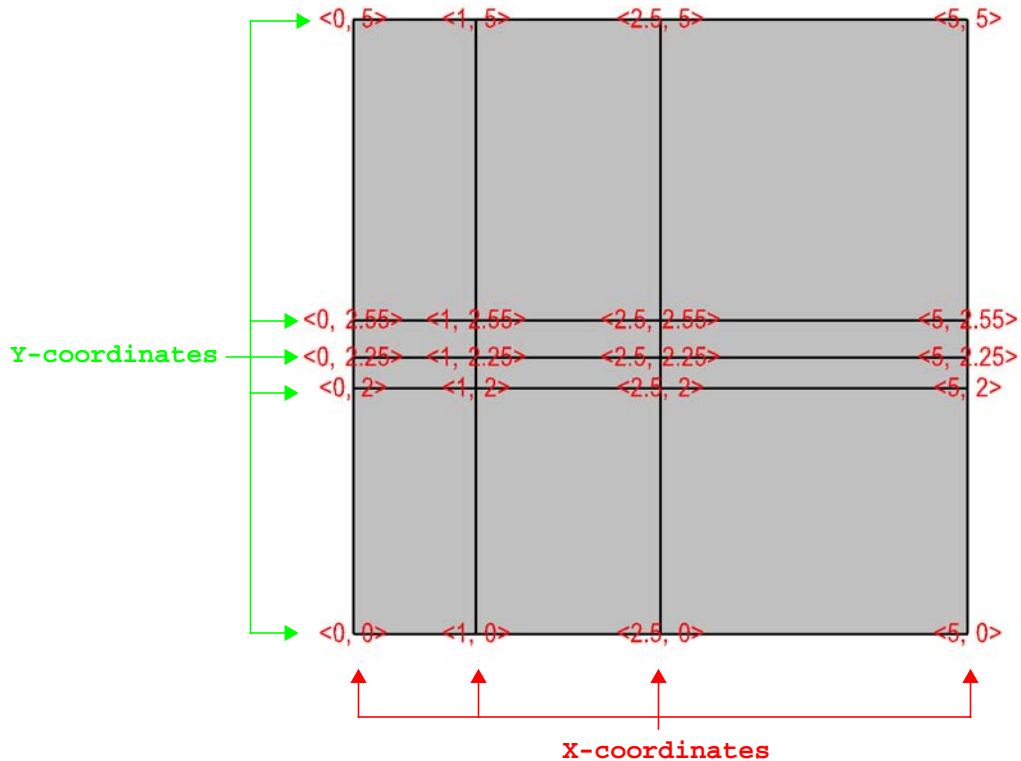


Figure 2-8: Rectilinear mesh and its X,Y node coordinates.

Listing 2-9: rect2d.c: C-Language example for writing a 2D rectilinear mesh.

```

/* Write a rectilinear mesh. */
float x[] = {0., 1., 2.5, 5.};
float y[] = {0., 2., 2.25, 2.55, 5.};
int dims[] = {4, 5};
int ndims = 2;
float *coords[] = {x, y};
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
              DB_FLOAT, DB_COLLINEAR, NULL);

```

Listing 2-10: frect2d.f: Fortran language example for writing a 2D rectilinear mesh.

```

c Write a rectilinear mesh
integer err, ierr, dims(2), ndims, NX, NY
parameter (NX = 4)
parameter (NY = 5)
real x(NX), y(NY)
data dims/NX, NY/
data x/0., 1., 2.5, 5./

```

```

data y/0., 2., 2.25, 2.55, 5./
ndims = 2
err = dbputqm (dbfile, "quadmesh", 8, "xc", 2,
. "yc", 2, "zc", 2, x, y, DB_F77NULL, dims, ndims,
. DB_FLOAT, DB_COLLINEAR, DB_F77NULL, ierr)
    
```

The previous code examples demonstrate how to write out a 2D rectilinear mesh using Silo's DBPutQuadmesh function (called dbputqm in Fortran). There are three pieces of important information passed to the DBPutQuadmesh function. The first important piece information is the name of the mesh being created. The name that you choose will be the name that you use when writing a variable to a Silo file and also the name that you will see in VisIt's plot menus when you want to create a Mesh plot in VisIt. After the name, you provide the coordinate arrays that contain the X and Y point values that ultimately form the set of X,Y coordinate pairs that describe the mesh. The C-interface to Silo requires that you pass pointers to the coordinate arrays in a single pointer array. The Fortran interface to Silo requires you to pass the names of the coordinate arrays, followed by the actual coordinate arrays, with a value of DB_F77NULL for any arrays that you do not use. The final critical pieces of information that must be passed to the DBPutQuadmesh function are the dimensions of the mesh, which correspond to the number of nodes, or coordinate values, along the mesh in a given dimension. The dimensions are passed in an array, along with the number of dimensions, which must be 2 or 3. Figure 2-11 shows an example of a 3D rectilinear mesh for the upcoming code examples.

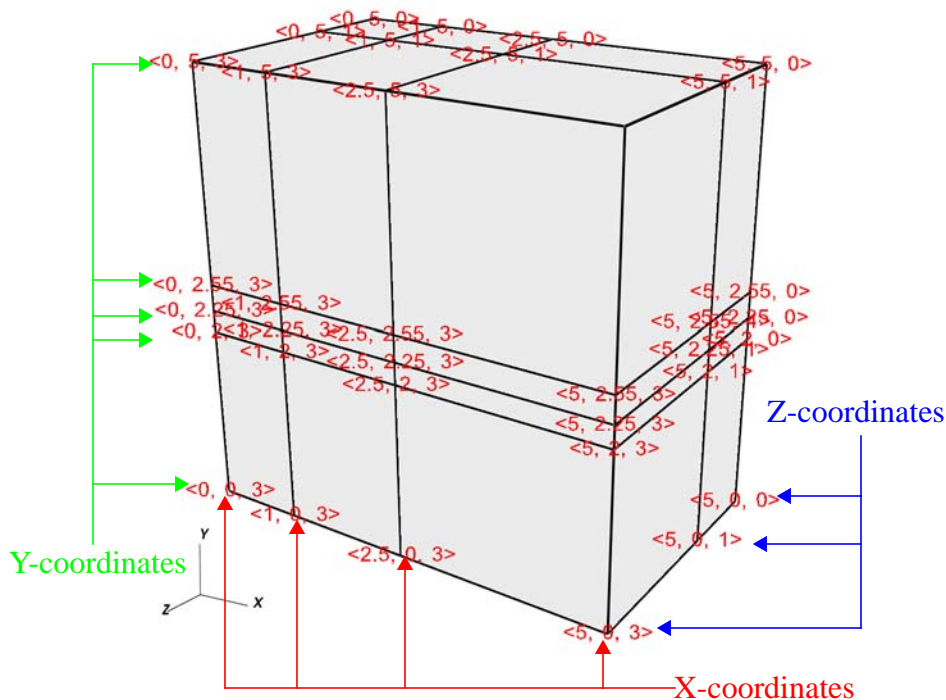


Figure 2-11: Rectilinear mesh and its X,Y,Z coordinates

Listing 2-12: rect3d.c: C-Language example for writing a 3D rectilinear mesh.

```

/* Write a rectilinear mesh. */
float x[] = {0., 1., 2.5, 5.};
float y[] = {0., 2., 2.25, 2.55, 5.};
float z[] = {0., 1., 3.};
int dims[] = {4, 5, 3};
int ndims = 3;
float *coords[] = {x, y, z};
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
              DB_FLOAT, DB_COLLINEAR, NULL);

```

Listing 2-13: frect3d.f: Fortran language example for writing a 3D rectilinear mesh.

```

integer err, ierr, dims(3), ndims, NX, NY, NZ
parameter (NX = 4)
parameter (NY = 5)
parameter (NZ = 3)
real x(NX), y(NY), z(NZ)
data x/0., 1., 2.5, 5./
data y/0., 2., 2.25, 2.55, 5./
data z/0., 1., 3./
ndims = 3
data dims/NX, NY, NZ/
err = dbputqm (dbfile, "quadmesh", 8, "xc", 2,
. "yc", 2, "zc", 2, x, y, z, dims, ndims,
. DB_FLOAT, DB_COLLINEAR, DB_F77NULL, ierr)

```

4.8 Writing a curvilinear mesh

A curvilinear mesh is similar to a rectilinear mesh. The main difference between the two mesh types is how coordinates are specified. Recall that in a rectilinear mesh, the coordinates are specified individually for each axis and only a small subset of the nodes in the mesh are provided. The coordinate arrays are used to assemble a point for each node in the mesh. In a curvilinear mesh, you must provide an X,Y,Z value for every node in the mesh. Providing the coordinates for every point explicitly allows you to specify more complex geometries than are possible using rectilinear meshes. Note how the mesh coordinates on the mesh in Figure 2-14 allow it to assume shapes that are not aligned to the coordinate axes.

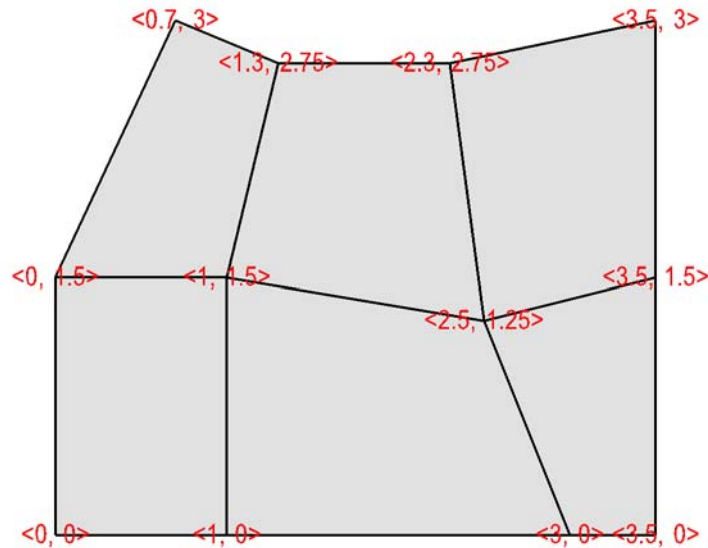


Figure 2-14: Curvilinear mesh and its X,Y node coordinates

The fine line between a rectilinear mesh and a curvilinear mesh comes down to how the coordinates are specified. Silo dictates that the coordinates be specified with an array of X-coordinates, an array of Y-coordinates, and an optional array of Z-coordinates. The difference, of course, is that in a curvilinear mesh, there are explicit values for each node's X,Y,Z points. Silo uses the same `DBPutQuadmesh` function to write out curvilinear meshes. The coordinate arrays are passed the same as for the rectilinear mesh, though the X,Y,Z arrays now point to larger arrays. You can pass the `DB_NONCOLLINEAR` flag to the `DBPutQuadmesh` function in order to indicate that the coordinate arrays contain values for every node in the mesh.

Listing 2-15: `curv2d.c`: C-Language example for writing a 2D curvilinear mesh.

```

/* Write a curvilinear mesh. */
#define NX 4
#define NY 3
float x[NY][NX] = {{0., 1., 3., 3.5}, {0., 1., 2.5, 3.5},
                  {0.7, 1.3, 2.3, 3.5}};
float y[NY][NX] = {{0., 0., 0., 0.}, {1.5, 1.5, 1.25, 1.5},
                  {3., 2.75, 2.75, 3.}};
int dims[] = {NX, NY};
int ndims = 2;
float *coords[] = {(float*)x, (float*)y};
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,

```

```
DB_FLOAT, DB_NONCOLLINEAR, NULL);
```

Listing 2-16: fcurv2d.f: Fortran language example for writing a 2D curvilinear mesh.

```
c Write a curvilinear mesh.
integer err, ierr, dims(2), ndims, NX, NY
parameter (NX = 4)
parameter (NY = 3)
real x(NX,NY), y(NX,NY)
data x/0., 1., 3., 3.5,
.      0., 1., 2.5, 3.5,
.      0.7, 1.3, 2.3, 3.5/
data y/0., 0., 0., 0.,
.      1.5, 1.5, 1.25, 1.5,
.      3., 2.75, 2.75, 3./
ndims = 2
data dims/NX, NY/
err = dbputqm (dbfile, "quadmesh", 8, "xc", 2,
. "yc", 2, "zc", 2, x, y, DB_F77NULL, dims, ndims,
. DB_FLOAT, DB_NONCOLLINEAR, DB_F77NULL, ierr)
```

Figure 2-17 shows a simple 3D curvilinear mesh that is 1 cell thick in the Z-dimension. The number of cells in a dimension is 1 less than the number of nodes in the same dimension. for structured meshes. As you increase the number of nodes in the Z-dimension, you must also add more X and Y coordinate values because the X,Y,Z values for node coordinates must be fully specified for a curvilinear mesh.

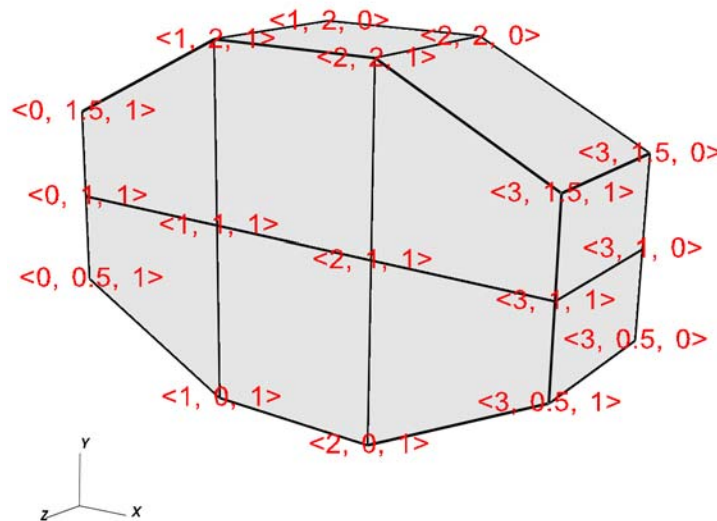


Figure 2-17: 3D Curvilinear mesh and its X,Y,Z coordinates

Listing 2-18: curv3d.c: C-Language example for writing a 3D curvilinear mesh.

```

/* Write a curvilinear mesh. */
#define NX 4
#define NY 3
#define NZ 2
float x[NZ][NY][NX] = {
    {{0.,1.,2.,3.},{0.,1.,2.,3.}, {0.,1.,2.,3.}},
    {{0.,1.,2.,3.},{0.,1.,2.,3.}, {0.,1.,2.,3.}}
};
float y[NZ][NY][NX] = {
    {{0.5,0.,0.,0.5},{1.,1.,1.,1.}, {1.5,2.,2.,1.5}},
    {{0.5,0.,0.,0.5},{1.,1.,1.,1.}, {1.5,2.,2.,1.5}}
};
float z[NZ][NY][NX] = {
    {{0.,0.,0.,0.},{0.,0.,0.,0.},{0.,0.,0.,0.}},
    {{1.,1.,1.,1.},{1.,1.,1.,1.},{1.,1.,1.,1.}}
};
int dims[] = {NX, NY, NZ};
int ndims = 3;
float *coords[] = {(float*)x, (float*)y, (float*)z};
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
    DB_FLOAT, DB_NONCOLLINEAR, NULL);

```

Listing 2-19: fcurv3d.f: Fortran language example for writing a 3D curvilinear mesh.

```

c Write a curvilinear mesh
    integer err, ierr, dims(3), ndims, NX, NY, NZ
    parameter (NX = 4)
    parameter (NY = 3)
    parameter (NZ = 2)
    real x(NX,NY,NZ), y(NX,NY,NZ), z(NX,NY,NZ)
    data x/0.,1.,2.,3., 0.,1.,2.,3., 0.,1.,2.,3.,
    . 0.,1.,2.,3., 0.,1.,2.,3., 0.,1.,2.,3./
    data y/0.5,0.,0.,0.5, 1.,1.,1.,1., 1.5,2.,2.,1.5,
    . 0.5,0.,0.,0.5, 1.,1.,1.,1., 1.5,2.,2.,1.5/
    data z/0.,0.,0.,0., 0.,0.,0.,0., 0.,0.,0.,0.,
    . 1.,1.,1.,1., 1.,1.,1.,1., 1.,1.,1.,1./
    ndims = 3
    data dims/NX, NY, NZ/
    err = dbputqm (dbfile, "quadmesh", 8, "xc", 2,
    . "yc", 2, "zc", 2, x, y, z, dims, ndims,
    . DB_FLOAT, DB_NONCOLLINEAR, DB_F77NULL, ierr)

```


4.9 Writing a point mesh

A point mesh is a set of 2D or 3D points where the nodes also constitute the cells in the mesh. Silo provides the `DBPutPointmesh` function so you can write out particle systems represented as point meshes.

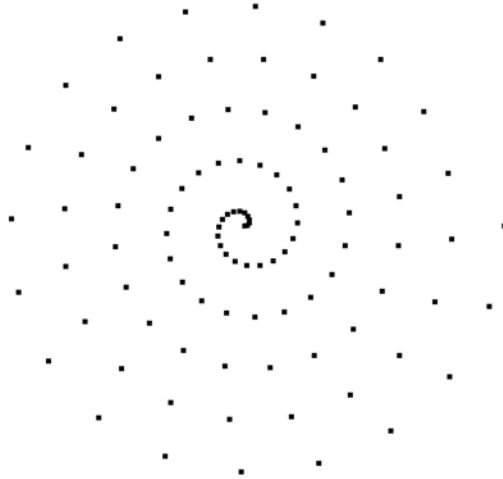


Figure 2-20: 2D point mesh

Listing 2-21: point2d.c: C-Language example for writing a 2D point mesh.

```

/* Create some points to save. */
#define NPTS 100
int i, ndims = 2;
float x[NPTS], y[NPTS];
float *coords[] = {(float*)x, (float*)y};
for(i = 0; i < NPTS; ++i)
{
    float t = ((float)i) / ((float)(NPTS-1));
    float angle = 3.14159 * 10. * t;
    x[i] = t * cos(angle);
    y[i] = t * sin(angle);
}
/* Write a point mesh. */
DBPutPointmesh(dbfile, "pointmesh", ndims, coords, NPTS,
    DB_FLOAT, NULL);

```

Listing 2-22: fpoint2d.f: Fortran language example for writing a 2D point mesh.

```

c Create some points to save.
    integer err, ierr, i, ndims, NPTS
    parameter (NPTS = 100)

```

```

real x(NPTS), y(NPTS), t, angle
do 10000 i = 0,NPTS-1
  t = float(i) / float(NPTS-1)
  angle = 3.14159 * 10. * t
  x(i+1) = t * cos(angle);
  y(i+1) = t * sin(angle);
10000 continue
ndims = 2
c Write a point mesh.
err = dbputpm (dbfile, "pointmesh", 9, ndims, x, y,
. DB_F77NULL, NPTS, DB_FLOAT, DB_F77NULL, ierr)

```

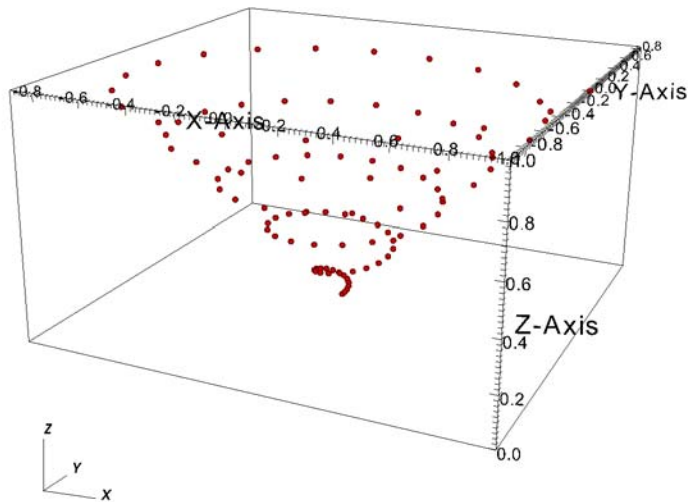


Figure 2-23: 3D point mesh

Writing a 3D point mesh is very similar to writing a 2D point mesh with the exception that for a 3D point mesh, you must specify a Z-coordinate. Figure 2-23 shows what happens when we extend our 2D point mesh example into 3D.

Listing 2-24: point3d.c: C-Language example for writing a 3D point mesh.

```

/* Create some points to save. */
#define NPTS 100
int i, ndims = 3;
float x[NPTS], y[NPTS], z[NPTS];
float *coords[] = {(float*)x, (float*)y, (float*)z};
for(i = 0; i < NPTS; ++i)

```

```

{
  float t = ((float)i) / ((float)(NPTS-1));
  float angle = 3.14159 * 10. * t;
  x[i] = t * cos(angle);
  y[i] = t * sin(angle);
  z[i] = t;
}
/* Write a point mesh. */
DBPutPointmesh(dbfile, "pointmesh", ndims, coords, NPTS,
  DB_FLOAT, NULL);

```

Listing 2-25: fpoint3d.f: Fortran language example for writing a 3D point mesh.

```

c Create some points to save
  integer err, ierr, i, ndims, NPTS
  parameter (NPTS = 100)
  real x(NPTS), y(NPTS), z(NPTS), t, angle
  do 10000 i = 0, NPTS-1
    t = float(i) / float(NPTS-1)
    angle = 3.14159 * 10. * t
    x(i+1) = t * cos(angle);
    y(i+1) = t * sin(angle);
    z(i+1) = t
  10000 continue
  ndims = 3
c Write a point mesh
  err = dbputpm (dbfile, "pointmesh", 9, ndims, x, y, z,
  . NPTS, DB_FLOAT, DB_F77NULL, ierr)

```

4.10 Writing an unstructured mesh

Unstructured meshes are collections of different types of zones and are useful because they can represent more complex mesh geometries than structured meshes can. This section explains the Silo functions that are used to write out an unstructured mesh.

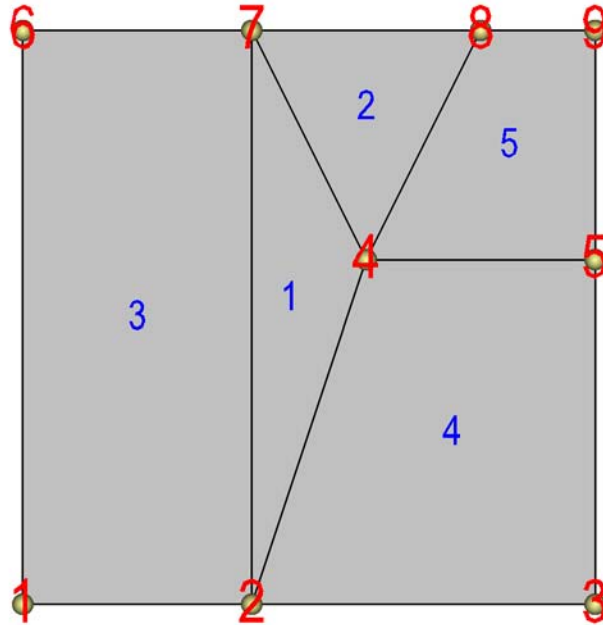


Figure 2-26: 2D unstructured mesh composed of triangles and quadrilaterals. The node numbers are labelled red and the zone numbers are labelled blue.

Silo supports the creation of 2D unstructured meshes composed of arbitrary polyhedral cells. However, of the myriad of possible polyhedral cells, VisIt's Silo reader plug-in will currently only accept cells that are triangles or quadrilaterals. Unstructured meshes are specified in terms of a set of nodes and then a zone list consisting of lists of nodes, called connectivity information, that make up the zones in the mesh. When creating connectivity information, be sure that the nodes in your zones are specified so that when you iterate over the nodes in the zone that a counter-clockwise pattern is observed. Silo provides the `DBPutZonelist` function to store out the connectivity information. The coordinates for the unstructured mesh itself is written out using the `DBPutUcdmesh` function.

Listing 2-27: `ucd2d.c`: C-Language example for writing a 2D unstructured mesh.

```
/* Node coordinates */
float x[] = {0., 2., 5., 3., 5., 0., 2., 4., 5.};
float y[] = {0., 0., 0., 3., 3., 5., 5., 5., 5.};
float *coords[] = {x, y};
/* Connectivity */
int nodelist[] = {
    2,4,7, /* tri zone 1 */
    4,8,7, /* tri zone 2 */
}
```

```

    1,2,7,6, /* quad zone 3 */
    2,3,5,4, /* quad zone 4 */
    4,5,9,8 /* quad zone 5 */
};
int lnodelist = sizeof(nodelist) / sizeof(int);
/* shape type 1 has 3 nodes (tri), shape type 2 is quad */
int shapysize[] = {3, 4};
/* We have 2 tris and 3 quads */
int shapecounts[] = {2, 3};
int nshapetypes = 2;
int nnodes = 9;
int nzones = 5;
int ndims = 2;
/* Write out connectivity information. */
DBPutZonelist(dbfile, "zonelist", nzones, ndims, nodelist, lnodelist,
    1, shapysize, shapecounts, nshapetypes);
/* Write an unstructured mesh. */
DBPutUcdmesh(dbfile, "mesh", ndims, NULL, coords, nnodes, nzones,
    "zonelist", NULL, DB_FLOAT, NULL);

```

Listing 2-28: fucd2d.f: Fortran language example for writing a 2D unstructured mesh.

```

    integer err, ierr, ndims, nshapetypes, nnodes, nzones
c Node coordinates
    real x(9) /0., 2., 5., 3., 5., 0., 2., 4., 5./
    real y(9) /0., 0., 0., 3., 3., 5., 5., 5., 5./
c Connectivity
    integer LNODELIST
    parameter (LNODELIST = 18)
    integer nodelist(LNODELIST) /2,4,7,
. 4,8,7,
. 1,2,7,6,
. 2,3,5,4,
. 4,5,9,8/
c Shape type 1 has 3 nodes (tri), shape type 2 is quad
    integer shapysize(2) /3, 4/
c We have 2 tris and 3 quads
    integer shapecounts(2) /2, 3/
    nshapetypes = 2
    nnodes = 9
    nzones = 5
    ndims = 2
c Write out connectivity information.
    err = dbputzl(dbfile, "zonelist", 8, nzones, ndims, nodelist,
. LNODELIST, 1, shapysize, shapecounts, nshapetypes, ierr)
c Write an unstructured mesh
    err = dbputum(dbfile, "mesh", 4, ndims, x, y, DB_F77NULL,
. "X", 1, "Y", 1, DB_F77NULL, 0, DB_FLOAT, nnodes, nzones,
. "zonelist", 8, DB_F77NULL, 0, DB_F77NULL, ierr)

```

3D unstructured meshes are created much the same way as 2D unstructured meshes are created. The main difference is that whereas in 2D, you use triangles and quadrilateral zone types, in 3D, you use hexahedrons, pyramids, prisms, and tetrahedrons to compose your mesh. The procedure for creating the node coordinates is the same with the exception that 3D meshes also require a Z-coordinate. The procedure for creating the zone list (connectivity information) is the same except that you specify cells using a larger number of nodes because they are 3D. The order in which the nodes are specified is also more important for 3D shapes because if the nodes are not given in the right order, the zones can become tangled. The proper zone ordering for each of the four supported 3D zone shapes is shown in Figure 2-29.

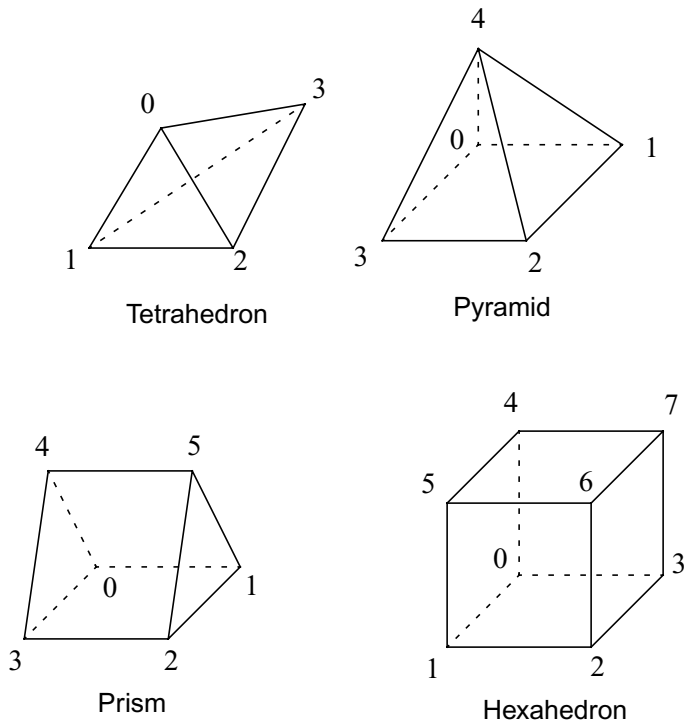


Figure 2-29: Node ordering for Silo's 3D unstructured zone types

Figure 2-30 shows an example of a simple 3D unstructured mesh consisting of 2 hexahedrons, 1 pyramid, 1 prism, and 1 tetrahedron.

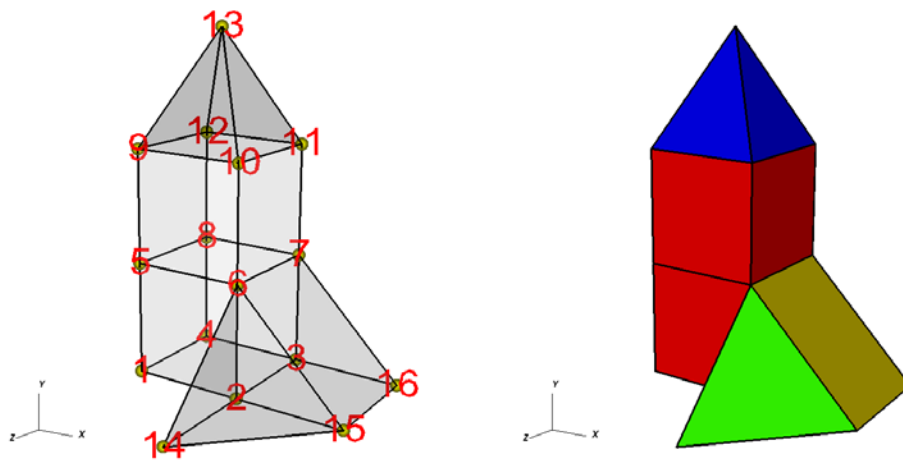


Figure 2-30: Node numbers on the left and the mesh, colored by zone type, on the right. Hexahedrons (red), Pyramid (blue), Prism (yellow), Tetrahedron (green).

Listing 2-31: ucd3d.c: C-Language example for writing a 3D unstructured mesh.

```

/* Node coordinates */
float x[] = {0.,2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,1.,2.,4.,4.};
float y[] = {0.,0.,0.,0.,2.,2.,2.,2.,4.,4.,4.,4.,6.,0.,0.,0.};
float z[] = {2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,0.,1.,4.,2.,0.};
float *coords[] = {x, y, z};
/* Connectivity */
int nodelist[] = {
    1,2,3,4,5,6,7,8,    /* hex,      zone 1 */
    5,6,7,8,9,10,11,12, /* hex,      zone 2 */
    9,10,11,12,13,     /* pyramid, zone 3 */
    2,3,16,15,6,7,     /* prism,   zone 4 */
    2,15,14,6          /* tet,      zone 5 */
};
int lnodelist = sizeof(nodelist) / sizeof(int);
/* shape type 1 has 8 nodes (hex) */
/* shape type 2 has 5 nodes (pyramid) */
/* shape type 3 has 6 nodes (prism) */
/* shape type 4 has 4 nodes (tet) */
int shapysize[] = {8,5,6,4};
/* We have 2 hex, 1 pyramid, 1 prism, 1 tet */
int shapecounts[] = {2,1,1,1};
int nshapetypes = 4;
int nnodes = 16;
int nzones = 5;
int ndims = 3;
/* Write out connectivity information. */
DBPutZonelist(dbfile, "zonelist", nzones, ndims, nodelist, lnodelist,
    1, shapysize, shapecounts, nshapetypes);
/* Write an unstructured mesh. */
DBPutUcdmesh(dbfile, "mesh", ndims, NULL, coords, nnodes, nzones,
    "zonelist", NULL, DB_FLOAT, NULL);

```

Listing 2-32: fucd3d.f: Fortran language example for writing a 3D unstructured mesh.

```

integer err, ierr, ndims, nzones
integer NSHAPETYPES, NNODES
parameter (NSHAPETYPES = 4)
parameter (NN = 16)
c Node coordinates
real x(NN) /0.,2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,1.,2.,4.,4./
real y(NN) /0.,0.,0.,0.,2.,2.,2.,2.,4.,4.,4.,4.,6.,0.,0.,0./
real z(NN) /2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,0.,1.,4.,2.,0./
c Connectivity
integer LNODELIST
parameter (LNODELIST = 31)
integer nodelist(LNODELIST) /1,2,3,4,5,6,7,8,
. 5,6,7,8,9,10,11,12,
. 9,10,11,12,13,

```

```

        . 2,3,16,15,6,7,
        . 2,15,14,6/
c Shape type 1 has 8 nodes (hex)
c Shape type 2 has 5 nodes (pyramid)
c Shape type 3 has 6 nodes (prism)
c Shape type 4 has 4 nodes (tet)
        integer shapysize(NSHAPETYPES) /8, 5, 6, 4/
c We have 2 hex, 1 pyramid, 1 prism, 1 tet
        integer shapecounts(NSHAPETYPES) /2, 1, 1, 1/
        nzones = 5
        ndims = 3
c Write out connectivity information.
        err = dbputz1(dbfile, "zonelist", 8, nzones, ndims, nodelist,
        . LNODELIST, 1, shapysize, shapecounts, NSHAPETYPES, ierr)
c Write an unstructured mesh
        err = dbputum(dbfile, "mesh", 4, ndims, x, y, z,
        . "X", 1, "Y", 1, "Z", 1, DB_FLOAT, NN, nzones,
        . "zonelist", 8, DB_F77NULL, 0, DB_F77NULL, ierr)
    
```

4.10.1 Adding axis labels and axis units

It is possible to add additional annotations to your meshes that you store to Silo files using Silo’s option list mechanism. This subsection covers how to change the axis titles and units that will be used when VisIt plots your mesh. By default, VisIt uses “X-Axis”, “Y-Axis”, and “Z-Axis” when labelling the coordinate axes. You can override the default labels using an option list. Option lists are created with the `DBMakeOptlist` function and freed with the `DBFreeOptlist` function. All of the Silo functions for writing meshes that we’ve demonstrated so far can accept option lists that contain custom axis labels and units. Refer to the *Silo User’s Manual* for more information on addition options that can be passed via option lists.

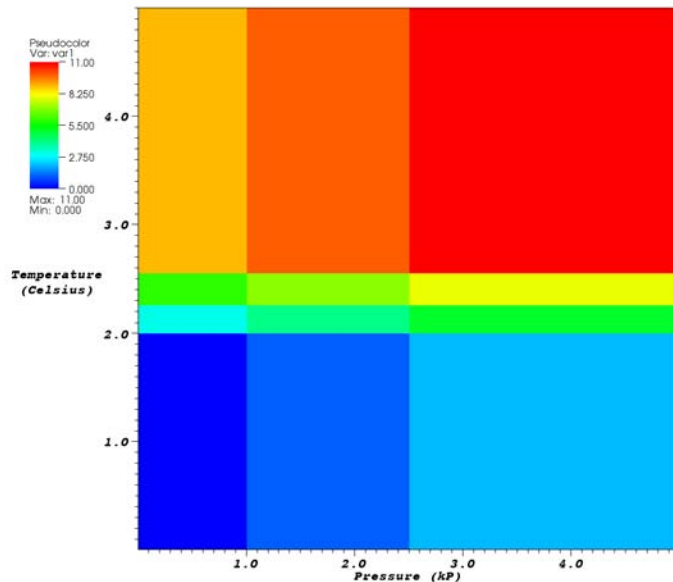


Figure 2-33: Custom mesh labels and units along the X and Y axes

Adding customized labels and units for a mesh by using option lists ensures that VisIt uses your customized labels and units instead of the default values. Figure 2-33 shows how the labels and units in the previous examples show up in VisIt's visualization window.

Listing 2-34: rect2d.c: C-Language example for associating new axis labels and units with a mesh.

```

/* Create an option list to contain labels and units. */
DBOptlist *optlist = DBMakeOptlist(4);
DBAddOption(optlist, DBOPT_XLABEL, (void *)"Pressure");
DBAddOption(optlist, DBOPT_XUNITS, (void *)"kP");
DBAddOption(optlist, DBOPT_YLABEL, (void *)"Temperature");
DBAddOption(optlist, DBOPT_YUNITS, (void *)"Degrees Celsius");
/* Write a quadmesh with an option list. */
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
    DB_FLOAT, DB_COLLINEAR, optlist);
/* Free the option list. */
DBFreeOptlist(optlist);

```

Listing 2-35: frect2d.f: Fortran language example for associating new axis labels and units with a mesh

```

c Create an option list to contain labels and units.
    integer err, ierr, optlistid
    err = dbmkoptlist(4, optlistid)
    err = dbaddcopt(optlistid, DBOPT_XLABEL, "Pressure", 8)
    err = dbaddcopt(optlistid, DBOPT_XUNITS, "kP", 2)
    err = dbaddcopt(optlistid, DBOPT_YLABEL, "Temperature", 11)
    err = dbaddcopt(optlistid, DBOPT_YUNITS, "Celsius", 7)
c Write a quadmesh with an option list.
    err = dbputqm (dbfile, "quadmesh", 8, "xc", 2,
    . "yc", 2, "zc", 2, x, y, DB_F77NULL, dims, ndims,
    . DB_FLOAT, DB_COLLINEAR, optlistid, ierr)
c Free the option list
    err = dbfreeoptlist(optlistid)

```

4.11 Writing a scalar variable

Silo provides several different functions for writing variables; one for each basic type of mesh: quadmesh (rectilinear and curvilinear), unstructured mesh, and point mesh. Each of these functions can be used to write either zone-centered or node-centered data. This section concentrates on how to write scalar variables; vector and tensor variable components can be written as scalar variables and reassembled into vectors and tensors using expressions, covered on page 45. This section's code examples use the rectilinear, curvilinear, point, and unstructured meshes that have appeared in previous code examples.

4.11.1 Zone centering vs. Node centering

VisIt supports two types of variable centering: zone-centering and node-centering. A variable's centering indicates how its values are attached to the mesh on which the variable is defined. When a variable is zone-centered, each zone is assigned a single value. If you were to plot a zone-centered value in VisIt, each zone would be drawn using a uniform color and picking anywhere in the zone would yield the same value. Arrays containing values that are to be zone-centered on a mesh must contain the same number of elements as there are zones in the mesh. Node-centered arrays, on the other hand, contain a value for every node in the mesh. When you plot a node-centered value in VisIt, VisIt interpolates the values from the nodes across the zone's surface, usually producing a smooth gradient of values across the zone.

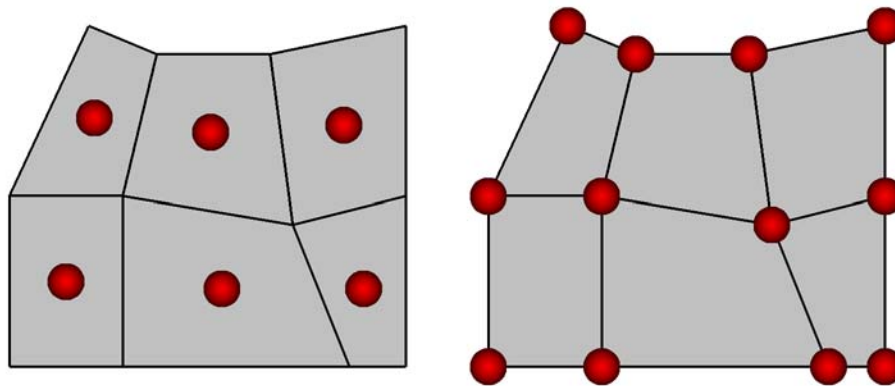


Figure 2-36: Zone-centering (left) and Node-centering (right)

4.11.2 API Commonality

Each of the provided functions for writing scalar variables does have certain arguments in common. For example, all of the functions must be provided the name of the variable to write out. The name that you pick is the name that will appear in VisIt's plot menus (see Figure 2-37). Be careful when you pick your variable names because you should avoid characters that include: punctuation marks, and spaces. Variable names should only contain letters and numbers and they should begin with a letter. These guidelines are in place to assure that your data files will have the utmost compatibility with VisIt's expression language, which is defined in the *VisIt User's Manual*.

All variables must be defined on a mesh. If you examine the code examples in this section, each Silo function that writes out a variable will be passed the name of the mesh on which the variable is to be defined.

Each of the Silo function calls will accept a pointer to the array that contains the variable's data. The data can be stored in several internal formats: `char`, `short`, `int`, `long`, `float`, and `double`. Since Silo's variable writing functions use a pointer to pass the data, you can pass a pointer that points to data in any of the mentioned types. In addition,

you must pass a flag that indicates to Silo the type of data stored in the array whose address you've passed.

Most of the remaining arguments to Silo's variable writing functions are specific to the types of meshes on which the variable is defined so the rest of this section will provide examples for writing out variables that are defined on various mesh types.

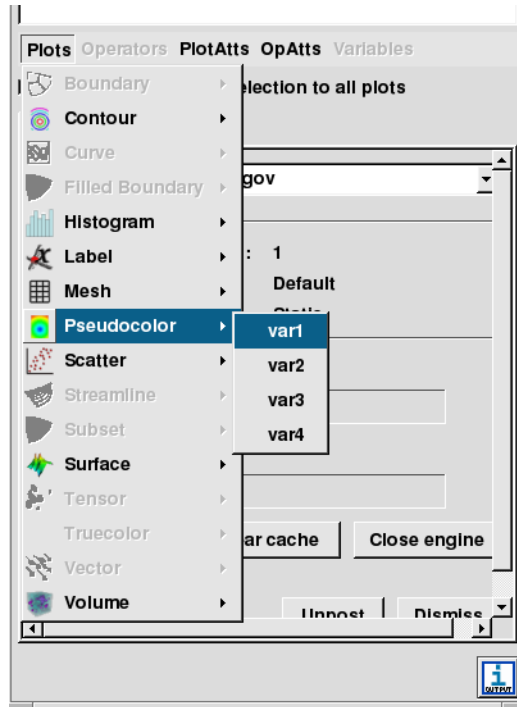


Figure 2-37: Variables in VisIt's plot menus

4.11.3 Rectilinear and curvilinear meshes

Recall from sections “Writing a rectilinear mesh” on page 18 and “Writing a curvilinear mesh” on page 21 that the procedure for creating rectilinear and curvilinear meshes was similar and the chief difference between the two mesh types was in how their coordinates were specified. While a rectilinear mesh's coordinates could be specified quite compactly as separate X,Y,Z arrays made up of unique values along a coordinate axis, the curvilinear mesh required X,Y,Z coordinate arrays that contained the X,Y,Z values for every node in the mesh. Regardless of how the coordinates were specified, both mesh types contain $(NX-1)*(NY-1)*(NZ-1)$ zones and $NX*NY*NZ$ nodes. This means that the code to write a variable on a rectilinear mesh will be identical to the code to write a zone-centered variable on a curvilinear mesh! Silo provides the `DBPutQuadvar1` function to write scalar variables for both rectilinear and curvilinear meshes,

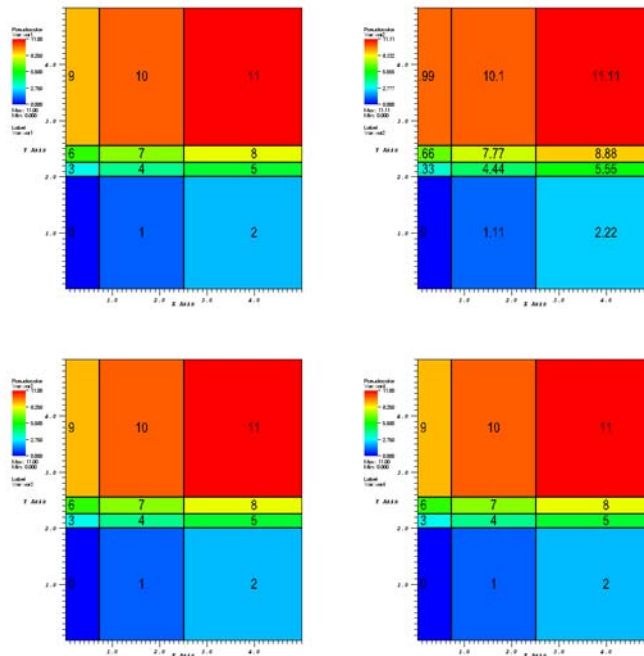


Figure 2-38: Zone-centered variables. Clock-wise from upper left, float, double-precision, integer, char

Listing 2-39: quadvar2d.c: C-Language example for writing zone-centered variables.

```

/* The data must be (NX-1) * (NY-1) since it is zonal. */
float var1[] = {
    0., 1., 2.,
    3., 4., 5.,
    6., 7., 8.,
    9., 10., 11.
};
double var2[] = {
    0.00, 1.11, 2.22,
    3.33, 4.44, 5.55,
    6.66, 7.77, 8.88,
    9.99, 10.1, 11.11
};
int var3[] = {
    0, 1, 2,
    3, 4, 5,
    6, 7, 8,
    9, 10, 11
};
char var4[] = {
    0, 1, 2,
    3, 4, 5,

```

```

        6, 7, 8,
        9, 10, 11
    };
    /* Note dims are 1 less than mesh's dims in each dimension. */
    int dims[]={3, 4};
    int ndims = 2;
    DBPutQuadvar1(dbfile, "var1", "quadmesh", var1, dims,
        ndims, NULL, 0, DB_FLOAT, DB_ZONECENT, NULL);
    /* Write a double-precision variable. */
    DBPutQuadvar1(dbfile, "var2", "quadmesh", (float*)var2, dims,
        ndims, NULL, 0, DB_DOUBLE, DB_ZONECENT, NULL);
    /* Write an integer variable */
    DBPutQuadvar1(dbfile, "var3", "quadmesh", (float*)var3, dims,
        ndims, NULL, 0, DB_INT, DB_ZONECENT, NULL);
    /* Write a char variable */
    DBPutQuadvar1(dbfile, "var4", "quadmesh", (float*)var4, dims,
        ndims, NULL, 0, DB_CHAR, DB_ZONECENT, NULL);

```

Listing 2-40: fquadvar2d.f: Fortran language example for writing zone-centered variables.

```

    integer err, ierr, dims(2), ndims, NX, NY, ZX, ZY
    parameter (NX = 4)
    parameter (NY = 5)
    parameter (ZX = NX-1)
    parameter (ZY = NY-1)
    real          var1(ZX,ZY)
    double precision var2(ZX,ZY)
    integer       var3(ZX,ZY)
    character     var4(ZX,ZY)
    data var1/0., 1., 2.,
    . 3., 4., 5.,
    . 6., 7., 8.,
    . 9., 10., 11./
    data var2/0.,1.11,2.22,
    . 3.33, 4.44, 5.55,
    . 6.66, 7.77, 8.88,
    . 9.99, 10.1, 11.11/
    data var3/0,1,2,
    . 3, 4, 5,
    . 6, 7, 8,
    . 9, 10, 11/
    data var4/0,1,2,
    . 3, 4, 5,
    . 6, 7, 8,
    . 9, 10, 11/
    data dims/ZX, ZY/
    ndims = 2
    err = dbputqv1(dbfile, "var1", 4, "quadmesh", 8, var1, dims,
    . ndims, DB_F77NULL, 0, DB_FLOAT, DB_ZONECENT, DB_F77NULL, ierr)
c Write a double-precision variable
    err = dbputqv1(dbfile, "var2", 4, "quadmesh", 8, var2, dims,
    . ndims, DB_F77NULL, 0, DB_DOUBLE, DB_ZONECENT,
    . DB_F77NULL, ierr)

```

```

c Write an integer variable
    err = dbputqv1(dbfile, "var3", 4, "quadmesh", 8, var3, dims,
        . ndims, DB_F77NULL, 0, DB_INT, DB_ZONECENT, DB_F77NULL, ierr)
c Write a char variable
    err = dbputqv1(dbfile, "var4", 4, "quadmesh", 8, var4, dims,
        . ndims, DB_F77NULL, 0, DB_CHAR, DB_ZONECENT, DB_F77NULL, ierr)

```

Both of the previous code examples produce a data file with 4 different scalar arrays as shown in Figure 2-38. Note that in both of the previous code examples, the same DBPutQuadvar1 function (or dbputqv1 in Fortran) function was used to write out data arrays of differing types.

The DBPutQuadvar1 function can also be used to write out node centered variables. There are two differences that you must observe when writing a node-centered variable as opposed to writing a zone-centered variable. First, the data array that you pass to the DBPutQuadvar1 function must be larger by 1 in each of its dimensions and you must pass DB_NODECENT instead of DB_ZONECENT.

Listing 2-41: quadvar2d.c: C-Language example for writing node-centered variables.

```

/* The data must be NX * NY since it is nodal. */
#define NX 4
#define NY 5
float nodal[] = {
    0., 1., 2., 3.,
    4., 5., 6., 7.,
    8., 9., 10., 11.,
    12., 13., 14., 15.,
    16., 17., 18., 19.
};
/* Nodal variables have same #values as #nodes in mesh */
int dims[]={NX, NY};
int ndims = 2;
DBPutQuadvar1(dbfile, "nodal", "quadmesh", nodal, dims,
    ndims, NULL, 0, DB_FLOAT, DB_NODECENT, NULL);

```

Listing 2-42: fquadvar2d.f: Fortran language example for writing node-centered variables.

```

c The data must be NX * NY since it is nodal.
    integer err, ierr, dims(2), ndims, NX, NY
    parameter (NX = 4)
    parameter (NY = 5)
    real    nodal(NX, NY)
    data dims/NX, NY/
    data nodal/0., 1., 2., 3.,
    . 4., 5., 6., 7.,
    . 8., 9., 10., 11.,
    . 12., 13., 14., 15.,
    . 16., 17., 18., 19./

```

```

    ndims = 2
    c Nodal variables have same #values as #nodes in mesh
    err = dbputqv1(dbfile, "nodal", 5, "quadmesh", 8, nodal,
        . dims, ndims, DB_F77NULL, 0, DB_FLOAT, DB_NODECENT,
        . DB_F77NULL, ierr)

```

Writing variables to 3D curvilinear and rectilinear meshes follows the same basic rules as writing variables for 2D meshes. For zone-centered variables, you must have $(NX-1)*(NY-1)*(NZ-1)$ data values and for node-centered variables, you must have $NX*NY*NZ$ data values. Figure 2-43 shows what the data values look like for the Silo files produced by the examples to come.

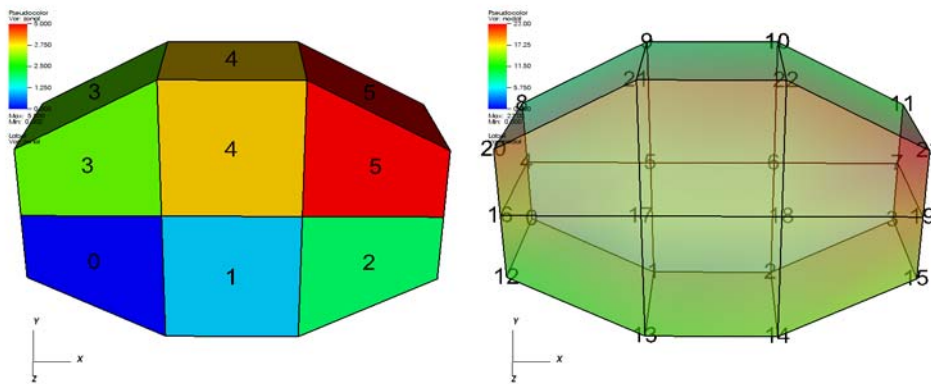


Figure 2-43: Zone-centered variable in 3D and a node-centered variable in 3D (shown with a partially transparent plot)

Listing 2-44: quadvar3d.c: C-Language example for writing variables on a 3D mesh.

```

#define NX 4
#define NY 3
#define NZ 2
/* Write a zone-centered variable. */
void write_zonecent_quadvar(DBfile *dbfile)
{
    int i, dims[3], ndims = 3;
    int ncells = (NX-1)*(NY-1)*(NZ-1);
    float *data = (float *)malloc(sizeof(float)*ncells);
    for(i = 0; i < ncells; ++i)
        data[i] = (float)i;
    dims[0] = NX-1; dims[1] = NY-1; dims[2] = NZ-1;
    DBPutQuadvar1(dbfile, "zonal", "quadmesh", data, dims,
        ndims, NULL, 0, DB_FLOAT, DB_ZONECENT, NULL);
    free(data);
}
/* Write a node-centered variable. */

```

```

void write_nodacent_quadvar(DBfile *dbfile)
{
    int i, dims[3], ndims = 3;
    int nnodes = NX*NY*NZ;
    float *data = (float *)malloc(sizeof(float)*nnodes);
    for(i = 0; i < nnodes; ++i)
        data[i] = (float)i;
    dims[0] = NX; dims[1] = NY; dims[2] = NZ;
    DBPutQuadvar1(dbfile, "nodal", "quadmesh", data, dims,
        ndims, NULL, 0, DB_FLOAT, DB_NODECENT, NULL);
    free(data);
}

```

Listing 2-45: fquadvar3d.f: Fortran language example for writing variables on a 3D mesh.

c Write a zone-centered variable.

```

subroutine write_zonecent_quadvar(dbfile)
    implicit none
    integer dbfile
    include "silo.inc"
    integer err, ierr, dims(3), ndims, i,j,k,index, ZX,ZY,ZZ
    parameter (ZX = 3)
    parameter (ZY = 2)
    parameter (ZZ = 1)
    integer zonal(ZX, ZY, ZZ)
    data dims/ZX, ZY, ZZ/
    index = 0
    do 10020 k=1,ZZ
    do 10010 j=1,ZY
    do 10000 i=1,ZX
        zonal(i,j,k) = index
        index = index + 1
10000 continue
10010 continue
10020 continue
    ndims = 3
    err = dbputqv1(dbfile, "zonal", 5, "quadmesh", 8, zonal, dims,
        . ndims, DB_F77NULL, 0, DB_INT, DB_ZONECENT, DB_F77NULL, ierr)
    end

```

c Write a node-centered variable.

```

subroutine write_nodacent_quadvar(dbfile)
    implicit none
    integer dbfile
    include "silo.inc"
    integer err, ierr, dims(3), ndims, i,j,k,index, NZ, NY, NX
    parameter (NX = 4)
    parameter (NY = 3)
    parameter (NZ = 2)
    real nodal(NX, NY, NZ)
    data dims/NX, NY, NZ/
    index = 0
    do 20020 k=1,NZ
    do 20010 j=1,NY

```



```

do 20000 i=1,NX
  nodal(i,j,k) = float(index)
  index = index + 1
20000 continue
20010 continue
20020 continue
  ndims = 3
  err = dbputqv1(dbfile, "nodal", 5, "quadmesh", 8, nodal, dims,
. ndims, DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL, ierr)
end

```

4.11.4 Point meshes

Point meshes, which were meshes composed of a set of points can, like other mesh types, have values associated with each point. Silo provides the `DBPutPointVar1` function that you can use to write out a scalar variable stored on a point mesh. Nodes and the zones are really the same thing in a point mesh so you can consider zone-centered scalars to be the same thing as node-centered scalars.

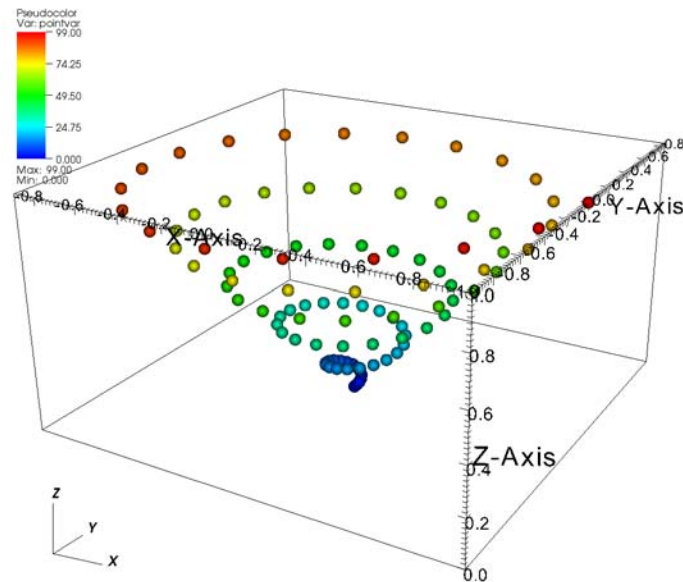


Figure 2-46: Scalar variable defined on a point mesh

Listing 2-47: `pointvar3d.c`: C-Language example for writing variables on a 3D point mesh.

```

/* Create some values to save. */
int i;
float var[NPTS];

```

```

for(i = 0; i < NPTS; ++i)
    var[i] = (float)i;
/* Write the point variable. */
DBPutPointvar1(dbfile, "pointvar", "pointmesh", var, NPTS,
    DB_FLOAT, NULL);

```

Listing 2-48: fpointvar3d.f: Fortran language example for writing variables on a 3D point mesh.

```

c Create some values to save.
    integer err, ierr, i, NPTS
    parameter (NPTS = 100)
    real var(NPTS)
    do 10010 i = 1,NPTS
        var(i) = float(i-1)
10010 continue
c Write the point variable
    err = dbputpv1(dbfile, "pointvar", 8, "pointmesh", 9,
        . var, NPTS, DB_FLOAT, DB_F77NULL, ierr)

```

4.11.5 Unstructured meshes

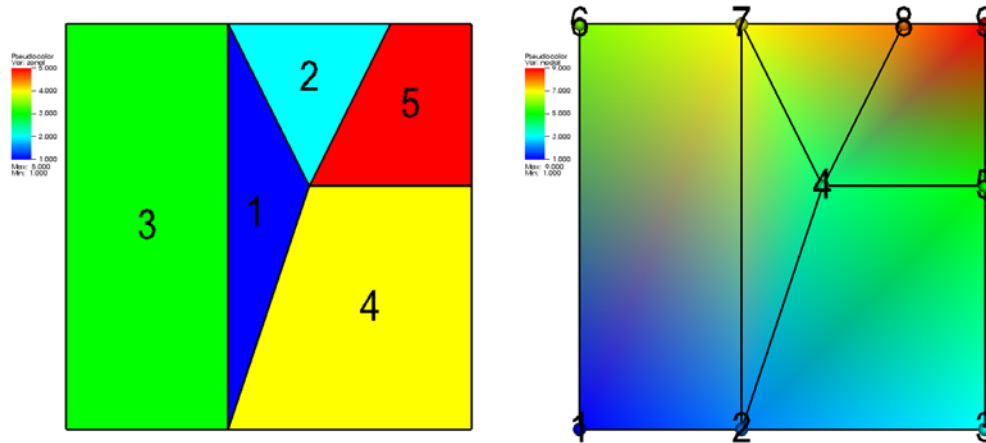


Figure 2-49: A 2D unstructured mesh with a zonal variable (left) and a nodal variable (right).

Writing a variable on an unstructured mesh is done following a procedure similar to that for writing a variable on a point mesh. As with other mesh types, a scalar variable defined on an unstructured grid can be zone-centered or node-centered. If the variable is zone-centered then the data array required to store the variable on the unstructured mesh must be a 1-D array with the same number of elements as the mesh has zones. If the variable to be stored is node-centered then the array containing the variable must be a 1-D array with the same number of elements as the mesh has nodes. Thinking of the data array as a 1-D array simplifies indexing since the number used to identify a particular node is the same

index that would be used to access data in the variable array (assuming 0-origin in C and 1-origin in Fortran). Since the data array is always 1-D for an unstructured mesh, the code to store variables on 2D and 3D unstructured meshes is identical. Figure 2-49 shows a 2D unstructured mesh with both zonal and nodal variables. Silo provides the `DBPutUcdvar1` function for writing scalar variables on unstructured meshes.

Listing 2-50: `ucdvar2d.c`: C-Language example for writing variables on an unstructured mesh.

```
float nodal[] = {1.,2.,3.,4.,5.,6.,7.,8.,9.};
float zonal[] = {1.,2.,3.,4.,5.};
int nnodes = 9;
int nzones = 5;
/* Write a zone-centered variable. */
DBPutUcdvar1(dbfile, "zonal", "mesh", zonal, nzones, NULL, 0,
             DB_FLOAT, DB_ZONECENT, NULL);
/* Write a node-centered variable. */
DBPutUcdvar1(dbfile, "nodal", "mesh", nodal, nnodes, NULL, 0,
             DB_FLOAT, DB_NODECENT, NULL);
```

Listing 2-51: `fucdvar2d.f`: Fortran language example for writing variables on an unstructured mesh.

```
integer err, ierr, NNODES, NZONES
parameter (NNODES = 9)
parameter (NZONES = 5)
real nodal(NNODES) /1.,2.,3.,4.,5.,6.,7.,8.,9./
real zonal(NZONES) /1.,2.,3.,4.,5./
c Write a zone-centered variable.
err = dbputuv1(dbfile, "zonal", 5, "mesh", 4, zonal, NZONES,
              . DB_F77NULL, 0, DB_FLOAT, DB_ZONECENT, DB_F77NULL, ierr)
c Write a node-centered variable.
err = dbputuv1(dbfile, "nodal", 5, "mesh", 4, nodal, NNODES,
              . DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL, ierr)
```

4.11.6 Adding variable units

All of the examples for writing scalar variables presented so far have focused on the basics of writing a variable array to a Silo file. Silo's option list mechanism allows a variable object to be annotated with various extra information. In the case of scalar variables, the option list passed to `DBPutQuadvar1` and `DBPutUcdvar1` can contain the units that describe the variable being stored. Refer to the *Silo User's Manual* for a complete list of the options accepted by the `DBPutQuadvar1` and `DBPutUcdvar1` functions. When a scalar variable has associated units, the units appear in the variable legend in VisIt's visualization window (see Figure 2-52).

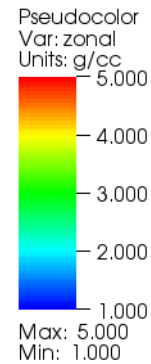


Figure 2-52: Plot legend with units

If you want to add units to the variable that you write, you must create an option list to pass to the function writing your variable. You may recall that option lists are created with the `DBMakeOptlist` function and freed with the `DBFreeOptlist` function. In order to add units to the option list, you must add the `DBOPT_UNITS` option.

Listing 2-53: ucdvar2d.c: C-Language example for writing a variables with units.

```
/* Create an option list and add "g/cc" units to it. */
DBOptlist *optlist = DBMakeOptlist(1);
DBAddOption(optlist, DBOPT_UNITS, (void*)"g/cc");
/* Write a variable that has units. */
DBPutUcdvar1(dbfile, "zonal", "mesh", zonal, nzones, NULL, 0,
             DB_FLOAT, DB_ZONECENT, optlist);
/* Free the option list. */
DBFreeOptlist(optlist);
```

Listing 2-54: fucdvar2d.f: Fortran language example for writing a variables with units.

```
c Create an option list and add "g/cc" units to it.
   integer err, optlistid
   err = dbmkoptlist(1, optlistid)
   err = dbaddcopt(optlistid, DBOPT_UNITS, "g/cc", 4)
c Write a variable that has units.
   err = dbputuv1(dbfile, "zonal", 5, "mesh", 4, zonal, NZONES,
   . DB_F77NULL, 0, DB_FLOAT, DB_ZONECENT, optlistid, ierr)
c Free the option list.
   err = dbfreeoptlist(optlistid)
```

4.12 Single precision vs. Double precision

After having written some variables to a Silo file, you've no doubt learned that you can pass a pointer to data of many different representations and precisions (char, int, float, double, etc.). When you pass data to a Silo function, you also must pass a flag that tells Silo how to interpret the data stored in your data array. For example, if you have single precision floating point data then you would tell Silo to traverse the data as such using the `DB_FLOAT` type flag in the function call to `DBPutQuadvar1`. Many of the functions in the Silo library require a type flag to indicate the type of data being passed to Silo. In fact, even the functions to write mesh coordinates can accept different data types. This means that you can use double-precision to specify your mesh coordinates, which can be immensely useful when dealing with very large or very small objects.

Listing 2-55: C-Language example for writing a mesh with double-precision coordinates.

```
/* The x,y arrays contain double-precision coordinates. */
double x[NY][NX], y[NY][NX];
int dims[] = {NX, NY};
int ndims = 2;
/* Note that x,y pointers are cast to float to conform to API. */
float *coords[] = {(float*)x, (float*)y};
/* Tell Silo that the coordinate arrays are actually doubles. */
DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
    DB_DOUBLE, DB_NONCOLLINEAR, NULL);
```

4.13 Writing expressions

You can plot derived quantities in VisIt by creating expressions that involve variables from your database. Sometimes, it is useful to include expression definitions in your Silo file so they are available to VisIt without you first having to create them. Silo provides the `DBPutdefvars` function so you can write your expressions to a Silo file. Expression names should be valid VisIt expression names, as defined in the *VisIt User's Manual*. Likewise, the expression definitions should contain only expressions that are supported by the VisIt expression language.

While VisIt's expression language can be useful for calculating a multitude of expressions, it can be particularly useful for grouping vector or tensor components into vector and tensor variables. If you store vector or tensor components as scalar variables in your Silo file then you can easily create expressions that assemble the components into real vector or tensor variables without significantly increasing your file's storage requirements. Writing out vector and tensor variables as expressions involving scalar variables also prevents you from having to use more complicated Silo functions in order to write out the vector or tensor data.

Listing 2-56: `defvars.c`: C-Language example for writing out expression definitions.

```

/* Write some expressions to the Silo file. */
const char *names[] = {"velocity", "speed"};
const char *defs[] = {"{xc,yc,zc}", "magnitude(velocity)"};
int types[] = {DB_VARTYPE_VECTOR, DB_VARTYPE_SCALAR};
DBPutDefvars(dbfile, "defvars", 2, names, types, defs, NULL);

```

Listing 2-57: fdefvars.f: Fortran language example for writing out expression definitions.

```

integer err, ierr, types(2), lnames(2), ldefs(2)
integer numexpressions, oldlen
c Initialize some 20 character length strings
character*20 names(2) //velocity      ',
. 'speed                          '/'
character*20 defs(2) //'{xc,yc,zc}    ',
. 'magnitude(velocity) '/'
c Store the length of each string
data lnames/8, 5/
data ldefs/10, 19/
data types/DB_VARTYPE_VECTOR, DB_VARTYPE_SCALAR/
c Set the maximum string length to 20 since that's how long
c our strings are
oldlen = dbget2dstrlen()
err = dbset2dstrlen(20)
c Write out the expressions
numexpressions = 2
err = dbputdefvars(dbfile, "defvars", 7, numexpressions,
. names, lnames, types, defs, ldefs, DB_F77NULL, ierr)
c Restore the previous value for maximum string length
err = dbset2dstrlen(oldlen)

```

In the previous Fortran example for writing expressions, there are more functions involved than just the `dbputdefvars` function. It is critical to set the maximum 2D string length for strings in the Silo library, using the `dbset2dstrlen` function, so the Fortran interface to Silo will be able to correctly traverse the string data passed to it from Fortran. In the previous example, we used 20 characters for both the expression names and definitions. We call `dbset2dstrlen` to set the maximum allowable 2d string length to 20 characters before we pass our arrays of 20 character strings to the `dbputdefvars` function. In addition, we must also pass valid lengths for the expression name and definition strings. The lengths should be at least 1 character long but no longer than the maximum allowable string length, which we set to 20 characters in the example program. Passing valid string lengths is important so the expressions that you save to your file do not contain any extra characters, such as trailing spaces.

4.14 Creating a master file for parallel

When a parallel program saves out its data files, often the most efficient method of I/O is for each processor to write its own piece of the simulation, or domain, to its own Silo file.

If each processor writes its own Silo file then no communication or synchronization must take place to manage access to a shared file. However, once the simulation has completed, there are many files and all of them are required to reconstitute the simulated object. Plotting each domain file in VisIt would be very tedious so Silo provides functions to create what is known as a “master file”, which is a top-level file that effectively unifies all of the domain files into a whole. When you open a master file in VisIt and plot variables out of it, all domains are plotted.

Master files contain what are known as multimeshes, multivars, and multimaterials. These objects are lists of filenames that contain the appropriate domain variable. They also contain some meta-information about each of the domains that helps VisIt perform better in parallel. Strategies for using metadata to improve VisIt’s I/O performance will be covered shortly.

4.14.1 Creating a multimesh

A multimesh is an object that unites smaller domain-sized meshes into a whole mesh. The multimesh object contains a list of the filenames that contain a piece of the named mesh. When you tell VisIt to plot a multimesh, VisIt reads the named mesh in all of the required domain files and processes the mesh in each file, to produce the entire mesh.

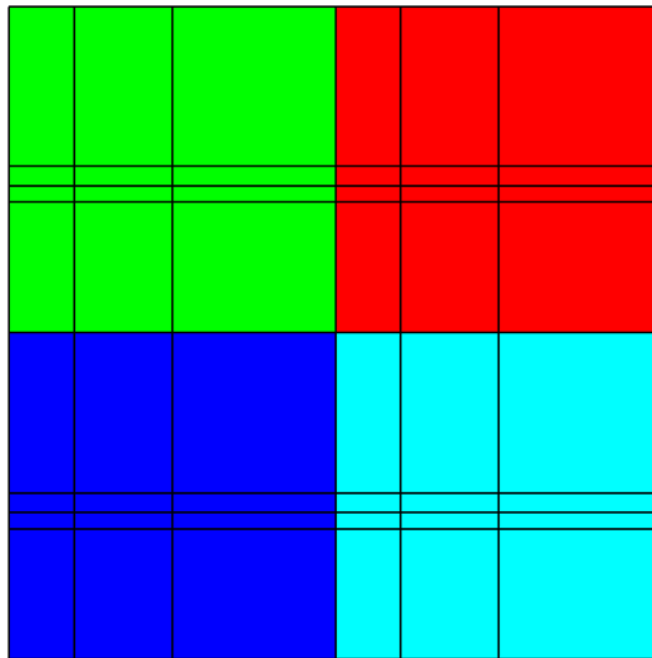


Figure 2-58: Multimesh colored by its domain number

The following example, shown in Figure 2-58, uses the mesh from the 2D rectilinear mesh example program and repeats it as 4 domains. Note that the mesh forming the domains is translated in X and Y so that the edges are shared. In the given example, the meshes that make up the entire mesh are stored in separate Silo files: *multimesh.1*, *multimesh.2*,

multimesh.3, and *multimesh.4*. The mesh and any data that may be defined on it is stored in those files. Remember that storing pieces of a single mesh is commonplace when parallel processes write their own file. Plotting each of the smaller files individually in VisIt is not necessary when a master file has been generated since plotting the multimesh object from the master file will cause VisIt to plot each of its constituent meshes. The code that will follow shows how to use Silo's `DBPutMultimesh` function to write out a multimesh object that reassembles meshes from many domain files into a whole mesh.

The list of meshes or items in a multi-object generally take the form: *path:item* where *path* is the file system path to the item and *item* is the name of the object being referenced. Note that the path may be specified as a relative or absolute path using names valid for the file system containing the master file. However, we strongly recommend using only relative paths so the master file does not reference directories that exist only on one file system. Using relative paths makes the master files much more portable since they allow the data files to be moved. The path may also refer to subdirectories within the file being referenced since Silo files may contain directories that help to organize related data. The following examples assume that the domain files will exist in the same directory as the master file since the path includes only the names of the domain files.

Listing 2-59: `multimesh.c`: C-Language example for writing a multimesh.

```
void write_masterfile(void)
{
    DBfile *dbfile = NULL;
    char **meshnames = NULL;
    int dom, nmesh = 4, *meshtypes = NULL;
    /* Create the list of mesh names. */
    meshnames = (char **)malloc(nmesh * sizeof(char *));
    for(dom = 0; dom < nmesh; ++dom)
    {
        char tmp[100];
        sprintf(tmp, "multimesh.%d:quadmesh", dom);
        meshnames[dom] = strdup(tmp);
    }
    /* Create the list of mesh types. */
    meshtypes = (int *)malloc(nmesh * sizeof(int));
    for(dom = 0; dom < nmesh; ++dom)
        meshtypes[dom] = DB_QUAD_RECT;
    /* Open the Silo file */
    dbfile = DBCreate("multimesh.root", DB_CLOBBER, DB_LOCAL,
        "Master file", DB_HDF5);
    /* Write the multimesh. */
    DBPutMultimesh(dbfile, "quadmesh", nmesh, meshnames,
        meshtypes, NULL);
    /* Close the Silo file. */
    DBClose(dbfile);
    /* Free the memory*/
    for(dom = 0; dom < nmesh; ++dom)
        free(meshnames[dom]);
    free(meshnames);
}
```



```

    free(meshtypes);
}

```

Listing 2-60: fmultimesh.f: Fortran language example for writing a multimesh.

```

subroutine write_master()
implicit none
include "silo.inc"
integer err, ierr, dbfile, nmesh, oldlen
character*20 meshnames(4) /'multimesh.1:quadmesh',
.                          'multimesh.2:quadmesh',
.                          'multimesh.3:quadmesh',
.                          'multimesh.4:quadmesh'/
integer lmeshnames(4) /20,20,20,20/
integer meshtypes(4) /DB_QUAD_RECT, DB_QUAD_RECT,
.                    DB_QUAD_RECT, DB_QUAD_RECT/
c Create a new silo file
err = dbcreate("multimesh.root", 14, DB_CLOBBER, DB_LOCAL,
. "multimesh root", 14, DB_HDF5, dbfile)
if(dbfile.eq.-1) then
write (6,*) 'Could not create Silo file!\n'
return
endif
c Set the maximum string length to 20 since that's how long our
c strings are
oldlen = dbget2dstrlen()
err = dbset2dstrlen(20)
c Write the multimesh object.
nmesh = 4
err = dbputmmesh(dbfile, "quadmesh", 8, nmesh, meshnames,
. lmeshnames, meshtypes, DB_F77NULL, ierr)
c Restore the previous value for maximum string length
err = dbset2dstrlen(oldlen)
c Close the Silo file
err = dbclose(dbfile)
end

```

Sometimes it can be advantageous to have each processor write its files to a unique subdirectory (e.g. proc-0, proc-1, proc-2, ...). You can also choose for each processor to write its files to a common directory so all files for a given time step are contained in a single place (e.g. cycle0000, cycle0001, cycle0002, ...). Generally, you will want to tailor your strategy to the strengths of your file system to spread the demands of writing files across as many I/O nodes as possible in order to increase throughput. The organization strategies mentioned so far are only suggestions and you will have to determine the optimum method for storing domain files on your computer system. Moving your domain files to subdirectories can make it easier to navigate your file system and can provide benefits later such as VisIt not having to check permissions, etc on so many files. Code to

create the list of mesh names where each processor writes its data to a different subdirectory that contains all files for a given time step might look like the following:

```

int cycle = 100;
for(dom = 0; dom < nmesh; ++dom)
{
    char tmp[100];
    sprintf(tmp, "proc-%d/multimesh.%04d:quadmesh", dom, cycle);
    meshnames[dom] = strdup(tmp);
}

```

4.14.2 Creating a multivar

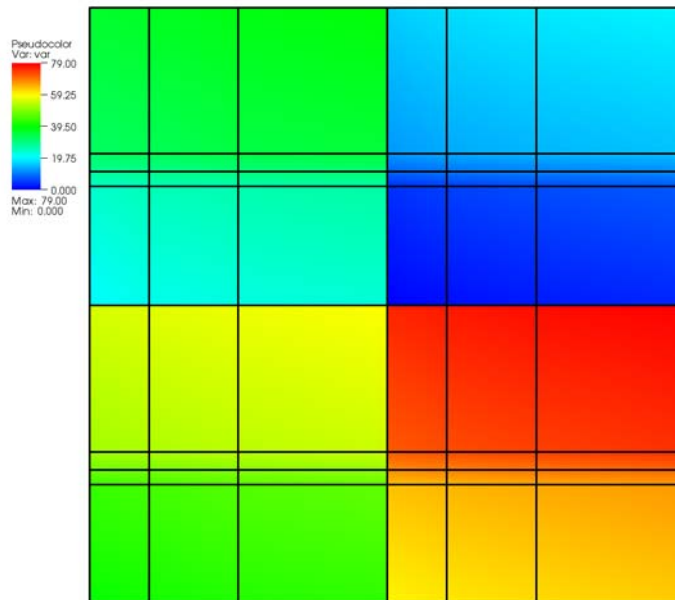


Figure 2-61: Multivar displayed on its multimesh

A multivar object is the variable equivalent of a multimesh object. Like the multimesh object, a multivar object contains a list of filenames that make up the variable represented by the multivar object. Silo provides the `DBPutMultivar` function for writing out multivar objects.

Listing 2-62: multivar.c: C-Language example for writing a multivar.

```

void write_multivar(DBfile *dbfile)
{

```

```

char **varnames = NULL;
int dom, nvar = 4, *vartypes = NULL;
/* Create the list of var names. */
varnames = (char **)malloc(nvar * sizeof(char *));
for(dom = 0; dom < nvar; ++dom)
{
    char tmp[100];
    sprintf(tmp, "multivar.%d:var", dom);
    varnames[dom] = strdup(tmp);
}
/* Create the list of var types. */
vartypes = (int *)malloc(nvar * sizeof(int));
for(dom = 0; dom < nvar; ++dom)
    vartypes[dom] = DB_QUADVAR;
/* Write the multivar. */
DBPutMultivar(dbfile, "var", nvar, varnames, vartypes, NULL);
/* Free the memory*/
for(dom = 0; dom < nvar; ++dom)
    free(varnames[dom]);
free(varnames);
free(vartypes);
}

```

Listing 2-63: fmultivar.f: Fortran language example for writing a multivar.

```

subroutine write_multivar(dbfile)
implicit none
include "silo.inc"
integer err, ierr, dbfile, nvar, oldlen
character*20 varnames(4) /'multivar.1:var    ',
.                               'multivar.2:var    ',
.                               'multivar.3:var    ',
.                               'multivar.4:var    '/
integer lvarnames(4) /14,14,14,14/
integer vartypes(4) /DB_QUADVAR,DB_QUADVAR,
.                   DB_QUADVAR,DB_QUADVAR/
c Set the maximum string length to 20 since that's how long
c our strings are
oldlen = dbget2dstrlen()
err = dbset2dstrlen(20)
c Write the multivar.
nvar = 4
err = dbputmvar(dbfile, "var", 3, nvar, varnames, lvarnames,
. vartypes, DB_F77NULL, ierr)
c Restore the previous value for maximum string length
err = dbset2dstrlen(oldlen)
end

```

4.14.3 EMPTY contributions

During the course of a calculation, sometimes only a subset of processors will contribute data. This means that they will not write data files. When some processors do not write data files, creating your multi-objects can become more complicated. Note that because of how VisIt represents its domain subsets, etc, you will want to keep the number of filenames in a multi-object equal to the number of processors that you are using (the maximum number of domains that you will generate). If the length of the list varies over time then VisIt's subsetting controls may not behave as expected. To keep things simple, if you have N processors that write N files, you will always want N entries in your multi-objects. If a processor does not contribute any data, insert the "EMPTY" keyword into the multi-object in place of the path and variable. The "EMPTY" keyword allows the size of the multi-object to remain fixed over time even as the number of processors that contribute data changes. Keeping the size of the multi-object fixed over time ensures that VisIt's subsetting controls will continue to function as expected. Note that if you use the "EMPTY" keyword in a multivar object then the same entry in the multimesh object for the variable must also contain the "EMPTY" keyword.

Listing 2-64: C-Language example using the EMPTY keyword.

```
/* Processors 3,4 did not contribute so use EMPTY. */
char *meshnames[] = {"proc-1/file000/mesh", "proc-2/file000/mesh",
                    "EMPTY", "EMPTY"};
int meshtypes[] = {DB_QUAD_RECT, DB_QUAD_RECT,
                  DB_QUAD_RECT, DB_QUAD_RECT};
int nmesh = 4;
/* Write the multimesh. */
DBPutMultimesh(dbfile, "mesh", nmesh, meshnames, meshtypes, NULL);
```

5.0 Writing VTK files

VTK (Visualization Toolkit) files provide a simple, flexible way to import data into VisIt. VTK files can be written in human-readable ASCII form or in binary form. VTK files may also be created in the legacy VTK file format or in their newer XML-based format. The human-readable ASCII form for legacy VTK files is described in the *VTK File Formats* document found on the Web at <http://public.kitware.com/VTK/pdf/file-formats.pdf>. You can create code in any language to write data to the VTK file format if you follow the format guidelines in the *VTK File Formats* document.

In order to simplify the creation of legacy VTK files, which can be susceptible to formatting mistakes, VisIt provides the `visit_writer` library. The `visit_writer` library is implemented in C and can be called from the C, C++, and Python programming languages. The `visit_writer` library provides a handful of easy-to-use functions for producing VTK files. This section will show how to use the `visit_writer` library to create VTK files that can be used to import data into VisIt.

5.1 Getting started with `visit_writer`

The `visit_writer` library is included in source code form in VisIt's source code distribution. The C-version of the library consists of 2 files called `visit_writer.c` and `visit_writer.h` that are stored in the `tools/writer` directory of VisIt's source code tree.

5.1.1 Using `visit_writer` in C programs

When you use the `visit_writer` library, you can include the `visit_writer.c` file directly in the list of source files for your project. Source files that use functions from the `visit_writer` library must include the `visit_writer.h` header file. The `visit_writer` library has no external dependencies so no additional libraries are required to link programs that use the `visit_writer` library, provided the `visit_writer.c` source code file was included in the project.

5.1.2 Using `visit_writer` in Python programs

The Python version of the `visit_writer` library is implemented as a Python extension module, which is a dynamically loaded executable file containing the `visit_writer` functions. The compiled `visit_writer` extension module is not currently distributed in VisIt's binary distributions so you will have to build it before you can use it in your Python programs. Fortunately, building the `visit_writer` module is easy if you allow Python to build it for you. To begin, open a terminal window and `cd` into VisIt's source code tree and then into the `tools/writer` directory. Next, type the following Python code into a file called `setup.py`:

```
from distutils.core import setup, Extension
module1 = Extension('visit_writer',
    include_dirs= ['.'],
    sources = ['visit_writer.c', 'py_visit_writer.c'])
setup (name = 'visit_writer',
    version = '1.0',
    description = 'This module lets us write VTK files.',
    ext_modules = [module1])
```

Once you have created the `setup.py` file, run the following command in your terminal window to build the `visit_writer` Python extension module.

```
python setup.py build
```

Once Python builds the `visit_writer` extension module, you can install it by running the following command:

```
python setup.py install
```

After the `visit_writer` module has been built and installed, it should be available when you run Python. To test whether the module was successfully installed, run `python` and type: `import visit_writer` at the Python prompt. If Python does not complain then the module was successfully built and loaded. Whenever you want to use the `visit_module` in

your Python scripts, you must first issue the `import visit_writer` directive. If you want to find out more information about a particular `visit_writer` function once you've imported the `visit_writer` module, you can type: `print visit_writer.__doc__` to make Python print out the documentation string for the `visit_writer` module.

5.2 Regular meshes with data

A regular mesh, or Cartesian mesh, is an implicit mesh in which all zones have the same size and are axis-aligned (see Figure 2-65). Furthermore, in this context, all zones are squares or cubes with a side length of 1. The extents are determined by the number of zones in each dimension. A regular mesh is a type of rectilinear mesh where the zones are not permitted to differ in size. The `visit_writer` library provides the `write_regular_mesh` function for writing out regular meshes and data to VTK files.

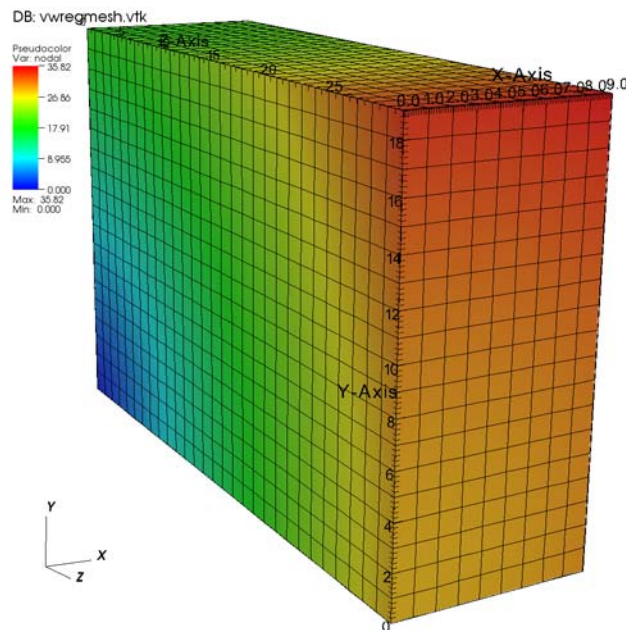


Figure 2-65: Regular mesh with data created using `visit_writer`

Listing 2-66: `vwregmesh.c`: C-Language example for writing a regular mesh with data.

```
#include <visit_writer.h>
#include <math.h>

int main(int argc, char *argv[])
{
#define NX 10
#define NY 20
```

```

#define NZ 30
  int i,j,k, index = 0;
  int dims[] = {NX, NY, NZ};
  int nvars = 2;
  int vardims[] = {1, 1};
  int centering[] = {0, 1};
  const char *varnames[] = {"zonal", "nodal"};
  float zonal[NZ-1][NY-1][NX-1], nodal[NZ][NY][NX];
  float *vars[] = {(float *)zonal, (float *)nodal};
  /* Create zonal variable */
  for(k = 0; k < NZ-1; ++k)
    for(j = 0; j < NY-1; ++j)
      for(i = 0; i < NX-1; ++i, ++index)
        zonal[k][j][i] = (float)index;
  /* Create nodal variable. */
  for(k = 0; k < NZ; ++k)
    for(j = 0; j < NY; ++j)
      for(i = 0; i < NX; ++i)
        nodal[k][j][i] = sqrt(i*i + j*j + k*k);
  /* Use visit_writer to write a regular mesh with data. */
  write_regular_mesh("vwregmesh.vtk", 0, dims, nvars, vardims,
    centering, varnames, vars);
  return 0;
}

```

Listing 2-67: vwregmesh.py: Python language example for writing a regular mesh with data.

```

import visit_writer, math
NX = 10
NY = 20
NZ = 30
# Create a zonal variable
zonal = []
index = 0
for k in range(NZ-1):
  for j in range(NY-1):
    for i in range(NX-1):
      zonal = zonal + [index]
      index = index + 1
# Create a nodal variable
nodal = []
for k in range(NZ):
  for j in range(NY):
    for i in range(NX):
      nodal = nodal + [math.sqrt(i*i + j*j + k*k)]
# Use visit_writer to write a regular mesh with data.
dims = (NX, NY, NZ)
vars = (("zonal", 1, 0, zonal), ("nodal", 1, 1, nodal))
visit_writer.WriteRegularMesh("vwregmesh2.vtk", 0, dims, vars)

```

5.3 Rectilinear meshes with data

Recall from “Writing a rectilinear mesh” on page 18 that a rectilinear mesh is a 2D or 3D mesh where all coordinates are aligned with the axes and coordinates along each axis can have different, non-uniform spacing. The `visit_writer` library provides the `write_rectilinear_mesh` function for writing rectilinear meshes. The following code examples will use the same 2D and 3D rectilinear meshes that were used for the Silo examples.

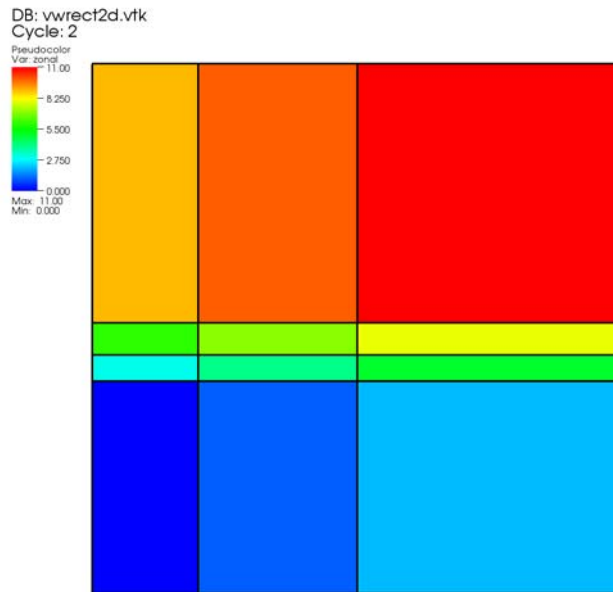


Figure 2-68: 2D rectilinear mesh with zonal variable

Listing 2-69: `vwrect2d.c`: C-Language example for writing a rectilinear mesh with data.

```
#include <visit_writer.h>

int main(int argc, char *argv[])
{
#define NX 4
#define NY 5
    /* Rectilinear mesh coordinates. */
    float x[] = {0., 1., 2.5, 5.};
    float y[] = {0., 2., 2.25, 2.55, 5.};
    float z[] = {0.};
    int dims[] = {NX, NY, 1};
    int ndims = 2;
    /* Zonal and Nodal variable data. */
    float zonal[NY-1][NX-1], nodal[NY][NX];
```



```

/* Info about the variables to pass to visit_writer. */
int nvars = 2;
int vardims[] = {1, 1};
int centering[] = {0, 1};
const char *varnames[] = {"zonal", "nodal"};
float *vars[] = {(float*)zonal, (float*)nodal};

/* Create a zonal variable. */
int i,j,index = 0;
for(j = 0; j < NY-1; ++j)
    for(i = 0; i < NX-1; ++i, ++index)
        zonal[j][i] = (float)index;

/* Create a nodal variable. */
index = 0;
for(j = 0; j < NY; ++j)
    for(i = 0; i < NX; ++i, ++index)
        nodal[j][i] = (float)index;

/* Pass the data to visit_writer to write a VTK file.*/
write_rectilinear_mesh("vwrect2d.vtk", 0, dims, x, y, z, nvars,
vardims, centering, varnames, vars);

return 0;
}

```

Listing 2-70: vwrect2d.py: Python language example for writing a rectilinear mesh with data.

```

import visit_writer

NX = 4
NY = 5
x = (0., 1., 2.5, 5.)
y = (0., 2., 2.25, 2.55, 5.)
z = 0.

# Create a zonal variable
zonal = []
index = 0
for j in range(NY-1):
    for i in range(NX-1):
        zonal = zonal + [index]
        index = index + 1

# Create a nodal variable
nodal = []
index = 0
for j in range(NY):
    for i in range(NX):
        nodal = nodal + [index]
        index = index + 1

```

```
vars = ("zonal", 1, 0, zonal), ("nodal", 1, 1, nodal))
visit_writer.WriteRectilinearMesh("vwrect2d.vtk", 0, x, y, z, vars)
```

5.4 Curvilinear meshes with data

A curvilinear mesh is similar to a rectilinear mesh; the main difference between the two mesh types is how coordinates are specified. Recall that in a rectilinear mesh, the coordinates are specified individually for each axis and only a small subset of the nodes in the mesh are provided. In a curvilinear mesh, you must provide an X,Y,Z value for every node in the mesh. The `visit_writer` library provides the `write_curvilinear_mesh` function to write out curvilinear meshes and any variables defined on them. Figure 2-71 shows an example of a 3D curvilinear mesh with a zonal variable.

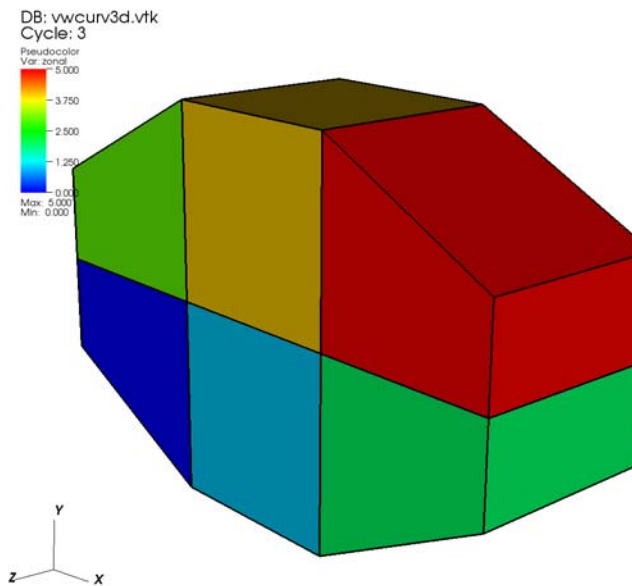


Figure 2-71: 3D curvilinear mesh with zonal variable

Listing 2-72: `vwcurv3d.c`: C-Language example for writing a curvilinear mesh with data.

```
#include <visit_writer.h>

#define NX 4
#define NY 3
#define NZ 2

int main(int argc, char *argv[])
{
```

```

/* Curvilinear mesh points stored x0,y0,z0,x1,y1,z1,...*/
float pts[] = {0, 0.5, 0, 1, 0, 0, 2, 0, 0,
               3, 0.5, 0, 0, 1, 0, 1, 1, 0,
               2, 1, 0, 3, 1, 0, 0, 1.5, 0,
               1, 2, 0, 2, 2, 0, 3, 1.5, 0,
               0, 0.5, 1, 1, 0, 1, 2, 0, 1,
               3, 0.5, 1, 0, 1, 1, 1, 1, 1,
               2, 1, 1, 3, 1, 1, 0, 1.5, 1,
               1, 2, 1, 2, 2, 1, 3, 1.5, 1
};
int dims[] = {NX, NY, NZ};
/* Zonal and nodal variable data. */
float zonal[NZ-1][NY-1][NX-1], nodal[NZ][NY][NX];
/* Info about the variables to pass to visit_writer. */
int nvars = 2;
int vardims[] = {1, 1};
int centering[] = {0, 1};
const char *varnames[] = {"zonal", "nodal"};
float *vars[] = {(float *)zonal, (float *)nodal};
int i,j,k, index = 0;

/* Create zonal variable */
for(k = 0; k < NZ-1; ++k)
    for(j = 0; j < NY-1; ++j)
        for(i = 0; i < NX-1; ++i, ++index)
            zonal[k][j][i] = (float)index;

/* Create nodal variable. */
index = 0;
for(k = 0; k < NZ; ++k)
    for(j = 0; j < NY; ++j)
        for(i = 0; i < NX; ++i, ++index)
            nodal[k][j][i] = index;

/* Pass the data to visit_writer to write a binary VTK file. */
write_curvilinear_mesh("vwcurv3d.vtk", 1, dims, pts, nvars,
                      vardims, centering, varnames, vars);

return 0;
}

```

Listing 2-73: vwcurv3d.py: Python language example for writing a curvilinear mesh with data.

```

import visit_writer

NX = 4
NY = 3
NZ = 2

# Curvilinear mesh points stored x0,y0,z0,x1,y1,z1,...
pts = (0, 0.5, 0, 1, 0, 0, 2, 0, 0,
       3, 0.5, 0, 0, 1, 0, 1, 1, 0,

```

```
    2, 1, 0, 3, 1, 0, 0, 1.5, 0,
    1, 2, 0, 2, 2, 0, 3, 1.5, 0,
    0, 0.5, 1, 1, 0, 1, 2, 0, 1,
    3, 0.5, 1, 0, 1, 1, 1, 1, 1,
    2, 1, 1, 3, 1, 1, 0, 1.5, 1,
    1, 2, 1, 2, 2, 1, 3, 1.5, 1)

# Create a zonal variable
zonal = []
index = 0
for k in range(NZ-1):
    for j in range(NY-1):
        for i in range(NX-1):
            zonal = zonal + [index]
            index = index + 1

# Create a nodal variable
nodal = []
index = 0
for k in range(NZ):
    for j in range(NY):
        for i in range(NX):
            nodal = nodal + [index]
            index = index + 1

# Pass data to visit_writer to write a binary VTK file.
dims = (NX, NY, NZ)
vars = (("zonal", 1, 0, zonal), ("nodal", 1, 1, nodal))
visit_writer.WriteCurvilinearMesh("vwcurv3d.vtk", 0, dims, pts, vars)
```

5.5 Point meshes with data

A point mesh is a set of 2D or 3D points where the nodes also constitute the cells in the mesh. The `visit_writer` library provides the `write_point_mesh` function to write out point meshes and data to VTK files.

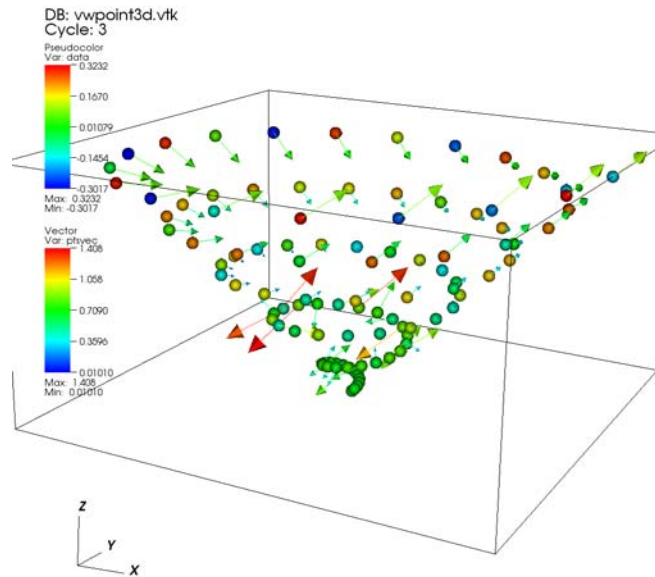


Figure 2-74: Point mesh with scalar data and vector data

Listing 2-75: `vwpoint3d.c`: C-Language example for writing a point mesh with data.

```
#include <visit_writer.h>

#define NPTS 100

int main(int argc, char *argv[])
{
    /* Create some points and data to save. */
    int i;
    float pts[NPTS][3], data[NPTS];
    int nvars = 2;
    int vardims[] = {1, 3};
    const char *varnames[] = {"data", "ptsvec"};
    float *vars[] = {(float *)pts, data};

    for(i = 0; i < NPTS; ++i)
    {
        /* Make a point. */
        float t = ((float)i) / ((float)(NPTS-1));
        float angle = 3.14159 * 10. * t;
```

```

        pts[i][0] = t * cos(angle);
        pts[i][1] = t * sin(angle);
        pts[i][2] = t;
        /* Make a scalar */
        data[i] = t * cos(angle);
    }

    /* Pass the mesh and data to visit_writer. */
    write_point_mesh("vwpoint3d.vtk", 1, NPTS, (float*)pts, nvars,
        vardims, varnames, vars);

    return 0;
}

```

Listing 2-76: vwpoint3d.py: Python language example for writing a point mesh with data.

```

import visit_writer, math
NPTS = 100
pts = []
data = []
for i in range(NPTS):
    # Make a point
    t = float(i) / float(NPTS-1)
    angle = 3.14159 * 10. * t
    pts = pts + [t * math.cos(angle), t * math.sin(angle), t]
    # Make a scalar
    data = data + [t * math.cos(angle)]

# Pass the mesh and data to visit_writer.
vars = (("data", 1, 1, pts), ("ptsvec", 3, 1, pts))
visit_writer.WritePointMesh("vwpoint3d.vtk", 1, pts, vars)

```

5.6 Unstructured meshes with data

Unstructured meshes are collections of different types of zones and are useful because they can represent more complex mesh geometries than the structured meshes can. Unstructured meshes are specified using the cell types and node orderings listed in “Writing an unstructured mesh” on page 28. This section explains how to use the `visit_writer` library’s `write_unstructured_mesh` function to write out unstructured meshes and data.

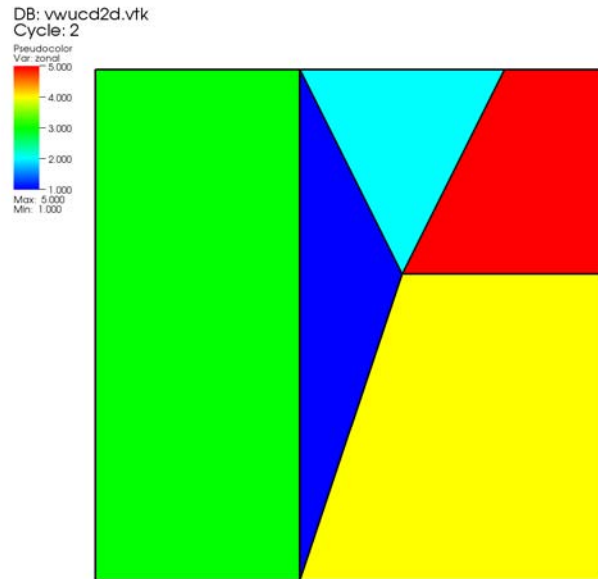


Figure 2-77: 2D unstructured mesh with zonal variable

Listing 2-78: vwrucd2d.c: C-Language example for writing an unstructured mesh with data.

```
#include <visit_writer.h>

int main(int argc, char *argv[])
{
    /* Node coordinates */
    int nnodes = 9;
    int nzones = 5;
    float pts[] = {0., 0., 0., 2., 0., 0., 5., 0., 0.,
                  3., 3., 0., 5., 3., 0., 0., 5., 0.,
                  2., 5., 0., 4., 5., 0., 5., 5., 0.};

    /* Zone types */
    int zonetypes[] = {VISIT_TRIANGLE, VISIT_TRIANGLE,
                      VISIT_QUAD, VISIT_QUAD, VISIT_QUAD};

    /* Connectivity */
    int connectivity[] = {
        1,3,6,    /* tri zone 1. */
        3,7,6,    /* tri zone 2. */
        0.,1,6,5, /* quad zone 3. */
        1,2,4,3,  /* quad zone 4. */
        3,4,8,7   /* quad zone 5. */
    };
};
```

```

/* Data arrays */
float nodal[] = {1,2,3,4,5,6,7,8,9};
float zonal[] = {1,2,3,4,5};

/* Info about the variables we're passing to visit_writer. */
int nvars = 2;
int vardims[] = {1, 1};
int centering[] = {0, 1};
const char *varnames[] = {"zonal", "nodal"};
float *vars[] = {zonal, nodal};

/* Pass the mesh and data to visit_writer. */
write_unstructured_mesh("vwucd2d.vtk", 1, nnodes, pts, nzones,
    zonetypes, connectivity, nvars, vardims, centering,
    varnames, vars);

return 0;
}

```

Listing 2-79: vwucd2d.py: Python language example for writing an unstructured mesh with data.

```

import visit_writer

# Node coordinates
pts = (0., 0., 0., 2., 0., 0., 5., 0., 0.,
       3., 3., 0., 5., 3., 0., 0., 5., 0.,
       2., 5., 0., 4., 5., 0., 5., 5., 0.)

# Connectivity
connectivity = (
    (visit_writer.triangle, 1,3,6),
    (visit_writer.triangle, 3,7,6),
    (visit_writer.quad, 0,1,6,5),
    (visit_writer.quad, 1,2,4,3),
    (visit_writer.quad, 3,4,8,7)
)

# Data arrays
nodal = (1,2,3,4,5,6,7,8,9)
zonal = (1,2,3,4,5)

# Pass the data to visit_writer
vars = (("zonal", 1, 0, zonal), ("nodal", 1, 1, nodal))
visit_writer.WriteUnstructuredMesh("vwucd2d.vtk", 1, pts,
    connectivity, vars)

```

5.7 Creating a master file for parallel (.visit file)

The `visit_writer` library creates legacy VTK files and the legacy VTK file format has no mechanism for storing more than a single mesh. Furthermore, legacy VTK files have

no concept of a master file or of multi-objects like Silo uses to unite domains into a whole. Fortunately, VisIt provides a construct called a `.visit` file that addresses this shortcoming. A `.visit` file is a text file, ending with the `“.visit”` extension, that contains the names of domain files that make up the whole. A `.visit` file can be created to group files for any file format that VisIt can read. Your parallel program can still write individual VTK files and you can create a `.visit` file before visualizing the files so VisIt knows to open all of the relevant files as opposed to you creating plots of each individual file. The following code example lists what a `.visit` file looks like if you have 4 VTK domain files that contain the same variables and all of them are to be plotted at once.

```
!NBLOCKS 4
proc-0.vtk
proc-1.vtk
proc-2.vtk
proc-3.vtk
```

The `.visit` file can be used for indicating which VTK files are part of a time-varying database in addition to indicating how to reassemble domain files into a whole. In the previous example, there were 4 domain files and only 1 time step. If you want to have more than 1 time step, just add more files to the list. The `!NBLOCKS` directive tells VisIt that every block of 4 files are related in a single time step. If you had two time steps then your `.visit` file might look like this:

```
!NBLOCKS 4
proc-0.0000.vtk
proc-1.0000.vtk
proc-2.0000.vtk
proc-3.0000.vtk
proc-0.0001.vtk
proc-1.0001.vtk
proc-2.0001.vtk
proc-3.0001.vtk
```

Chapter 3

Creating compatible files II

Advanced topics

1.0 Overview

This chapter elaborates on some of the advanced topics involved in creating files that VisIt can read. Most applications should be able to write out all of their data using information contained in the previous chapter. This chapter introduces advanced topics such as incorporating metadata to accelerate VisIt's performance as well as some less common data representations. Many of the examples in this chapter use the Silo library, which was introduced in the previous chapter. For more information on getting started with the Silo library, see "Writing Silo files" on page 12.

2.0 Writing vector data

The components of vector data are often stored to files as individual scalar variables and VisIt uses an expression to compose the scalars back into a vector field. If you use the Silo library, you can always choose instead to store your vector data as a multi-component variable. The previous chapter provided several examples that use the Silo library to write scalar variables on rectilinear, curvilinear, point, and unstructured meshes. The functions that were used to write the scalars were simplified forms of the functions that are used to write vector data. The scalar functions that were used to write data for a specific mesh type as well as the vector function equivalents are listed in the following table:

Mesh type	Scalar function	Vector function
Rectilinear mesh	DBPutQuadvar1	DBPutQuadvar
Curvilinear mesh	DBPutQuadvar1	DBPutQuadvar
Point mesh	DBPutPointvar1	DBPutPointvar

Mesh type	Scalar function	Vector function
Unstructured mesh	DBPutUcdvar1	DBPutUcdvar

The differences between a scalar function and a vector function are small. In fact, the argument lists for a scalar function and a vector function are nearly identical in the Silo library's C-Language interface. The chief difference is that the vector functions take two additional arguments and the meaning of one existing argument is modified. The first new argument is an integer indicating the number of components contained by the variable to be written. The next difference is that you must pass an array of pointers to character strings that represent the names of each individual component. Finally, the argument that was used to pass the data to the DBPutQuadvar1 function, now in the DBPutQuadvar function, accepts an array pointers to the various arrays that contain the variable components. For more complete information on each of the arguments to the functions that Silo uses to write multi-component data, refer to the *Silo User's Manual*.

Listing 3-1: vectorvar.c: C-Language example for writing vector data using Silo.

```
int i, dims[3], ndims = 3;
int nnodes = NX*NY*NZ;
float *comp[3];
char *varnames[] = {"nodal_comp0", "nodal_comp1", "nodal_comp2"};
comp[0] = (float *)malloc(sizeof(float)*nnodes);
comp[1] = (float *)malloc(sizeof(float)*nnodes);
comp[2] = (float *)malloc(sizeof(float)*nnodes);
for(i = 0; i < nnodes; ++i)
{
    comp[0][i] = (float)i; /*vector component 0*/
    comp[1][i] = (float)i; /*vector component 1*/
    comp[2][i] = (float)i; /*vector component 2*/
}
dims[0] = NX; dims[1] = NY; dims[2] = NZ;
DBPutQuadvar(dbfile, "nodal", "quadmesh",
    3, varnames, comp, dims,
    ndims, NULL, 0, DB_FLOAT, DB_NODECENT, NULL);
free(comp[0]);
free(comp[1]);
free(comp[2]);
```

Silo's Fortran interface does not provide functions to write out multi-component data such as vectors. If you use the Fortran interface to Silo, you will have to write out the vector components as separate scalar variables and then write an expression to your Silo file that composes the components into a single vector variable.

Listing 3-2: fvectorvar.f: Fortran-Language example for writing vector data using Silo.

```
subroutine write_nodecent_quadvar(dbfile)
```

```

implicit none
integer dbfile
include "silo.inc"
integer err, ierr, dims(3), ndims,i,j,k,index,NX,NY,NZ
parameter (NX = 4)
parameter (NY = 3)
parameter (NZ = 2)
real    comp0(NX,NY,NZ), comp1(NX,NY,NZ), comp2(NX,NY,NZ)
data dims/NX,NY,NZ/
index = 0
do 20020 k=1,NZ
do 20010 j=1,NY
do 20000 i=1,NX
    comp0(i,j,k) = float(index)
    comp1(i,j,k) = float(index)
    comp2(i,j,k) = float(index)
    index = index + 1
20000 continue
20010 continue
20020 continue
    ndims = 3
    err = dbputqv1(dbfile, "n_comp0", 11, "quadmesh", 8, comp0,
. dims, ndims, DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL,
. ierr)
    err = dbputqv1(dbfile, "n_comp1", 11, "quadmesh", 8, comp1,
. dims, ndims, DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL,
. ierr)
    err = dbputqv1(dbfile, "n_comp2", 11, "quadmesh", 8, comp2,
. dims, ndims, DB_F77NULL, 0, DB_FLOAT, DB_NODECENT, DB_F77NULL,
. ierr)
end

subroutine write_defvars(dbfile)
implicit none
integer dbfile
include "silo.inc"
integer    err, ierr, types(2), lnames(2), ldefs(2), oldlen
c Initialize some 20 character length strings
character*40 names(2) /'zonalvec           ',
.
.          'nodalvec                       '/
character*40 defs(2)  /'{z_comp0,z_comp1,z_comp2} ',
.
.          '{n_comp0,n_comp1,n_comp2}  '/
c Store the length of each string
data lnames/8, 8/
data ldefs/37, 37/
data types/DB_VARTYPE_VECTOR, DB_VARTYPE_VECTOR/
c Set the maximum string length to 40 since that's how long our
c strings are
oldlen = dbget2dstrlen()
err = dbset2dstrlen(40)
c Write out the expressions
err = dbputdefvars(dbfile, "defvars", 7, 2, names, lnames,
. types, defs, ldefs, DB_F77NULL, ierr)
c Restore the previous value for maximum string length

```

```
err = dbset2dstrlen(oldlen)
end
```

3.0 Adding metadata for performance boosts

VisIt incorporates several performance boosting strategies that make use of metadata, if it is available. Most of the metadata applies to increasing parallel performance by reducing the amount of I/O and subsequent processing that is required. The I/O reductions are realized by not reading in and processing domains that will contribute nothing to the final image on the screen. In order to prevent domains from being read in, your multi-objects must have associated metadata for each of the domains that they contain. When a Silo multi-object contains metadata about all of its constituent domains, VisIt can make work-saving decisions since it knows the properties of each domain without having to read in the data for each domain.

This section explains how to add metadata to your Silo multi-objects using option lists. Metadata attached to multi-objects allow VisIt to determine important data characteristics such as data extents or the spatial extents of the mesh without having to first read and process all domains. Such knowledge allows VisIt to restrict the number of domains that are processed, thus reducing the amount of work and the time required to display images on your screen.

3.1 Writing data extents

Providing data extents can help VisIt only read in and process those domains that will contribute to the final image. Many types of plots and operators use data extents for each domain, when they are provided, to perform a simple upfront test to determine if a domain contains the values which will be used. If a domain is not needed then VisIt will not read that domain because it is known beforehand that the domain does not contain the desired value.

An example of a plot that uses data extents in order to save work is VisIt's Contour plot. The Contour plot creates contours (lines or surfaces where the data has the same value) through a dataset. Consider the example shown in Figure 3-3, where the entire mesh and scalar field are divided into four smaller domains where the data extents of each domain are stored to the file so VisIt can perform optimizations. Before the Contour plot executes, it tells VisIt the data values for which it will make contours. Suppose that that you wanted to see the areas where the value in the scalar field are equal to 11.5. The Contour plot takes that 11.5 contour value and compares it to the data extents for all of the domains to see which domains will be needed. If a domain will not be needed then VisIt will make no further effort to read the domain or process it, thus saving work and making the plot appear on the screen faster than it could if the data extents were not available in the file

metadata. In the above example, the value of 11.5 is only present in domain 3, which means that the Contour plot will only return a result if it processes data from domain 3.

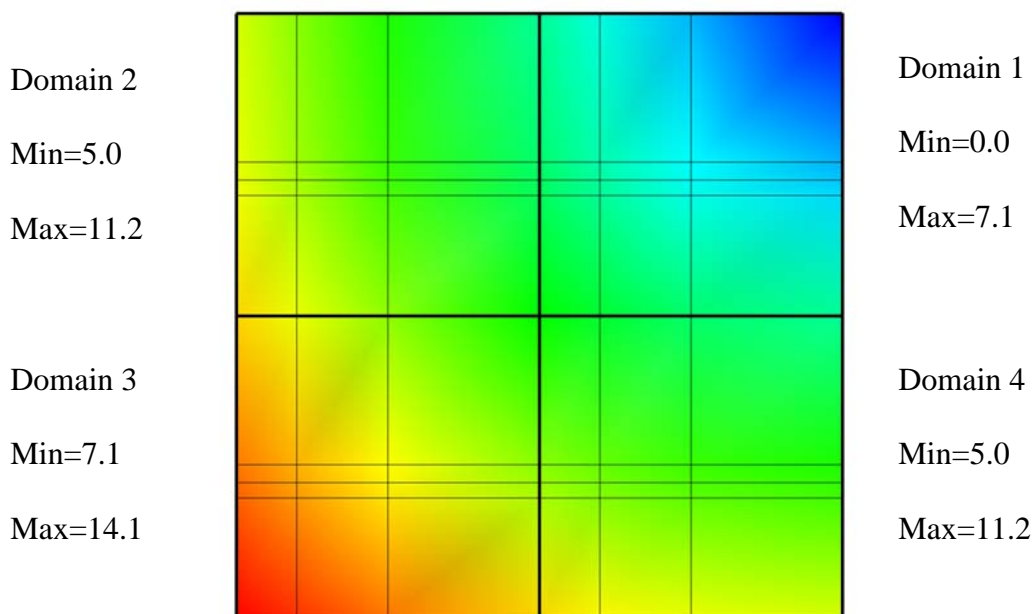


Figure 3-3: Example Mesh and Pseudocolor plots with the data extents for each domain of the Pseudocolor plot's scalar variable.

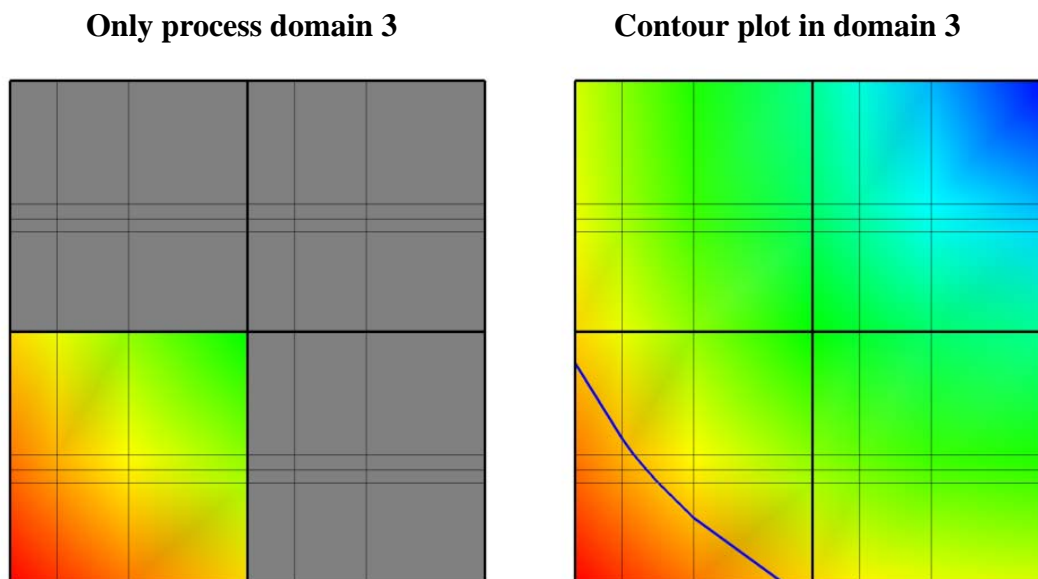


Figure 3-4: Only process domain 3 (left) to yield the Contour plot of value 11.5 (right).

The other domains are not processed in this case because they do not contain the required value of 11.5. After the comparisons have been made, VisIt knows which domains will

have to be processed and it can divide the set of domains (just domain 3 in this case) that will contribute to the visualization among processors so they can execute the plot and return data to VisIt's viewer where it can be displayed.

To add the data extents for each processor to the metadata using Silo, you must add the data extents to the option list that you pass to the `DBPutMultivar` function call. Having the data extents for each domain readily available in the Multivar object ensures that VisIt will have enough information to determine which domains will be necessary for operations such as Contour without having to read all of the data to determine which domains contribute to the visualization. The data extents must be stored in a double precision array that has enough entries to accommodate the min and max values for each domain in the multivar object. The layout of the min and max values within that array are as follows: `min_dom1, max_dom1, min_dom2, max_dom2, . . . , min_domN, max_domN`

Listing 3-5: `dataextents.c`: C-Language example for writing data extents using Silo.

```
const int two = 2;
double extents[NDOMAINS][2];
DBoptlist *optlist = NULL;
/* Calculate the per-domain data extents for this variable. */
/* Write the multivar. */
optlist = DBMakeOptlist(2);
DBAddOption(optlist, DBOPT_EXTENTS_SIZE, (void *)&two);
DBAddOption(optlist, DBOPT_EXTENTS, (void *)extents);
DBPutMultivar(dbfile, "var", nvar, varnames, vartypes, optlist);
DBFreeOptlist(optlist);
```

Listing 3-6: `fdataextents.f`: Fortran language example for writing data extents using Silo.

```
double precision extents(2,NDOMAINS)
integer err, optlist
c Calculate the per-domain data extents for this variable.
c Write the multivar.
err = dbmkoptlist(2, optlist)
err = dbaddiopt(optlist, DBOPT_EXTENTS_SIZE, 2)
err = dbaddopt(optlist, DBOPT_EXTENTS, extents)
err = dbputmvar(dbfile, "var", 3, nvar, varnames, lvarnames,
. vartypes, optlist, ierr)
err = dbfreeoptlist(optlist)
```


3.2 Writing spatial extents

If you provide spatial extents for each domain in your database then VisIt can use that information during spatial data reduction operations, such as slicing, to reduce the number of domains that must be read from disk and processed.

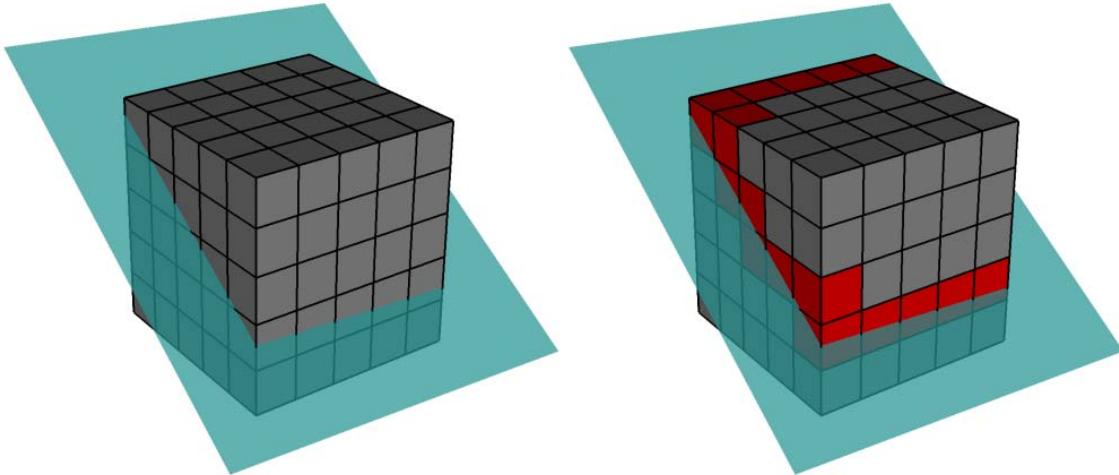


Figure 3-7: Only the red domains need to be processed to compute the slice plane if spatial extents are provided.

Spatial extents for a domain contain the minimum and maximum values of the coordinates within that domain, also called the domain's bounding box. The spatial extents must be stored in a double precision array that has enough entries to accommodate the min and max coordinate values for each domain in the multimesh object. The layout of the min and max values within that array for 3D domains are as follows: `xmin_dom1`, `ymin_dom1`, `zmin_dom1`, `xmax_dom1`, `ymax_dom1`, `zmax_dom1`, ..., `xmin_domN`, `ymin_domN`, `zmin_domN`, `xmax_domN`, `ymax_domN`, `zmax_domN`. In the event that you have 2D domains then you can omit the z-components of the min and max values and tell Silo that there are 4 values per min/max tuple instead of the 6 values required to specify min and max values for 3D domains.

Listing 3-8: `spatialextents.c`: C-Language example for writing 3D spatial extents using Silo.

```
const int six = 6;
double spatial_extents[NDOMAINS][6];
DBoptlist *optlist = NULL;
/* Calculate the per-domain spatial extents for this mesh. */
for(int i = 0; i < NDOMAINS; ++i)
{
    spatial_extents[i][0] = xmin; /* xmin for i'th domain */
    spatial_extents[i][1] = ymin; /* ymin for i'th domain */
    spatial_extents[i][2] = zmin; /* zmin for i'th domain */
}
```

```

        spatial_extents[i][3] = xmin; /* xmax for i'th domain */
        spatial_extents[i][4] = ymax; /* ymax for i'th domain */
        spatial_extents[i][5] = zmax; /* zmax for i'th domain */
    }
    /* Write the multimesh. */
    optlist = DBMakeOptlist(2);
    DBAddOption(optlist, DBOPT_EXTENTS_SIZE, (void *)&six);
    DBAddOption(optlist, DBOPT_EXTENTS, (void *)spatial_extents);
    DBPutMultimesh(dbfile, "mesh", nmesh, meshnames, meshtypes, optlist);
    DBFreeOptlist(optlist);

```

Listing 3-9: fspatial extents.f: Fortran language example for writing 3D spatial extents using Silo.

```

        double precision spatial_extents(6,NDOMAINS)
        integer optlist, err, dom
    c Calculate the per-domain spatial extents for this mesh.
        do 10000 dom=1,NDOMAINS
            spatial_extents(1,dom) = xmin
            spatial_extents(2,dom) = ymin
            spatial_extents(3,dom) = zmin
            spatial_extents(4,dom) = xmin
            spatial_extents(5,dom) = ymax
            spatial_extents(6,dom) = zmax
        10000 continue
    c Write the multimesh
        err = dbmkoptlist(2, optlist)
        err = dbaddiopt(optlist, DBOPT_EXTENTS_SIZE, 6)
        err = dbadddopt(optlist, DBOPT_EXTENTS, spatial_extents)
        err = dbputmmesh(dbfile, "quadmesh", 8, nmesh, meshnames,
        . lmeshnames, meshtypes, optlist, ierr)
        err = dbfreeoptlist(optlist)

```

4.0 Ghost zones

Ghost zones are zones external to a domain, which correspond to zones in an adjacent domain. Ghost zones allow VisIt to ensure continuity between domains containing zone-centered data, making surfaces such as Contour plots continuous across domain boundaries instead of creating surfaces with ugly gaps at the domain boundaries. Ghost zones also allow VisIt to remove internal surfaces from the visualized data for plots such as Pseudocolor, which only wants to keep the surfaces that are external to the model. Removing internal surfaces results in fewer primitives that must be rendered on the graphics card and that increases interactivity with plots. See Figure 3-10 for examples of the problems that ghost zones allow VisIt to fix.

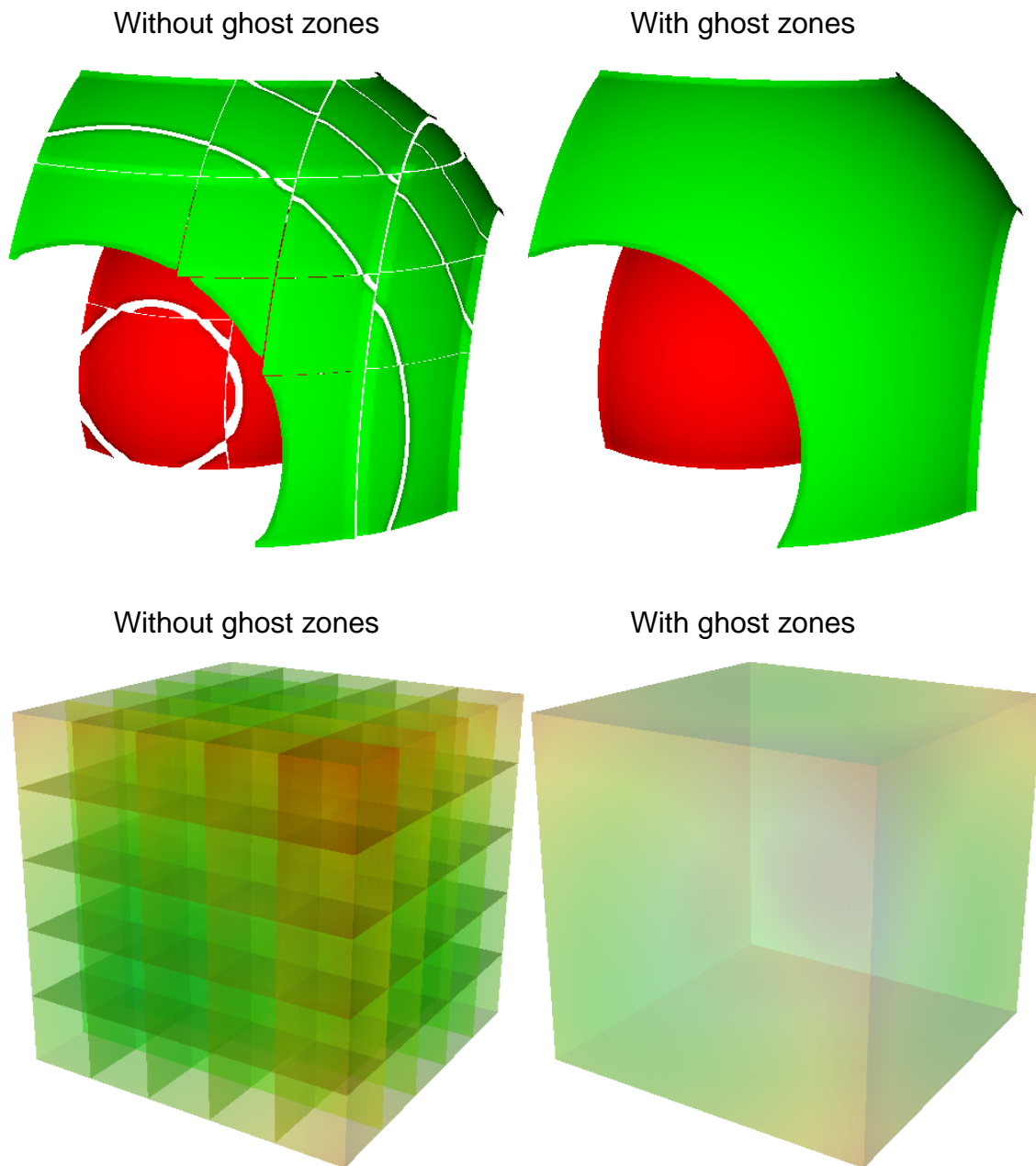


Figure 3-10: VisIt can use ghost zones to ensure continuity and to remove internal surfaces

Ghost zones can be stored into the database so VisIt can read them when the data is visualized. Ghost zones can also be created on-the-fly for structured (rectilinear and curvilinear) meshes if multimesh adjacency information is provided. This section will show how to write ghost zones to the file. If you are interested in providing multimesh adjacency information so you can write smaller files and so VisIt can automatically create ghost zones then refer to the documentation for the `DBPutMultimeshadj` function in the *Silo User's Guide*.

4.1 Writing ghost zones to your files

You can write ghost zones to your files using the Silo library or you can instead write a multimesh adjacency object, covered in the *Silo User's Guide*, that VisIt can use to automatically create ghost zones. This section will cover how to use the Silo library to store ghost zones explicitly in your files.

The first step in creating ghost zones is to add a layer of zones around the mesh in each domain of your database where a domain boundary exists. Each zone in the layer of added ghost zones must match the location and have the same data value as the zone in the domain that it is meant to mirror in order for VisIt to be able to successfully use ghost zones to remove domain decomposition artifacts. This means that you must change your code for writing out meshes and variables so your meshes have an addition layer of zones for each domain boundary that is internal to the model. Your variables must also contain valid data values in the ghost zones since providing a domain with knowledge of the data values of its neighboring domains is the entire point of adding ghost zones. Note that you should not add ghost zones on the surface of a domain where the surface is external to the model. When ghost zones are erroneously added to external surfaces of the model, VisIt removes the external faces and this can cause plots to be invisible.

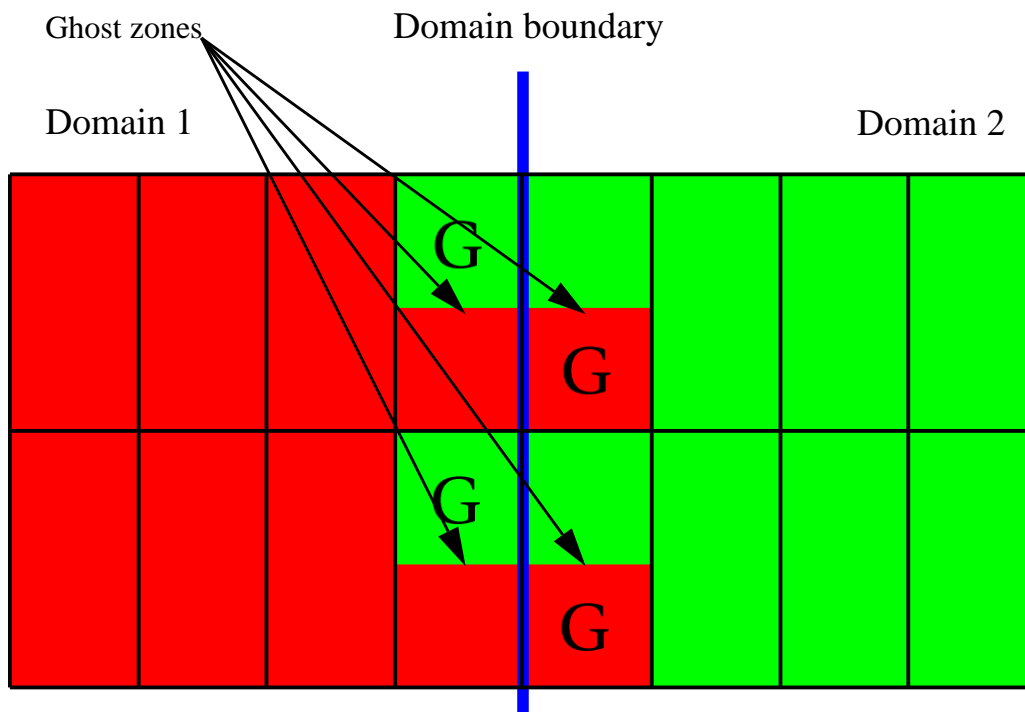


Figure 3-11: The zones that are both red and green are real zones in one domain and ghost zones in another.

Figure 3-11 shows two domains: domain1 (red) and domain2 (green). The boundary between (blue) the two domains is the interface that would exist between the domains if there were no ghost zones. When you add a layer of ghost zones, each domain intrudes a

little into the other domain's bounding box so the zones in one domain's layer of ghost zones match the zones in the other domain's external layer of zones. Of course, domains on both sides of the domain boundary have ghost zones to assure that the VisIt will know the proper zone-centered data values whether it approaches the domain boundary from the left or from the right. The first row of cells on either side of the domain boundary are ghost zones. For example, if you look at the upper left zone containing the "G" for ghost zone, the "G" is drawn in the green part of the zone, while the red part of the zone contains no "G". This means that the zone in question is a zone in domain1, the red domain, but that domain2 has a zone that exactly matches the location and values of the zone in the red domain. The corresponding zone in domain2 is a ghost zone.

Listing 3-12: spatialextents.c: C-Language example for writing a 3D, domain-decomposed rectilinear mesh without ghost zones.

```

/* Create each of the domain meshes. */
int dom = 0, xdom, ydom, zdom;
for(zdom = 0; zdom < NZDOMS; ++zdom)
for(ydom = 0; ydom < NYDOMS; ++ydom)
for(xdom = 0; xdom < NXDOMS; ++xdom, ++dom)
{
    float xc[NX], yc[NY], zc[NZ];
    float *coords[] = {xc, yc, zc};
    int index = 0;
    float xstart, xend, ystart, yend, zstart, zend;
    int xzones, yzones, zzones, nzones;
    int xnodes, ynodes, znodes;

    /* Create a new directory. */
    char dirname[100];
    sprintf(dirname, "Domain%03d", dom);
    DBMkDir(dbfile, dirname);
    DBSetDir(dbfile, dirname);

    /* Determine default start, end coordinates */
    xstart = (float)xdom * XSIZE;
    xend   = (float)(xdom+1) * XSIZE;
    xzones = NX-1;
    ystart = (float)ydom * YSIZE;
    yend   = (float)(ydom+1) * YSIZE;
    yzones = NY-1;
    zstart = (float)zdom * ZSIZE;
    zend   = (float)(zdom+1) * ZSIZE;
    zzones = NZ-1;

    xnodes = xzones + 1;
    ynodes = yzones + 1;
    znodes = zzones + 1;

    /* Create the mesh coordinates. */
    for(i = 0; i < xnodes; ++i)
    {

```

```

        float t = (float)i / (float)(xnodes-1);
        xc[i] = (1.-t)*xstart + t*xend;
    }
    for(i = 0; i < ynodes; ++i)
    {
        float t = (float)i / (float)(ynodes-1);
        yc[i] = (1.-t)*ystart + t*yend;
    }
    for(i = 0; i < znodes; ++i)
    {
        float t = (float)i / (float)(znodes-1);
        zc[i] = (1.-t)*zstart + t*zend;
    }
    /* Write a rectilinear mesh. */
    dims[0] = xnodes;
    dims[1] = ynodes;
    dims[2] = znodes;
    DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
        DB_FLOAT, DB_COLLINEAR, NULL);

    /* Go back to the top directory. */
    DBSetDir(dbfile, "..");
}

```

Once you have changed your mesh-writing code to add a layer of ghost zones, where appropriate, you must indicate that the extra layer of zones are ghost zones. If you use Silo's DBPutQuadmesh function to write your mesh, you can indicate which zones are ghost zones by adding DBOPT_LO_OFFSET and DBOPT_HI_OFFSET to pass arrays containing high and low zone index offsets in the option list. If you are adding ghost zones to an unstructured mesh, you would instead adjust the `lo_offset` and `hi_offset` arguments that you pass to the DBPutZonelist2 function. The next code listing shows the additions made in order to support ghost zones in a domain-decomposed rectilinear mesh. The additions are underlined.

Listing 3-13: ghostzonesinfile.c: C-Language example for writing a 3D, domain-decomposed rectilinear mesh with ghost zones.

```

/* Determine the size of a zone. */
float cx, cy, cz;
cx = XSIZE / (float)(NX-1);
cy = YSIZE / (float)(NY-1);
cz = ZSIZE / (float)(NZ-1);
    /* Create each of the domain meshes. */
    int dom = 0, xdom, ydom, zdom;
    for(zdom = 0; zdom < NZDOMS; ++zdom)
    for(ydom = 0; ydom < NYDOMS; ++ydom)
    for(xdom = 0; xdom < NXDOMS; ++xdom, ++dom)
    {
        float xc[NX], yc[NY], zc[NZ];
        float *coords[] = {xc, yc, zc};
    }

```

```

int index = 0;
float xstart, xend, ystart, yend, zstart, zend;
int xzones, yzones, zzones, nzones;
int xnodes, ynodes, znodes;
int hi offset[3], lo offset[3];
DBOptlist *optlist = NULL;

/* Create a new directory. */
char dirname[100];
sprintf(dirname, "Domain%03d", dom);
DBMkDir(dbfile, dirname);
DBSetDir(dbfile, dirname);

/* Determine default start, end coordinates */
xstart = (float)xdom * XSIZE;
xend   = (float)(xdom+1) * XSIZE;
xzones = NX-1;
ystart = (float)ydom * YSIZE;
yend   = (float)(ydom+1) * YSIZE;
yzones = NY-1;
zstart = (float)zdom * ZSIZE;
zend   = (float)(zdom+1) * ZSIZE;
zzones = NZ-1;

/* Set the starting hi/lo offsets. */
lo offset[0] = 0;
lo offset[1] = 0;
lo offset[2] = 0;
hi offset[0] = 0;
hi offset[1] = 0;
hi offset[2] = 0;

/* Adjust the start and end coordinates based on whether
 * or not we have ghost zones.
 */
if(xdom > 0)
{
    xstart -= cx;
    lo offset[0] = 1;
    ++xzones;
}
if(xdom < NXDOMS-1)
{
    xend += cx;
    hi offset[0] = 1;
    ++xzones;
}
if(ydom > 0)
{
    ystart -= cy;
    lo offset[1] = 1;
    ++yzones;
}
if(ydom < NYDOMS-1)

```

```

    {
        yend += cy;
        hi offset[1] = 1;
        ++yzones;
    }
    if(zdom > 0)
    {
        zstart -= cz;
        lo offset[2] = 1;
        ++zzones;
    }
    if(zdom < NZDOMS-1)
    {
        zend += cz;
        hi offset[2] = 1;
        ++zzones;
    }

    xnodes = xzones + 1;
    ynodes = yzones + 1;
    znodes = zzones + 1;

    /* Create the mesh coordinates. */
    for(i = 0; i < xnodes; ++i)
    {
        float t = (float)i / (float)(xnodes-1);
        xc[i] = (1.-t)*xstart + t*xend;
    }
    for(i = 0; i < ynodes; ++i)
    {
        float t = (float)i / (float)(ynodes-1);
        yc[i] = (1.-t)*ystart + t*yend;
    }
    for(i = 0; i < znodes; ++i)
    {
        float t = (float)i / (float)(znodes-1);
        zc[i] = (1.-t)*zstart + t*zend;
    }
    /* Write a rectilinear mesh. */
    dims[0] = xnodes;
    dims[1] = ynodes;
    dims[2] = znodes;
    optlist = DBMakeOptlist(2);
    DBAddOption(optlist, DBOPT HI OFFSET, (void *)hi offset);
    DBAddOption(optlist, DBOPT LO OFFSET, (void *)lo offset);
    DBPutQuadmesh(dbfile, "quadmesh", NULL, coords, dims, ndims,
        DB FLOAT, DB COLLINEAR, optlist);
    DBFreeOptlist(optlist);

    /* Go back to the top directory. */
    DBSetDir(dbfile, "..");
}

```


There are two changes to the code in the previous listing that allow it to write ghost zones. First of all, the code calculates the size of a zone in the `cx`, `cy`, `cz` variables and then uses those sizes along with the location of the domain within the model to determine which domain surfaces will receive a layer of ghost zones. The layer of ghost zones is added by altering the start and end locations of the coordinate arrays as well as incrementing the number of zones and nodes in the dimensions that will have added ghost zones. The knowledge of which surfaces get a layer of ghost zones is recorded in the `lo_offset` and `hi_offset` arrays. By setting `lo_offset[0]` to 1, Silo knows that the first layer of zones in the X dimension will all be ghost zones. Similarly, by setting `high_offset[0]` to 1, Silo knows that the last layer of zones in the X dimension are ghost zones. The `lo_offset` and `hi_offset` arrays are associated with the mesh by adding them to the option list that is passed to the `DBPutQuadmsh` function. The example program `fghostzonesinfile.f` demonstrates how to add ghost zones to a file using Silo's Fortran interface.

5.0 Materials

Many simulations use materials to define the composition of regions so the response of the materials can be taken into account during the calculation. Materials are represented as a list of integers with associated material names such as: "steel". Each zone in the mesh gets one or more material numbers to indicate its composition. When a zone has a single material number, it is said to be a "clean zone". When there is more than one material number in a zone, it is said to be a "mixed zone". When zones are mixed, they have a list of material numbers and a list of volume fractions (floating point numbers that sum to one) that indicate how much of each material is contained in a zone. VisIt provides the `FilledBoundary` and `Boundary` plots for plotting materials and VisIt provides the **Subset** window so you can selectively turn off certain materials.

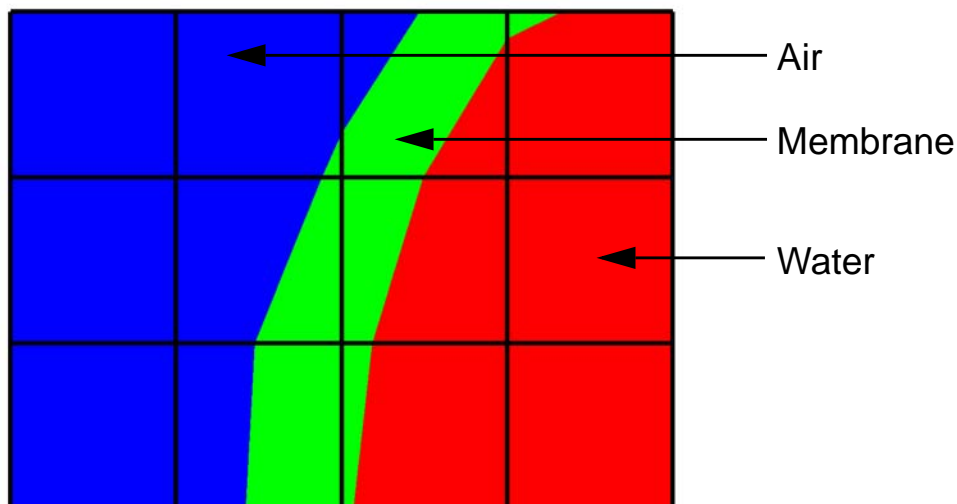


Figure 3-14: A mesh with both clean and mixed material zones

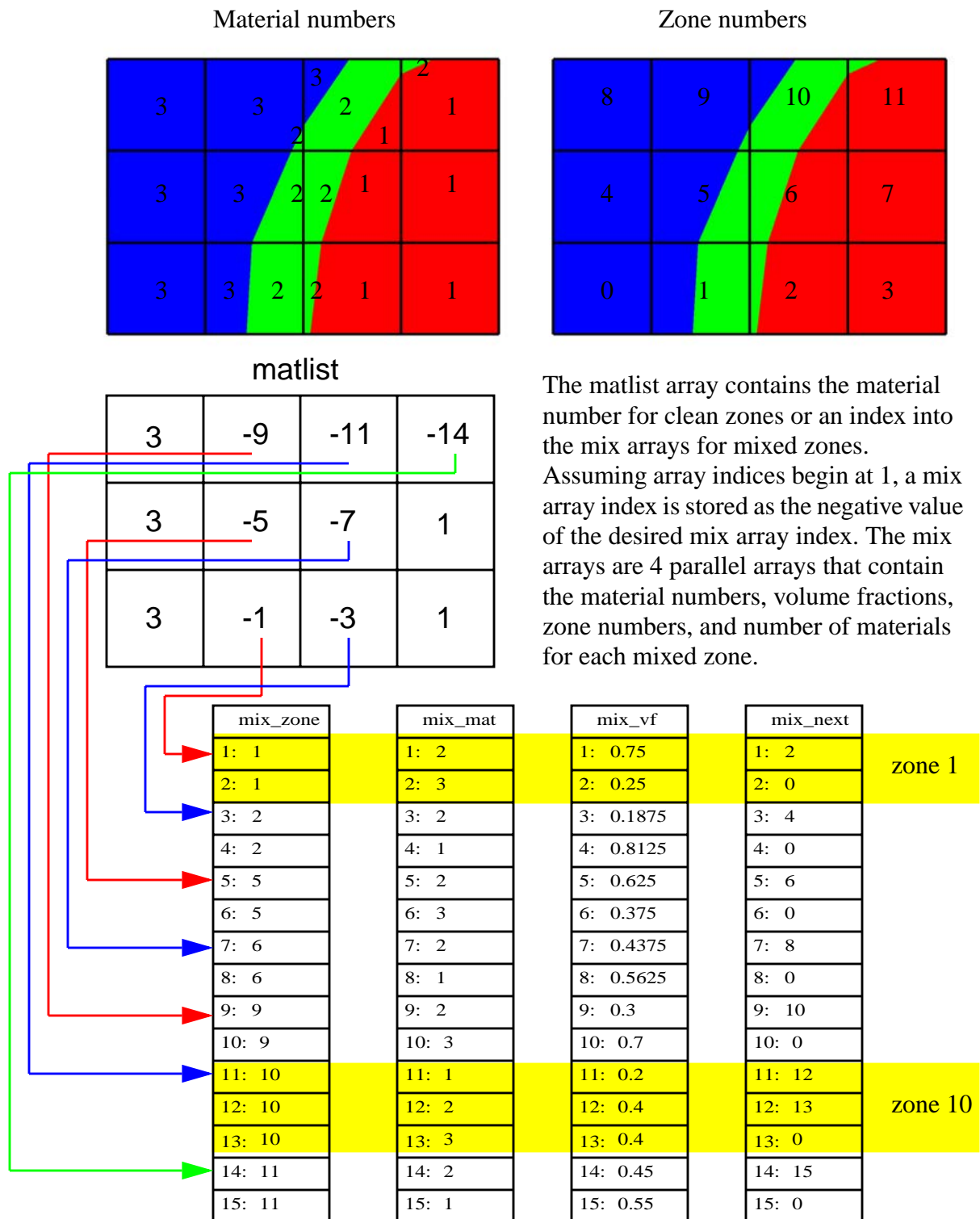


Figure 3-15: Mixed material example

The plot of the material object shown in Figure 3-14 and Figure 3-15 contains three materials: “Water” (1), “Membrane” (2), and “Air” (3). Materials use a `matlist` array to indicate which zone are clean and which are mixed. The `matlist` array is a zone-centered array of integers that contain the material numbers for the materials in the zone. If a zone has only one material then the `matlist` array entry for that zone will contain the material number of the material that fills the zone. If a zone contains more than one material then the `matlist` array entry for that zone will contain an index into the mixed material arrays. Indices into the mixed material arrays are equal to the negative value of the desired mixed material array entry. When creating your mixed material arrays, assume that array indices for the mixed material arrays begin at 1. When you begin assigning material information into the mixed material arrays, use one array index per material in the mixed material zone. The index that you use for the beginning index for the next mixed material zone is the current index minus the number of materials in the current zone. Study the `matlist` array in Figure 3-15. The first mixed material zone is zone 1 and since it is mixed, instead of containing a material number, the `matlist` array for zone 1 contains the starting index into the mixed material arrays, or -1. If you negate the -1, you arrive at index 1, which is the starting index for zone 1 in the mixed material arrays. Since zone 1 will contain two materials, we use indices 1 and 2 in the mixed material arrays to store information for zone 1. The next available array for other zones wanting to add mixed materials to the mixed material arrays is element 3. Thus, when zone 2, which is also a mixed zone, needs to have its information added to the mixed material arrays, you store -3 into the `matlist` array to indicate that zone 2’s values begin at zone 3 in the mixed material arrays.

The mixed material arrays are a set of 4 parallel arrays: `mix_zone`, `mix_mat`, `mix_vf`, and `mix_next`. All of the arrays have the number of elements but that number varies depending on how many mixed zones there are in the material object. The `mix_zone` array contains the index of the zone that owns the material information for the current array element. That is, if you examine element 14 in the `mix_zone` array, you will know that element 14 in all of the mixed material arrays contain information about zone 11.

The `mix_mat` array contains the material numbers of the materials that occupy a zone. Material numbers correspond to the names of materials (e.g. 1 = Water) and should begin at 1 and increment from there. The range of material numbers used may contain gaps without causing any problems in VisIt. However, if you create databases that have many domains that vary over time, you’ll want to make sure that each domain has the same list of materials at every time step. It is not necessary to use a material number in the `matlist` array or in the mixed material arrays in order to include it in a material object. Look at element 11 in the `mix_mat` array in Figure 3-15. Element 11 contains material 1, element 12 contains material 2, and element 13 contains material 3. Since those three material numbers are supposed to all be present in zone 10, they are all added to the `mix_mat` array. The same array elements in the `mix_vf` array record the amount of each material in zone 10. The values in the `mix_vf` array for zone 10 are: 0.2, 0.4, 0.4 and those numbers mean that 20% of zone 10 is filled with material 1, 40% is filled with material 2, and 40% is filled with material 3. Note that all of the numbers for a zone in the `mix_vf` array must sum to 1., or 100%.

The `mix_next` array contains indices to the next element in the mixed material arrays that contains values for the mixed material zone under consideration. The `mix_next` array allows you to construct a linked-list of material numbers for a zone within the mixed material arrays. This means that the information for one zone's mixed materials could be scattered through the mixed material arrays but in practice the mixed material information for one zone is usually contiguous within the mixed material arrays. The `mix_next` array contains the next index to use within the mixed material arrays or it contains a zero to indicate that no more information for the zone is available.

To write materials to a Silo file, you use the `DBPutMaterial` function. The `DBPutMaterial` function is covered in the *Silo User's Guide* but it is worth noting here that it can be called to write either mixed materials or clean materials. The examples so far have illustrated the more complex case of writing out mixed materials. You can pass the `matlist` array and the mixed material arrays to the `DBPutMaterial` function or, in the case of writing clean materials, you can pass only the `matlist` array and `NULL` for all of the mixed material arrays. Note that when you write clean materials, your `matlist` array will contain only the numbers of valid materials. That is, the `matlist` array does not contain any negative mixed material array indices when you write out clean material objects.

Listing 3-16: mixedmaterials.c: C-Language example for writing mixed materials using Silo.

```
/* Material arrays */
int nmats = 2, mdims[2];
int matnos[] = {1,2,3};
char *matnames[] = {"Water", "Membrane", "Air"};
int matlist[] = {
    3, -1, -3, 1,
    3, -5, -7, 1,
    3, -9, -11, -14
};
float mix_vf[] = {
    0.75,0.25,    0.1875,0.8125,
    0.625,0.375,    0.4375,0.56250,
    0.3,0.7,    0.2,0.4,0.4,    0.45,0.55
};
int mix_zone[] = {
    1,1,  2,2,
    5,5,  6,6,
    9,9, 10,10,10, 11,11
};
int mix_mat[] = {
    2,3,  2,1,
    2,3,  2,1,
    2,3,  1,2,3,  2,1
};
int mix_next[] = {
    2,0,  4,0,
    6,0,  8,0,
    10,0, 12,13,0,  15,0
}
```

```

};
int mixlen = 15;

/* Write out the material */
mdims[0] = NX-1;
mdims[1] = NY-1;
optlist = DBMakeOptlist(1);
DBAddOption(optlist, DBOPT_MATNAMES, matnames);
DBPutMaterial(dbfile, "mat", "quadmesh", nmats, matnos, matlist,
  mdims, ndims, mix_next, mix_mat, mix_zone, mix_vf, mixlen,
  DB_FLOAT, optlist);
DBFreeOptlist(optlist);

```

Listing 3-17: fmixedmaterials.f: Fortran language example for writing mixed materials using Silo.

```

subroutine write_mixedmaterial(dbfile)
  implicit none
  integer dbfile
  include "silo.inc"
  integer NX, NY
  parameter (NX = 5)
  parameter (NY = 4)
  integer err, ierr, optlist, ndims, nmats, mixlen
  integer mdims(2) /NX-1, NY-1/
  integer matnos(3) /1,2,3/

  integer matlist(12) /3, -1, -3, 1,
. 3, -5, -7, 1,
. 3, -9, -11, -14/

  real mix_vf(15) /0.75,0.25,      0.1875,0.8125,
. 0.625,0.375,      0.4375,0.56250,
. 0.3,0.7,          0.2,0.4,0.4,      0.45,0.55/

  integer mix_zone(15) /1,1,  2,2,
. 5,5,  6,6,
. 9,9, 10,10,10, 11,11/

  integer mix_mat(15) /2,3,  2,1,
. 2,3,  2,1,
. 2,3,  1,2,3,  2,1/

  integer mix_next(15) /2,0,  4,0,
. 6,0,  8,0,
. 10,0, 12,13,0, 15,0/

  ndims = 2
  nmats = 3
  mixlen = 15
c Write out the material
  err = dbputmat(dbfile, "mat", 3, "quadmesh", 8, nmats, matnos,
. matlist, mdims, ndims, mix_next, mix_mat, mix_zone, mix_vf,
. mixlen, DB_FLOAT, DB_F77NULL, ierr)

```

end



Chapter 4

Creating a database reader plug-in

1.0 Overview

This chapter shows how to extend VisIt by writing a new database reader plug-in so you can use VisIt to access data files that you have already generated. Writing a database reader plug-in has several advantages over other approaches to importing data into VisIt such as writing a conversion program. First of all, if VisIt can natively read your file format then there is no need to convert files and consume extra disk space. Converting files may not even be possible if the data files are prohibitively large. Secondly, plug-ins offer the advantage of not having to alter a complex simulation code to write out data that VisIt can read. New plug-ins are free to read the simulation code's native file format. While many approaches to importing data into VisIt require new, specialized, code, when you write a database plug-in, the code that you write is external to your simulation and it is not a convertor that you have to maintain. There is no doubt that there is some maintenance involved in writing a database reader plug-in for VisIt but there is always the option of contributing your plug-in back into the VisIt source code tree where the code maintenance burden is shared among the developer community.

This chapter first reviews the VisIt architecture and describes where plug-ins fit into that scheme. After plug-ins are discussed, the steps that you must follow in order to create a plug-in are outlined. After covering the basics, you can dive into the section that covers how to implement your plug-in. Finally, once you have a working plug-in, you can add advanced features.

2.0 Structure of VisIt

VisIt is a parallel, distributed application that consists of four component processes that work in tandem to produce your visualizations. The two components that you may already

be familiar with are the client and the viewer. VisIt has GUI, Python interface, and Java clients that control the visualization operations performed by the viewer, which is the central state repository and graphics rendering component. The other components, which are not immediately visible, are the database server and the compute engine. The database server (sometimes called the meta-data server) is responsible for browsing the file system and letting you know which files can be opened. Once you decide on a file to open, the database server attempts to open that file, loading an appropriate database reader plug-in to do so. Once the database server has opened a file, it sends file metadata such as the list of available variables to the client and the viewer. The compute engine comes into play when you want to create a plot to process your data into a form that can be rendered on the screen. The compute engine, like the database server, loads a plug-in to read a data file and does the actual work of reading the problem-sized data from the file and translating it into Visualization Toolkit (VTK) objects that VisIt can process. Once the data has been read, it is fed through the visualization pipeline and returned to the viewer component where it can be displayed.

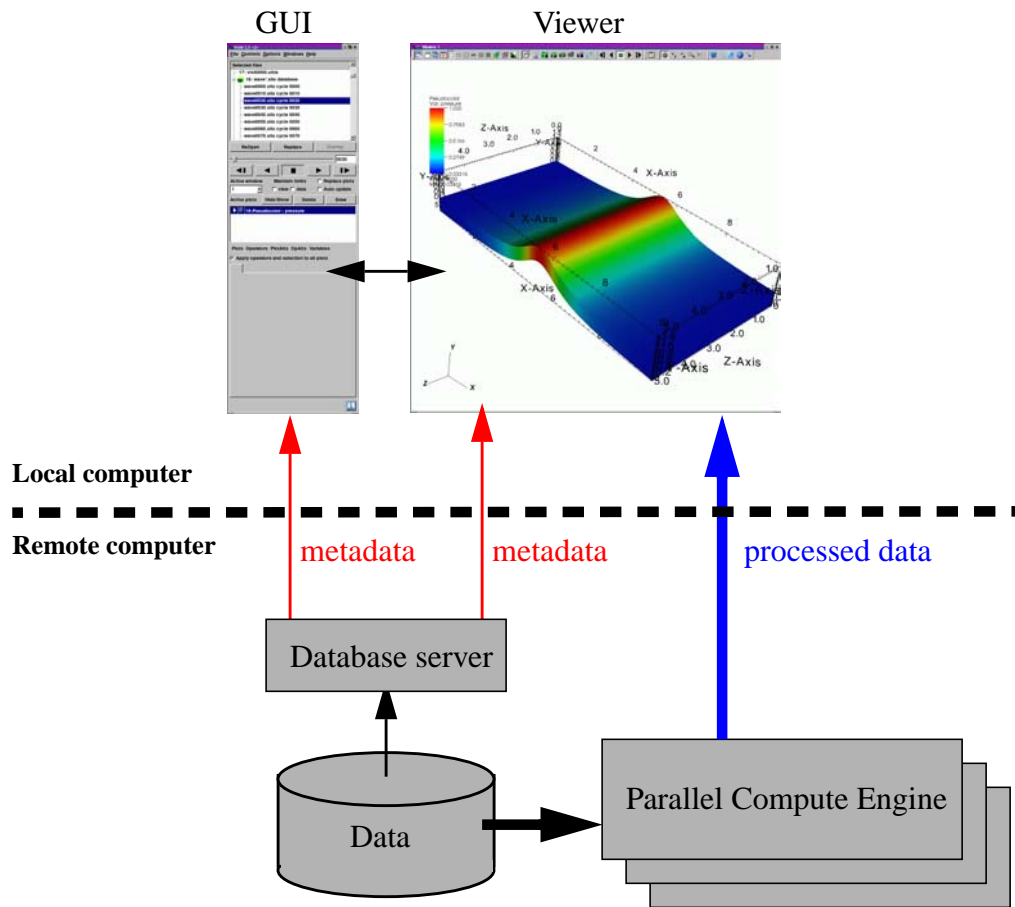


Figure 4-1: VisIt's architecture

2.1 plug-ins

VisIt supports three types of plug-ins: plot plug-ins, operator plug-ins, and database reader plug-ins. This chapter explores database reader plug-ins as a method of importing data from new file formats into VisIt. A database reader plug-in is made of three shared libraries, which are dynamically loaded by the appropriate VisIt components when data from a file must be read. The VisIt components involved in reading data from a file are the database server and the compute engine. Each database reader plug-in has a database server component, a compute engine component, and an independent component, for a total of three shared libraries (`libM`, `libE`, `libI`).

The independent plug-in component, or `libI` plug-in component, is a very lightweight shared library containing little more than the name and version of a plug-in as well as the file extensions that should be associated with it. When the database server and compute engine initialize at runtime, one of their first actions is to scan VisIt's plug-in directories for available `libI` plug-ins and then load all of the `libI` plug-ins to assemble an internal list of known plug-ins along with the table of file extensions for each file.

When VisIt needs to open a file, the filename is first passed to the database server, which tries to extract a file extension from the end of the filename so an appropriate plug-in can be selected from the list of available plug-ins. Once one or more matches are made, the database factory object in the database server loads the `libM` plug-in component for the first plug-in in the list of matching plug-ins. The `libM` plug-in component is the piece of the plug-in used by the database server and it is used to read the metadata from the file in question. If the plug-in cannot open the file then it should throw an exception to make the database factory attempt to open the file using the next matching plug-in. If there are no plug-ins that match the file's file extension then a default database plug-in is used. If that plug-in cannot open the file then VisIt issues an error message. Once the `libM` plug-in has read the metadata from the file, that information is sent to the VisIt clients where it can be used to populate variable menus, etc.

When you add a plot in VisIt and click the **Draw** button, the first step that the compute engine takes to process your request is to open the file that contains the data. The procedure for opening the file that contains the data in the compute engine is the same as that for the database server. In fact, the same database factory code is used internally. However, the database factory in the compute engine loads the `libE` plug-in component. The `libE` and `libM` plug-in components are essentially the same except that, when possible, database server plug-in components do less work. Both the `libE` and `libM` plug-in components contain code to read a file's metadata and both contain code to read variables and create meshes. The difference between the two plug-in types is that the code to read the variables and create meshes is only called from the `libE` plug-in component.

3.0 Starting your plug-in

Now that you know the basics of how VisIt uses database reader plug-ins in order to read different types of files, it is time to begin your plug-in. This section explains the different interfaces available for coding your plug-in and also covers the steps involved to create your plug-in code skeleton and run it for the first time.

3.1 Picking a database reader plug-in interface

Database reader plug-ins have 4 possible interfaces, which affect how files are mapped to plug-in file format objects. The 4 possible interfaces are shown in the table below:

	SD	MD
ST	STSD - Single time state per file and it contains just 1 domain.	STMD - Single time state per file but each file contains multiple domains.
MT	MTSD - Multiple time states per file and each file contains just 1 domain	MTMD - Multiple time states per file and each file contains multiple domains.

In order to pick which plug-in interface is most appropriate for your particular file format, you must consider how your file format treats time and domains. If your file format contains multiple time states in each file then you have an MT file format; otherwise you have an ST file format. If your file format comes from a parallel simulation then you will often have some type of domain decomposition, which breaks up the entire simulation into smaller pieces called domains that are divided among processors. If your simulation has domains and the domains are written to a single file then you have an MD file format; otherwise, if your simulation processors wrote out their own files then you have an SD file format. When you consider both how your file format deals with time and how it deals with domains, you should be able to select which plug-in interface you will need when you write your database reader plug-in.

3.2 Using XMLEdit

Once you pick which database interface you will use to write your database plug-in, the next step is to use VisIt's XMLEdit tool to get started with some interface definitions. XMLEdit is a graphical application that lets you create an XML file that describes some of the basic attributes for your database reader plug-in. The XML file contains information such as the name of the plug-in, its version, which interface is used, the plug-in's list of file extensions, and any additional libraries or source code files that need to be included in the plug-in in order to build it.

To get started with building your plug-in, the first step is to create a source code directory to contain all of the files that will be created to generate your plug-in. It is best that the directory name be the name of your file format or the name of your simulation. Once you have created a directory for your plug-in files, you can run VisIt's XMLEdit program. To start XMLEdit on UNIX systems where VisIt is installed, open a command window and type `xmledit`. On Windows systems, XMLEdit should be available in the **Start** menu under VisIt's plug-in development options.

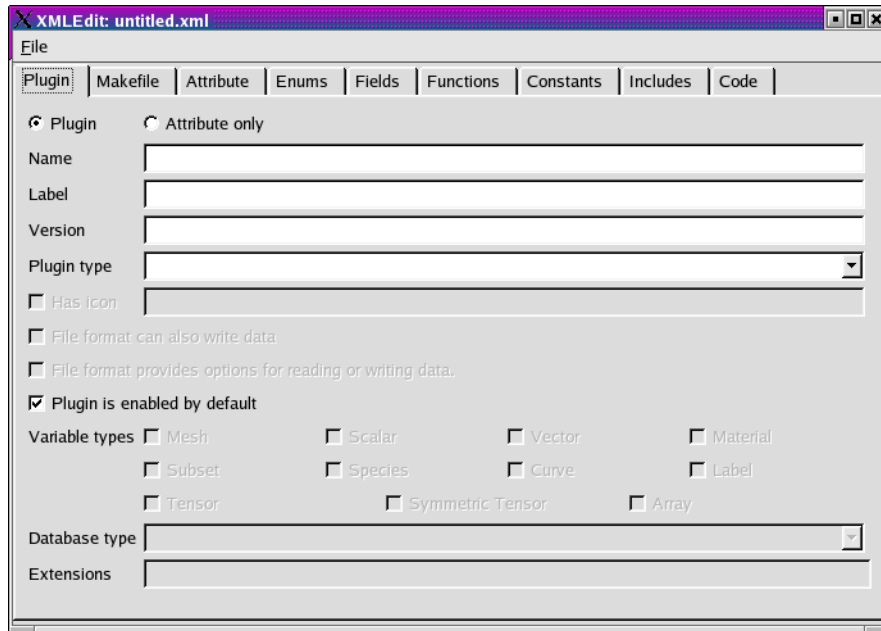


Figure 4-2: XMLEdit plug-in tab

Once XMLEdit is active you can see that it has a number of tabs that are devoted to various aspects of plug-in development. Most of the tabs are used for developing plot and operator plug-ins only so this section will focus on the actions that you need to take to create your database reader plug-in. First of all, you must type the name of your plug-in into the **Name** text field. The name should match the name of the source code directory that you created - be sure that you pick a name that can be used inside of C++ class names since the name is used to help generate the plug-in code skeleton that will form the basis of your database reader plug-in. Next, type in a label into the **Label** text field. The label for a database plug-in can contain a longer identifier that will be displayed when VisIt uses your plug-in to read files. The label may contain spaces and punctuation. Next, enter the version of your plug-in into the **Version** text field. The version for initial development should be: *1.0*. Now, choose *Database* from the **Plugin type** combo box to tell XMLEdit that you want to build a database reader plug-in. Once you choose *Database* for your plug-

in type, some additional options will become enabled. You can ignore these options for now since they contain reasonable default values.

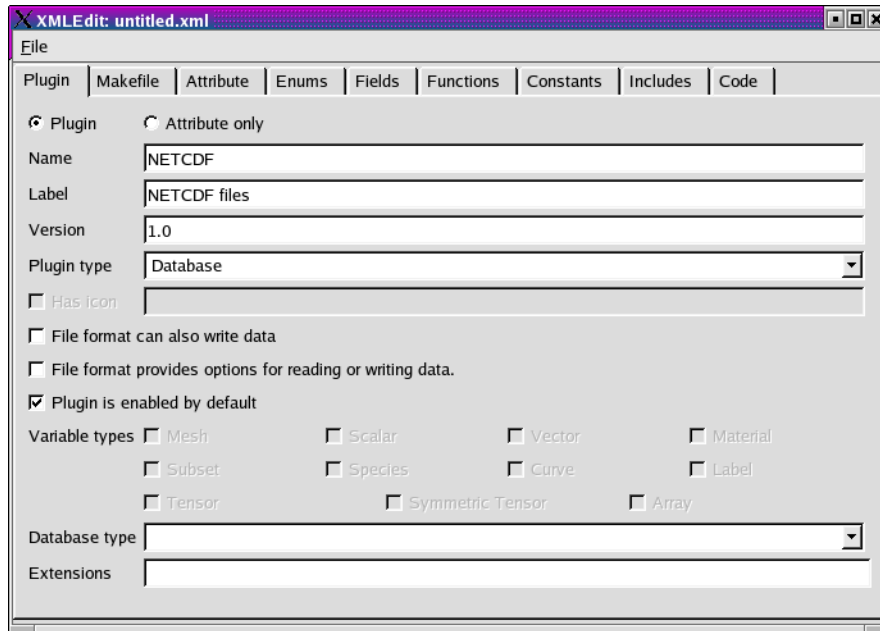


Figure 4-3: XMLEdit plug-in tab with plug-in name and type selected

The next step in creating your database plug-in using XMLEdit is to set the database type to *STSD*, *STMD*, *MTSD*, *MTMD* by selecting one of those options from the **Database type** combo box. Note that it is possible to instead choose to create a fully custom

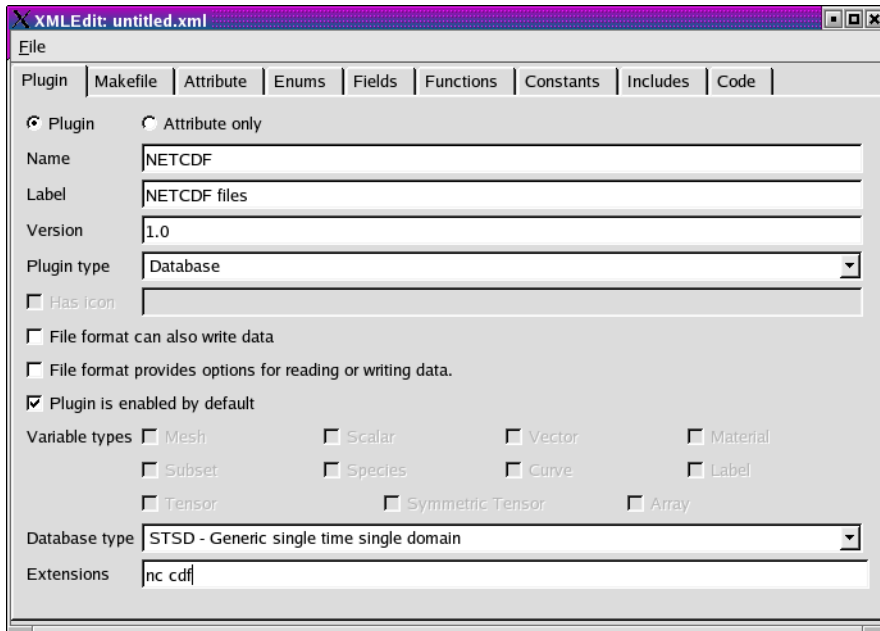


Figure 4-4: XMLEdit plug-in tab with database type and extensions selected

database type but do not choose that option since most formats do not need that level of customizeability. Once you have selected a database type for your plug-in, type in the list of file formats that you want to associate with your plug-in. You can enter as many space-delimited file extensions as you want.

The information that you entered is the minimum amount of information required to create your database reader plug-in. Save your XMLEdit session to an XML file by selecting **Save** from the **File** menu. Be sure to use the same name as you used for the directory name that will contain your plug-in files and also be sure to save your XML file to that directory. At this point, you can skip ahead to generating your plug-in code skeleton or you can continue adding options to your XML file.

3.2.1 Makefile options

XMLEdit contains controls on its Makefile tab that allow you to add options to your XML file that will influence how your plug-in code is built when you go to compile it. For example, the Makefile tab includes options that allow you to specify compiler options such as **CXXFLAGS**, **LDFLAGS**, and **LIBS**. Adding options to these fields can be particularly useful if your plug-in uses an external library such as NETCDF or HDF5. You can add the include file and library file locations to ensure that the compiler will know where to look for your external library when your plug-in is built. You can also add extra files to the `libE` and `libM` plug-ins by adding a list of files to the **Engine files** and **MDServer files** text fields, respectively. If you change any of these options, shown in Figure 4-5, be sure to save your XML file before quitting XMLEdit.

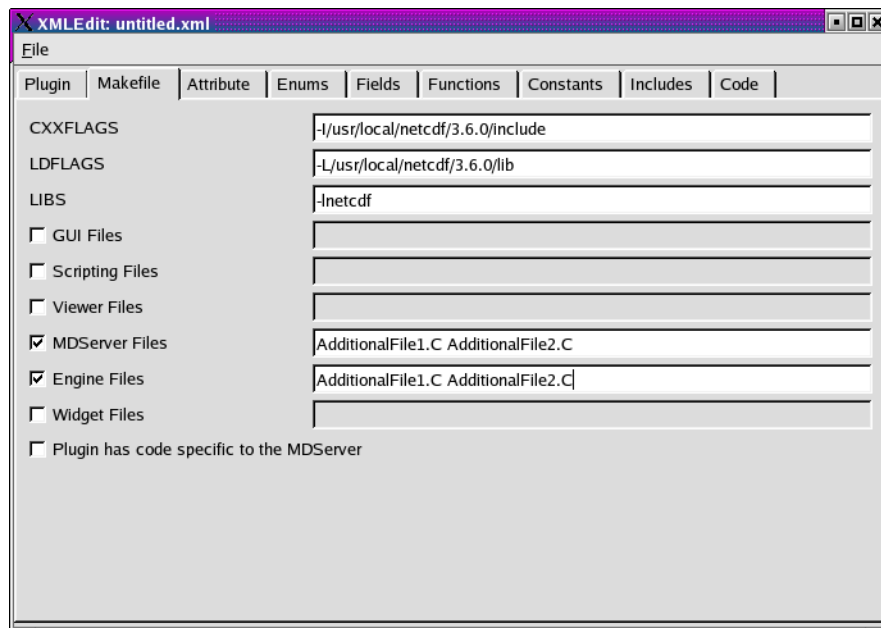


Figure 4-5: XMLEdit Makefile tab with compiler options and additional files specified.

3.3 Generating a plug-in code skeleton

Once you save your work from XMLEdit, you will find an XML file containing the options that you provided in the directory where you store your plug-in files. VisIt provides more XML tools to generate the necessary code skeleton for your plug-in. The important tools when building a database plug-in are: `xml2makefile`, `xml2info`, `xml2plugin`. The `xml2plugin` program is actually a script that automates calling the required `xml2*` programs. In order to generate your plug-in code skeleton, open a command window, go to the directory containing your XML file, and run `xml2plugin`. On UNIX systems, the command that you will run is:

```
xml2plugin -clobber FILE.xml
```

Be sure to replace *FILE.xml* with the name of your own XML file. Once you run the `xml2plugin` program, if you look in your directory, you will see several new files.

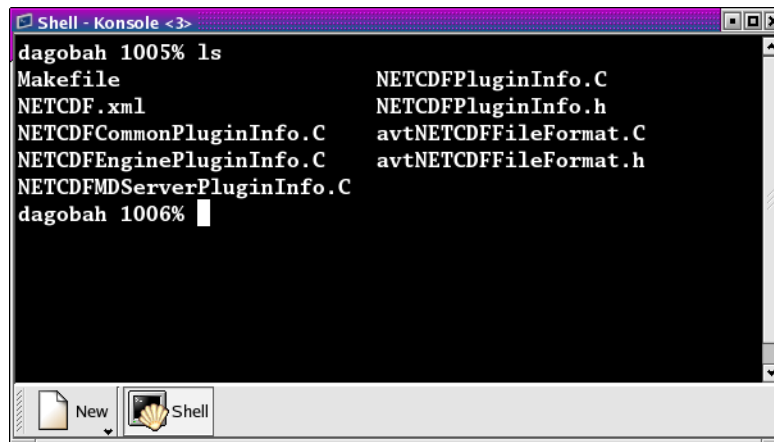
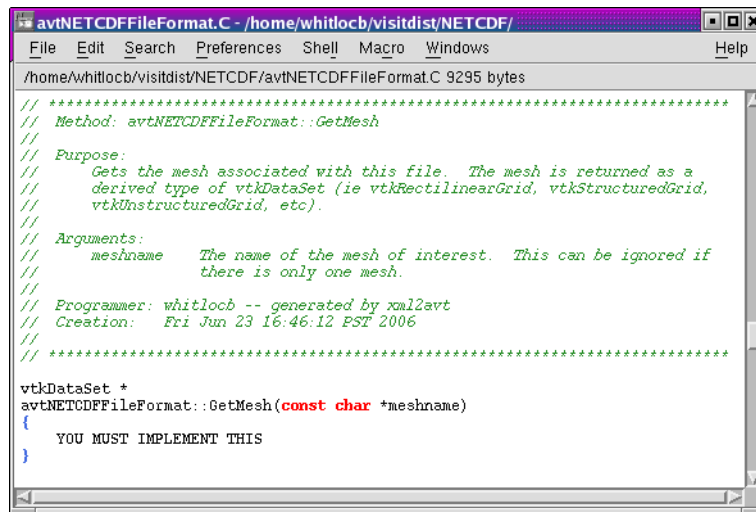


Figure 4-6: Files generated by `xml2plugin`

For database reader plug-ins, there are essentially three classes of files that `xml2plugin` creates. First of all, `xml2plugin` creates the plug-in code skeleton, which includes the plug-in entry points that are used to load the plug-in dynamically at runtime. These files have “*Info*” in their name and they are generated by the `xml2info` program. If you change the name, version, or file extensions that your plug-in uses then you should re-run `xml2info` instead of running `xml2plugin`. The next set of files are the AVT file format source and header files. The AVT file format source code files are C++ source code files that you will complete using new code to read your file format. Finally, `xml2makefile`, created a `Makefile` for your plug-in so all you have to do in order to build your plug-in is type: `make` at the command prompt.

3.4 Building your plug-in

So far, we have created an XML file using the XMLEdit program and then used the XML file with VisIt's XML tools to generate plug-in source code. The static portions of the generated source code is complete but there are still some pieces that you need to write yourself in order to make VisIt read your data files. The automatically generated files that are called *avtXXXXFileFormat.C* and *avtXXXXFileFormat.h*, where XXXX is the name of your plug-in, are incomplete. These two AVT files contain a derived class of one of the STSD, STMD, MTSD, MTMD file format classes that VisIt provides for reading different file types. Your job is to fill in the missing code in the methods for the AVT classes so they can read data from your file format and translate that data into VTK objects. By default, the AVT files contain some messages in the source code like “*YOU MUST IMPLEMENT THIS*”, which are meant to prevent the source code from compiling and to call attention to areas of the plug-in that you need to implement (See Figure 4-7).



```

avtNETCDFFileFormat.C - /home/whitlocb/visitdist/NETCDF/
File Edit Search Preferences Shell Macro Windows Help
/home/whitlocb/visitdist/NETCDF/avtNETCDFFileFormat.C 9295 bytes
// *****
// Method: avtNETCDFFileFormat::GetMesh
//
// Purpose:
// Gets the mesh associated with this file. The mesh is returned as a
// derived type of vtkDataSet (ie vtkRectilinearGrid, vtkStructuredGrid,
// vtkUnstructuredGrid, etc).
//
// Arguments:
// meshname The name of the mesh of interest. This can be ignored if
// there is only one mesh.
//
// Programmer: whitlocb -- generated by xml2avt
// Creation: Fri Jun 23 16:46:12 EST 2006
// *****

vtkDataSet *
avtNETCDFFileFormat::GetMesh(const char *meshname)
{
    YOU MUST IMPLEMENT THIS
}

```

Figure 4-7: Example of a “YOU MUST IMPLEMENT THIS” message

The first step in building a plug-in is to make sure that the automatically generated source code compiles. Open the AVT files and look for instances of the “*YOU MUST IMPLEMENT THIS*” message and, when you find them, write down a note of where they appear. Comment out each of the messages in the C++ source code and add “`return 0;`” statements (See Figure 4-8). By commenting out the offending messages, the automatically generated source code will compile when you attempt to compile the plug-in. You will also have a list of some of the plug-in methods that you will have to write later when you really begin developing your plug-in.

```

avtNETCDFFileFormat.C - /home/whitlocb/visitdist/NETCDF/
File Edit Search Preferences Shell Macro Windows Help
/home/whitlocb/visitdist/NETCDF/avtNETCDFFileFormat.C line 199, col 13, 9311 bytes
// *****
// Method: avtNETCDFFileFormat::GetMesh
//
// Purpose:
// Gets the mesh associated with this file. The mesh is returned as a
// derived type of vtkDataSet (ie vtkRectilinearGrid, vtkStructuredGrid,
// vtkUnstructuredGrid, etc).
//
// Arguments:
// meshname The name of the mesh of interest. This can be ignored if
// there is only one mesh.
//
// Programmer: whitlocb -- generated by xml2avt
// Creation: Fri Jun 23 16:46:12 EST 2006
// *****

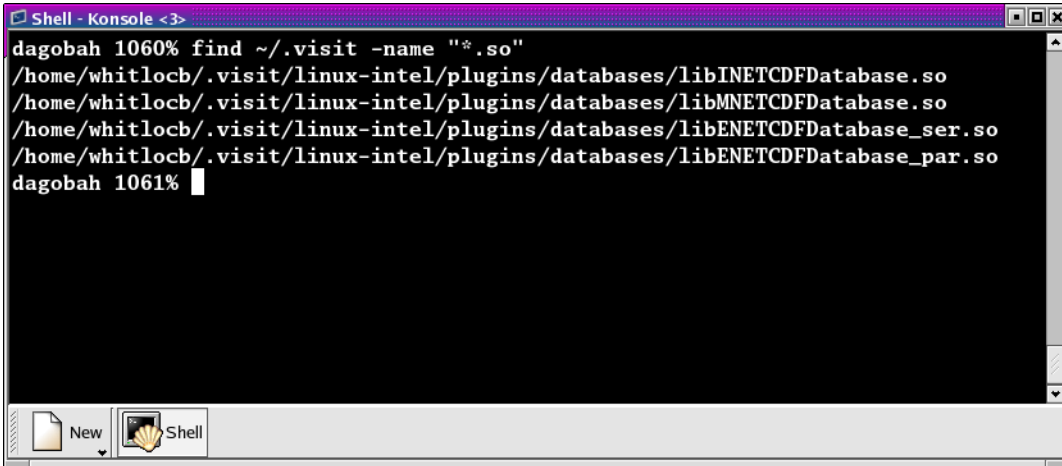
vtkDataSet *
avtNETCDFFileFormat::GetMesh(const char *meshname)
{
    //YOU MUST IMPLEMENT THIS
    return 0;
}

```

Figure 4-8: Example of corrections made to a “YOU MUST IMPLEMENT THIS” message needed to make the source code compile

Once you have changed the AVT files so there are no stray messages about implementing a plug-in feature, go back to your command terminal and type the make command for your system (commonly `make` or `gmake`). The make command takes the automatically generated Makefile that was generated by `xml2makefile` and starts building your plug-in against the installed version of VisIt. If you encounter compilation errors, such as syntax errors, then you most likely need to make further changes to your AVT files before trying to build your plug-in. A good C++ language reference can help you understand the types of errors that may be printed to your command window in the event that you have not successfully changed the AVT files. If your source code seems to compile but fails due to missing libraries such as NETCDF or HDF5 then you can edit your Makefile so it points to the right library installation locations.

Once your plug-in is built, it will be stored in a platform-specific subdirectory of the `.visit` directory in your home directory (`~/ .visit`). If you type: `find ~/ .visit -name "*.so"` into your command window, you will be able to locate the `libE`, `libI`, and `libM` files that make up your compiled plug-in (see Figure 4-9). If you develop for MacOS X, you should substitute `*.dylib` for `*.so` in the previous command because shared libraries on MacOS X have a `.dylib` file extension instead of a `.so` file extension. Note that when a parallel compute engine is available in the installed version of VisIt, you will get two `libE` plug-ins; one with a `_ser` suffix and one with a `_par` suffix. The `libE` files that have a `_ser` suffix are loaded by the serial compute engine and the `_par` `libE` file is loaded by the parallel compute engine and may contain parallel function calls, such as calls to the MPI library.



```

Shell - Konsole <3>
dagobah 1060% find ~/.visit -name "*.so"
/home/whitlocb/.visit/linux-intel/plugins/databases/libINETCDFDatabase.so
/home/whitlocb/.visit/linux-intel/plugins/databases/libMNETCDFDatabase.so
/home/whitlocb/.visit/linux-intel/plugins/databases/libENETCDFDatabase_ser.so
/home/whitlocb/.visit/linux-intel/plugins/databases/libENETCDFDatabase_par.so
dagobah 1061%

```

Figure 4-9: Files are created in the `.visit` directory when a plug-in is built.

When VisIt's database server and compute engine execute, they look in your `~/.visit` directory for available plug-ins and load any that are available. This means that even if you build plug-ins against the installed version of VisIt, it will still be able to find your private plug-ins.

It is recommended that while you develop your plug-ins, you only install them in your `~/.visit` directory so other VisIt users will not be affected. However, if you develop your plug-in on MacOS X, you will have to make sure that your plug-ins are installed publicly so that they can be loaded at runtime. You can also choose to install your plug-ins publicly once you have completed development. To install plug-ins publicly, first remove the files that were installed to your `~/.visit` directory by typing the `make clean` command in your command window. Next, re-run the `xml2makefile` program like this: `xml2makefile -public -clobber FILE.xml`. Adding the `-public` argument on the command line causes `make` to install your plug-in files publicly so all VisIt users can access them.

3.5 Calling your plug-in for the first time

Once you have completed building your plug-in for the first time, all that you need to do is run VisIt and try to open one of your files. When you open one of your files, the database server should match the file extension of the file that you tried to open with the list of file extensions that your plug-in accepts, causing your plug-in to be loaded and used for opening the file. You can verify that VisIt used your plug-in by opening the **File Information** window (see Figure 4-10) in the VisIt GUI and looking for the name of your plug-in in the listed information.

Note that at this stage, the database server should be properly loading your database reader plug-in but since no code to actually read your files has yet been added to the AVT source code files, no plottable meshes or variables will be available.

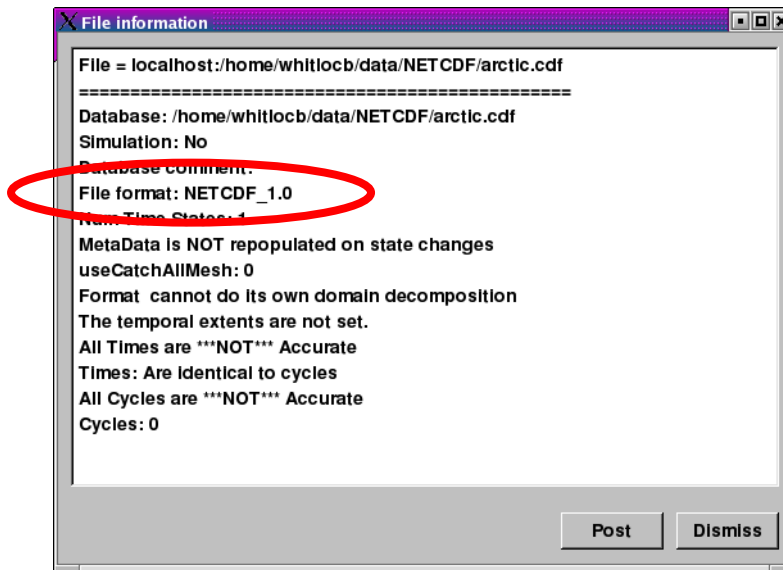


Figure 4-10: File Information window confirming use of your plug-in.

4.0 Implementing your plug-in

Now that you have built a working plug-in framework, you are ready to begin adding code to your plug-in that will make it capable of opening your file format, reading data, and translating that data into VTK objects. This section explores the details of writing the AVT code for your database reader plug-in, providing necessary background and then diving into specific topics such as how to return data for a particular mesh type. **Before starting, remember that building a plug-in is an incremental process and you should proceed in small steps, saving your work, building, and testing your plug-in each step of the way.**

4.1 Required plug-in methods

Most of the code in a VisIt database plug-in is automatically generated and, for the most part, the only code that you need to modify is the AVT code. The AVT code contains a class definition and implementation for a derived type of the STSD, STMD, MTSD, or MTMD file format classes and your job as a plug-in developer is to write the required methods for your derived file format class so that VisIt can read your file. There are many methods in the file format class interface that you can override to make your plug-in perform specialized operations. The only methods that you absolutely must implement are

the `PopulateDatabaseMetaData`, `GetMesh`, `GetVar`, and `GetVectorVar` methods. The purpose of each of these plug-in methods is listed in the following table.

Method	Purpose
<code>PopulateDatabaseMetaData</code>	VisIt calls the <code>PopulateDatabaseMetaData</code> method when file metadata is needed. File metadata is returned in a pass-by-reference <code>avtDatabaseMetaData</code> object. File metadata consists of the list of names of meshes, scalar variables, vector variables, tensor variables, label variables, array variables, expressions, cycles, and times contained in the file. These lists of variables and meshes let VisIt know the names of the objects that can be plotted from your file. The metadata is used primarily to populate the plot menus in the GUI and viewer components. The <code>PopulateDatabaseMetaData</code> method is called by both the <code>libM</code> and <code>libE</code> plug-ins.
<code>GetMesh</code>	VisIt calls the <code>GetMesh</code> method in a <code>libE</code> plug-in when it needs to plot a mesh. This method is the first method to return “problem-sized” data, meaning that the mesh data can be as large as the data in your file. The <code>GetMesh</code> method must return a mesh object in the form of one of the VTK dataset objects (<code>vtkRectilinearGrid</code> , <code>vtkStructuredGrid</code> , <code>vtkUnstructuredGrid</code> , <code>vtkPolyData</code>)
<code>GetVar</code>	VisIt calls the <code>GetVar</code> method in a <code>libE</code> plug-in when it needs to read a scalar variable. Like the <code>GetMesh</code> method, this method returns “problem-sized” data. <code>GetVar</code> reads data values from the file format, possibly performing calculations to alter the data, and stores the data into a derived type <code>vtkDataArray</code> object such as <code>vtkFloatArray</code> or <code>vtkDoubleArray</code> . If your file format does not need to return scalar data then you can leave the “ <code>return 0;</code> ” implementation that you added in order to get your plug-in to build.

Method	Purpose
GetVectorVar	VisIt calls the GetVectorVar method in a libE plug-in when it needs to read a vector or tensor variable. GetVectorVar performs the same function as GetVar but returns vtkFloatArray or vtkDoubleArray objects that have more than one value per tuple. A tuple is the equivalent of a value associated with a zone or node but it can store more than one value. If your file format does not need to return scalar data then you can leave the “return 0;” implementation that you added in order to get your plug-in to build.

4.2 Debugging your plug-in

Before beginning to write code for your plug-in, you should know a few techniques for debugging your plug-in since debugging VisIt can be tricky because of its distributed architecture.

4.2.1 Debugging logs

The first method debugging in VisIt is by using VisIt’s debug logs. When you run `visit` on the command line, you can optionally add the `-debug 5` arguments to make VisIt write out debugging logs. The number of debugging logs can be 1, 2, 3, 4, or 5, with debugging log 5 being the most detailed. When VisIt’s components are told to run with debugging logs turned on, each component writes a set of debugging logs. For example, the database server component will write `mdserver.1.log`, `mdserver.2.log`, ..., `mdserver.5.log` debugging logs if you pass `-debug 5` on the VisIt command line. Since you are writing a database reader plug-in, you will want to look at the `mdserver*.log` and `engine*.log` files since those components load your `libM` and `libE` plug-ins.

The debugging logs will contain information written to them by the debugging statements in VisIt’s source code. If you want to add debugging statements to your AVT code then you can use the `debug1`, `debug2`, `debug3`, `debug4`, or `debug5` streams as shown in the next code listing.

Listing 4-11: `debugstream.C`: C++-Language example for using debug streams.

```
// NOTE - This code incomplete and is for example purposes only.

// Include this header for debug streams.
#include <DebugStream.h>

vtkDataSet *
```

```

avtXXXXFileFormat::GetMesh(const char *meshname)
{
    // Write messages to different levels of the debug logs.
    debug1 << "Hi from avtXXXXFileFormat::GetMesh" << endl;

    debug4 << "Many database plug-ins prefer debug4" << endl;

    debug5 << "Lots of detail from avtXXXXFileFormat::GetMesh"
            << endl;

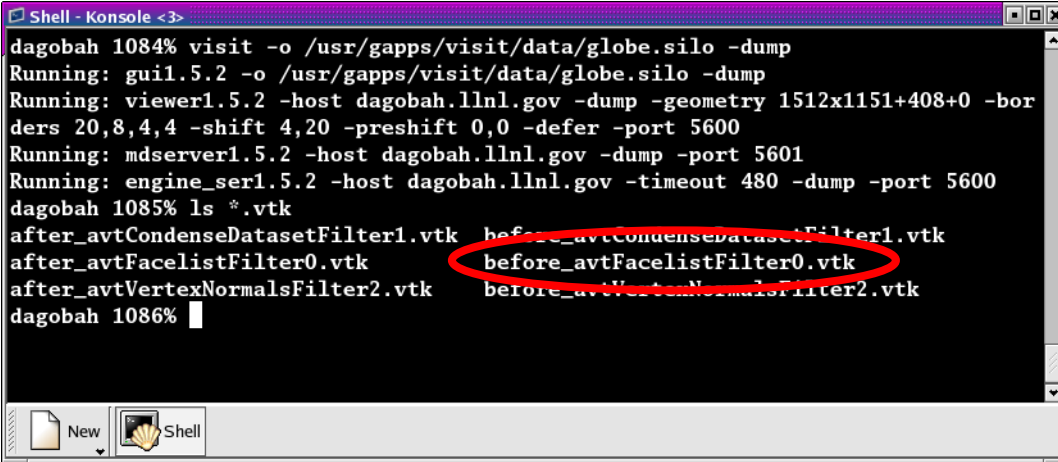
    return 0;
}

```

4.2.2 Dumping VTK objects to disk

In addition to the `-debug` argument, VisIt also supports a `-dump` argument. The `-dump` argument tells VisIt's compute engine to write VTK files containing the data for every stage of the pipeline execution so you can view the changes to the data made by each AVT filter. While this option is more useful when writing plots and operators, you can use it to examine the data at the beginning of the pipeline since, at that stage, the data will contain the VTK object that was created by your database reader plug-in.

When you run VisIt with the `-dump` argument, many VTK files will be created since the data is saved at every stage in the execution of VisIt's data processing pipeline. Each VTK file contains a number indicating the order of the filter in the pipeline that saved the data. Look for the filename of the form: *before*0.vtk*. The list of files created by using the `-dump` argument is shown in Figure 4-12.



```

Shell - Konsole <3>
dagobah 1084% visit -o /usr/gapps/visit/data/globe.silo -dump
Running: guil.5.2 -o /usr/gapps/visit/data/globe.silo -dump
Running: viewer1.5.2 -host dagobah.llnl.gov -dump -geometry 1512x1151+408+0 -bor
ders 20,8,4,4 -shift 4,20 -preshift 0,0 -defer -port 5600
Running: mdserver1.5.2 -host dagobah.llnl.gov -dump -port 5601
Running: engine_ser1.5.2 -host dagobah.llnl.gov -timeout 480 -dump -port 5600
dagobah 1085% ls *.vtk
after_avtCondenseDatasetFilter1.vtk  before_avtCondenseDatasetFilter1.vtk
after_avtFacelistFilter0.vtk         before_avtFacelistFilter0.vtk
after_avtVertexNormalsFilter2.vtk    before_avtVertexNormalsFilter2.vtk
dagobah 1086%

```

Figure 4-12: Output of running with the `-dump` command line argument

4.3 Opening your file

When VisIt receives a list of files to open, it tries to determine which plug-in should be loaded to access the data in those files. The match is performed by comparing the file extension of the files against the known file extensions for all database reader plug-ins. Each plug-in in the list of matches is loaded and VisIt creates instances of the plug-in's AVT file format classes that are then used to access the data in the files. When an AVT object is created, its constructor can open the data file and make sure that the file is of the appropriate type. If the file is not the right type, or if it contains errors, or if it cannot be accessed for some other reason, the constructor must throw an `InvalidDBTypeException` exception. When the `InvalidDBTypeException` exception is thrown from the constructor of an AVT file format derived type, VisIt's database factory catches the exception and then tries to open the file with the next matching plug-in. This procedure continues until the file is opened by a suitable plug-in or the file cannot be opened at all.

Listing 4-13: `invaliddbtype.C`: C++-Language example for a file format constructor that must throw an exception.

```
// NOTE - This code incomplete and is for example purposes only.

#include <InvalidDBTypeException.h>

avtXXXXFileFormat::avtXXXXFileFormat(const char *filename)
    : avtSTSDFileFormat(filename)
{
    bool fileOpened = false;

    // Open the file specified by the filename argument here using
    // your file format API and set fileOpened accordingly.
    YOU MUST IMPLEMENT THIS

    // If your file format API could not open the file then throw
    // an exception.
    if (!fileOpened)
    {
        EXCEPTION1(InvalidDBTypeException,
            "The file could not be opened");
    }
}
```

If you use a file extension that is already used by other VisIt database reader plug-ins, your file format's constructor should open the file to ensure that the file is of the right type. If your database reader plug-in uses a unique file extension then you have the option of deferring any file opens until later when metadata is required. This is the preferred approach because VisIt may create many instances of your file format class and doing less work in the constructor makes opening files faster.

Once you decide whether your file format can defer opening a file or whether it must open the file in the constructor, you can begin adding code to your AVT class. Since opening files can be a costly operation, you might want to open a file and keep it open if you have a random access file format. If you open a file in one method and want to keep the file open so it is available to multiple plug-in methods, you will need to add a new class member to your AVT class to contain the handle to your open file. If your file format consists of sequential text then you might consider reading the file once and keeping the data in memory in a format that you can conveniently translate into VTK objects. Both approaches require the addition of a new class member - either a handle to the file or a pointer to data that was read from the file.

4.4 Returning file metadata

Once you have decided how your plug-in will manage access to the file that it must read, the next step in writing your database reader plug-in is to implement the `PopulateDatabaseMetaData` method. The `PopulateDatabaseMetaData` method is called by VisIt's database infrastructure when information about a file's meshes and variables must be obtained. The `PopulateDatabaseMetaData` method is usually called only the first time that a file format's metadata is being read, though some time-varying formats can have time-varying metadata, which requires that `PopulateDatabaseMetaData` is called each time VisIt requests data for a new time state. However, most file formats call `PopulateDatabaseMetaData` once.

The `PopulateDatabaseMetaData` method arguments can vary, depending on whether your file format is STSD, STMD, MTSD, or MTMD but in all cases the first argument is an `avtDatabaseMetaData` object. The `avtDatabaseMetaData` object is a class that is pervasively used in VisIt; it contains information about the files that you plot such as the number of domains, times, meshes, and variables that the files can provide. When you implement your plug-in's `PopulateDatabaseMetaData` method, you must populate the `avtDatabaseMetaData` object with the list of meshes and variables, etc. that you want VisIt to be able to plot. You can hard-code a fixed list of meshes and variables if your file format always contains the same entities or you can open your file and provide a dynamic list of meshes and variables. This section covers how to add meshes and various variable types to the `avtDatabaseMetaData` object so your file format's data will be exposed in VisIt. For a complete listing of the `avtDatabaseMetaData` object's methods, see the `avtDatabaseMetaData.h` header file. It is worth noting that the following code examples create metadata objects and manually add them to the metadata object instead of using convenience functions. This is done because the convenience functions used in automatically generated plug-in code do not provide support for less often used metadata settings such as units and labels.

4.4.1 Returning mesh metadata

In order for you to be able to plot any data from your file format, your database reader plug-in must add at least one mesh to the `avtDatabaseMetaData` object that is passed into the `PopulateDatabaseMetaData` method. Adding information about a mesh to

the `avtDatabaseMetaData` object is done by creating an `avtMeshMetaData` object, populating its important members, and adding it to the `avtDatabaseMetaData`. At a minimum, each mesh must have a name, spatial dimension, topological dimension, and a mesh type. The mesh's name is the identifier that will be displayed in VisIt's plot menus and it is also the name that will be passed later on into the plug-in's `GetMesh` method.

The spatial dimension attribute corresponds to how many dimensions are needed to specify the coordinates for the points that make up your mesh. If your mesh exists in a 2D plane then choose 2, otherwise choose 3. Note that when you create the points for your mesh later in the `GetMesh` method, you will always create points that contain X,Y,Z points.

The topological dimension attribute describes the number of logical dimensions used by your mesh, regardless of the dimension of the space that it sits in. For example, you may have a planar surface of triangles sitting in 3D space. Such a mesh would be topologically 2D even though it sits in 3D space. The rule of thumb that VisIt follows is that if your mesh's cells are points then you have a mesh that is topologically 0D, lines are 1D, surfaces are 2D, and volumes are 3D. This point is illustrated in Figure 4-14.

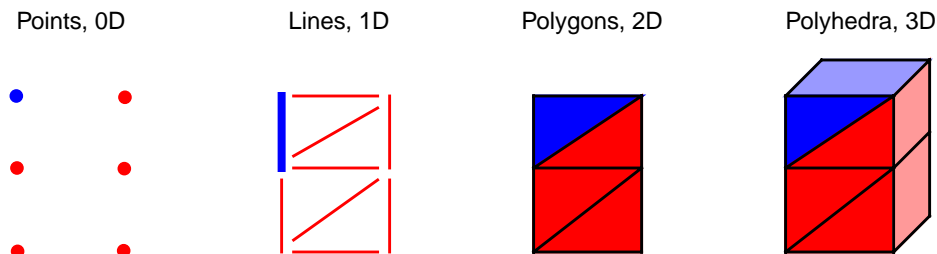


Figure 4-14: Topological dimensions. One zone is highlighted blue.

Once you have set the other basic attributes for your mesh object, consider which type of mesh you have. VisIt supports several different mesh types and the value that you provide in the metadata allows VisIt to tailor how it applies filters that process your data. If you have a mesh composed entirely of particles then choose `AVT_POINT_MESH`. If you have a structured mesh where the coordinates are specified by small vectors of values for each axis and the rest of the coordinates are implied then you probably have a rectilinear mesh and you should choose `AVT_RECTILINEAR_MESH`. If you have a structured mesh and every node has its own specific location in space then you probably have a curvilinear mesh and you should choose `AVT_CURVILINEAR_MESH`. If you have a mesh for which you specify a large list of nodes and then create cells using indices into that list of nodes then you probably have an unstructured mesh and you should choose `AVT_UNSTRUCTURED_MESH` for the mesh type. If you have a mesh that adaptively refines then choose `AVT_AMR_MESH`. Finally, if your mesh is specified using shapes such as cones and spheres that are unioned or differenced using boolean operations then you

have a constructive solid geometry mesh and you should choose `AVT_CSG_MESH` for your mesh's mesh type.

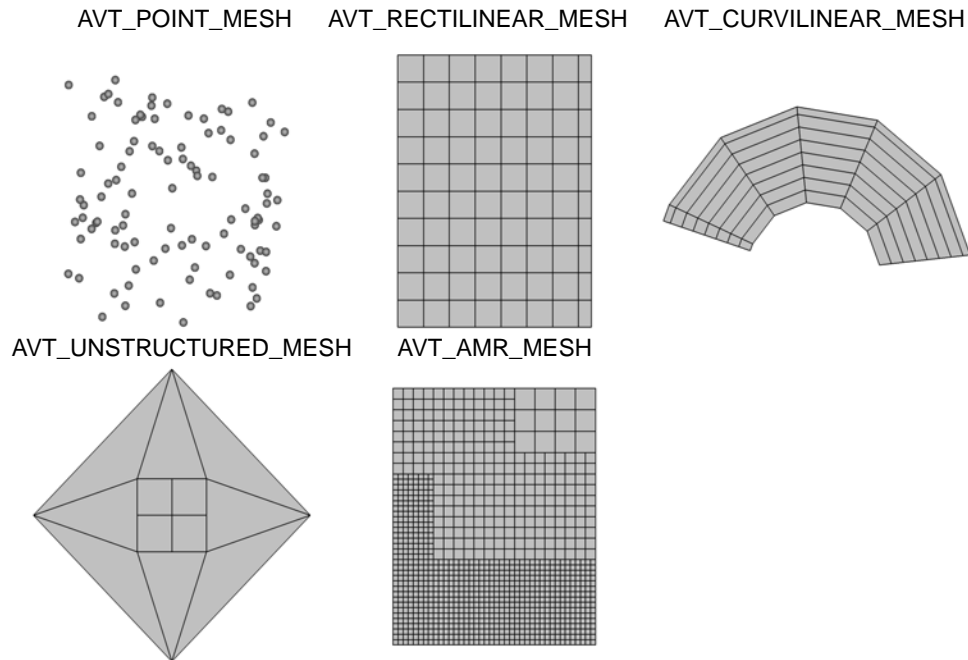


Figure 4-15: AVT mesh types (AVT_CSG_MESH not pictured).

If your mesh consists of multiple domains then you will need to set the number of domains into the `numBlocks` member of the `avtMeshMetaData` object. Remember that the number of domains tells VisIt how many pieces make up your mesh and it is especially important to specify this number if your plug-in is derived from an MD file format interface. You may also choose to tell VisIt what the domains are called for your file format. Some file formats use the word: “domains” while others use “brick” or “block”. If you choose to set the name that VisIt uses for domains then that term will be used in parts of VisIt’s GUI such as the **Subset** window. Set the `blockPieceName` member of the `avtMeshMetaData` object to a suitable term that describes a domain in the context of your simulation code. Alternatively, you can provide proper names by providing a vector of strings containing the names by setting the `blockNames` member.

Now that the most important attributes of the `avtMeshMetaData` object have been specified, you can add extra information such as the names or units of the coordinate dimensions. Once all attributes are set to your satisfaction, you must add the `avtMeshMetaData` object to the `avtDatabaseMetaData` object.

Listing 4-16: `meshmetadata.C`: C++-Language example for returning mesh metadata.

// NOTE - This code incomplete and is for example purposes only.

```

void
avtXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a point mesh to the metadata. Note that this example will
    // always expose a mesh called "particles" to VisIt. A real
    // plug-in may want to read a list of meshes from the data
    // file.
    avtMeshMetaData *mmd = new avtMeshMetaData;
    mmd->name = "particles";
    mmd->spatialDimension = 3;
    mmd->topologicalDimension = 0;
    mmd->meshType = AVT_POINT_MESH;
    mmd->numBlocks = 1;
    md->Add(mmd);

    // Add other objects to the metadata object.
}

```

4.4.2 Returning scalar metadata

Once you have exposed a mesh to VisIt by adding mesh metadata to the `avtDatabaseMetaData` object, you can add scalar field metadata to the metadata. A scalar field is a set of floating point values defined for all cells or nodes of a mesh. You can expose as many scalar variables as you want on any number of meshes. The list of scalar fields that a plug-in exposes is often determined by the data file being processed. Like mesh metadata, scalar metadata requires a name so the scalar can be added to VisIt's menus. The name that you choose is the same name that later is passed to the `GetVar` plug-in method. Once you select a name for your scalar variable, you must indicate the name of the mesh on which the variable is defined by setting the `meshName` member of the `avtScalarMetaData` object. Once you have set the name and `meshName` members, you can set the centering member. The centering member of the `avtScalarMetaData` object can be set to `AVT_NODECENT` or `AVT_ZONECENT`, indicating that the data is defined on the nodes or at the zone centers, respectively. If you want to indicate units that are associated with the scalar variable, set the `hasUnits` member to `true` and set the `units` string to the appropriate unit names.

Listing 4-17: `scalarmetadata.C`: C++-Language example for returning scalar metadata.

```

// NOTE - This code incomplete and is for example purposes only.

void
avtXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a mesh called "mesh" to the metadata object.

    // Add a scalar to the metadata. Note that this plug-in will
    // always expose a scalar called "temperature" to VisIt. A real
    // plug-in may want to read a list of scalars from the data

```

```

// file.
avtScalarMetaData *smd = new avtScalarMetaData;
smd->name = "temperature";
smd->meshName = "mesh";
smd->centering = AVT_ZONECENT;
smd->hasUnits = true;
smd->units = "Celsius";
md->Add(smd);

// Add other objects to the metadata object.
}

```

4.4.3 Returning vector metadata

The procedure for returning vector metadata is similar to that for returning scalar metadata. In fact, if you change the object type that you create from `avtScalarMetaData` to `avtVectorMetaData` then you are almost done. After you set the basic vector metadata attributes, you must set the `varDim` member to 2 if you have a 2-component vector or 3 if you have a 3-component vector.

Listing 4-18: `vectormetadata.C`: C++-Language example for returning vector metadata.

```

// NOTE - This code incomplete and is for example purposes only.

void
avtXXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a mesh called "mesh" to the metadata object.

    // Add a vector to the metadata. Note that this plug-in will
    // always expose a vector called "velocity" to VisIt. A real
    // plug-in may want to read a list of vectors from the data
    // file.
    avtVectorMetaData *vmd = new avtVectorMetaData;
    vmd->name = "velocity";
    vmd->meshName = "mesh";
    vmd->centering = AVT_ZONECENT;
    vmd->hasUnits = true;
    vmd->units = "m/s";
    vmd->varDim = 3;
    md->Add(vmd);

    // Add other objects to the metadata object.
}

```

4.4.4 Returning material metadata

Like the other types of mesh variables that we have seen so far, a material is defined on a specific mesh. However, unlike the other variables types, materials can be used to name regions of the mesh and can also be used by VisIt to break the mesh down into smaller pieces that can be turned on and off using the **Subset** window. Material metadata is stored in an `avtMaterialMetaData` object and it consists of: the name of the material object, the mesh on which it is defined, the number of materials, and the names of the materials. If you had a material called “mat1” defined on “mesh” and “mat1” was composed of: “Steel”, “Wood”, “Glue”, and “Air” then the metadata object needed to expose “mat1” to VisIt would look like the following code listing:

Listing 4-19: materialmetadata.C: C++-Language example for returning material metadata.

```
// NOTE - This code incomplete and is for example purposes only.

void
avtXXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a mesh called "mesh" to the metadata object.

    // Add a material to the metadata. Note that this plug-in will
    // always expose a material called "mat1" to VisIt. A real
    // plug-in may want to use from the data file to construct
    // a material.
    avtMaterialMetaData *matmd = new avtMaterialMetaData;
    matmd->name = "mat1";
    matmd->meshName = "mesh";
    matmd->numMaterials = 4;
    matmd->materialNames.push_back("Steel");
    matmd->materialNames.push_back("Wood");
    matmd->materialNames.push_back("Glue");
    matmd->materialNames.push_back("Air");
    md->Add(matmd);

    // Add other objects to the metadata object.
}
```

4.4.5 Returning expressions

VisIt provides support for defining expressions to calculate new data based on the data in your file. VisIt provides the **Expression** window in the GUI for managing expression definitions. It can be convenient for users in certain fields, where custom expressions are used frequently, to store the expression definitions directly in the file format or to encode the custom expressions directly in the file metadata so they are always available when a given file is visualized. VisIt’s `avtDatabaseMetaData` object can contain custom expressions. Thus you can add custom expressions to the `avtDatabaseMetaData` object inside of your database reader plug-in. Custom expressions are added to the

avtDatabaseMetaData object by creating `Expression` (defined in `Expression.h`) objects and adding them by calling the `avtDatabaseMetaData::AddExpression` method. The `Expression` object lets you provide the name and definition of an expression as well as the expression's expected return type (scalar, vector, tensor, etc.) and whether the expression should be hidden from the user. Hidden expressions can be useful if you build a complex expression that makes use of smaller sub-expressions that do not need to be exposed in the VisIt user interface.

Listing 4-20: expressionmetadata.C: C++-Language example for returning expression metadata.

```
// NOTE - This code incomplete and is for example purposes only.

#include <Expression.h>

void
avtXXXXXFileFormat::PopulateDatabaseMetaData(avtDatabaseMetaData *md)
{
    // Add a mesh called "mesh" to the metadata object.

    // Add scalars to the metadata object.

    // Add expression definitions to the metadata object.
    Expression *e0 = new Expression;
    e0->SetName("speed");
    e0->SetDefinition("{u,v,w}");
    e0->SetType(Expression::VectorMeshVar);
    e0->SetHidden(false);
    md->AddExpression(e0);

    Expression *e1 = new Expression;
    e1->SetName("density");
    e1->SetDefinition("mass/volume");
    e1->SetType(Expression::ScalarMeshVar);
    e1->SetHidden(false);
    md->AddExpression(e1);

    // Add other objects to the metadata object.
}
```

4.5 Returning a mesh

Once your database reader plug-in can successfully return metadata about one or more meshes, you can proceed to implementing your plug-in's `GetMesh` method. When you make a plot in VisIt, the plot is set up using the file metadata returned by your plug-in. When you click the **Draw** button in the VisIt GUI, it causes a series of requests that make the compute engine load your `libE` plug-in and call its `GetMesh` method with the name of the mesh being used by the plot as well as the time state and domain numbers (MT or MD formats only).

A database reader plug-in's job is to read relevant data from a file format and translate the data into a VTK object that VisIt can process. The `GetMesh` method's job is to read the mesh information from the file and create a VTK object that describes the mesh in the data file. VisIt can process many different mesh types (See Figure 4-15 on page 105) and you can return different types of VTK objects that best describe your mesh type. This section gives example code to show how you would take data read from your file format and turn it into VTK objects that describe your mesh. The details of reading data from your file format are omitted from the example code listings because those details change for each file format. The central message in this section is how to use data from a file format to construct different mesh types.

4.5.1 Determining which mesh to return

The `GetMesh` method is always passed a string containing the name of the mesh that should be returned from the plug-in. If your file format only ever has one mesh then you can ignore the `meshname` argument. However, if your file format can contain more than one mesh then you should check the name of the requested mesh before returning a VTK object so you create and return the correct mesh.

Listing 4-21: `getmesh1.C`: C++ Language example for which mesh to return in `GetMesh`.

```
// NOTE - This code incomplete and is for example purposes only.

#include <InvalidVariableException.h>

vtkDataSet *
avtXXXXFileFormat::GetMesh(const char *meshname)
{
    // Determine which mesh to return.
    if (strcmp(meshname, "mesh") == 0)
    {
        // Create a VTK object for "mesh"
        return mesh;
    }
    else if (strcmp(meshname, "mesh2") == 0)
    {
        // Create a VTK object for "mesh2"
        return mesh2;
    }
    else
    {
        // No mesh name that we recognize.
        EXCEPTION1(InvalidVariableException, meshname);
    }

    return 0;
}
```

If your database reader plug-in is derived from one of the MT or MD file format interfaces then the `GetMesh` method will have, in addition to the `meshname` argument, either a `timestep` argument, `domain` argument, or both. These extra arguments are both integers that VisIt passes to your plug-in so your plug-in can select the right mesh for the specified time state or domain. If your `GetMesh` method accepts a `timestep` argument then you can use it to return the mesh for the specified time state, which is in the range $[0, \text{NTS} - 1]$, where `NTS` is the number of time states that your plug-in returned from its `GetNTimesteps` method. The range for the `domain` argument, if it is present, is $[0, \text{NDOMS} - 1]$ where `NDOMS` is the number of domains that your file format added to the `numBlocks` member in the `avtMeshMetaData` object corresponding to the mesh named by the `meshname` argument.

4.5.2 Rectilinear meshes

A rectilinear mesh is a 2D or 3D mesh where all coordinates are aligned with the axes. Each axis of the rectilinear mesh can have different, non-uniform spacing, allowing for details to be concentrated in certain regions of the mesh. Rectilinear meshes are specified by lists of coordinate values for each axis. Since the mesh is aligned to the axes, it is only necessary to specify one set of X, Y, and Z values to generate all of the coordinates for the entire mesh.

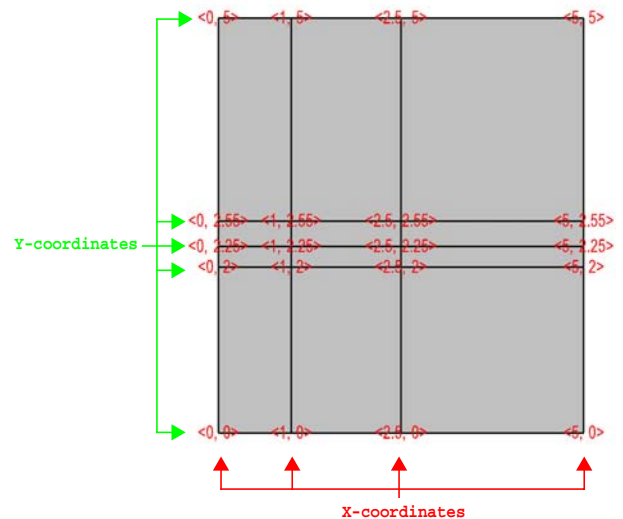


Figure 4-22: Rectilinear mesh and its X,Y node coordinates.

Once you read the X,Y, and Z coordinates from your data file, you can use them to assemble a `vtkRectilinearGrid` object. The procedure for creating a `vtkRectilinearGrid` object and returning it from `GetMesh` is shown in the next code listing. The underlined portions of the code listing indicate incomplete code that you must replace with code to read values from your file format. The first such piece requires you to read the number of dimensions for your mesh from the file format and store the value into the `ndims` variable. Once you have done that, read the number of nodes in each of the X,Y,Z dimensions and store those values in the `dims` array. Finally, fill in the code for reading the X coordinate values into the `xarray` array and do the same for the Y and Z coordinate arrays. Once you have replaced the underlined code portions with code that reads values from your file format, your plug-in should be able to return a valid `vtkRectilinearGrid` object once you rebuild it.

Listing 4-23: `getmesh_rect.C`: C++ Language example for creating `vtkRectilinearGrid` in `GetMesh`.

// NOTE - This code incomplete and requires underlined portions

```

// to be replaced with code to read values from your file format.

#include <vtkFloatArray.h>
#include <vtkRectilinearGrid.h>

vtkDataSet *
avtXXXFileFormat::GetMesh(const char *meshname)
{
    int ndims = 2;
    int dims[3] = {1,1,1};
    vtkFloatArray *coords[3] = {0,0,0};

    // Read the ndims and number of X,Y,Z nodes from file.
    ndims = NUMBER OF MESH DIMENSIONS;
    dims[0] = NUMBER OF NODES IN X-DIMENSION;
    dims[1] = NUMBER OF NODES IN Y-DIMENSION;
    dims[2] = NUMBER OF NODES IN Z-DIMENSION, OR 1 IF 2D;

    // Read the X coordinates from the file.
    coords[0] = vtkFloatArray::New();
    coords[0]->SetNumberOfTuples(dims[0]);
    float *xarray = (float *)coords[0]->GetVoidPointer(0);
    READ dims[0] FLOAT VALUES INTO xarray

    // Read the Y coordinates from the file.
    coords[1] = vtkFloatArray::New();
    coords[1]->SetNumberOfTuples(dims[1]);
    float *yarray = (float *)coords[1]->GetVoidPointer(0);
    READ dims[1] FLOAT VALUES INTO yarray

    // Read the Z coordinates from the file.
    coords[2] = vtkFloatArray::New();
    if(ndims > 2)
    {
        coords[2]->SetNumberOfTuples(dims[2]);
        float *zarray = (float *)coords[2]->GetVoidPointer(0);
        READ dims[2] FLOAT VALUES INTO zarray
    }
    else
    {
        coords[2]->SetNumberOfTuples(1);
        coords[2]->SetComponent(0, 0, 0.);
    }

    //
    // Create the vtkRectilinearGrid object and set its dimensions
    // and coordinates.
    //
    vtkRectilinearGrid *rgrid = vtkRectilinearGrid::New();
    rgrid->SetDimensions(dims);
    rgrid->SetXCoordinates(coords[0]);
    coords[0]->Delete();
    rgrid->SetYCoordinates(coords[1]);
    coords[1]->Delete();

```



```

rgrid->SetZCoordinates(coords[2]);
coords[2]->Delete();

return rgrid;
}

```

4.5.3 Curvilinear meshes

Curvilinear meshes are structured meshes as are rectilinear meshes. Whereas in a rectilinear mesh, a small set of independent X,Y,Z coordinate arrays are used to generate the coordinate values for each node in the mesh, in a curvilinear mesh, the node coordinates are explicitly given for each node in the mesh. This means that the sizes of the X,Y,Z coordinate arrays in a curvilinear mesh are all $NX*NY*NZ$ where NX is the number of nodes in the X-dimension, NY is the number of nodes in the Y-dimension, and NZ is the number of nodes in the Z-dimension. Providing the coordinates for every node permits you to create more complex geometries than are possible using rectilinear meshes (See Figure 4-24).

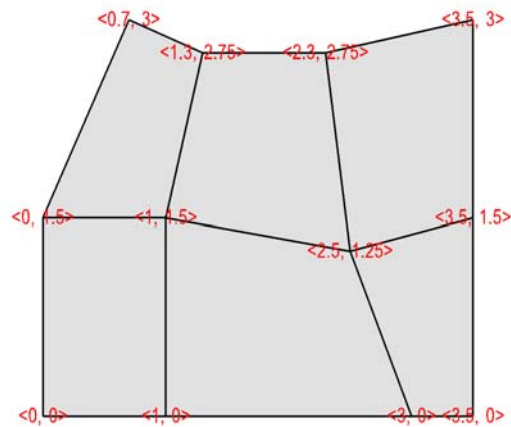


Figure 4-24: Curvilinear mesh and its X,Y node coordinates

Curvilinear meshes are created using the `vtkStructuredGrid` class. The next code listing shows how to create a `vtkStructuredGrid` object once you have read the required information from your file format. The underlined portions of the code listing indicate incomplete code that you will need to replace with code that can read data from your file format. First, read the number of dimensions for your mesh from the file format and store the value into the `ndims` variable. Once you have done that, read the number of nodes in each of the X,Y,Z dimensions and store those values in the `dims` array. Finally, fill in the code for reading the X coordinate values into the `xarray` array and do the same for the Y and Z coordinate arrays. Once you have replaced the underlined code portions with code that reads values from your file format, your plug-in should be able to return a valid `vtkStructuredGrid` object once you rebuild it

Listing 4-25: `getmesh_curv.C`: C++ Language example for creating `vtkStructuredGrid` in `GetMesh`.

```

// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

#include <vtkPoints.h>
#include <vtkStructuredGrid.h>

```

```

vtkDataSet *
avtXXXFileFormat::GetMesh(const char *meshname)
{
    int ndims = 2;
    int dims[3] = {1,1,1};

    // Read the ndims and number of X,Y,Z nodes from file.
    ndims = NUMBER OF MESH DIMENSIONS;
    dims[0] = NUMBER OF NODES IN X-DIMENSION;
    dims[1] = NUMBER OF NODES IN Y-DIMENSION;
    dims[2] = NUMBER OF NODES IN Z-DIMENSION, OR 1 IF 2D;
    int nnodes = dims[0]*dims[1]*dims[2];

    // Read the X coordinates from the file.
    float *xarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO xarray

    // Read the Y coordinates from the file.
    float *yarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO yarray

    // Read the Z coordinates from the file.
    float *zarray = 0;
    if(ndims > 2)
    {
        zarray = new float[nnodes];
        READ dims[2] FLOAT VALUES INTO zarray
    }

    //
    // Create the vtkStructuredGrid and vtkPoints objects.
    //
    vtkStructuredGrid *sgrid = vtkStructuredGrid::New();
    vtkPoints          *points = vtkPoints::New();
    sgrid->SetPoints(points);
    sgrid->SetDimensions(dims);
    points->Delete();
    points->SetNumberOfPoints(nnodes);

    //
    // Copy the coordinate values into the vtkPoints object.
    //
    float *pts = (float *) points->GetVoidPointer(0);
    float *xc = xarray;
    float *yc = yarray;
    float *zc = zarray;
    if(ndims == 3)
    {
        for(int k = 0; k < dims[2]; ++k)
            for(int j = 0; j < dims[1]; ++j)
                for(int i = 0; i < dims[0]; ++i)
                {
                    *pts++ = *xc++;
                }
    }
}

```

```

        *pts++ = *yc++;
        *pts++ = *zc++;
    }
}
else if(ndims == 2)
{
    for(int j = 0; j < dims[1]; ++j)
    for(int i = 0; i < dims[0]; ++i)
    {
        *pts++ = *xc++;
        *pts++ = *yc++;
        *pts++ = 0.;
    }
}

// Delete temporary arrays.
delete [] xarray;
delete [] yarray;
delete [] zarray;

return sgrid;
}

```

4.5.4 Point meshes

Point meshes are collections of particle positions that can be displayed in VisIt as points or small glyphed icons. Point meshes can be returned from the `GetMesh` method as `vtkUnstructuredGrid` objects that contain the locations of the points and connectivity composed entirely of vertex cells.

The next code listing shows how to create a `vtkUnstructuredGrid` object once you have read the required information from your file format. The underlined portions of the code listing indicate incomplete code that you will need to replace with code that can read data from your file format. First, read the number of dimensions for your mesh from the file format and store the value into the `ndims` variable. Next, read the number of points that make up the point mesh into the `nnodes` variable. Finally, fill in the code for reading the X coordinate values into the `xarray` array and do the same for the Y and Z coordinate arrays. Once you have replaced the underlined code portions with

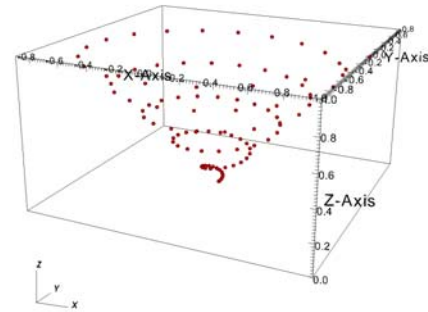


Figure 4-26: 3D point mesh

code that reads values from your file format, your plug-in should be able to return a valid `vtkUnstructuredGrid` object once you rebuild it.

Listing 4-27: `getmesh_point.C`: C++ Language example for returning a point mesh from `GetMesh`.

```
// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

#include <vtkPoints.h>
#include <vtkUnstructuredGrid.h>

vtkDataSet *
avtXXXFileFormat::GetMesh(const char *meshname)
{
    int ndims = 2;
    int nnodes;

    // Read the ndims and number of nodes from file.
    ndims = NUMBER OF MESH DIMENSIONS;
    nnodes = NUMBER OF NODES IN THE MESH;

    // Read the X coordinates from the file.
    float *xarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO xarray

    // Read the Y coordinates from the file.
    float *yarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO yarray

    // Read the Z coordinates from the file.
    float *zarray = 0;
    if(ndims > 2)
    {
        zarray = new float[nnodes];
        READ dims[2] FLOAT VALUES INTO zarray
    }

    //
    // Create the vtkPoints object and copy points into it.
    //
    vtkPoints *points = vtkPoints::New();
    points->SetNumberOfPoints(nnodes);
    float *pts = (float *) points->GetVoidPointer(0);
    float *xc = xarray;
    float *yc = yarray;
    float *zc = zarray;
    if(ndims == 3)
    {
        for(int i = 0; i < nnodes; ++i)
        {
            *pts++ = *xc++;
            *pts++ = *yc++;
        }
    }
}
```

```
        *pts++ = *zc++;
    }
}
else if(ndims == 2)
{
    for(int i = 0; i < nnodes; ++i)
    {
        *pts++ = *xc++;
        *pts++ = *yc++;
        *pts++ = 0.;
    }
}

//
// Create a vtkUnstructuredGrid to contain the point cells.
//
vtkUnstructuredGrid *ugrid = vtkUnstructuredGrid::New();
ugrid->SetPoints(points);
points->Delete();
ugrid->Allocate(nnodes);
vtkIdType onevertex;
for(int i = 0; i < nnodes; ++i)
{
    onevertex = i;
    ugrid->InsertNextCell(VTK_VERTEX, 1, &onevertex);
}

// Delete temporary arrays.
delete [] xarray;
delete [] yarray;
delete [] zarray;

return ugrid;
}
```

4.5.5 Unstructured meshes

Unstructured meshes are collections of cells of various geometries that are specified using indices into an array of points. When you write your `GetMesh` method, if your mesh is best described as an unstructured mesh then you can return a `vtkUnstructuredGrid` object.

Like some of the other mesh objects, the `vtkUnstructuredGrid` object also uses a `vtkPoints` object to contain its node array. In addition to the `vtkPoints` array, the `vtkUnstructuredGrid` object maintains a list of cells whose connectivity is determined by setting the cell type to one of VTK's predefined unstructured cell types (`VTK_VERTEX`, `VTK_LINE`, `VTK_TRIANGLE`, `VTK_QUAD`, `VTK_TETRA`, `VTK_PYRAMID`, `VTK_WEDGE`, and `VTK_HEXAHEDRON`), shown in Figure 4-29. When you add a cell using one of the predefined unstructured cell types, you must also provide a list of node indices that are used as the nodes for the cell. The number of nodes that each cell contains is determined by its cell type.

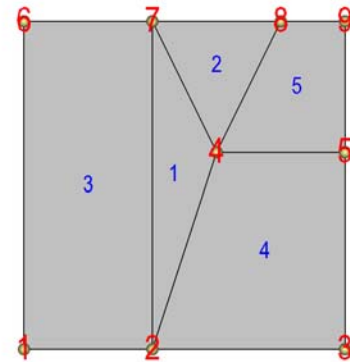


Figure 4-28: 2D unstructured mesh composed of triangles and quadrilaterals. The node numbers are labelled red and the cell numbers are labelled blue.

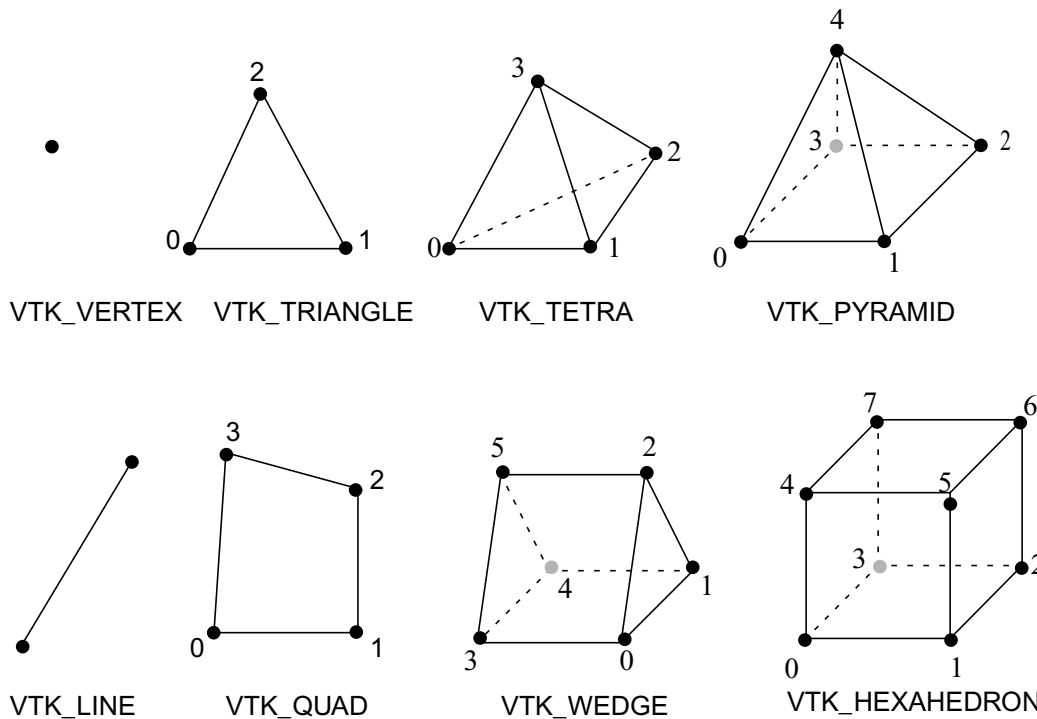


Figure 4-29: Node ordering for some VTK unstructured cell types

The next code listing shows how to create a `vtkUnstructuredGrid` object. The connectivity for an unstructured grid can be stored in a file format using a myriad of different approaches. The example code assumes that the connectivity will be stored in an integer array that contains the information for each cell, beginning with the cell type for the first cell, followed by a list of node indices that are used in the cell. After that, the cell type for the second cell appears, followed by its node indices, and so on. For example, if you wanted to store connectivity for cells 1 and 2 in the example shown in Figure 4-28 then the connectivity array would contain: `[VTK_TRIANGLE, 2, 4, 7, VTK_TRIANGLE, 4, 8, 7, ...]`. Note that the node indices in the example begin at one so the example code will subtract one from all of the node indices to ensure that they begin at zero, the starting index for the `vtkPoints` array.

Listing 4-30: `getmesh_ugrid.C`: C++ Language example for returning an unstructured mesh from `GetMesh`.

```
// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

#include <vtkPoints.h>
#include <vtkUnstructuredGrid.h>
#include <InvalidVariableException.h>

vtkDataSet *
avtXXXFileFormat::GetMesh(const char *meshname)
{
    int ndims = 2;
    int nnodes, ncells, origin = 1;

    // Read the ndims, nnodes, ncells, origin from file.
    ndims = NUMBER OF MESH DIMENSIONS;
    nnodes = NUMBER OF NODES IN THE MESH;
    ncells = NUMBER OF CELLS IN THE MESH;
    origin = GET THE ARRAY ORIGIN (0 or 1);

    // Read the X coordinates from the file.
    float *xarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO xarray

    // Read the Y coordinates from the file.
    float *yarray = new float[nnodes];
    READ nnodes FLOAT VALUES INTO yarray

    // Read the Z coordinates from the file.
    float *zarray = 0;
    if(ndims > 2)
    {
        zarray = new float[nnodes];
        READ dims[2] FLOAT VALUES INTO zarray
    }

    // Read in the connectivity array. This example assumes that
```

```

// the connectivity will be stored: type, indices, type,
// indices, ... and that there will be a type/index list
// pair for each cell in the mesh.
int *connectivity = 0;
ALLOCATE connectivity ARRAY AND READ VALUES INTO IT.

//
// Create the vtkPoints object and copy points into it.
//
vtkPoints *points = vtkPoints::New();
points->SetNumberOfPoints(nnodes);
float *pts = (float *) points->GetVoidPointer(0);
float *xc = xarray;
float *yc = yarray;
float *zc = zarray;
if(ndims == 3)
{
    for(int i = 0; i < nnodes; ++i)
    {
        *pts++ = *xc++;
        *pts++ = *yc++;
        *pts++ = *zc++;
    }
}
else if(ndims == 2)
{
    for(int i = 0; i < nnodes; ++i)
    {
        *pts++ = *xc++;
        *pts++ = *yc++;
        *pts++ = 0.;
    }
}

// Delete temporary arrays.
delete [] xarray;
delete [] yarray;
delete [] zarray;

//
// Create a vtkUnstructuredGrid to contain the point cells.
//
vtkUnstructuredGrid *ugrid = vtkUnstructuredGrid::New();
ugrid->SetPoints(points);
points->Delete();
ugrid->Allocate(ncells);
vtkIdType verts[8];
int *conn = connectivity
for(int i = 0; i < ncells; ++i)
{
    int fileCellType = *conn++;
    // You file's cellType will likely not match so you
    // will have to translate fileCellType to a VTK
    // cell type.
}

```



```

int cellType = MAP fileCellType TO VTK CELL TYPE.

// Determine number of vertices for each cell type.
if(cellType == VTK_VERTEX)
    nverts = 1;
else if(cellType == VTK_LINE)
    nverts = 2;
else if(cellType == VTK_TRIANGLE)
    nverts = 3;
else if(cellType == VTK_QUAD)
    nverts = 4;
else if(cellType == VTK_TETRA)
    nverts = 4;
else if(cellType == VTK_PYRAMID)
    nverts = 5;
else if(cellType == VTK_WEDGE)
    nverts = 6;
else if(cellType == VTK_HEXAHEDRON)
    nverts = 8;
else
{
    delete [] connectivity;
    ugrid->Delete();
    // Other cell type - need to add a case for it.
    // In the meantime, throw exception or if you
    // know enough, skip the cell.
    EXCEPTION0(InvalidVariableException, meshname);
}

// Make a list of node indices that make up the cell.
for(int j = 0; j < nverts; ++j)
    verts[j] = conn[j] - origin;
conn += nverts;

// Insert the cell into the mesh.
ugrid->InsertNextCell(cellType, nverts, verts);
}

delete [] connectivity;

return ugrid;
}

```

The previous code listing shows how to create an unstructured mesh in a `vtkUnstructuredGrid` object. The code listing contains underlined portions that you must replace with working code to read the relevant data from your file format. The first instance of code that must be replaced are the lines that read `ndims`, `nnodes`, `ncells`, and `origin` from the file format. The `ndims` variable should contain 2 or 3, depending on whether your data is 2D or 3D. The `nnodes` variable should contain the number of nodes that are used in the set of vertices that describe your unstructured mesh. The `ncells` variable should contain the number of cells that will be added to your

unstructured mesh. The `origin` variable should contain 0 or 1, depending on whether your connectivity indices begin at 0 or 1. Once you have set those variables to the appropriate values, you must read in the X,Y, and Z coordinate arrays from the file format and store the values into the `xarray`, `yarray`, and `zarray` array variables. If your file format keeps X,Y,Z values together in a single array then you may be able to read the coordinate values directly into the `vtkPoint` object's memory, skipping the step of copying the X,Y,Z coordinate components into the `vtkPoint` object.

After reading in the coordinate values from your file format, unstructured meshes require two more changes to the code in the listing. The next change requires you to allocate memory for a `connectivity` array, which stores the type of cells and the nodes indices of the nodes that are used in the cells. The final change that you must make to the source code in the listing is located further down in the loop that adds cells to the `vtkUnstructuredGrid` object. The cell type read from your file format will most likely not use the same enumerated type values that VTK uses for its cell types (`VTK_VERTEX`, `VTK_LINE`, ...) so you will need to add code to translate from your cell type designation to VTK cell type numbers. After making the necessary changes and rebuilding your plug-in, your plug-in's `GetMesh` method should be capable of returning a valid `vtkUnstructuredGrid` object for VisIt to plot.

4.6 Returning a scalar variable

Now that you can successfully create a Mesh plot of the meshes from your file format, you can focus on other types of data such as scalars. If you exposed scalar variables in your plug-in's `PopulateDatabaseMetaData` method then those variable names will appear in the plot menus for plots that can use scalar variables (e.g. the Pseudocolor plot). When you create a plot of a scalar variable and click the **Draw** button in the VisIt GUI, VisIt will tell your database reader plug-in to open your file, read the mesh, and then your plug-in's `GetVar` method will be called with the name of the variable that you want to plot. The `GetVar` method, like the `GetMesh` method, takes a variable name as an argument. When you receive the variable name in the `GetVar` method you should access your file and read out the desired variable and return it in a VTK data array such as a `vtkFloatArray` or a `vtkDoubleArray`. A `vtkFloatArray` is a VTK object that encapsulates a dynamically allocated array of a given length. The length of the array that you allocate to contain your variable must match either the number of cells in your mesh or the number of nodes in your mesh. The length is determined by the scalar variable's centering (cell-centered, node-centered).

Listing 4-31: `getvar.C`: C++ Language example for returning data from `GetVar`.

```
// NOTE - This code incomplete and requires underlined portions  
// to be replaced with code to read values from your file format.  
  
#include <vtkFloatArray.h>  
  
vtkDataArray *
```

```

avtXXXFileFormat::GetVar(const char *varname)
{
    int nvals;
    // Read the number of vaues contained in the array
    // specified by varname.
    nvals = NUMBER OF VALUES IN ARRAY NAMED BY varname;

    // Allocate the return vtkFloatArray object. Note that
    // you can use vtkFloatArray, vtkDoubleArray,
    // vtkUnsignedCharArray, vtkIntArray, etc.
    vtkFloatArray *arr = vtkFloatArray::New();
    arr->SetNumberOfTuples(nvals);
    float *data = (float *)arr->GetVoidPointer(0);
    READ nvals FLOAT NUMBERS INTO THE data ARRAY.

    return arr;
}

```

In the previous code listing, there are two underlined areas that need to have code added to them in order to have a completed `GetVar` method. The first change that you must make is to add code to read the size of the array to be created into the `nvals` variable. The value that is read into the `nvals` variable must be either the number of cells in the mesh on which the variable is defined if you have a cell-centered variable or it must be the number of nodes in the mesh. Once you have successfully set the proper value into the `nvals` variable, you can proceed to read values from your file format into the data array, which points to storage owned by the `vtkFloatArray` object that will be returned from the `GetVar` method. Once you have made these changes, you can rebuilt your plug-in and begin plotting scalar variables.

4.7 Returning a vector variable

If you exposed vector variables in your plug-in's `PopulateDatabaseMetaData` method then those variable names will appear in the plot menus for plots that can use vector variables (e.g. the Vector plot). When you create a plot of a vector variable and click the **Draw** button in the VisIt GUI, VisIt will tell your database reader plug-in to open your file, read the mesh, and then your plug-in's `GetVectorVar` method will be called with the name of the variable that you want to plot. The `GetVectorVar` method, like the `GetMesh` method, takes a variable name as an argument. When you receive the variable name in the `GetVectorVar` method you should access your file and read out the desired variable and return it in a VTK data array such as a `vtkFloatArray` or a `vtkDoubleArray`. A `vtkFloatArray` is a VTK object that encapsulates a dynamically allocated array of a given length. The length of the array that you allocate to contain your variable must match either the number of cells in your mesh or the number of nodes in your mesh. The length is determined by the scalar variable's centering (cell-centered, node-centered). In addition to setting the length, which like a scalar variable is tied to the number of cells or nodes, you must also set the number of vector components.

In `Visit`, vector variables always have three components. If the third component is not needed then all values in the third component should be set to zero.

The `GetVectorVar` code listing shows how to return a `vtkFloatArray` with multiple components from the `GetVectorVar` method. As with the code listing for `GetVar`, this code listing requires you to replace underlined lines of code with code that reads data from your file format and stores the results in the variables provided.

Listing 4-32: `getvectorvar.C`: C++ Language example for returning data from `GetVectorVar`.

```
// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

#include <vtkFloatArray.h>
#include <InvalidVariableException.h>

vtkDataArray *
avtXXXFileFormat::GetVectorVar(const char *varname)
{
    int nvals, ncomps = 3;

    // Read the number of values contained in the array
    // specified by varname.
    nvals = NUMBER OF VALUES IN ARRAY NAMED BY varname;
    ncomps = NUMBER OF VECTOR COMPONENTS IN ARRAY NAMED BY varname;

    // Read component 1 from the file.
    float *comp1 = new float[nvals];
    READ nvals FLOAT VALUES INTO comp1

    // Read component 2 from the file.
    float *comp2 = new float[nvals];
    READ nvals FLOAT VALUES INTO comp2

    // Read component 3 from the file.
    float *comp3 = 0;
    if(ncomps > 2)
    {
        comp3 = new float[nvals];
        READ nvals FLOAT VALUES INTO comp3
    }

    // Allocate the return vtkFloatArray object. Note that
    // you can use vtkFloatArray, vtkDoubleArray,
    // vtkUnsignedCharArray, vtkIntArray, etc.
    vtkFloatArray *arr = vtkFloatArray::New();
    arr->SetNumberOfComponents(3);
    arr->SetNumberOfTuples(nvals);
    float *data = (float *)arr->GetVoidPointer(0);
    float *c1 = comp1;
    float *c2 = comp2;
    float *c3 = comp3;
```

```

if(ncomps == 3)
{
    for(int i = 0; i < nvals; ++i)
    {
        *data++ = *c1++;
        *data++ = *c2++;
        *data++ = *c3++;
    }
}
else if(ncomps == 2)
{
    for(int i = 0; i < nvals; ++i)
    {
        *data++ = *c1++;
        *data++ = *c2++;
        *data++ = 0.;
    }
}
else
{
    delete [] comp1;
    delete [] comp2;
    delete [] comp3;
    arr->Delete();
    EXCEPTION1(InvalidVariableException, varname);
}

// Delete temporary arrays.
delete [] comp1;
delete [] comp2;
delete [] comp3;

return arr;
}

```

4.8 Using a VTK reader class

The implementations so far for the `GetMesh`, `GetVar`, and `GetVectorVar` plug-in methods have assumed that the database plug-in would do the work of interacting with the file format to read data into VTK form. Most of the work of reading a file and creating VTK objects from it can be handled at the VTK level if you wish. This means that it is possible to use an existing VTK reader class to read data into VisIt if you are willing to implement your plug-in methods so that they in turn call the VTK reader object's methods. See VisIt's VTK database reader plug-in for an example of how to call VTK reader objects from inside a VisIt database reader plug-in.

5.0 Advanced topics

If you've implemented your database reader plug-in using only the techniques outlined in this chapter so far then you likely have a database reader plug-in that works and correctly serves up its data to VisIt in VTK form. This part of the chapter explains some of the more advanced, though not necessarily required, techniques that you can use to enhance your plug-in. For instance, you can enhance your plug-in so it returns the correct simulation times from the data files. You can also add code to return data and spatial extents for your data, enabling VisIt to make more optimization decisions when processing files with multiple domains.

5.1 Returning cycles and times

Simulations often iterate for many thousands of cycles while they solve their systems of equations. Generally, each simulation cycle has an associated cycle number and time value. Many file formats save this information so it can be made available later to post-processing tools such as VisIt. VisIt uses cycles and times to help you navigate through time in your database by providing the same time frame of reference that your simulation used. VisIt's **File panel** can display times next to each time state in a database and can also show the current time value as you scroll through time using the time slider. Cycle and time values for the current time state are often displayed in the visualization window.

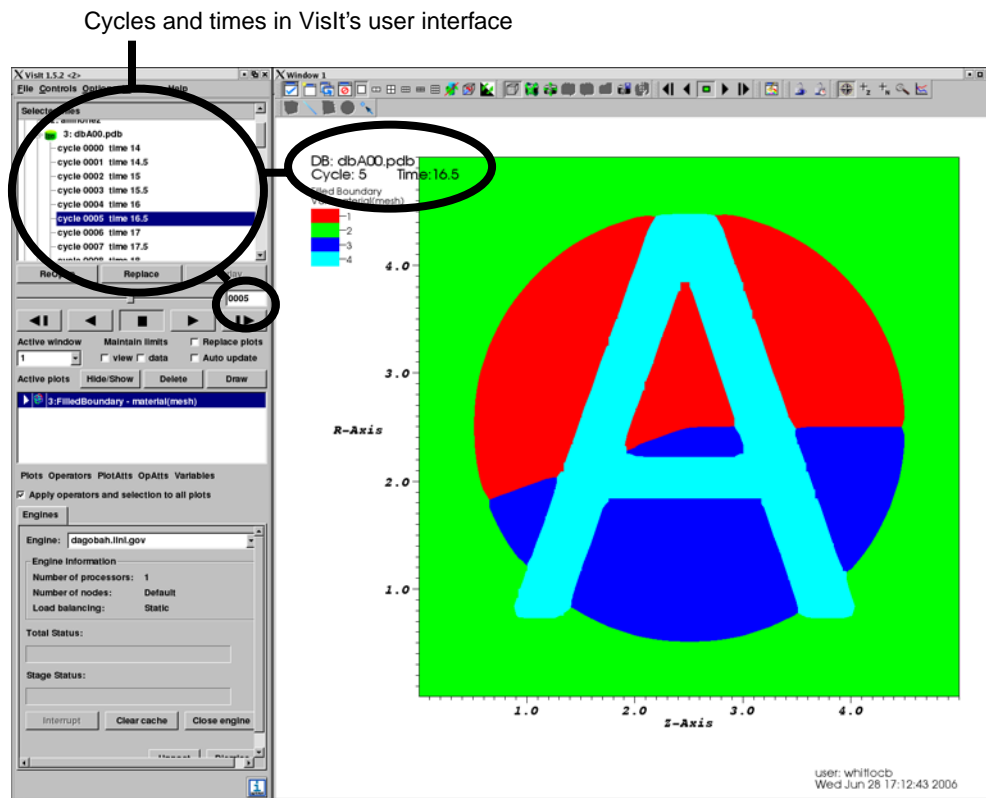


Figure 4-33: Cycles and times values are used to help you navigate through time

Returning cycle and time values from your plug-in is completely optional. In fact, returning cycle and time values for data such as CAD drawings does not make sense. Since returning cycles and times is optional in a VisIt database reader plug-in, you can choose to not implement the methods that return cycles and times. You can also implement code to return time but not cycles or vice-versa.

The mechanics of returning cycles and times are a little different depending on whether you have written an ST or an MT database reader plug-in. In any case, if your plug-in implements the methods to return cycles or times then those methods will be some of the first methods called when VisIt accesses your database reader plug-in. VisIt calls the methods to get cycles and times and if the returned values appear to be valid then they are added to the metadata for your file so they can be returned to the VisIt clients and used to populate windows such as the **File Information** window, shown in Figure 4-34.

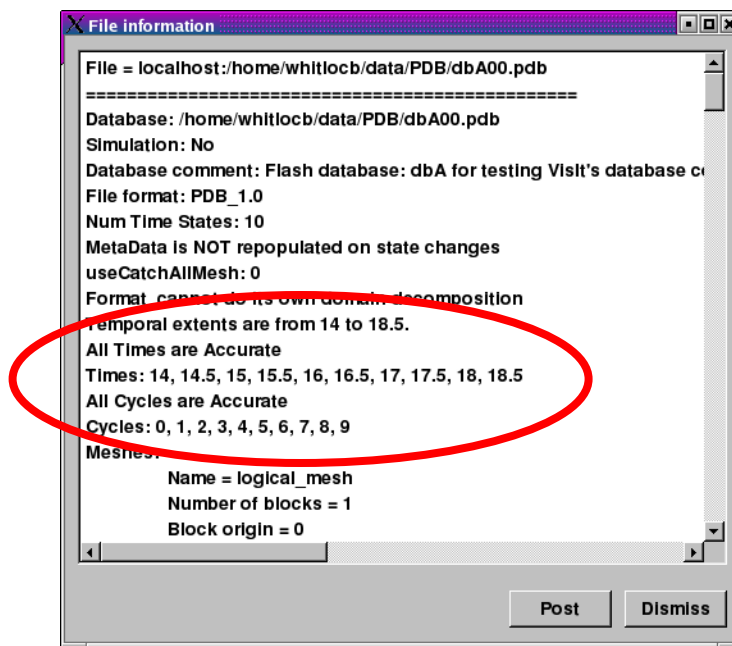


Figure 4-34: The File Information window can be used to inspect the cycles and times returned from your plug-in.

5.1.1 Returning cycles and times in an ST plug-in

When VisIt creates plug-in objects to handle a list of files using an ST plug-in, there is one plug-in object per file in the list of files. Since each plug-in object can only ever be associated with one file, the programming interface for returning cycles and times for an ST plug-in provides methods that return a single value. The methods for returning cycles and times for an ST plug-in are:

```
virtual bool ReturnsValidCycle() const { return true; }
virtual int GetCycle(void);
```

```

virtual bool    ReturnsValidTime() const { return true; }
virtual double  GetTime(void);

```

Implementing valid cycles and times can be done independently of one another and there is no requirement that you have to implement both or either of them, for that matter. The ReturnsValidCycle method is a simple method that you should expose if you plan to provide a custom GetCycle method in your database reader plug-in. If you provide GetCycle then the ReturnsValidCycle method should return true. The same pattern applies if you implement GetTime - except that you would also implement the ReturnsValidTime method. Replace the underlined sections of code in the listing with code to read the correct cycle and time values from your file format.

Listing 4-35: cycletime_st.C: C++ Language example for returning cycles, times from ST plug-in.

```

// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

```

```

int
avtXXXXFileFormat::GetCycle(void)
{
    int cycle = OPEN FILE AND READ THE CYCLE VALUE;
    return cycle;
}

double
avtXXXXFileFormat::GetTime(void)
{
    double dtime = OPEN FILE AND READ THE TIME VALUE;
    return dtime;
}

```

In the event that you implement the GetCycle method but no cycle value is available in the file, you can return the INVALID_CYCLE value to make VisIt discard your plug-in's cycle number and guess the cycle number from the filename. If you want VisIt to successfully guess the cycle number from the filename then you must implement the GetCycleFromFilename method.

```

int
avtXXXXFileFormat::GetCycleFromFilename(const char *f) const
{
    return GuessCycle(f);
}

```


5.1.2 Returning cycles and times in an MT plug-in

An MT database reader plug-in may return cycles and times for multiple time states so the programming interface for MT plug-ins allows you to return vectors of cycles and times. In addition, an MT database reader plug-in prefers to know upfront how many time states will be returned from the file format so in addition to `GetCycles` and `GetTimes` methods, there is a `GetNTimesteps` method that is among the first methods called from your database reader plug-in.

```
virtual void GetCycles(std::vector<int> &);
virtual void GetTimes(std::vector<double> &);
virtual int GetNTimesteps(void);
```

As with ST plug-ins, there is no requirement that an MT plug-in must provide a list of cycles or times. However, an MT plug-in must provide a `GetNTimesteps` method. If you are enhancing your database reader plug-in to return cycles and times then it is convenient to implement your `GetNTimesteps` method such that it just calls your `GetCycles` or `GetTimes` method and returns the length of the vector returned by those methods. This simplifies the implementation and ensures that the number of time states reported by your database reader plug-in matches the length of the cycle and time vectors returned from `GetCycles` and `GetTimes`. Replace the underlined sections of code in the listing with code to read the correct cycles and times from your file format.

Listing 4-36: `cycletime_mt.C`: C++ Language example for returning cycles, times from MT plug-in.

```
// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

void
avtXXXFileFormat::GetCycles(std::vector<int> &cycles)
{
    int ncycles, *vals = 0;
    ncycles = OPEN FILE AND READ THE NUMBER OF CYCLES;
    READ ncycles INTEGER VALUES INTO THE vals ARRAY;

    // Store the cycles in the vector.
    for(int i = 0; i < ncycles; ++i)
        cycles.push_back(vals[i]);

    delete [] vals;
}

void
avtXXXFileFormat::GetTime(std::vector<double> &times)
{
    int ntimes;
    double *vals = 0;
```

```

    ntimes = OPEN FILE AND READ THE NUMBER OF TIMES;
    READ ntimes DOUBLE VALUES INTO THE vals ARRAY;

    // Store the times in the vector.
    for(int i = 0; i < ntimes; ++i)
        times.push_back(vals[i]);

    delete [] vals;
}

int
avtXXXXFileFormat::GetNTimesteps(void)
{
    std::vector<double> times;
    GetTimes(times);
    return times.size();
}

```

5.2 Auxiliary data

This section describes how to enable your MD database reader plug-in so it can provide auxiliary data such as data extents, spatial extents, and materials to VisIt if they are available in your file format. “Auxiliary data”, is the generic term for many types of data that VisIt’s pipeline can use to perform specific tasks such as I/O reduction or material selection. VisIt’s database reader plug-in interfaces provide a method called `GetAuxiliaryData` that you can implement if you want your plug-in to be capable of returning auxiliary data. Note however that if your plug-in is MTMD then you will have to cache your spatial and data extents in the plug-in’s variable cache in the `PopulateDatabaseMetaData` method instead of returning that information from the `GetAuxiliaryData` method. This subtle difference in how certain metadata is accessed by VisIt must be observed by an MTMD plug-in in order for it to return spatial and data extents.

The method arguments for the `GetAuxiliaryData` method may vary somewhat depending on whether your database reader plug-in is based on the STSD, STMD, MTSD, MTMD interfaces. There is an extra integer argument for the time state if your plug-in is MT and there is another integer argument for the domain if your plug-in is MD. Those differences aside, the `GetAuxiliaryData` method always accepts the name of a variable, a string indicating the type of data being requested, a pointer to optional data required by the type of auxiliary data being requested, and a return reference for a destructor function that will be responsible for freeing resources for the returned data. The variable name that VisIt passes to the `GetAuxiliaryData` method is the name of a variable such as those passed to the `GetVar` method when VisIt wants to read a variable’s data.

5.2.1 Returning data extents

When an MD database reader plug-in provides data extents for each of its domains, VisIt has enough information to make important optimization decisions in filters that support data extents. For example, if you create a Contour plot using a specific contour value, VisIt can check the data extents for each domain before any domains are read from disk and determine the list of domains that contain the desired contour value. After determining which subset of the domains will contribute to the final image, VisIt's compute engine then reads and processes only those domains, saving work and accelerating VisIt's computations. For a more complete explanation of data extents, see "Writing data extents" on page 70.

In the context of returning data extents, VisIt first checks a plug-in's variable cache for extents. If the desired extents are not available then VisIt calls the plug-in's `GetAuxiliaryData` method with the name of the scalar variable for which data extents are required and also passes `AUXILIARY_DATA_DATA_EXTENTS` as the type argument, indicating that the `GetAuxiliaryData` method is being called to obtain the data extents for the specified scalar variable. If the data extents for the specified variable are not available then the `GetAuxiliaryData` method should return 0. If the data extents are available then the list of minimum and maximum values for the specified variable are assembled into an interval tree structure that VisIt uses for fast comparisons of different data ranges. Once the interval tree is constructed, as shown in the code listing, the `GetAuxiliaryData` method must return the interval tree object and set the destructor function argument to a function that can be called to later destroy the interval tree. To add support for data extents to your database reader plug-in, copy the `GetAuxiliaryData` method in the code listing and replace the underlined lines of code with code that reads the required information from your file format.

Listing 4-37: `dataextents.C`: C++ Language example for returning data extents.

```
// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

#include <avtIntervalTree.h>

// STMD version of GetAuxiliaryData.
void *
avtXXXXFileFormat::GetAuxiliaryData(const char *var,
    int domain, const char *type, void *,
    DestructorFunction &df)
{
    void *retval = 0;

    if(strcmp(type, AUXILIARY_DATA_DATA_EXTENTS) == 0)
    {
        // Read the number of domains for the mesh.
        int ndoms = READ NUMBER OF DOMAINS FROM FILE;
    }
}
```

```

// Read the min/max values for each domain of the
// "var" variable. This information should be in
// a single file and should be available without
// having to read the real data.
double *minvals = new double[ndoms];
double *maxvals = new double[ndoms];
READ ndoms DOUBLE VALUES INTO minvals ARRAY.
READ ndoms DOUBLE VALUES INTO maxvals ARRAY.

// Create an interval tree
avtIntervalTree *itree = new avtIntervalTree(ndoms, 1);
for(int dom = 0; dom < ndoms; ++dom)
{
    double range[2];
    range[0] = minvals[dom];
    range[1] = maxvals[dom];
    itree->AddElement(dom, range);
}
itree->Calculate(true);

// Delete temporary arrays.
delete [] minvals;
delete [] maxvals;

// Set return values
retval = (void *)itree;
df = avtIntervalTree::Destruct;
}

return retval;
}

```

5.2.2 Returning spatial extents

Another type of auxiliary data that VisIt supports for MD file formats are spatial extents. When VisIt knows the spatial extents for all of the domains that comprise a mesh, VisIt can optimize operations such as the Slice operator by first determining whether the slice will intersect a given domain. The Slice operator is thus able to use spatial extents to determine which set of domains must be read from disk and processed in order to produce the correct visualization. Spatial extents are used in this way by many filters to reduce the set of domains that must be processed.

When VisIt asks the database reader plug-in for spatial extents, the `GetAuxiliaryData` method is called with its type argument set to `AUXILIARY_DATA_SPATIAL_EXTENTS`. When VisIt creates spatial extents, they are stored in an interval tree structure as they are with data extents. The main difference is the input into the interval tree. When adding information about a specific domain to the interval tree, you must provide the minimum and maximum spatial values for the domain's X, Y, and Z dimensions. The spatial extents for one domain are expected to be provided in

the following order: xmin, xmax, ymin, ymax, zmin, zmax. To add support for spatial extents to your database reader plug-in, copy the `GetAuxiliaryData` method in the code listing and replace the underlined lines of code with code that reads the required information from your file format.

Listing 4-38: `spatialextents.C`: C++ Language example for returning spatial extents.

```
// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

#include <avtIntervalTree.h>

// STMD version of GetAuxiliaryData.
void *
avtXXXXFileFormat::GetAuxiliaryData(const char *var,
    int domain, const char *type, void *,
    DestructorFunction &df)
{
    void *retval = 0;

    if(strcmp(type, AUXILIARY_DATA_SPATIAL_EXTENTS) == 0)
    {
        // Read the number of domains for the mesh.
        int ndoms = READ NUMBER OF DOMAINS FROM FILE;

        // Read the spatial extents for each domain of the
        // mesh. This information should be in a single
        // and should be available without having to
        // read the real data. The expected format for
        // the data in the spatialextents array is to
        // repeat the following pattern for each domain:
        // xmin, xmax, ymin, ymax, zmin, zmax.
        double *spatialextents = new double[ndoms * 6];
        READ ndoms*6 DOUBLE VALUES INTO spatialextents ARRAY.

        // Create an interval tree
        avtIntervalTree *itree = new avtIntervalTree(ndoms, 3);
        double *extents = spatialextents;
        for(int dom = 0; dom < ndoms; ++dom)
        {
            itree->AddElement(dom, extents);
            extents += 6;
        }
        itree->Calculate(true);

        // Delete temporary array.
        delete [] spatialextents;

        // Set return values
        retval = (void *)itree;
        df = avtIntervalTree::Destruct;
    }
}
```

```

    return retval;
}

```

5.2.3 Returning materials

Materials are another type of auxiliary data that database plug-ins can provide. A material classifies different pieces of the mesh into different named subsets that can be turned on and off using VisIt's **Subset** window. In the simplest case, you can think of a material as a cell-centered variable, or matlist, defined on your mesh where each cell contains an integer that identifies a particular material such as "Steel" or "Air". VisIt's `avtMaterial` object is used to encapsulate knowledge about materials. The `avtMaterial` object contains the matlist array and a list of names corresponding to each unique material number in the matlist array. Materials can also be structured so that instead of providing just one material number for each cell in the mesh, you can provide multiple materials per cell with volume fractions occupied by each. So-called "mixed materials" are created using additional arrays, described in "Materials" on page 81. To add support for materials in your database reader plug-in's `GetAuxiliaryData` method, replace the underlined lines in the code example with code that read the necessary values from your file format.

Listing 4-39: `matclean.C`: C++ Language example for returning material data.

```

// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

#include <avtMaterial.h>

// STMD version of GetAuxiliaryData.
void *
avtXXXXFileFormat::GetAuxiliaryData(const char *var,
    int domain, const char *type, void *,
    DestructorFunction &df)
{
    void *retval = 0;

    if(strcmp(type, AUXILIARY_DATA_MATERIAL) == 0)
    {
        int dims[3] = {1,1,1}, ndims = 1;
        // Structured mesh case
        ndims = MESH_DIMENSION, 2 OR 3;
        dims[0] = NUMBER OF ZONES IN X DIMENSION;
        dims[1] = NUMBER OF ZONES IN Y DIMENSION;
        dims[2] = NUMBER OF ZONES IN Z DIMENSION, OR 1 IF 2D;

        // Unstructured mesh case
        dims[0] = NUMBER OF ZONES IN THE MESH
        ndims = 1;

        // Read the number of materials from the file. This

```

```

// must have already been read from the file when
// PopulateDatabaseMetaData was called.
int nmats = NUMBER OF MATERIALS;

// The matnos array contains the list of numbers that
// are associated with particular materials. For example,
// matnos[0] is the number that will be associated with
// the first material and any time it is seen in the
// matlist array, that number should be taken to mean
// material 1. The numbers in the matnos array must
// all be greater than or equal to 1.
int *matnos = new int[nmats];
READ nmats INTEGER VALUES INTO THE matnos ARRAY.

// Read the material names from your file format or
// make up names for the materials. Use the same
// approach as when you created material names in
// the PopulateDatabaseMetaData method.
char **names = new char *[nmats];
READ MATERIAL NAMES FROM YOUR FILE FORMAT UNTIL EACH
ELEMENT OF THE names ARRAY POINTS TO ITS OWN STRING.

// Read the matlist array, which tells what the material
// is for each zone in the mesh.
int nzones = dims[0] * dims[1] * dims[2];
int *matlist = new int[nzones];
READ nzones INTEGERS INTO THE matlist array.

// Optionally create mix_mat, mix_next, mix_zone, mix_vf
// arrays and read their contents from the file format.

// Use the information to create an avtMaterial object.
avtMaterial *mat = new avtMaterial(
    nmats,
    matnos,
    names,
    ndims,
    dims,
    0,
    matlist,
    0, // length of mix arrays
    0, // mix_mat array
    0, // mix_next array
    0, // mix_zone array
    0 // mix_vf array
);

// Clean up.
delete [] matlist;
delete [] matnos;
for(int i = 0; i < nmats; ++i)
    delete [] names[i];
delete [] names;

```

```

        // Set the return values.
        retval = (void *)mat;
        df = avtMaterial::Destruct;
    }

    return retval;
}

```

5.3 Returning ghost zones

Ghost zones are mesh zones that should not be visible in the visualization but may provide additional information such as values along domain boundaries. VisIt uses ghost zones for ensuring variable continuity across domain boundaries, for removing internal domain boundary faces, and for blanking out specific zones. This section covers the code that must be added to make your database reader plug-in in order for it to return ghost zones to VisIt.

5.3.1 Blanking out zones

Blanking out specific zones so they do not appear in a visualization is a common practice for creating holes in structured meshes so cells zones that overlap or tangle on top of one another can be removed from the mesh. If you want to create a mesh that contains voids where zones have been removed then you can add a special cell-centered array to your mesh before you return it from your plug-in's GetMesh method. The code in the listing can be used to remove zones from any mesh type and works by looking through a mesh-sized array containing on/off values for each zone and sets the appropriate values into the ghost zone array that gets added to the mesh object. Replace any underlined code with code that can read the necessary values from your file format.

Listing 4-40: gz_blank.C: C++ Language example for returning a mesh with blanked out zones.

```

// NOTE - This code incomplete and requires underlined portions
// to be replaced with code to read values from your file format.

#include <avtGhostData.h>
#include <vtkUnsignedCharArray.h>

vtkDataSet *
avtXXXXFileFormat::GetMesh(const char *meshname)
{
    // Code to create your mesh goes here.
    vtkDataSet *retval = CODE TO CREATE YOUR MESH;

    // Now that you have your mesh, figure out which cells need
    // to be removed.
    int nCells = retval->GetNumberOfCells();
    int *blanks = new int[nCells];
    READ nCells INTEGER VALUES INTO blanks ARRAY.
}

```



```

// Now that we have the blanks array, create avtGhostZones.
unsigned char realVal = 0, ghost = 0;
avtGhostData::AddGhostZoneType(ghost,
    ZONE_NOT_APPLICABLE_TO_PROBLEM);
vtkUnsignedCharArray *ghostCells = vtkUnsignedCharArray::New();
ghostCells->SetName("avtGhostZones");
ghostCells->Allocate(nCells);
for(int i = 0; i < nCells; ++i)
{
    if(blanks[i])
        ghostCells->InsertNextValue(realVal);
    else
        ghostCells->InsertNextValue(ghost);
}
retval->GetCellData()->AddArray(ghostCells);
retval->SetUpdateGhostLevel(0);
ghostCells->Delete();

// Clean up
delete [] blanks;

return retval;
}

```

5.3.2 Ghost zones at the domain boundaries

When ghost zones are used to ensure continuity across domains, an extra layer of zones must be added to the mesh boundaries where the boundary is shared with another domain. Once you have done that step, the approach for providing ghost zones is the same as for blanking out cells using ghost zones if your `blanks` array contains zeroes for only the zones that appear on domain boundaries. The one minor difference is that you must substitute the `DUPLICATED_ZONE_INTERNAL_TO_PROBLEM` ghost zone type for the `ZONE_NOT_APPLICABLE_TO_PROBLEM` ghost zone type in the code example.

5.4 Parallelizing your reader

VisIt is a distributed program made up of multiple software processes that act as a whole. The software process that reads in data and processes it is the compute engine, which comes in serial and parallel versions. All of the `libE` plug-ins in VisIt also have both serial and parallel versions. The parallel `libE` plug-ins can contain specialized MPI communication to support the communication patterns needed by the algorithms used. If you want to parallelize your database reader plug-in then, in most cases, you will have to use the MD interface or convert from SD to MD. There are some SD formats that can adaptively decompose their data so each processor has work (see the `ViSUS` plug-in) but most database plug-ins that benefit from parallelism instead are implemented as MD plug-ins. MD plug-ins are a natural fit for the parallel compute engine because they serve data that is already decomposed into domains. Some database reader plug-ins, such as the `BOV`

plug-in, take single domain meshes and automatically decompose them into multiple domains for faster processing on multiple processors.

Deriving your plug-in from an MD interface is useful since it naturally tells VisIt to expect data from more than one domain when reading your file format. There are a number of parallel optimizations that can be made inside of your MD database reader plug-in. For example, you might have one processor read the metadata and broadcast it to all other processors so when you visualize your data with a large number of processors, they are not all trying to read the file that contains the metadata.

VisIt's parallel compute engine can use one of two different load balancing schemes: static or dynamic. In static load balancing, each processor is assigned a fixed list of domains and each of those domains is processed one at a time in parallel visualization pipelines until the result is computed. When static load balancing is used, the same code is executed on all processors with different data and there are more opportunities for parallel, global communication. When VisIt's parallel compute engine uses dynamic load balancing, the master process acts as an executive that assigns work as needed to each processor. When a processor needs work, it requests a domain from the executive and it processes the domain in its visualization pipeline until the results for the domain have been calculated. After that, the processor asks the executive for another domain. In dynamic load balancing, each processor can be working on very different operations so there is no opportunity to do global communication. VisIt attempts to do dynamic load balancing unless any one of the filters in its visualization pipeline requires global communication, in which case static load balancing must be used. This means that the places where global communication can occur are few.

VisIt's database plug-in interfaces provide the `ActivateTimestep` method as a location where global, parallel communication can be performed safely. If your parallel database reader needs to do parallel communication such as broadcasting metadata to all processors, or figuring out data extents in parallel then that code must be added in the `ActivateTimestep` method.

Chapter 5

Instrumenting a simulation code

1.0 Overview

Some simulation programs include a runtime graphics package, which creates visualizations of simulation results during execution. Runtime graphics have a number of advantages over writing out graphics files that can be visualized after the fact by a visualization tool. First of all, graphics files are written far less frequently than the simulation calculates its data because of time and disk space limitations. Secondly, runtime graphics packages have access to all of the variables that a simulation calculates, whereas a graphics file usually contains a small subset of the variables. Finally, by using runtime graphics, users can visualize simulation results as the simulation executes and the user can possibly intercede to change how the simulation runs.

VisIt provides a library that can be used by simulation codes in order to expose data to VisIt, allowing you to use VisIt as a runtime graphics package. This chapter explains in detail the steps required to instrument your C or Fortran simulation so that VisIt can access its data for the purpose of runtime graphics.

2.0 Architecture

Parallel simulations often use a technique called domain decomposition (see Figure 5-1) to break up the simulated problem into smaller pieces called domains. We've learned in earlier chapters how to store data from different domains in a variety of file formats such as Silo and VTK. Simulations often write out 1 domain file per processor, and VisIt

processes all of the individual domain files to produce a unified picture with contributions from all of the relevant domains.

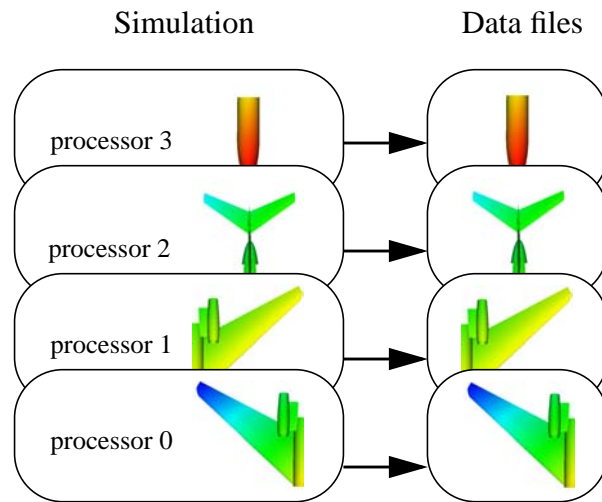


Figure 5-1: Simulation writing data files in parallel

VisIt has a distributed architecture which allows various functions to be grouped into cooperating processes. VisIt's compute engine is particularly relevant when discussing runtime graphics. The compute engine is responsible for reading data from files, generating plots from the data, and sending the plots to VisIt's viewer where the plot can be displayed. In short, VisIt's compute engine is the VisIt component that handles all of the data. Figure 5-2 depicts VisIt's compute engine reading data files in parallel.

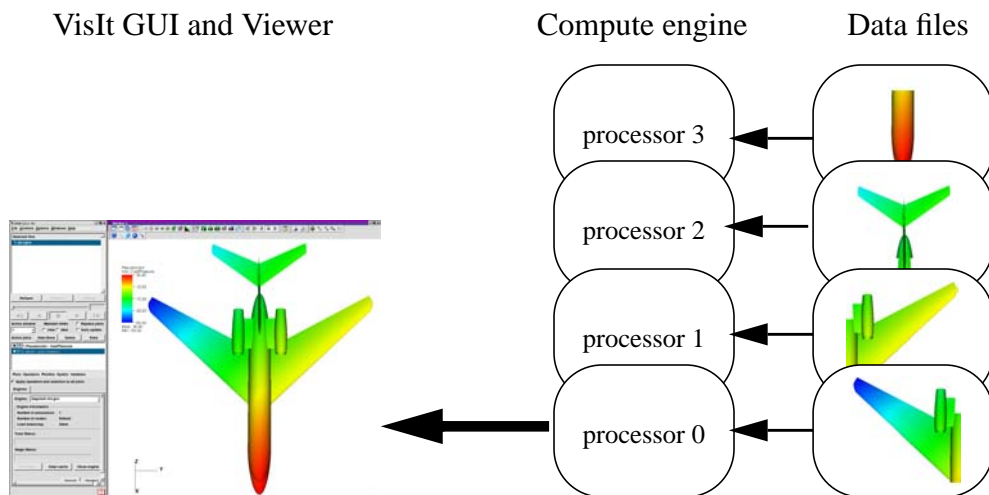


Figure 5-2: VisIt's compute engine reads data files in parallel and sends data to the viewer component.

VisIt users often import their data via files that have been written to disk, making data visualization and analysis a post-processing step. VisIt's `libsim` simulation instrumentation library can be inserted into a simulation program to make the simulation act in many ways like a VisIt compute engine. The `libsim` library, coupled with some data access code that you must write and build into your simulation, gives VisIt's data processing routines access to the simulation's calculated data without the need for the simulation to write files to disk (see Figure 5-3). An instrumented simulation may begin its processing while periodically listening for connections from an instance of VisIt using `libsim`. When `libsim` detects that VisIt wants to connect to the simulation so its data can be visualized, `libsim` loads the VisIt Compute Engine Library (VCEL). VCEL is a dynamically loaded library that contains all of the VisIt compute engine's data processing functions. Once VCEL is loaded, your simulation connects back to VisIt's viewer so requests for plots and data can be made as though your simulation was a regular VisIt compute engine.

When a request for data comes in from VisIt's viewer, your simulation is asked to provide data via some data access code. Data access code consists of a set of callback functions that your simulation must provide in order to serve data to VisIt. Data access code is written in the same language as your simulation program and it serves as the "glue" that allows the VCEL to access your simulation's data so it can be processed and plotted in VisIt. Though the initial portion of this chapter illustrates how to integrate `libsim` routines into your simulation, much of the rest of this chapter will be devoted to writing data access code.

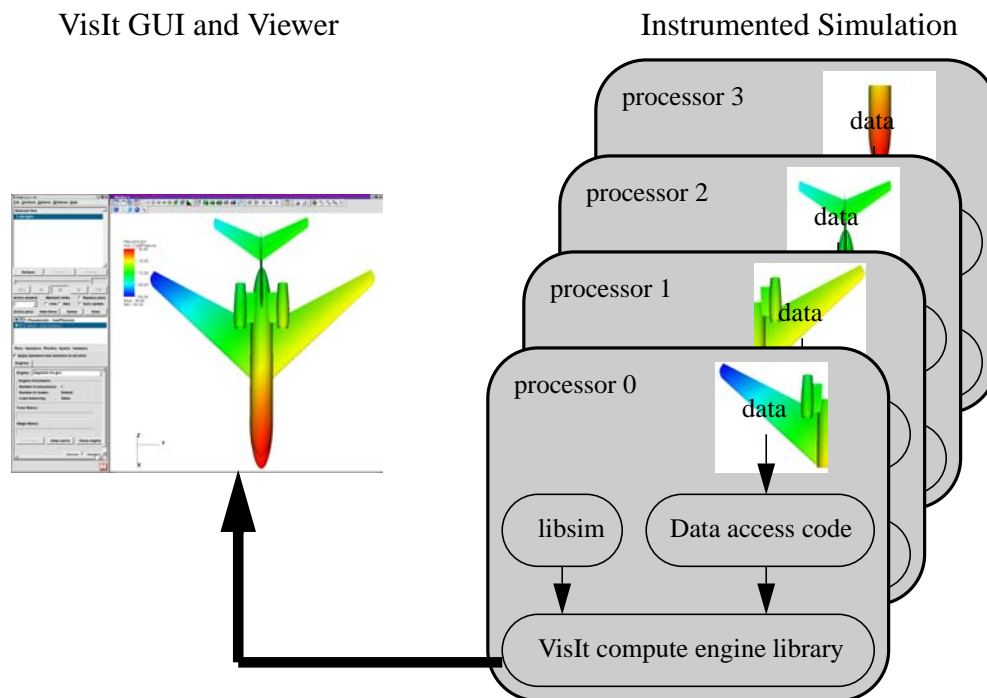


Figure 5-3: VisIt getting data from an instrumented parallel simulation

3.0 Using libsim

The first step in instrumenting a simulation so it can serve up data to VisIt is to add the `libsim` library. The `libsim` library is responsible for listening for incoming VisIt connections, connecting to them, and for dynamically loading VCEL (the piece that allows the simulation to act as a VisIt compute engine). The `libsim` library can listen for input from incoming VisIt instances, establish connections to VisIt, and respond to console input or input from VisIt. As one might imagine, this implies that your simulation's main loop will need to be changed so it calls critical routines from `libsim`. Restructuring the main loop will be covered shortly.

3.1 Getting libsim

VisIt's `libsim` library is located in the `libsim/VI` directory, which is installed under the version and platform directories when VisIt is installed. For example, if you are building against a Linux/Intel version of VisIt 1.5.4 installed in `/usr/local/apps/visit` then the full path to the `libsim` directory would be: `/usr/local/apps/visit/1.5.4/linux-intel/libsim`. Note that there may be multiple versions of the `libsim` library in the future so the current version 1 `libsim` library is installed in a `VI` subdirectory. The `VI` subdirectory contains `include` and `lib` directories that give you easy access to the required C and Fortran include files and static libraries.

The files that you need in order to instrument a simulation vary depending on the language that you used to write your simulation.

Language	Include files
C/C++	<code>VisItControlInterface_V1.h</code> <code>VisItDataInterface_V1.h</code>
Fortran	<code>visitfortransiminterface.inc</code>

3.2 Building in libsim support

When you write your simulation in C or C++, you must include `VisItControlInterface_V1.h` in your simulation's source file. In addition, you must add `libsim.a` to the list of libraries against which your program is linked. When your simulation is written in Fortran, you must also take care to include `visitfortransiminterface.inc` in your Fortran simulation code to assure that the compiler knows the names of the functions that come from `libsim`. You must link your Fortran program against both `libsim.a` and `libsimg.a`.

Listing 5-4: Including `libsim` header file in C-Language simulation.

```
#include <VisItControlInterface_V1.h>
int main(int argc, char **argv)
```

```

{
    return 0;
}

```

Listing 5-5: Including libsim header file in Fortran-Language simulation.

```

program main
implicit none
include "visitfortransiminterface.inc"
stop
end

```

Using `libsim` on UNIX platforms, such as Linux, will most likely require you to link your simulation with the dynamic loader library (`-ldl`) because `libsim` uses the system's `dlopen` function to dynamically load the VisIt Compute Engine Library.

3.3 Initialization

This section discusses the changes to the main program that are involved when instrumenting a simulation code with `libsim`. The following examples are cartoonish but they show how the main program evolves from something very simple into a main program that can serve as the skeleton of a simulation that can act as a VisIt compute engine. Once you adapt one of your programs to use `libsim`, it is easy to use that program as a template for future simulations. Additions to the example programs in this section will be underlined unless otherwise stated.

Listing 5-6: `sim1.c`: C-Language simulation example before adding `libsim`

```

/* SIMPLE SIMULATION SKELETON */
void simulate_one_timestep()
{
    /* Simulate 1 timestep. */
}
int main(int argc, char **argv)
{
    read_input_deck();
    do
    {
        simulate_one_timestep();
        write_vis_dump();
    } while(!simulation_done());
    return 0;
}

```

3.3.1 Setting up the environment and creating a .sim file

The first step in instrumenting a simulation with `libsim` is to call `libsim`'s initialization functions, starting with the `VisitSetupEnvironment` function. The `VisitSetupEnvironment` function adds important visit-related environment variables to the environment, ensuring that `VisIt` has the environment that it needs to find its plug-ins, etc.

Step 2 in instrumenting a simulation is to call the `VisitInitializeSocketAndDumpSimFile` function, which initializes the `libsim` library and writes out a `.sim` file to your `.visit` directory in your home directory. A `.sim` file is a small text file that contains details that tell `VisIt` how to connect to your running simulation. The `.sim` file contains such information as the name of the computer where your simulation is running, the port that should be used to connect to the simulation, and the key that should be returned when you successfully connect to the simulation. The first argument to the `VisitInitializeSocketAndDumpSimFile` function is the base name that will be used to construct a filename for the `.sim` file. The name for a `.sim` file is typically the specified file base with the time that the simulation started appended to it, allowing you to distinguish between multiple simulations that may be running concurrently. The second argument is a comment that can be used to further identify your simulation. The third argument contains the directory path to where your simulation was started, though it is mainly reserved for future use. The fourth argument, which is optional, contains the path and name to the simulation's input file. The final argument, which is also optional, contains the name of an XML user interface file that `VisIt` can use to create a custom user interface for controlling your simulation.

Listing 5-7: `sim2.c`: C-Language simulation example including `libsim` initialization

```
/* SIMPLE SIMULATION SKELETON */
#include <VisitControlInterface V1.h>
void simulate_one_timestep()
{
    /* Simulate 1 timestep. */
}
int main(int argc, char **argv)
{
    /* Initialize environment variables. */
    VisitSetupEnvironment();
    /* Write out .sim file that VisIt uses to connect. */
    VisitInitializeSocketAndDumpSimFile("simname",
        "Simulation Comment", "/path/to/where/sim/was/started",
        NULL, NULL);

    read_input_deck();
    do
    {
        simulate_one_timestep();
        write_vis_dump();
    } while(!simulation_done());
}
```



```

    return 0;
}

```

3.3.2 Parallel initialization

Parallel programs often require global communication to ensure that all processors are working on the same activity. The `libsim` library requires periodic global communication to ensure that all processors service the same plot requests from VisIt's viewer process. Using `libsim` in a parallel simulation requires a little bit of extra setup. The code in Listing 5-8 differs from the previous code listing in three important ways, each labelled in the listing using comments: *CHANGE 1*, *CHANGE 2*, *CHANGE 3*, respectively.

The first change in the code listing adds two broadcast functions that `libsim` will use when it needs to broadcast integers or strings. The two callback functions from the code listing can most likely be copied directly into your simulation. Note that the callback functions are conditionally compiled since they are not needed in a serial simulation. The first change also includes two static integer variables that will contain the number of processors that are used to run the simulation as well as the processor's rank within that group of processors. Various routines that we'll add in future code examples will use the `par_rank`, and `par_size` integers for control flow because processor 0 needs to behave a little differently from the rest of the processors because it communicates with VisIt's viewer.

The second change in Listing 5-8 includes initialization of the MPI library, `par_rank`, `par_size`, and `libsim`. Once MPI is initialized, the processor rank and size is queried and stored in `par_rank` and `par_size` so they can be used to initialize `libsim` as well as later for control flow. Note that the broadcast functions defined in the first change are registered with `libsim`, using `VisItSetBroadcastIntFunction` and `VisItSetBroadcastStringFunction`, so `libsim` can broadcast integers and strings among processors. Once the broadcast callbacks are installed, `par_rank` and `par_size` are used to tell `libsim` how many processors there are and whether the simulation is parallel using the `VisItSetParallel` and `VisItSetParallelRank` functions.

Listing 5-8: `sim2p.c`: C-Language simulation example including parallel `libsim` initialization

```

/* SIMPLE PARALLEL SIMULATION SKELETON */
#include <VisItControlInterface_V1.h>
#include <mpi.h>
void simulate_one_timestep()
{
    /* Simulate 1 timestep. */
}
/* CHANGE 1 */
#ifdef PARALLEL

```

```

static int visit broadcast int callback(int *value, int sender)
{
    return MPI Bcast(value, 1, MPI INT, sender, MPI COMM WORLD);
}
static int visit broadcast string callback(char *str, int len,
int sender)
{
    return MPI Bcast(str, len, MPI CHAR, sender, MPI COMM WORLD);
}
#endif
static int par rank = 0;
static int par size = 1;

int main(int argc, char **argv)
{
    /* Initialize environment variables. */
    VisItSetupEnvironment();
    /* CHANGE 2 */
#ifdef PARALLEL
    /* Initialize MPI */
    MPI Init(&argc, &argv);
    MPI Comm rank (MPI COMM WORLD, &par rank);
    MPI Comm size (MPI COMM WORLD, &par size);

    /* Install callback functions for global communication. */
    VisItSetBroadcastIntFunction(visit broadcast int callback);
    VisItSetBroadcastStringFunction(visit broadcast string callback);
    /* Tell libsim whether the simulation is parallel. */
    VisItSetParallel(par size > 1);
    VisItSetParallelRank(par rank);
#endif

    /* Write out .sim file that VisIt uses to connect. Only do it
    * on processor 0.
    */
    /* CHANGE 3*/
    if(par rank == 0)
    {
        VisItInitializeSocketAndDumpSimFile("simname",
            "Simulation Comment", "/path/to/where/sim/was/started",
            NULL, NULL);
    }
    read_input_deck();
    do
    {
        simulate_one_timestep();
        write_vis_dump();
    } while(!simulation_done());

#ifdef PARALLEL
    MPI Finalize();
#endif

    return 0;

```

```
}
|_____|
```

3.4 Restructuring the main loop

Given the example code from the previous example, the `do . . while` loop that serves as the simulation's main loop can be separated out into a new function called `mainloop`.

3.4.1 Creating a mainloop function

Moving the `do . . while` loop into a separate `mainloop` function will help in the next stage where additional `libsim` functions will be called. If your simulation does not have a well-defined function for simulating one time step, as in the previous example code, then it is strongly recommended that you refactor your simulation so that code to simulate 1 time can be called from `mainloop` using either a single function or a small block of code. The next examples assume that the simulation provides a function called: `simulate_one_timestep` that can be called over and over again to perform one cycle of the simulation.

Listing 5-9: `sim3.c`: C-Language simulation example with a mainloop function.

```
/* SIMPLE SIMULATION SKELETON */
#include <VisItControlInterface_V1.h>
void simulate_one_timestep()
{
    /* Simulate 1 timestep. */
}

void mainloop(void)
{
    do
    {
        simulate_one_timestep();
        write_vis_dump();
    } while(!simulation_done());
}

int main(int argc, char **argv)
{
    /* Initialize environment variables. */
    VisItSetupEnvironment();
    /* Write out .sim file that VisIt uses to connect. */
    VisItInitializeSocketAndDumpSimFile("simname",
        "Simulation Comment",
        "/no/useful/path/path/to/where/sim/was/started", NULL, NULL);

    /* Read input problem setup, geometry, data. */
    read_input_deck();
}
```

```

    /* Call the main loop. */
    mainloop();

    return 0;
}

```

3.4.2 Adding libsim functions to mainloop

Now that the main loop of the program has been extracted from the main piece of the simulation, we can perform an even larger change on the `mainloop` function. The following code example keeps only the `do . . while` loop and the call to `simulate_one_timestep`; everything else is new. The structure of the `mainloop` function will be very similar between simulations since most of the code is devoted to detecting input from `VisIt` using `libsim` and doing the right thing based on that input.

Listing 5-10: `sim4.c`: C-Language simulation example with fully instrumented `mainloop` function.

```

/* Is the simulation in run mode (not waiting for VisIt input) */
static int runFlag = 1;

void mainloop(void)
{
    int blocking, visitstate, err = 0;

    do
    {
        blocking = runFlag ? 0 : 1;
        /* Get input from VisIt or timeout so the simulation can run. */
        visitstate = VisItDetectInput(blocking, -1);

        /* Do different things depending on the output from
        VisItDetectInput. */
        if(visitstate <= -1)
        {
            fprintf(stderr, "Can't recover from error!\n");
            err = 1;
        }
        else if(visitstate == 0)
        {
            /* There was no input from VisIt, return control to sim. */
            simulate_one_timestep();
        }
        else if(visitstate == 1)
        {
            /* VisIt is trying to connect to sim. */
            if(VisItAttemptToCompleteConnection())
                fprintf(stderr, "VisIt connected\n");
            else
                fprintf(stderr, "VisIt did not connect\n");
        }
    }
}

```

```
else if(visitstate == 2)
{
    /* VisIt wants to tell the engine something. */
    runFlag = 0;
    if(!VisItProcessEngineCommand())
    {
        /* Disconnect on an error or closed connection. */
        VisItDisconnect();
        /* Start running again if VisIt closes. */
        runFlag = 1;
    }
}
} while(!simulation_done() && err == 0);
}
```

There are several functions from `libsim` that are called in the new `mainloop` function. The first `libsim` function that we call is the `VisItDetectInput` function, which listens for inbound `VisIt` connections on a port that was allocated when `libsim` was initialized. The `VisItDetectInput` function can be called so that it blocks indefinitely, or so that it times out after a brief period. When the simulation starts up, `VisItDetectInput` is called in non-blocking mode so that it times out. When a timeout occurs, the `VisItDetectInput` function returns zero and we call the `simulate_one_timestep` function. Since the `VisItDetectInput` function will continue to time out until `VisIt` connects to it, this augmented main loop allows the simulation to keep iterating, while still periodically listening for inbound `VisIt` connections.

When `visItDetectInput` returns one, there is an inbound `visIt` connection to which the simulation should try and connect. In this situation, we call the

`visItAttemptToCompleteConnection` function, which is responsible for two crucial actions. The first action is to dynamically load VCEL (VisIt Compute Engine Library), which is the piece of the puzzle that allows the simulation to perform compute engine operations. After loading VCEL, the

`visItAttemptToCompleteConnection` function tries to connect back to `visIt`'s viewer. In the event of a successful connection, the viewer and the simulation will be connected and the simulation will appear in the GUI's **Compute Engines** and **Simulation** windows (see Figure 5-11).

When `visItDetectInput` returns two, `visIt`'s viewer is sending commands to generate plots to the simulation. The simulation can handle commands from the viewer simply by calling the `visItProcessEngineCommand` function. The `visItProcessEngineCommand` function reads the commands coming from the viewer and uses them to make requests of VCEL, which ends up requesting data through your data access code and processing it. If the `visItProcessEngineCommand` function fails for any reason, it usually means that either `visIt` quit or the communication link between `visIt` and the simulation was severed. When the simulation can no longer communicate with `visIt`, it is important for it to call `libsIm`'s `visItDisconnect` function. The `visItDisconnect` function resets `libsIm` so it is ready to once again accept a new incoming `visIt` connection. Note that after calling `visItDisconnect`, we also set the `runFlag` variable to ensure that the simulation begins to again run autonomously.

3.4.3 Setting up mainloop for a parallel simulation

In `visIt`'s parallel compute engine, only the first processor, processor 0, communicates in any way with `visIt`'s viewer. When requests for plots come in, processor 0 broadcasts the requests to all of the other processors so all can begin working on the request. Instead of calling `visItProcessEngineCommand` directly in a parallel simulation, you will have to add code to ensure that all slave processors also call `visItProcessEngineCommand` when needed. Listing 5-12 shows how instead of calling `visItProcessEngineCommand` directly, you can call it and broadcast the appropriate cues to other processors, ensuring they also process input from `visIt`'s viewer. Note that command communication also requires calling the



Figure 5-11: Simulation window

VisitSetSlaveProcessCallback function and registering a slave process callback to be used in command communication.

Listing 5-12: sim4p.c: C-Language simulation example with fully instrumented parallel mainloop function.

```

#define VISIT_COMMAND_PROCESS 0
#define VISIT_COMMAND_SUCCESS 1
#define VISIT_COMMAND_FAILURE 2

/* Helper function for ProcessVisitCommand */
static void BroadcastSlaveCommand(int *command)
{
#ifdef PARALLEL
    MPI_Bcast(command, 1, MPI_INT, 0, MPI_COMM_WORLD);
#endif
}

/* Callback involved in command communication. */
void SlaveProcessCallback()
{
    int command = VISIT_COMMAND_PROCESS;
    BroadcastSlaveCommand(&command);
}

/* Process commands from viewer on all processors. */
int ProcessVisitCommand(void)
{
    int command;
    if (par_rank == 0)
    {
        int success = VisitProcessEngineCommand();
        if (success)
        {
            command = VISIT_COMMAND_SUCCESS;
            BroadcastSlaveCommand(&command);
            return 1;
        }
        else
        {
            command = VISIT_COMMAND_FAILURE;
            BroadcastSlaveCommand(&command);
            return 0;
        }
    }
    else
    {
        /* Note: only through the SlaveProcessCallback callback
        * above can the rank 0 process send a VISIT_COMMAND_PROCESS
        * instruction to the non-rank 0 processes. */
        while (1)
        {
            BroadcastSlaveCommand(&command);
            switch (command)
            {

```

```

        case VISIT_COMMAND_PROCESS:
            VisItProcessEngineCommand();
            break;
        case VISIT_COMMAND_SUCCESS:
            return 1;
        case VISIT_COMMAND_FAILURE:
            return 0;
    }
}
}

/* Is the simulation in run mode (not waiting for VisIt input) */
static int runFlag = 1;

/* New function to contain the program's main loop. */
void mainloop(void)
{
    int blocking, visitstate, err = 0;

    do
    {
        blocking = runFlag ? 0 : 1;
        /* Get input from VisIt or timeout so the simulation can run. */
        if(par rank == 0)
            visitstate = VisItDetectInput(blocking, -1);
        MPI_Bcast(visitstate, 1, MPI_INT, 0, MPI_COMM_WORLD);

        /* Do different things depending on the output from
        VisItDetectInput. */
        if(visitstate >= -5 && visitstate <= -1)
        {
            fprintf(stderr, "Can't recover from error!\n");
            err = 1;
        }
        else if(visitstate == 0)
        {
            /* There was no input from VisIt, return control to sim. */
            simulate_one_timestep();
        }
        else if(visitstate == 1)
        {
            /* VisIt is trying to connect to sim. */
            if(VisItAttemptToCompleteConnection())
            {
                fprintf(stderr, "VisIt connected\n");
                VisItSetSlaveProcessCallback(SlaveProcessCallback);
            }
            else
                fprintf(stderr, "VisIt did not connect\n");
        }
        else if(visitstate == 2)
        {
            /* VisIt wants to tell the engine something. */

```



```

runFlag = 0;
if(!ProcessVisItCommand())
{
    /* Disconnect on an error or closed connection. */
    VisItDisconnect();
    /* Start running again if VisIt closes. */
    runFlag = 1;
}
}
} while(!simulation_done() && err == 0);
}

```

3.5 Using libsim in a Fortran simulation

So far, most of the examples for using `libsim` have been expressed in the C programming language. It is also possible to instrument Fortran simulations so they can serve their data up to VisIt. This subsection will list the entire code skeleton for a `libsim`-instrumented Fortran simulation since the transitions that evolved a simple program into one that can connect to VisIt have already been demonstrated in C. The principles for instrumenting a Fortran program are the same. If you want to inspect the intermediate steps involved in converting a simple Fortran simulation program, examine the sample programs that accompany this book.

The primary source of differences between the following code listing and the code in Listing 5-10 result from Fortran's treatment of string variables. Strings are not always null-terminated in Fortran as they are in C, so any `libsim` function that takes string arguments will require the length of each string argument to be passed as well. The length argument immediately follows any string argument in the argument list of a `libsim` function.

The Fortran interface to `libsim` differs in another significant way; it requires certain functions to be defined in order to link successfully. The `libsim` library uses callback functions, or functions that must be provided by your simulation, in order to perform certain operations. Since the Fortran programming language lacks pointers, it is not possible to pass the address of a function that will perform a certain action to `libsim`. The Fortran interface to `libsim`, called `libsimf`, gets around this limitation by registering internal callback functions, which reference Fortran functions that must be provided by your simulation. The data access functions required to pass simulation data to VCEL are handled using the same method, thus instrumenting a Fortran simulation initially requires more steps than instrumenting a C simulation. The number of steps to instrument simulations in either language is ultimately the same.

Listing 5-13: `fsim4.f`: Fortran language simulation example with fully instrumented mainloop function.

```
c Program: main
c
c-----
      program main
      implicit none
      include "visitfortransiminterface.inc"
ccc  local variables
      integer err

      err = visitsetupenv()
      err = visitinitializesim("fsim4", 5,
. "Fortran prototype simulation connects to VisIt", 46,
. "/no/useful/path", 15,
. VISIT_F77NULLSTRING, VISIT_F77NULLSTRINGLEN,
. VISIT_F77NULLSTRING, VISIT_F77NULLSTRINGLEN)
      call mainloop()
      stop
      end

c-----
c mainloop
c-----

      subroutine mainloop()
      implicit none
      include "visitfortransiminterface.inc"
ccc  local variables
      integer visitstate, result, runflag, blocking

c  main loop
      runflag = 1
      do 10
         if(runflag.eq.1) then
            blocking = 0
         else
            blocking = 1
         endif

         visitstate = visitdetectinput(blocking, -1)

         if (visitstate.lt.0) then
            goto 1234
         elseif (visitstate.eq.0) then
            call simulate_one_timestep()
         elseif (visitstate.eq.1) then
            runflag = 0
            result = visitattemptconnection()
            if (result.eq.1) then
               write (6,*) 'VisIt connected!'
            else
               write (6,*) 'VisIt did not connect!'
            endif
         elseif (visitstate.eq.2) then
            runflag = 0
            if (visitprocessenginecommand().eq.0) then
```

```

        result = visitdisconnect()
        runflag = 1
    endif
endif
10    continue
1234 end

subroutine simulate_one_timestep()
c Simulate one time step
write (6,*) 'Simulating time step'
call sleep(1)
end

```

The above code listing lists the functions from `libsimg` that must be called from the program's main function and main loop for a serial simulation. When instrumenting a Fortran simulation using `libsimg`, you must define the following functions in order to link your program successfully:

Required subroutine/function	Argument types
subroutine <code>visitcommandcallback (cmd, lcmnd, intdata, floatdata, stringdata, lstringdata)</code>	character* 8 cmd, stringdata integer lcmnd, lstringdata , intdata real floatdata
integer function <code>visitbroadcastintfunction(value, sender)</code>	integer value, sender
integer function <code>visitbroadcaststringfunction(str, lstr, sender)</code>	character* 8 str integer lstr, sender
subroutine <code>visitslaveprocesscallback ()</code>	

These functions are primarily for using `libsimg` with a parallel simulation but they must always be defined. Extending a parallel Fortran simulation will be covered shortly. In addition, there are functions related to data access code that must also be defined in order to get your Fortran simulation to link successfully. Look at the `fsimg4.f` source code file for examples of which functions must also be defined. Those additional functions will be covered later in this chapter.

3.6 Using `libsimg` in a parallel Fortran simulation

A parallel Fortran simulation's `mainloop` function should look very similar to its serial counterpart in terms of how code is organized. Once you have adapted your simulation so it can be instrumented with `libsimg`, it is possible to make further changes that allow each processor to serve data to VisIt in parallel. There are many changes that need to happen in order to instrument a parallel simulation so the process will be broken into

stages. The changes begin with telling VisIt the number of processors and the rank of the current processor within the group before the call to the `visitinitializesim` function. You can provide this information to VisIt by calling MPI's `MPI_COMM_RANK` and `MPI_COMM_SIZE` functions and then passing the resulting rank and size data to the `visitsetparallel` and `visitsetparallelrank` functions. Once the rank and size data have been given to `libsिम`, the next change is to ensure that only the master, or rank zero, process calls the `visitinitializesim` function from `libsिम`. Only the master process should call the `visitinitializesim` function to ensure that only one `sim1` file is created.

Listing 5-14: `fscalarp.f`: Fortran language simulation example for parallel initialization.

```

c-----
c Program: main
c
c-----
      program main
      implicit none
      include "visitfortransiminterface.inc"
      include "mpif.h"
ccc  local variables
      integer err
ccc  PARALLEL state common block
      integer par rank, par size
      common /PARALLEL/ par rank, par size
      save /PARALLEL/

      call MPI_INIT(err)

c Determine the rank and size of this MPI task so we can tell
c VisIt's libsिम about it.
      call MPI_COMM_RANK(MPI_COMM_WORLD, par rank, err)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, par size, err)
      if(par size.gt.1) then
         err = visitsetparallel(1)
      endif
      err = visitsetparallelrank(par rank)

      err = visitsetupenv()
c Have the master process write the sim file.
      if(par rank.eq.0) then
         err = visitinitializesim("fscalarp", 8,
         .   "Demonstrates scalar data access function", 40,
         .   "/no/useful/path", 15,
         .   VISIT_F77NULLSTRING, VISIT_F77NULLSTRINGLEN,
         .   VISIT_F77NULLSTRING, VISIT_F77NULLSTRINGLEN)
      endif

      call mainloop()

      call MPI_FINALIZE(err)

```

```

stop
end

```

The next step in instrumenting a parallel Fortran simulation is to change the `mainloop` function. The first change that you must make is to ensure that only the master process calls `visitdetectinput`. Remember that only the master process talks to VisIt's viewer process so the `visitdetectinput` function should not be called by slave processes. However, the slaves need to know the instructions that came from the viewer so we must insert an MPI broadcast function to ensure that all processes get the value sent from the viewer to the master process. In addition the `visitprocessenginecommand` function must be exchanged for a function that can call `visitprocessenginecommand` on all processes. For now, let's call that new function `processvisitcommand`.

Listing 5-15: `fscalarp.f`: Fortran language simulation example for parallel `mainloop` function.

```

-----
c mainloop
-----
      subroutine mainloop()
      implicit none
      include "mpif.h"
      include "visitfortransiminterface.inc"
ccc  functions
      integer processvisitcommand
ccc  local variables
      integer visitstate, result, blocking, ierr
ccc  SIMSTATE common block
      integer runflag, simcycle
      real simtime
      common /SIMSTATE/ runflag,simcycle,simtime
      save /SIMSTATE/
ccc  PARALLEL state common block
      integer par rank, par size
      common /PARALLEL/ par rank, par size
c    main loop
      runflag = 1
      simcycle = 0
      simtime = 0
      do 10
         if(runflag.eq.1) then
            blocking = 0
         else
            blocking = 1
         endif

ccc  Detect input from VisIt on processor 0 and then broadcast
ccc  the results of that input to all processors.
         if(par rank.eq.0) then

```

```

        visitstate = visitdetectinput(blocking, -1)
    endif
    call MPI_BCAST(visitstate,1,MPI_INTEGER,0,
. MPI_COMM_WORLD,ierr)

    if (visitstate.lt.0) then
        goto 1234
    elseif (visitstate.eq.0) then
        call simulate_one_timestep()
    elseif (visitstate.eq.1) then
        runflag = 0
        result = visitattemptconnection()
        if (result.eq.1) then
            write (6,*) 'VisIt connected!'
        else
            write (6,*) 'VisIt did not connect!'
        endif
    elseif (visitstate.eq.2) then
        runflag = 0
        if (processvisitcommand().eq.0) then
            result = visitdisconnect()
            runflag = 1
        endif
    endif
endif
10    continue
1234 end

```

Now that you have changed the mainloop function it is time to define the processvisitcommand function. The processvisitcommand function is used by the mainloop function as a replacement for the visitprocessenginecommand function. The new processvisitcommand function must call the visitprocessenginecommand function and it must do so in a way that ensures the function is called on all processors. Since the processvisitcommand function is completely new, you will probably be able to paste it into your simulation with few changes.

Listing 5-16: fscalarp.f: Fortran language simulation example for parallel processvisitcommand function.

```

c-----
c processvisitcommand
c-----

integer function processvisitcommand()
implicit none
include "mpif.h"
include "visitfortransiminterface.inc"
ccc PARALLEL state common block
integer par_rank, par_size
common /PARALLEL/ par_rank, par_size

```

```

integer command, e, doloop, success, ret
integer VISIT_COMMAND_PROCESS
integer VISIT_COMMAND_SUCCESS
integer VISIT_COMMAND_FAILURE
parameter (VISIT_COMMAND_PROCESS = 0)
parameter (VISIT_COMMAND_SUCCESS = 1)
parameter (VISIT_COMMAND_FAILURE = 2)

if(par_rank.eq.0) then
    success = visitprocessenginecommand()

    if(success.gt.0) then
        command = VISIT_COMMAND_SUCCESS
        ret = 1
    else
        command = VISIT_COMMAND_FAILURE
        ret = 0
    endif

    call MPI_BCAST(command,1,MPI_INTEGER,0,MPI_COMM_WORLD,e)
else
    doloop = 1
2345    call MPI_BCAST(command,1,MPI_INTEGER,0,MPI_COMM_WORLD,e)
    if(command.eq.VISIT_COMMAND_PROCESS) then
        success = visitprocessenginecommand()
    elseif(command.eq.VISIT_COMMAND_SUCCESS) then
        ret = 1
        doloop = 0
    else
        ret = 0
        doloop = 0
    endif
    if(doloop.ne.0) then
        goto 2345
    endif
endif
processvisitcommand = ret
end

```

The alterations to the code that have been listed thus far are nearly enough to complete the changes required for a parallel Fortran simulation to use `libsिम`. The main program and the `mainloop` function have been changed to support the extra processing that needs to happen to ensure that all processors properly receive instructions from VisIt's viewer. However, there are some broadcast callback functions that must now be implemented to ensure that `libsिम` can communicate with all processors. The callback functions: `visitbroadcastintfunction`, `visitbroadcaststringfunction`, and `visitslaveprocesscallback` have to date been stub functions that did not do any

real work. When you instrument a parallel Fortran simulation, those callback functions need to perform broadcasts so `libsimg` can properly communicate with all processors.

Listing 5-17: `fscalarp.f`: Fortran language simulation example for parallel broadcast functions.

```

-----
c visitbroadcastintfunction
-----
      integer function visitbroadcastintfunction(value, sender)
      implicit none
      include "mpif.h"
      integer value, sender
      integer ierr
      call MPI_BCAST(value,1,MPI_INTEGER,sender,MPI_COMM_WORLD,ierr)
      visitbroadcastintfunction = 0
      end

-----
c visitbroadcaststringfunction
-----
      integer function visitbroadcaststringfunction(str, lstr,
      sender)
      implicit none
      include "mpif.h"
      character*8 str
      integer lstr, sender
      integer ierr
      call MPI_BCAST(str,lstr,MPI_CHARACTER,sender,MPI_COMM_WORLD,
      . ierr)
      visitbroadcaststringfunction = 0
      end

-----
c visitslaveprocesscallback
-----
      subroutine visitslaveprocesscallback ()
      implicit none
      include "mpif.h"
      integer c, ierr, VISIT_COMMAND_PROCESS
      parameter (VISIT_COMMAND_PROCESS = 0)
      c = VISIT_COMMAND_PROCESS
      call MPI_BCAST(c,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
      end

```

After making all of these changes, your parallel Fortran simulation should be ready to run for the first time as an application to which VisIt can connect. You will not be able to extract any data from your simulation just yet but you can begin to run connected to VisIt and once you have that working you can begin to expose your data to VisIt.

3.7 Running an instrumented simulation

Once you've added `libsim` functions to your simulation and created a `mainloop` function capable of connecting to VisIt, you can run your modified simulation. The current `libsim` implementation must be told where to locate VisIt's shared libraries and plug-ins in order to have VCEL function properly. VisIt uses environment variables to locate its shared libraries and plug-ins. If you use a Linux version of VisIt 1.5.4 installed in `/usr/local/apps/visit` then use the following commands to ensure that VCEL can find the necessary VisIt libraries when it runs:

```
# Set VISIT to the directory where a version of VisIt is intalled
setenv VISIT /usr/local/apps/visit/1.5.4/linux-intel
env LD_LIBRARY_PATH=$VISIT/lib VISITPLUGINDIR=$VISIT/plugins ./sim
```

If you use a different version of VisIt or run VisIt on a platform other than Linux, make the appropriate substitutions in the `VISIT` environment variable before trying to run.

3.8 Connecting to an instrumented simulation from VisIt

Once you've successfully launched your simulation, you can attempt to connect to it using VisIt. Open a terminal window and run VisIt. When VisIt comes up, open the **File selection window** and browse to `~/.visit/simulations`, the directory where `.sim1` files are stored. You should see a file in that directory with a `.sim1` file extension. The `.sim1` file was created by your simulation when it started and called the `VisItInitializeSocketAndDumpSimFile` function from `libsim`. The `.sim1` file contains all of the information that VisIt needs to connect to your simulation. If you add the file to your selected files list and open it in VisIt's **Main window**, VisIt will initiate contact with your simulation.

If your environment was not properly set when you ran your simulation, VisIt will not be able to connect to it and you might see error messages like the following:

```
Simulating time step
Simulating time step
VisIt did not connect
Simulating time step
Simulating time step
```

Error messages such as those above appear in the terminal window where your simulation was launched and they result from your environment not being set properly. Be sure that the paths that you use for `LD_LIBRARY_PATH` and `VISITPLUGINDIR` are valid paths that contain VisIt files. The current VisIt implementation lacks some robustness with respect to connecting to simulations. If VisIt cannot connect to your simulation then the viewer process will be hung and you will have to kill it. Note that killing VisIt will not have any impact on your simulation.

At this stage in instrumenting your simulation, if it was able to successfully create a connection to VisIt then you will see the name of your simulation in the **Compute**

engines window and the **Simulations window**. Furthermore, you will see the following messages in the window where you launched your simulation:

```
Simulating time step
Simulating time step
VisIt connected
Error opening plug-in file: /usr/gapps/visit/1.5.4/linux-
intel/plugins/databases/libESimV1Database_ser.so: undefined symbol:
visitCallbacks
Error opening plug-in file: /usr/gapps/visit/1.5.4/linux-
intel/plugins/databases/libESimV1Database_ser.so: undefined symbol:
visitCallbacks
```

The above messages indicate that VisIt successfully connected back to your simulation. Also notice that there are error messages about an undefined symbol called `visitCallbacks`. This is to be expected since `visitCallbacks` is part of the data access code that must be added to your simulation. Since no data access code has yet been added to your simulation, VCEL cannot find the `visitCallbacks` object that allows it to call your data access functions, which ultimately pass your simulation's data to VCEL. Do not worry about the error messages because the next section explains how to write data access code for your simulation. In the meantime, note how quitting VisIt causes your simulation to resume calculations.

4.0 Writing data access code

If you have made it this far then you probably have a simulation that has been restructured to use `libsim`. Once a simulation has been instrumented using `libsim`, it should be possible for VisIt to connect to it. Adding the code to allow VisIt to connect to your simulation is only the first part of instrumenting your simulation. The next phase in instrumenting your simulation code is adding data access code to your simulation so VCEL, the VisIt Compute Engine Library, can access your simulation's data.

Writing data access code is much like writing a database reader plug-in. It all starts with writing a function to provide metadata to VisIt so that it knows the names of the meshes and variables that are available for plotting. After your simulation is capable of telling VisIt about its variables, the next step is to write functions that can pass your mesh or data arrays to VisIt so they can be used in plots. If your data is not in a format that VisIt readily supports, you can create a more VisIt-friendly representation of the data in the data access functions and hand it off to VisIt.

4.1 The VisIt Data Interface

VisIt relies on the VisIt Data Interface (VDI), a C header file containing the structures and formats of the data that are supported. The VDI determines how all of the objects passed as data to VCEL are to be stored in memory. When you write a data access function for your simulation, you create objects of types defined in the VDI, populate their data, and

return them. From there, the objects are used to serve data up to VCEL where your plots are processed.

The VDI C-Language header file is called `VisitDataInterface_V1.h` and it defines the types and structures that are used when creating objects that pass data to VisIt. The header file is installed with the binary VisIt distribution. If a Linux version of VisIt 1.5.4 was installed in `/usr/local/apps/visit` then the header file would be located in `/usr/local/apps/visit/1.5.4/linux-intel/include/visit/libsim/V1/include`. Of course, the actual path depends on where VisIt was installed, the version of VisIt that was installed, and the platform.

If you are writing your simulation in Fortran then the `VisitDataInterface_V1.h` header file will be of no consequence to you. Everything you need to instrument a Fortran simulation code is located in `visitfortransiminterface.inc`, the same file that you've already used to instrument your simulation so far. Fortran simulations do not create structures directly to pass their data to VisIt. Since the structures defined in `VisitDataInterface_V1.h` are more advanced than what can be easily expressed in Fortran, `visitfortransiminterface.inc` defines many functions that can be used to create objects of the right type. These functions are actually an intermediate layer, making up the `libsimf` library, that accept the data passed as arguments to the functions and package them up in the form of the structures defined in `VisitDataInterface_V1.h` before the data is passed to VisIt for processing. The differences will become apparent in the remainder of this chapter.

4.2 How data access functions are called

VisIt data access functions are made known to VisIt using a special object called: `visitCallbacks`. The `visitCallbacks` object is an instance of the `Visit_SimulationCallback` structure and it contains pointers to the data access functions that are to be used by VisIt. When VisIt opens the `.sim1` file corresponding to your running simulation, VisIt knows that the data will come from a simulation because the `.sim1` file is opened by the SimV1 database reader plug-in. The SimV1 plug-in is a special VisIt database reader plug-in that uses the functions in the `visitCallbacks` object to access data from your simulation. When VCEL is loaded into your simulation and VisIt tells the simulation to make a plot, the request ends up in the SimV1 database reader plug-in. When the SimV1 plug-in wants to read metadata, for example, it looks for the `visitCallbacks` object and uses it to get the pointer to the function that you've provided in your simulation when VisIt wants to retrieve metadata. Once the function to call in order to get metadata has been determined, VisIt calls it, which ends up calling your function. Once your function returns a populated metadata object, the SimV1 plug-in transcribes the metadata from your metadata object into the `avtDatabaseMetaData` object that the SimV1 plug-in must populate. The basic procedure by which all of the other data access methods are called is similar.

4.3 Compiler and platform issues

Instrumenting a simulation code on different platforms, using different compilers and linkers can require different steps to be taken. This section notes some of the special methods that must be employed in order to get your instrumented simulation working.

4.3.1 Linking your simulation

The SimV1 database reader plug-in must look for the `visitCallbacks` object within the symbols exposed by your simulation in order to find it successfully. If the SimV1 plug-in cannot find the `visitCallbacks` object then it fails to load and VisIt will not be able to retrieve data from your simulation. The current approach for resolving `visitCallbacks` in the SimV1 database reader plug-in relies on the dynamic linker, which often must have additional information in the simulation executable in order to properly perform the runtime linking. In short, the current approach means that you have to add a special linker flag when linking your simulation. If your Makefile uses `LDFLAGS` to contain command line arguments that are passed to the linker, then add this line to your Makefile after `LDFLAGS` has been defined.

```
LDFLAGS=$(LDFLAGS) -Wl,--export-dynamic
```

The `--export-dynamic` linker flag is a GNU-specific linker flag that tells the linker to export all public symbols to the dynamic symbol table. Adding this flag ensures that the runtime linker can resolve the references to `visitCallbacks` inside of the SimV1 database plug-in, using the `visitCallbacks` object that you provide in your simulation. If you do not use the `--export-dynamic` linker flag, or an equivalent, when linking your simulation then the SimV1 plug-in will fail to load and VisIt will not be able to access your simulation's data.

4.3.2 The Windows platform

The `libsिम` library has not been fully ported to the Windows platform at the time of this writing. Preliminary results suggest that the dynamic linker approach to resolving `visitCallbacks` in the SimV1 plug-in will not work. Work-arounds have been explored and have even been successful but no fully productized Windows port of `libsिम` has yet been made available.

4.4 Making data access functions available

The previous sections have established the role and the importance of the `visitCallbacks` object in an instrumented simulation. Now that you know what the `visitCallbacks` object does, it is time to see how it is used to make data access functions available. The `visitCallbacks` object is nothing more than a C-Language structure that contains a set of function pointers that can be set to point to the data access functions that you provide within your simulation. If you want to make a data access function accessible to the SimV1 database reader plug-in so VisIt can read your data,

simply create a `Visit_SimulationCallback` struct called *visitCallbacks* and set its `GetMetaData` function pointer to the address of the function that you wrote to provide metadata about your simulation.

Listing 5-18: sim5.c: C-Language example for making a data access function available.

```
#include <VisitDataInterface_V1.h>

Visit_SimulationMetaData *VisitGetMetaData(void)
{
    /* Create a metadata object with no variables. */
    size_t sz = sizeof(Visit_SimulationMetaData);
    Visit_SimulationMetaData *md =
        (Visit_SimulationMetaData *)malloc(sz);
    memset(md, 0, sz);
    return md;
}

Visit_SimulationCallback visitCallbacks =
{
    &VisitGetMetaData,
    NULL, /* GetMesh */
    NULL, /* GetMaterial */
    NULL, /* GetSpecies */
    NULL, /* GetScalar */
    NULL, /* GetCurve */
    NULL, /* GetMixedScalar */
    NULL /* GetDomainList */
};
```

Data access functions for Fortran simulations do not have to be made available explicitly because that is taken care of in `visitfortransiminterface.c`, the file that defines the Fortran-callable wrapper functions for `libsim` and `VDI`. Instead of defining the data access function and including it in `visitCallbacks`, you only need to define it. In fact, all data access functions for Fortran simulations must be defined to successfully link your simulation.

Listing 5-19: fsm5.f: Fortran language example for making a data access function available.

```
integer function visitgetmetadata(handle)
implicit none
integer handle
include "visitfortransiminterface.inc"
visitgetmetadata = VISIT_OKAY
end
```

4.5 Data access function for metadata

The first data access function that you write should be the one that populates a metadata object. VisIt uses metadata to determine which meshes and variables are in a database and reading a database's metadata is the first thing VisIt does when accessing a new database. The object of the data access function for returning metadata is to allocate and return a `VisIt_SimulationMetaData` object. The `VisIt_SimulationMetaData` object contains lists of the other metadata objects. In Fortran, you do not explicitly create a `VisIt_SimulationMetaData` object. Instead, one is created in `libsimgf` and a handle to it is passed to your data access function, which then passes the handle to helper functions in `libsimgf` that perform various operations on the allocated object. Good starting points for a data access function that returns metadata are found in Listing 5-18 and Listing 5-19. The code listings found in this section may reproduce those listings, however, as the listings get longer, the following code listings may instead contain code fragments required to perform a particular operation. The code fragments can be included into your simulation and modified until they expose the right variables for your simulation.

4.5.1 Returning simulation state metadata

Simulation state metadata is important because it indicates the running state of the simulation as well as its cycle iteration and simulated time. The C-Language example in Listing 5-20 shows that the simulation state can be set directly into the metadata object. The Fortran language example in Listing 5-21 shows how to set the simulation state into the metadata object using the `visitmdsetcycletime` and the `visitmdsetrunning` functions..

Listing 5-20: `sim6.c`: C-Language example for returning simulation state metadata.

```
static int    simcycle = 0;
static double simtime = 0.;
VisIt_SimulationMetaData *VisItGetMetaData(void)
{
    /* Create a metadata object with no variables. */
    size_t sz = sizeof(VisIt_SimulationMetaData);
    VisIt_SimulationMetaData *md =
        (VisIt_SimulationMetaData *)malloc(sz);
    memset(md, 0, sz);

    /* Set the simulation state. */
    md->currentMode = runFlag ? VISIT_SIMMODE_RUNNING :
        VISIT_SIMMODE_STOPPED;
    md->currentCycle = simcycle;
    md->currentTime = simtime;
    return md;
}
```

 }

Listing 5-21: fsm6.f: Fortran language example for returning simulation state metadata.

```

    integer function visitgetmetadata(handle)
    implicit none
    integer handle
    include "visitfortransiminterface.inc"
    c SIMSTATE common block (data shared with mainloop and
    c simulate_one_timestep)
    integer runflag, simcycle
    real simtime
    common /SIMSTATE/ runflag, simcycle, simtime
    integer err

    err = visitmdsetcycletime(handle, simcycle, simtime)
    if(runflag.eq.1) then
        err = visitmdsetrunning(handle, VISIT_SIMMODE_RUNNING)
    else
        err = visitmdsetrunning(handle, VISIT_SIMMODE_STOPPED)
    endif

    visitgetmetadata = VISIT_OKAY
  end

```

4.5.2 Returning mesh metadata

If you want VisIt to be able to plot any of your simulation's data then you must expose at least one of your simulation's meshes in the metadata. Remember that VisIt can support several different mesh types from simple point meshes all the way up to complex multi-domain unstructured meshes.

Mesh metadata is stored in the `Visit_SimulationMetaData` as a dynamically allocated array of `Visit_MeshMetaData` objects. Each `Visit_MeshMetaData` object contains information about a mesh such as its name, type, dimensions, units, labels, etc. Note that when you create new `Visit_MeshMetaData` objects and add them to the `Visit_SimulationMetaData` object, they become the property of the `Visit_SimulationMetaData` object and should not be deallocated by you. The same principle applies to any string members in the `Visit_MeshMetaData` object; be sure to use the `strdup` function to create duplicate copies of strings so your strings are not destroyed when `Visit` deletes the `Visit_SimulationMetaData` object.

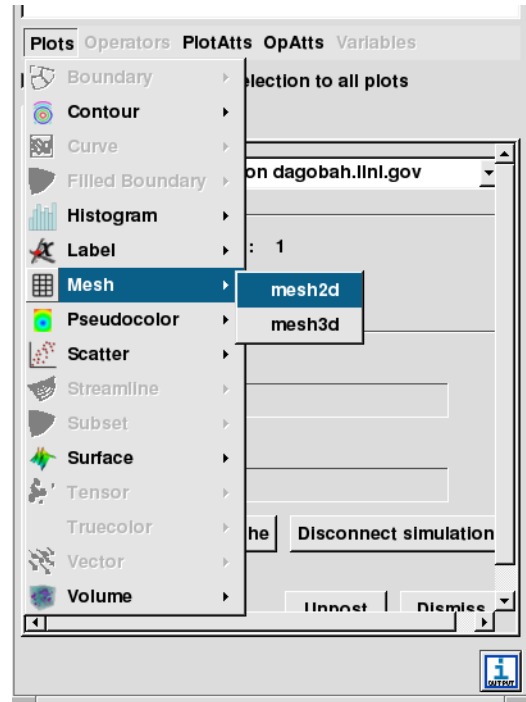


Figure 5-22: Mesh variables in the plot menu

It is not important to set values for all of the members in the `Visit_MeshMetaData` object so long as you do set values for the `name`, `meshType`, `topologicalDimension`, `spatialDimension`, and `numBlocks` structure members. The value that you use for the mesh's name is the name that will appear in `Visit`'s **Plot menu** (see Figure 5-22) as well as the name that will be passed to your data access function when `Visit` wants to plot your mesh. The `meshType` value specifies the mesh's type and can be any of the following values: `VISIT_MESHTYPE_RECTILINEAR`, `VISIT_MESHTYPE_CURVILINEAR`, `VISIT_MESHTYPE_UNSTRUCTURED`, `VISIT_MESHTYPE_POINT`, `VISIT_MESHTYPE_SURFACE`. The `topologicalDimension` and `spatialDimension` values should be either 2 or 3, depending on whether your mesh exists in 2D or 3D. Finally, the `numBlocks` value should be set to the total number of domains that comprise your mesh.

Listing 5-23: `sim7.c`: C-Language example for returning mesh metadata.

```
#define NDOMAINS 1
/* Allocate enough room for 2 meshes in the metadata. */
size_t sz;
md->numMeshes = 2;
sz = sizeof(Visit_MeshMetaData) * md->numMeshes;
md->meshes = (Visit_MeshMetaData *)malloc(sz);
memset(md->meshes, 0, sz);

/* Set the first mesh's properties.*/
md->meshes[0].name = strdup("mesh2d");
```



```

md->meshes[0].meshType = VISIT_MESHTYPE_RECTILINEAR;
md->meshes[0].topologicalDimension = 2;
md->meshes[0].spatialDimension = 2;
md->meshes[0].numBlocks = NDOMAINS;
md->meshes[0].blockTitle = strdup("Domains");
md->meshes[0].blockPieceName = strdup("domain");
md->meshes[0].numGroups = 0;
md->meshes[0].units = strdup("cm");
md->meshes[0].xLabel = strdup("Width");
md->meshes[0].yLabel = strdup("Height");
md->meshes[0].zLabel = strdup("Depth");

/* Set the second mesh's properties.*/
md->meshes[1].name = strdup("mesh3d");
md->meshes[1].meshType = VISIT_MESHTYPE_CURVILINEAR;
md->meshes[1].topologicalDimension = 3;
md->meshes[1].spatialDimension = 3;
md->meshes[1].numBlocks = NDOMAINS;
md->meshes[1].blockTitle = strdup("Domains");
md->meshes[1].blockPieceName = strdup("domain");
md->meshes[1].numGroups = 0;
md->meshes[1].units = strdup("Miles");
md->meshes[1].xLabel = strdup("Width");
md->meshes[1].yLabel = strdup("Height");
md->meshes[1].zLabel = strdup("Depth");

```

The Fortran interface does not deal directly with `Visit_MeshMetaData` objects and it hides the complexities of inserting them into the `Visit_SimulationMetaData` object. This difference in how metadata is added between the C and Fortran interfaces for `libsims` is due primarily to Fortran's lack of direct support for dynamically allocated objects. Instead of directly creating `Visit_MeshMetaData` objects in Fortran, the interface provides the `visitmdmeshcreate` function which creates a new `Visit_MeshMetaData` object, inserts it into the `Visit_SimulationMetaData` object, and returns an integer handle. The handle can be passed to other mesh-related metadata functions such as `visitmdmeshsetunits` in order to set additional mesh properties.

Listing 5-24: `fsim7.f`: Fortran language example for returning mesh metadata.

```

integer err, tdim, sdim, mesh, mt
c   Add a 2D rectilinear mesh
mt = VISIT_MESHTYPE_RECTILINEAR
tdim = 2
sdim = 2
mesh = visitmdmeshcreate(handle, "mesh2d", 6, mt, tdim,
. sdim, 1)
if(mesh.ne.VISIT_INVALID_HANDLE) then
    err = visitmdmeshsetunits(handle, mesh, "cm", 2)
    err = visitmdmeshsetlabels(handle, mesh, "Width", 5,
. "Height", 6, "Depth", 5)

```

```

        err = visitmdmeshsetblocktitle(handle, mesh, "Domains", 7)
        err = visitmdmeshsetblockpiecename(handle, mesh, "domain",
. 6)
    endif

c    Add a 3D curvilinear mesh
    tdim = 3
    sdim = 3
    mt = VISIT_MESHTYPE_CURVILINEAR
    mesh = visitmdmeshcreate(handle, "mesh3d", 6, mt, tdim,
. sdim, 1)
    if(mesh.ne.VISIT_INVALID_HANDLE) then
        err = visitmdmeshsetunits(handle, mesh, "Miles", 5)
        err = visitmdmeshsetlabels(handle, mesh, "Width", 5,
. "Height", 6, "Depth", 5)
        err = visitmdmeshsetblocktitle(handle, mesh, "Domains", 7)
        err = visitmdmeshsetblockpiecename(handle, mesh, "domain",
. 6)
    endif

```

4.5.3 Returning scalar variable metadata

Scalar variables must be exposed via the metadata if they are to be plotted in VisIt. You need not expose all of the scalar variables that you have; only those you want to plot in VisIt. The `VisIt_SimulationMetaData` object contains a list of `VisIt_ScalarMetaData` objects, which contain the metadata for all of the scalars that you expose to VisIt. Specifying a scalar variable only requires you to create a new entry in the list of `VisIt_ScalarMetaData` objects. You must set the `name`, `meshName`, and `centering` fields in the `VisIt_ScalarMetaData` structure. The Fortran interface provides the `visitmdscalarcreate` function to add metadata for a new scalar variable to the metadata.

Listing 5-25: sim8.c: C-Language example for returning scalar metadata.

```

/* Add some scalar variables. */
md->numScalars = 2;
sz = sizeof(VisIt_ScalarMetaData) * md->numScalars;
md->scalars = (VisIt_ScalarMetaData *)malloc(sz);
memset(md->scalars, 0, sz);

/* Add a zonal variable on mesh2d. */
md->scalars[0].name = strdup("zonal");
md->scalars[0].meshName = strdup("mesh2d");
md->scalars[0].centering = VISIT_VARCENTERING_ZONE;

/* Add a nodal variable on mesh3d. */
md->scalars[1].name = strdup("nodal");
md->scalars[1].meshName = strdup("mesh3d");

```

```
md->scalars[1].centering = VISIT_VARCENTERING_NODE;
```

Listing 5-26: fsm8.f: Fortran language example for returning scalar metadata.

```

integer scalar
c   Add a zonal variable on mesh2d.
scalar = visitmdscalarcreate(handle, "zonal", 5, "mesh2d", 6,
. VISIT_VARCENTERING_ZONE)
c   Add a nodal variable on mesh3d.
scalar = visitmdscalarcreate(handle, "nodal", 5, "mesh3d", 6,
. VISIT_VARCENTERING_NODE)

```

4.5.4 Returning curve variable metadata

As with other variable types, curve variables (X-Y plot data) must also be exposed in the metadata if they are to be plotted in VisIt. The `Visit_SimulationMetaData` object contains a list of `Visit_CurveMetaData` objects, which contain the attributes of the curve variables that will be exposed to VisIt from the simulation. The only required field that must be set in the `Visit_CurveMetaData` object is the name field, which specifies the name of the curve as it will be used in the **Plot list** and in your data access function.

Listing 5-27: sim9.c: C-Language example for returning curve metadata.

```

/* Add a curve variable. */
md->numCurves = 1;
sz = sizeof(Visit_CurveMetaData) * md->numCurves;
md->curves = (Visit_CurveMetaData *)malloc(sz);
memset(md->curves, 0, sz);

md->curves[0].name = strdup("sine");
md->curves[0].xUnits = strdup("radians");
md->curves[0].xLabel = strdup("angle");
md->curves[0].yLabel = strdup("amplitude");

```

Listing 5-28: fsm9.f: Fortran language example for returning curve metadata.

```

integer err, curve
c   Add a curve variable
curve = visitmdcurvecreate(handle, "sine", 4)
if(curve.ne.VISIT_INVALID_HANDLE) then
    err = visitmdcurvesetlabels(handle, curve, "angle", 5,
. "amplitude", 9)
    err = visitmdcurvesetunits(handle, curve, "radians", 7,
. VISIT_F77NULLSTRING, VISIT_F77NULLSTRINGLEN)
endif

```

The Fortran interface provides the `visitmdcurvecreate` function to add a curve to the metadata. The `visitmdcurvecreate` function takes the metadata handle, the name of the new curve, and the length of the curve name string as arguments. If the `visitmdcreatecurve` function succeeds then it returns a handle to the new curve metadata object, which can be passed to `visitmdcurvesetlabels` and `visitmdcurvesetunits` to set additional attributes.

4.5.5 Returning material metadata

In addition to the variable types mentioned so far, the `Visit_SimulationMetaData` object also contains a list of material variables. The list of material variables is stored in the `materials` member and is composed of `Visit_MaterialMetaData` objects. A `Visit_MaterialMetaData` object contains the name of the material, the mesh on which it is defined, and the list of possible material names that can be used.

Listing 5-29: `sim10.c`: C-Language example for returning material metadata.

```

/* Add a material variable. */
md->numMaterials = 1;
sz = sizeof(Visit_MaterialMetaData) * md->numMaterials;
md->materials = (Visit_MaterialMetaData *)malloc(sz);
memset(md->materials, 0, sz);

md->materials[0].name = strdup("mat");
md->materials[0].meshName = strdup("mesh2d");
md->materials[0].numMaterials = 3;
/* Allocate memory to store the list of material names. */
md->materials[0].materialNames = (const char **)malloc(
sizeof(char *) * md->materials[0].numMaterials);
md->materials[0].materialNames[0] = strdup("Iron");
md->materials[0].materialNames[1] = strdup("Copper");
md->materials[0].materialNames[2] = strdup("Nickel");

```

The Fortran interface provides a different interface once more to circumvent the difficulties imposed by dynamic memory allocation. Instead of directly allocating a `Visit_MaterialMetaData` object, in the Fortran interface, you call `visitmdmaterialcreate` to create material metadata and acquire a handle to it. The returned handle can be used with the `visitmdmaterialadd` function to add materials one at a time to the list of material types in the material metadata object.

Listing 5-30: `fsim10.f`: Fortran language example for returning material metadata.

```

integer err, mat
c   Add a material
mat = visitmdmaterialcreate(handle, "mat", 3, "mesh2d", 6)
if(mat.ne.VISIT_INVALID_HANDLE) then
  err = visitmdmaterialadd(handle, mat, "Iron", 4)
  err = visitmdmaterialadd(handle, mat, "Copper", 6)

```

```

        err = visitmdmaterialadd(handle, mat, "Nickel", 6)
    endif

```

4.5.6 Returning expression metadata

VisIt allows databases to return user-defined expressions that can be plotted or used to create new expressions in the **Expressions window**. The `VisIt_SimulationMetaData` object contains an array of `VisIt_ExpressionMetaData` objects that each contain the information for one expression. An expression consists of an expression name, definition, and expression type. The expression definition is a string that must contain a valid VisIt expression, as defined in by the expression language documented in the *VisIt User's Manual*.

Listing 5-31: sim11.c: C-Language example for returning material metadata.

```

/* Add some expressions. */
md->numExpressions = 2;
sz = sizeof(VisIt_ExpressionMetaData) * md->numExpressions;
md->expressions = (VisIt_ExpressionMetaData *)malloc(sz);
memset(md->expressions, 0, sz);

md->expressions[0].name = strdup("zvec");
md->expressions[0].definition = strdup("{zonal, zonal, zonal}");
md->expressions[0].vartype = VISIT_VARTYPE_VECTOR;

md->expressions[1].name = strdup("nid");
md->expressions[1].definition = strdup("nodeid(mesh3d)");
md->expressions[1].vartype = VISIT_VARTYPE_SCALAR;

```

The Fortran interface provides a function called `visitmdexpressioncreate` that you can use to create expressions. The function takes the new expression's name, definition and variable type as arguments and inserts a new expression definition into the `VisIt_SimulationMetaData` object.

Listing 5-32: fsim11.f: Fortran language example for returning material metadata.

```

c      Add some expressions
      e = visitmdexpressioncreate(handle, "zvec", 4,
        . "{zonal, zonal, zonal}", 21, VISIT_VARTYPE_VECTOR)
      e = visitmdexpressioncreate(handle, "nid", 3,
        . "nodeid(mesh3d)", 14, VISIT_VARTYPE_SCALAR)

```

4.5.7 Returning simulation-defined command metadata

VisIt allows your simulation to provide the names of user-defined commands in the metadata object. When such commands appear in a simulation's metadata, it influences VisIt to create special command buttons in the **Simulations window**. When you open the **Simulations window** and click on the buttons, it causes a chain of events that ends up calling your simulation's command callback function, which then performs some action based on the name of the command being executed. These custom commands give you the opportunity to perform limited steering of your simulation from within VisIt. More advanced methods of simulation steering will be covered later in this chapter.

Example of simple simulation commands that you might want to expose in the metadata are the “run”, “halt”, “step”. Imagine that you use VisIt to connect to your simulation and you create some plots. Once you are done analyzing a particular time step, you may want to click the “run” button in the **Simulations window** (shown in Figure 5-33) to let your simulation proceed for a while. After your simulation has advanced, you could click the “halt” button to pause it while you investigate features that have developed in the data for the simulation's current time step.

The C-Language `mainloop` function that was created in Section 3.4.2 did not have support for a command callback function. The following code listing shows what the command callback function would look like for a simulation that exposes three simple commands: halt, step, and run. The code listing also shows how the command callback function is registered with `libsim` using the `VisItSetCommandCallback` function. The new command callback function and the change to the `mainloop` function are underlined.



Figure 5-33: VisIt's Simulations window with custom simulation commands.

Listing 5-34: `sim12.c`: C-Language example for installing a command callback function.

```

/* Is the simulation in run mode (not waiting for VisIt input) */
static int    runFlag = 1;
static int    simcycle = 0;
static double simtime = 0.;

/* Callback function for control commands. */
void ControlCommandCallback(const char *cmd,
    int intdata, float floatdata,
    const char *stringdata)
{

```

```

    if(strcmp(cmd, "halt") == 0)
        runFlag = 0;
    else if(strcmp(cmd, "step") == 0)
        simulate_one_timestep();
    else if(strcmp(cmd, "run") == 0)
        runFlag = 1;
}

void mainloop(void)
{
    int blocking, visitstate, err = 0;

    do
    {
        blocking = runFlag ? 0 : 1;
        /* Get input from VisIt or timeout so the simulation can run. */
        visitstate = VisItDetectInput(blocking, -1);

        /* Do different things depending on the output from
        VisItDetectInput. */
        if(visitstate >= -5 && visitstate <= -1)
        {
            fprintf(stderr, "Can't recover from error!\n");
            err = 1;
        }
        else if(visitstate == 0)
        {
            /* There was no input from VisIt, return control to sim. */
            simulate_one_timestep();
        }
        else if(visitstate == 1)
        {
            /* VisIt is trying to connect to sim. */
            if(VisItAttemptToCompleteConnection())
            {
                fprintf(stderr, "VisIt connected\n");
                VisItSetCommandCallback(ControlCommandCallback);
            }
            else
                fprintf(stderr, "VisIt did not connect\n");
        }
        else if(visitstate == 2)
        {
            /* VisIt wants to tell the engine something. */
            runFlag = 0;
            if(!VisItProcessEngineCommand())
            {
                /* Disconnect on an error or closed connection. */
                VisItDisconnect();
                /* Start running again if VisIt closes. */
                runFlag = 1;
            }
        }
    } while(!simulation_done() && err == 0);
}

```

```
}

```

Listing 5-35: sim12.c: C-Language example for returning simulation commands in the metadata.

```
/* Add some custom commands. */
md->numGenericCommands = 3;
sz = sizeof(Visit_SimulationControlCommand) * md->numGenericCommands;
md->genericCommands = (Visit_SimulationControlCommand *)malloc(sz);
memset(md->genericCommands, 0, sz);

md->genericCommands[0].name = strdup("halt");
md->genericCommands[0].argType = VISIT_CMDARG_NONE;
md->genericCommands[0].enabled = 1;

md->genericCommands[1].name = strdup("step");
md->genericCommands[1].argType = VISIT_CMDARG_NONE;
md->genericCommands[1].enabled = 1;

md->genericCommands[2].name = strdup("run");
md->genericCommands[2].argType = VISIT_CMDARG_NONE;
md->genericCommands[2].enabled = 1;
```

Since the Fortran interface, defined in `visitfortransiminterface.c` requires callbacks to be in place when the simulation is linked, the Fortran simulation examples so far have already contained a command callback function. No change is required to the `mainloop` function in the Fortran simulations because the callback is already installed. The command callback function, which is always named `visitcommandcallback` in a Fortran simulation, previously did nothing. The following code example shows how to compare the names of a command coming from a button click in VisIt's **Simulations window** with the names of the supported commands and how to perform the desired action. The Fortran interface provides the `visitstrcmp` function, which is analogous to the C-Language's `strcmp` function in order to make string comparisons easier in Fortran. After the Listing 5-36, Listing 5-37 shows how to use the Fortran interface's `visitaddsimcommand` function to add simulation commands to the metadata.

Listing 5-36: fsim12.f: Fortran language implementation of the command callback function.

```
-----
c visitcommandcallback
-----
      subroutine visitcommandcallback (cmd, lcmd, intdata,
. floatdata, stringdata, lstringdata)
      implicit none
      character*8 cmd, stringdata
      integer      lcmd, lstringdata, intdata
      real         floatdata
      include "visitfortransiminterface.inc"
ccc  SIMSTATE common block
```



```

integer runflag, simcycle
real simtime
common /SIMSTATE/ runflag, simcycle, simtime
c Handle the commands that we define in visitgetmetadata.
if(visitstrcmp(cmd, lcmd, "halt", 4).eq.0) then
    runflag = 0
elseif(visitstrcmp(cmd, lcmd, "step", 4).eq.0) then
    call simulate_one_timestep()
elseif(visitstrcmp(cmd, lcmd, "run", 3).eq.0) then
    runflag = 1
endif
end

```

Listing 5-37: fsm12.f: Fortran language example for returning simulation commands in metadata..

```

integer err
c Add simulation commands
err = visitmdaddsimcommand(handle, "halt", 4,
. VISIT_CMDARG_NONE, 1)
err = visitmdaddsimcommand(handle, "step", 4,
. VISIT_CMDARG_NONE, 1)
err = visitmdaddsimcommand(handle, "run", 3,
. VISIT_CMDARG_NONE, 1)

```

4.6 Data access function for meshes

Now that you've implemented a function to return metadata about the meshes and variables in your simulation, you can write a new data access function to return the actual mesh. Adding a new data access function means that you will be adding a new function pointer to the `visitCallbacks` object. If your simulation is written in Fortran, you must implement the `visitgetmesh` function to return your mesh's data.

The data access function for meshes returns a `Visit_MeshData` object. The `Visit_MeshData` object is a simple structure, defined in `VisitDataInterface_V1.h`, consisting of pointers to structures, which contain data for the different mesh types that VisIt supports. This section will first show how to return the right mesh to VisIt and will then focus on passing different types of meshes back to VisIt so they can be visualized.

4.6.1 Adding a mesh data access function

Adding a mesh data access function means that you have to first write a function and set the `visitCallbacks` object's `GetMesh` member so it points to your function. The mesh data access function takes 2 arguments if you program in C. The first argument is a domain number, which you can use to return smaller pieces of the whole mesh. The second argument is the name of the mesh that VisIt wants to read. The mesh name will be one of the meshes that you added to the metadata. The basic procedure involved in writing

a mesh data access function is to first check the incoming name against the names of the meshes that your simulation is prepared to return and when one is found, return it to VisIt in a `VisIt_MeshData` object. If your mesh data access routine does not recognize the name of the mesh then you can return `NULL` instead of returning a `VisIt_MeshData` object.

Listing 5-38: mesh.c: C-Language example for installing a mesh data access function.

```
VisIt_MeshData *VisItGetMesh(int domain, const char *name)
{
    VisIt_MeshData *mesh = NULL;
    size_t sz = sizeof(VisIt_MeshData);

    if(strcmp(name, "mesh2d") == 0)
    {
        /* Allocate VisIt_MeshData. */
        mesh = (VisIt_MeshData *)malloc(sz);
        memset(mesh, 0, sz);
        /* Make VisIt_MeshData contain a VisIt_RectilinearMesh. */
        sz = sizeof(VisIt_RectilinearMesh);
        mesh->rmesh = (VisIt_RectilinearMesh *)malloc(sz);
        memset(mesh->rmesh, 0, sz);

        /* Fill in the attributes of the VisIt_RectilinearMesh. */
    }
    else if(strcmp(name, "mesh3d") == 0)
    {
        /* Allocate VisIt_MeshData. */
        mesh = (VisIt_MeshData *)malloc(sz);
        memset(mesh, 0, sz);
        /* Make VisIt_MeshData contain a VisIt_CurvilinearMesh. */
        sz = sizeof(VisIt_CurvilinearMesh);
        mesh->cmesh = (VisIt_CurvilinearMesh *)malloc(sz);
        memset(mesh->cmesh, 0, sz);

        /* Fill in the attributes of the VisIt_CurvilinearMesh. */
    }

    return mesh;
}

VisIt_SimulationCallback visitCallbacks =
{
    &VisItGetMetaData,
    &VisItGetMesh,
    NULL, /* GetMaterial */
    NULL, /* GetSpecies */
    NULL, /* GetScalar */
    NULL, /* GetCurve */
    NULL, /* GetMixedScalar */
    NULL /* GetDomainList */
}
```

```
};
```

Remember that when writing a Fortran simulation, all of the data access functions must be defined before you can actually link your simulation. That means that up until now, the Fortran example programs have been using a simple implementation of the `visitgetmesh` function, which did nothing. The rest of this section will cover how to add an appropriate, working implementation of the `visitgetmesh` data access function.

Listing 5-39: `fmesh.f`: Fortran language example of a mesh data access function.

```

c-----
c visitgetmesh
c-----
      integer function visitgetmesh(handle, domain, name, lname)
      implicit none
      character*8 name
      integer      handle, domain, lname
      include "visitfortransiminterface.inc"
      integer m
      m = VISIT_ERROR
      if(visitstrcmp(name, lname, "mesh2d", 6).eq.0) then
c Create a rectilinear mesh here
      elseif(visitstrcmp(name, lname, "mesh3d", 6).eq.0) then
c Create a curvilinear mesh here
      endif
      visitgetmesh = m
      end

```

4.6.2 Rectilinear meshes

Rectilinear meshes can be returned by the mesh data access function by allocating a `VisIt_RectilinearMesh` object and inserting it into the returned `VisIt_MeshData` object. Don't forget to set the `VisIt_MeshData`'s `meshType` member to `VISIT_MESHTYPE_RECTILINEAR`. Once you've allocated the `VisIt_RectilinearMesh` object, start initializing its members using information about the mesh. For starters, set the `ndims` member to 2 or 3, depending on the number of dimensions occupied by the mesh. Next, set the components of the `dims` array so `VisIt` will know the size of each of the coordinate arrays. The values that you store in the `dims` array are the number of mesh nodes in each dimension. The `dims` array must always contain 3 elements. If you are creating a 2D mesh, set the last element to one.

After setting the elements in the `dims` array to the right values for your mesh, you can set the `baseIndex` member, which is an offset in X,Y,Z that will be added to your mesh's zone numbers and node numbers when `VisIt` displays information about your mesh. You can leave these values set at zero. However, when you want to create a multi-domain mesh

that has global zone and node numbers, you should set the values for `baseIndex`. Global node and zone numbers can make it easier to think of your domain-decomposed mesh as a single entity by making VisIt features such as `pick` return global node or zone numbers instead of per-domain node or zone numbers.

Now that you've set the values in the `VisIt_RectilinearMesh` object that indicate its logical size, you must tell VisIt whether the mesh has ghost zones. The `VisIt_RectilinearMesh` object indicates whether there are ghost zones by using the values stored in the `minRealIndex` and `maxRealIndex` members. If you initialized the entire object to zeroes using the `memset` function then you can omit code to set the values in the `minRealIndex` array. If your mesh has no ghost zones then you can set the elements in the `maxRealIndex` array to the number of cells in each dimension. If your mesh has ghost zones in any of the dimensions then be sure that you add 1 to the values stored in the `minRealIndex` array for the dimensions that have ghost zones. Also be sure to subtract 1 from the elements in the `maxRealIndex` array for the dimensions that have ghost zones.

The final stage in specifying your rectilinear mesh is to provide VisIt with the coordinate arrays. The `VisIt_RectilinearMesh` object contains three data array objects that can be used to contain references to your simulation's X,Y,Z mesh coordinate arrays or they can contain copies of those arrays if you do not want to share them with VisIt. The `VisIt_CreateDataArrayFromFloat` utility function is used to store a reference to the simulation-owned mesh coordinate arrays into the `VisIt_RectilinearMesh` object. The coordinate arrays will be the same arrays, thus the same memory locations, as the simulation's coordinate arrays if you use the `VISIT_OWNER_SIM` flag when creating data arrays for VisIt. If you instead pass `VISIT_OWNER_VISIT` then VisIt will create a copy of the coordinate array, requiring additional memory. Copying the arrays when giving them to VisIt is a good choice if your data are not readily stored in a format that VisIt can process. If you use the `VISIT_OWNER_VISIT` flag then VisIt will free the data arrays when they are no longer required.

Listing 5-40: `mesh.c`: C-Language example for returning a rectilinear mesh.

```
/* Simulation mesh */
float mesh_x[] = {0., 1., 2.5, 5.};
float mesh_y[] = {0., 2., 2.25, 2.55, 5.};
int mesh_dims[] = {4, 5, 1};
int mesh_ndims = 2;

VisIt_MeshData *VisItGetMesh(int domain, const char *name)
{
    VisIt_MeshData *mesh = NULL;
    size_t sz = sizeof(VisIt_MeshData);

    if(strcmp(name, "mesh2d") == 0)
    {
        /* Allocate VisIt_MeshData. */
        mesh = (VisIt_MeshData *)malloc(sz);
    }
}
```

```

memset(mesh, 0, sz);
/* Make VisIt_MeshData contain a VisIt_RectilinearMesh. */
sz = sizeof(VisIt_RectilinearMesh);
mesh->rmesh = (VisIt_RectilinearMesh *)malloc(sz);
memset(mesh->rmesh, 0, sz);

/* Tell VisIt which mesh object to use. */
mesh->meshType = VISIT_MESHTYPE_RECTILINEAR;

/* Set the mesh's number of dimensions. */
mesh->rmesh->ndims = mesh_ndims;

/* Set the mesh dimensions. */
mesh->rmesh->dims[0] = mesh_dims[0];
mesh->rmesh->dims[1] = mesh_dims[1];
mesh->rmesh->dims[2] = mesh_dims[2];

mesh->rmesh->baseIndex[0] = 0;
mesh->rmesh->baseIndex[1] = 0;
mesh->rmesh->baseIndex[2] = 0;

mesh->rmesh->minRealIndex[0] = 0;
mesh->rmesh->minRealIndex[1] = 0;
mesh->rmesh->minRealIndex[2] = 0;
mesh->rmesh->maxRealIndex[0] = mesh_dims[0]-1;
mesh->rmesh->maxRealIndex[1] = mesh_dims[1]-1;
mesh->rmesh->maxRealIndex[2] = mesh_dims[2]-1;

/* Let VisIt use simulation's copy of the mesh coordinates. */
mesh->rmesh->xcoords = VisIt_CreateDataArrayFromFloat(
    VISIT_OWNER_SIM, mesh_x);
mesh->rmesh->ycoords = VisIt_CreateDataArrayFromFloat(
    VISIT_OWNER_SIM, mesh_y);
}

return mesh;
}

```

The Fortran interface provides the `visitmeshrectilinear` function to create a rectilinear mesh that can be passed back to VisIt. The `visitmeshrectilinear` function essentially packages up the code from the C-Language example, making it possible to dynamically create a `VisIt_RectilinearMesh` object and populate its members. The data arrays that make up the rectilinear mesh in the upcoming Fortran example are stored in a Fortran common block, making the data accessible to the `simulate_one_timestep` function and the `visitgetmesh` function. If you store your data in common blocks, it is easy to make it accessible to VisIt.

Listing 5-41: `fmesh.f`: Fortran language example for returning a rectilinear mesh.

```

subroutine simulate_one_timestep()

```

```

ccc  RECTMESH common block
      integer NX, NY
      parameter (NX = 4)
      parameter (NY = 5)
      real rmx(NX), rmy(NY)
      integer rmdims(3), rmdims
      common /RECTMESH/ rmdims, rmdims, rmx, rmy
      save /RECTMESH/
c Initial rectilinear mesh
      data rmdims /2/
      data rmdims /4, 5, 1/
      data rmx/0., 1., 2.5, 5./
      data rmy/0., 2., 2.25, 2.55, 5./
c Simulate one time step
      end

c-----
c visitgetmesh
c-----
      integer function visitgetmesh(handle, domain, name, lname)
      implicit none
      character*8 name
      integer      handle, domain, lname
      include "visitfortransiminterface.inc"
ccc  RECTMESH common block (shared with simulate_one_timestep)
      integer NX, NY
      parameter (NX = 4)
      parameter (NY = 5)
      real rmx(NX), rmy(NY)
      integer rmdims(3), rmdims
      common /RECTMESH/ rmdims, rmdims, rmx, rmy
ccc  local variables
      integer m, baseindex(3), minrealindex(3), maxrealindex(3)
      real rmz

      m = VISIT_ERROR
      if(visitstrcmp(name, lname, "mesh2d", 6).eq.0) then
          baseindex(1) = 1
          baseindex(2) = 1
          baseindex(3) = 1
          minrealindex(1) = 0
          minrealindex(2) = 0
          minrealindex(3) = 0
          maxrealindex(1) = rmdims(1)-1
          maxrealindex(2) = rmdims(2)-1
          maxrealindex(3) = rmdims(3)-1
c Create a rectilinear rmesh here
          m = visitmeshrectilinear(handle, baseindex, minrealindex,
      .           maxrealindex, rmdims, rmdims, rmx, rmy, rmz)
          elseif(visitstrcmp(name, lname, "mesh3d", 6).eq.0) then
c Create a curvilinear mesh here
      endif
      visitgetmesh = m

```

end

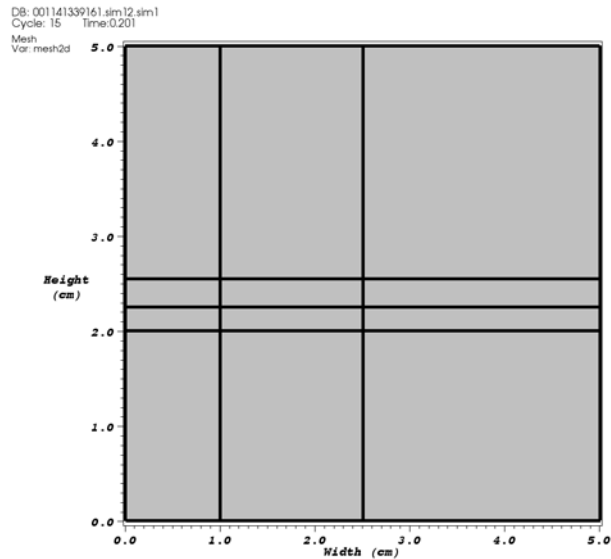


Figure 5-42: 2D rectilinear mesh returned by the previous code examples.

4.6.3 Curvilinear meshes

Curvilinear meshes can be returned from your mesh data access function by creating a `Visit_CurvilinearMesh` object and storing it inside a `Visit_MeshData` object. With the exception of using a different structure name in C programs, the procedure for creating a curvilinear mesh is exactly the same as that for creating a rectilinear mesh. Remember that the only difference that `Visit` recognizes between the two mesh types is the size of the coordinate arrays. A curvilinear mesh must have the `X,Y,Z` coordinates of each node in the mesh explicitly provided, whereas most of the coordinates are implicitly defined in a rectilinear mesh. Since the code for handling curvilinear meshes is so similar to that for handling rectilinear meshes, refer to Section 4.6.2 for more detail on setting values into the `Visit_CurvilinearMesh`.

Listing 5-43: `mesh.c`: C-Language example for returning a curvilinear mesh.

```

/* Curvilinear mesh */
float cmesh_x[2][3][4] = {
    {{0.,1.,2.,3.},{0.,1.,2.,3.}, {0.,1.,2.,3.}},
    {{0.,1.,2.,3.},{0.,1.,2.,3.}, {0.,1.,2.,3.}}
};
float cmesh_y[2][3][4] = {
    {{0.5,0.,0.,0.5},{1.,1.,1.,1.}, {1.5,2.,2.,1.5}},

```

```

    {{0.5,0.,0.,0.5},{1.,1.,1.,1.}, {1.5,2.,2.,1.5}}
};
float cmesh_z[2][3][4] = {
    {{0.,0.,0.,0.},{0.,0.,0.,0.},{0.,0.,0.,0.}},
    {{1.,1.,1.,1.},{1.,1.,1.,1.},{1.,1.,1.,1.}}
};
int cmesh_dims[] = {4, 3, 2};
int cmesh_ndims = 3;

VisIt_MeshData *VisItGetMesh(int domain, const char *name)
{
    VisIt_MeshData *mesh = NULL;
    size_t sz = sizeof(VisIt_MeshData);
    if(strcmp(name, "mesh3d") == 0)
    {
        /* Allocate VisIt_MeshData. */
        mesh = (VisIt_MeshData *)malloc(sz);
        memset(mesh, 0, sz);
        /* Make VisIt_MeshData contain a VisIt_CurvilinearMesh. */
        sz = sizeof(VisIt_CurvilinearMesh);
        mesh->cmesh = (VisIt_CurvilinearMesh *)malloc(sz);
        memset(mesh->cmesh, 0, sz);

        /* Tell VisIt which mesh object to use. */
        mesh->meshType = VISIT_MESHTYPE_CURVILINEAR;

        /* Set the mesh's number of dimensions. */
        mesh->cmesh->ndims = cmesh_ndims;

        /* Set the mesh dimensions. */
        mesh->cmesh->dims[0] = cmesh_dims[0];
        mesh->cmesh->dims[1] = cmesh_dims[1];
        mesh->cmesh->dims[2] = cmesh_dims[2];

        mesh->cmesh->baseIndex[0] = 0;
        mesh->cmesh->baseIndex[1] = 0;
        mesh->cmesh->baseIndex[2] = 0;

        mesh->cmesh->minRealIndex[0] = 0;
        mesh->cmesh->minRealIndex[1] = 0;
        mesh->cmesh->minRealIndex[2] = 0;
        mesh->cmesh->maxRealIndex[0] = cmesh_dims[0]-1;
        mesh->cmesh->maxRealIndex[1] = cmesh_dims[1]-1;
        mesh->cmesh->maxRealIndex[2] = cmesh_dims[2]-1;

        /* Let VisIt use simulation's copy of the mesh coordinates. */
        mesh->cmesh->xcoords = VisIt_CreateDataArrayFromFloat(
            VISIT_OWNER_SIM, (float *)cmesh_x);
        mesh->cmesh->ycoords = VisIt_CreateDataArrayFromFloat(
            VISIT_OWNER_SIM, (float *)cmesh_y);
        mesh->cmesh->zcoords = VisIt_CreateDataArrayFromFloat(
            VISIT_OWNER_SIM, (float *)cmesh_z);
    }
}

```



```

    return mesh;
}

```

The Fortran interface provides the `visitmeshcurvilinear` function to create a rectilinear mesh that can be passed back to VisIt. The `visitmeshcurvilinear` function essentially packages up the code from the C-Language example, making it possible to dynamically create a `Visit_CurvilinearMesh` object and populate its members. The data arrays that make up the curvilinear mesh in the upcoming Fortran example are stored in a Fortran common block, making the data accessible to the `simulate_one_timestep` function and the `visitgetmesh` function.

Listing 5-44: `fmesh.f`: Fortran language example for returning a curvilinear mesh.

```

    subroutine simulate_one_timestep()
ccc  CURVMESH common block
    integer CNX, CNY, CNZ
    parameter (CNX = 4)
    parameter (CNY = 3)
    parameter (CNZ = 2)
    integer cmdims(3), cmndims
    real cmx(CNX,CNY,CNZ), cmy(CNX,CNY,CNZ), cmz(CNX,CNY,CNZ)
    common /CURVMESH/ cmdims, cmndims, cmx, cmy, cmz
    save /CURVMESH/
c  Curvilinear mesh data
    data cmx/0.,1.,2.,3., 0.,1.,2.,3., 0.,1.,2.,3.,
      . 0.,1.,2.,3., 0.,1.,2.,3., 0.,1.,2.,3./
    data cmy/0.5,0.,0.,0.5, 1.,1.,1.,1., 1.5,2.,2.,1.5,
      . 0.5,0.,0.,0.5, 1.,1.,1.,1., 1.5,2.,2.,1.5/
    data cmz/0.,0.,0.,0., 0.,0.,0.,0., 0.,0.,0.,0.,
      . 1.,1.,1.,1., 1.,1.,1.,1., 1.,1.,1.,1./
    data cmndims /3/
    data cmdims/CNX,CNY,CNZ/
c  Simulate one time step
    end

c-----
c  visitgetmesh
c-----

    integer function visitgetmesh(handle, domain, name, lname)
    implicit none
    character*8 name
    integer      handle, domain, lname
    include "visitfortransiminterface.inc"
ccc  CURVMESH common block (shares with simulate_one_timestep)
    integer CNX, CNY, CNZ
    parameter (CNX = 4)
    parameter (CNY = 3)
    parameter (CNZ = 2)
    integer cmdims(3), cmndims
    real cmx(CNX,CNY,CNZ), cmy(CNX,CNY,CNZ), cmz(CNX,CNY,CNZ)
    common /CURVMESH/ cmdims, cmndims, cmx, cmy, cmz

```

```

ccc  local variables
integer m, baseindex(3), minrealindex(3), maxrealindex(3)

m = VISIT_ERROR
if(visitstrcmp(name, lname, "mesh3d", 6).eq.0) then
  baseindex(1) = 1
  baseindex(2) = 1
  baseindex(3) = 1
  minrealindex(1) = 0
  minrealindex(2) = 0
  minrealindex(3) = 0
  maxrealindex(1) = cmdims(1)-1
  maxrealindex(2) = cmdims(2)-1
  maxrealindex(3) = cmdims(3)-1
  c Create a curvilinear mesh here
  m = visitmeshcurvilinear(handle, baseindex, minrealindex,
    .      maxrealindex, cmdims, cmdims, cmx, cmx, cmz)
endif
visitgetmesh = m
end

```

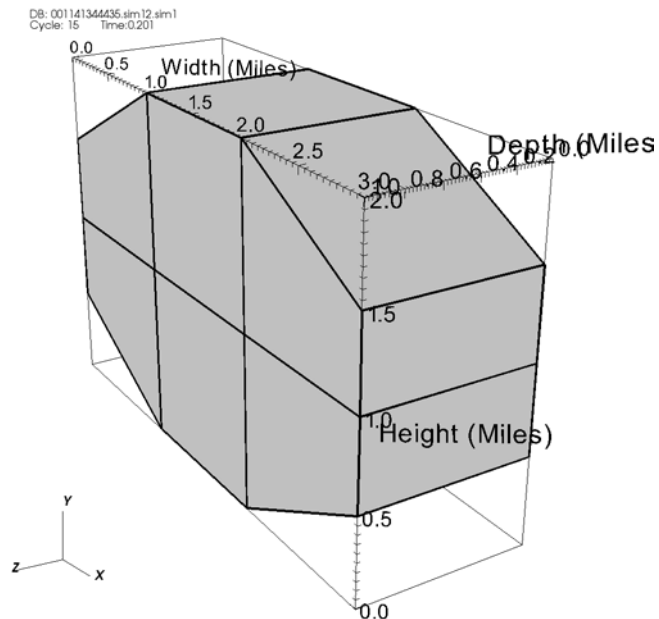


Figure 5-45: 3D curvilinear mesh returned by the previous code examples

4.6.4 Point meshes

Point meshes can be returned by the mesh data access function by allocating a `VisIt_PointMesh` object and inserting it into the returned `VisIt_MeshData` object. Don't forget to set the `VisIt_MeshData`'s `meshType` member to

VISIT_MESHTYPE_POINT. Once you've allocated the `Visit_PointMesh` object, start initializing its members using information about the mesh. Point meshes contain relatively few elements - little more than a list of vertices. Be sure to set the `ndims` member, which tells `Visit` how many of the coordinate arrays that your point mesh will use: 2 or 3. After setting the number of dimensions, set the `nnodes` member - the number of nodes in the point mesh. Finally, use the `Visit_CreateDataArrayFromFloat` function to set the coordinate arrays for your point mesh. The coordinate arrays can either be owned by the simulation (`VISIT_OWNER_SIM`), in which case `Visit` will not free the arrays. If you use `VISIT_OWNER_VISIT` then `Visit` will free the arrays once they are no longer required.

Listing 5-46: `point.c`: C-Language example for returning a point mesh.

```
#define NPTS 100
float angle = 0.;
int pmesh_ndims = 3;
float pmesh_x[NPTS], pmesh_y[NPTS], pmesh_z[NPTS];
void simulate_one_timestep(void)
{
    int i;
    for(i = 0; i < NPTS; ++i)
    {
        float t = ((float)i) / ((float)(NPTS-1));
        float a = 3.14159 * 10. * t;
        pmesh_x[i] = t * cos(a + (0.5 + 0.5 * t) * angle);
        pmesh_y[i] = t * sin(a + (0.5 + 0.5 * t) * angle);
        pmesh_z[i] = t;
    }
    angle = angle + 0.05;
}
Visit_MeshData *VisitGetMesh(int domain, const char *name)
{
    Visit_MeshData *mesh = NULL;
    size_t sz = sizeof(Visit_MeshData);

    if(strcmp(name, "point3d") == 0)
    {
        /* Allocate Visit_MeshData. */
        mesh = (Visit_MeshData *)malloc(sz);
        memset(mesh, 0, sz);
        /* Make Visit_MeshData contain a Visit_PointMesh. */
        sz = sizeof(Visit_PointMesh);
        mesh->pmesh = (Visit_PointMesh *)malloc(sz);
        memset(mesh->pmesh, 0, sz);

        /* Tell Visit which mesh object to use. */
        mesh->meshType = VISIT_MESHTYPE_POINT;

        /* Set the mesh's number of dimensions. */
        mesh->pmesh->ndims = pmesh_ndims;
    }
}
```

```

/* Set the number of points in the mesh. */
mesh->pmesh->nnodes = NPTS;

/* Let VisIt use simulation's copy of the mesh coordinates. */
mesh->pmesh->xcoords = VisIt_CreateDataArrayFromFloat(
    VISIT_OWNER_SIM, (float *)pmesh_x);
mesh->pmesh->ycoords = VisIt_CreateDataArrayFromFloat(
    VISIT_OWNER_SIM, (float *)pmesh_y);
mesh->pmesh->zcoords = VisIt_CreateDataArrayFromFloat(
    VISIT_OWNER_SIM, (float *)pmesh_z);
}
return mesh;
}

```

The Fortran interface provides the `visitmeshpoint` function so you can create a `VisIt_PointMesh` object that can be returned to VisIt. The `visitmeshpoint` function takes 6 arguments. The first argument is an integer handle to the mesh object that was passed into the `visitgetmesh` function. The second argument allows you to set the number of dimensions that your point mesh will use: 2 or 3. The third argument lets you set the number of nodes in your point mesh. The final three REAL arguments contain the X,Y,Z coordinates, respectively.

Listing 5-47: `fpoint.f`: Fortran language example for returning a point mesh.

```

subroutine simulate_one_timestep()
ccc POINTMESH common block (shared with visitgetmesh)
integer NPTS
parameter (NPTS = 100)
real pmx(NPTS), pmy(NPTS), pmz(NPTS), angle
integer pmndims, pmnnodes
common /RECTMESH/ pmx, pmy, pmz, pmndims, pmnnodes, angle
ccc local variables
real a, t
c Simulate one time step
pmndims = 3
pmnnodes = NPTS
do 10000 i = 0,NPTS-1
    t = float(i) / float(NPTS-1)
    a = 3.14159 * 10. * t
    pmx(i+1) = t * cos(a + (0.5 + 0.5 * t) * angle);
    pmy(i+1) = t * sin(a + (0.5 + 0.5 * t) * angle);
    pmz(i+1) = t
10000 continue
angle = angle + 0.05
end

integer function visitgetmesh(handle, domain, name, lname)
implicit none
character*8 name
integer handle, domain, lname

```

```

include "visitfortransiminterface.inc"
ccc POINTMESH common block (shared with simulate_one_timestep)
integer NPTS
parameter (NPTS = 100)
real pmx(NPTS), pmy(NPTS), pmz(NPTS), angle
integer pmndims, pmnnodes
common /RECTMESH/ pmx, pmy, pmz, pmndims, pmnnodes, angle
ccc local variables
integer m

m = VISIT_ERROR
if(visitstrcmp(name, lname, "point3d", 7).eq.0) then
c Create a point mesh here
m = visitmeshpoint(handle, pmndims, pmnnodes, pmx, pmy, pmz)
endif
visitgetmesh = m
end

```

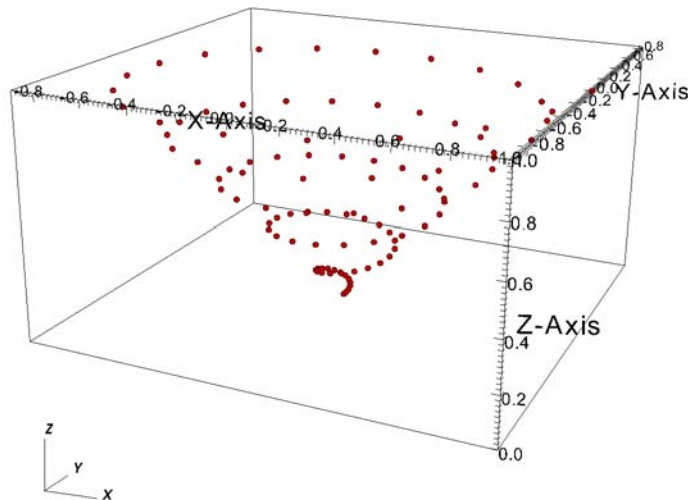


Figure 5-48: 3D point mesh returned by the previous code examples

4.6.5 Unstructured meshes

Unstructured meshes can be returned by the mesh data access function by allocating a `Visit_UnstructuredMesh` object and inserting it into the returned `Visit_MeshData` object. Don't forget to set the `Visit_MeshData`'s `meshType` member to `VISIT_MESHTYPE_UNSTRUCTURED`. Once you've allocated the `Visit_UnstructuredMesh` object, start initializing its members using information

about the mesh. The first member that you set should be the `ndims` member, which tells `VisIt` if the mesh is 2D or 3D. Set the `ndims` member to 2 for a 2D mesh and 3 for a 3D mesh. Next, set the `nnodes` and `nzones` members so `VisIt` will know how many nodes make up the mesh and how many zones are connected out of that set of nodes.

You can specify the mesh's coordinates by using the `VisIt_CreateDataArrayFromFloat` function to create data arrays for the `xcoords`, `ycoords`, and `zcoords` members. You can use the simulation's data arrays by passing `VISIT_OWNER_SIM` or you can create copies of them by passing `VISIT_OWNER_VISIT`.

Now you must tell `VisIt` whether the mesh has ghost zones. The `VisIt_UnstructuredMesh` object indicates whether there are ghost zones by using the values stored in the `firstRealZone` and `lastRealZone` members. You can use those members to indicate that the first *N* zones are ghost zones and that the last *M* zones are ghost zones. If your mesh has no ghost zones then you can set the `lastRealZone` member to the number of zones in the mesh minus one. If your mesh has ghost zones then be sure to set both `firstRealZone` and `lastRealZone` so they tell `VisIt` the indices of the zones in the zone list where the real zones begin and end. If you do not set these members then `VisIt` may become confused.

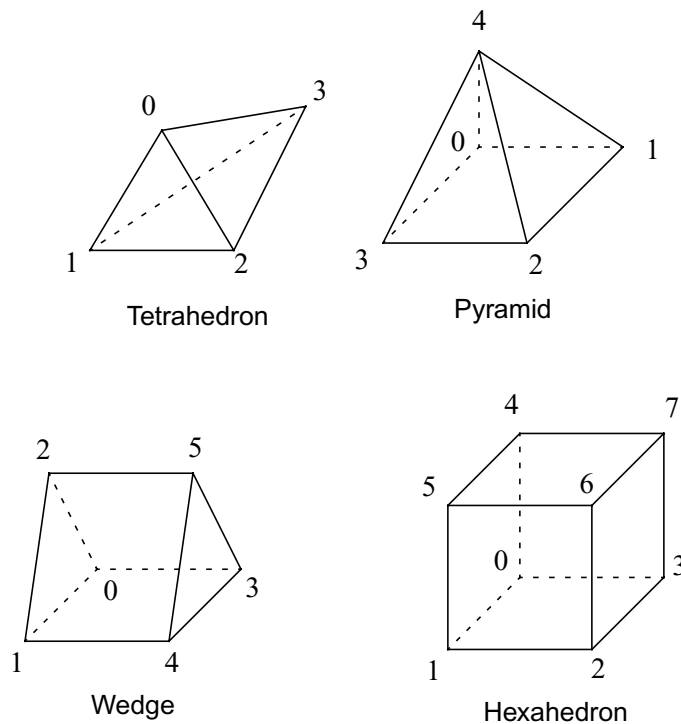


Figure 5-49: Node ordering for 3D unstructured zone types

The final step in creating an unstructured mesh is providing the zone connectivity information. Connectivity information indicates how nodes are connected into zones of

varying types. The connectivity information is stored in a linear array of integers in sequences that list the zone type, followed by the node indices being used for that zone. The node indices should begin at node zero, even in languages where the first array element is one, such as in Fortran. This pattern is repeated until all zones in the mesh have been identified. Figure 5-49 shows the node ordering that must be used to create cells for an unstructured mesh. Note that the node ordering (VTK's node ordering) for all zone types is the same as for creating Silo files, except for the wedge zone type. You can use the `VisIt_CreateDataArrayFromInt` function to create an integer data array that can be passed to VisIt. After supplying the connectivity array, be sure to set the `connectivityLen` member so the length of the connectivity array since VisIt uses that value to determine when to stop iterating through the connectivity array.

Listing 5-50: unstructured.c: C-Language example for returning an unstructured mesh.

```
float umx[] = {0.,2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,1.,2.,4.,4.};
float umy[] = {0.,0.,0.,0.,2.,2.,2.,2.,4.,4.,4.,4.,6.,0.,0.,0.};
float umz[] = {2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,0.,1.,4.,2.,0.};
/* Connectivity */
int connectivity[] = {
    VISIT_CELL_HEX,    0,1,2,3,4,5,6,7,    /* hex,    zone 1 */
    VISIT_CELL_HEX,    4,5,6,7,8,9,10,11, /* hex,    zone 2 */
    VISIT_CELL_PYR,    8,9,10,11,12,     /* pyramid, zone 3 */
    VISIT_CELL_WEDGE,  1,14,5,2,15,6,     /* wedge,  zone 4 */
    VISIT_CELL_TET,    1,14,13,5         /* tet,    zone 5 */
};
int lconnectivity = sizeof(connectivity) / sizeof(int);
int umnnodes = 16;
int umnzones = 5;

VisIt_MeshData *VisItGetMesh(int domain, const char *name)
{
    VisIt_MeshData *mesh = NULL;
    size_t sz = sizeof(VisIt_MeshData);

    if(strcmp(name, "unstructured3d") == 0)
    {
        /* Allocate VisIt_MeshData. */
        mesh = (VisIt_MeshData *)malloc(sz);
        memset(mesh, 0, sz);
        /* Make VisIt_MeshData contain a VisIt_PointMesh. */
        sz = sizeof(VisIt_UnstructuredMesh);
        mesh->umesh = (VisIt_UnstructuredMesh *)malloc(sz);
        memset(mesh->umesh, 0, sz);

        /* Tell VisIt which mesh object to use. */
        mesh->meshType = VISIT_MESHTYPE_UNSTRUCTURED;

        /* Set the mesh's number of dimensions. */
        mesh->umesh->ndims = 3;
        /* Set the number of nodes and zones in the mesh. */
        mesh->umesh->nnodes = umnnodes;
    }
}
```

```

mesh->umesh->nzones = umnzones;

/* Set the indices for the first and last real zones. */
mesh->umesh->firstRealZone = 0;
mesh->umesh->lastRealZone = umnzones-1;

/* Let VisIt use simulation's copy of the mesh coordinates. */
mesh->umesh->xcoords = VisIt_CreateDataArrayFromFloat(
    VISIT_OWNER_SIM, umx);
mesh->umesh->ycoords = VisIt_CreateDataArrayFromFloat(
    VISIT_OWNER_SIM, umy);
mesh->umesh->zcoords = VisIt_CreateDataArrayFromFloat(
    VISIT_OWNER_SIM, umz);

/* Let VisIt use the simulation's copy of the connectivity. */
mesh->umesh->connectivity = VisIt_CreateDataArrayFromInt(
    VISIT_OWNER_SIM, connectivity);
mesh->umesh->connectivityLen = lconnectivity;
}
return mesh;
}

```

The Fortran interface provides the `visitmeshunstructured` function for creating unstructured meshes and returning them to VisIt. The `visitmeshunstructured` function takes 11 arguments. The first argument is an integer handle to the mesh object that will contain your unstructured mesh. This handle was passed to your `visitgetmesh` function from `visitfortransiminterface.c`. The next argument is `ndims`, which lets you tell VisIt whether your unstructured mesh is 2D or 3D. The third argument is the `nnodes` argument, indicating the number of nodes in your mesh's coordinate arrays. The fourth argument, `nzones`, tells VisIt how many zones are contained in your mesh. The fifth argument is `firstrealzone`, which is a ghost zone argument indicating the index of the mesh's first real zone. The first real zone is set to zero in many cases. The sixth argument is the `lastrealzone` argument, which lets you indicate the index of the last real zone, above which are found the ghost zones. The next three arguments (7,8,9) let you specify the mesh's X,Y,Z coordinates in 32-bit floating point form. The tenth argument is the length of the connectivity array. The final argument is an integer zone connectivity array, which tells VisIt how to connect your mesh's nodes into zones.

Listing 5-51: `funstructured.f`: Fortran language example for returning an unstructured mesh.

```

subroutine simulate_one_timestep()
implicit none
include "visitfortransiminterface.inc"
ccc UNSTRUCTURED common block (shared with visitgetmesh)
integer NNODES, NZONES, LCONN
parameter (NNODES = 16)
parameter (NZONES = 5)
parameter (LCONN = 36)

```



```

real umx(NNODES), umy(NNODES), umz(NNODES)
integer connectivity(LCONN)
common /UNSTRUCTURED/ umx, umy, umz, connectivity
save /UNSTRUCTURED/
c Data values
data umx/0.,2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,1.,2.,4.,4./
data umy/0.,0.,0.,0.,2.,2.,2.,2.,4.,4.,4.,4.,6.,0.,0.,0./
data umz/2.,2.,0.,0.,2.,2.,0.,0.,2.,2.,0.,0.,1.,4.,2.,0./
data connectivity/VISIT_CELL_HEX, 0,1,2,3,4,5,6,7,
. VISIT_CELL_HEX, 4,5,6,7,8,9,10,11,
. VISIT_CELL_PYR, 8,9,10,11,12,
. VISIT_CELL_WEDGE, 1,14,5,2,15,6,
. VISIT_CELL_TET, 1,14,13,5/
end

-----
c visitgetmesh
-----

integer function visitgetmesh(handle, domain, name, lname)
implicit none
character*8 name
integer handle, domain, lname
include "visitfortransiminterface.inc"
ccc UNSTRUCTURED common block (shared with simulate_one_timestep)
integer NNODES, NZONES, LCONN
parameter (NNODES = 16)
parameter (NZONES = 5)
parameter (LCONN = 36)
real umx(NNODES), umy(NNODES), umz(NNODES)
integer connectivity(LCONN)
common /UNSTRUCTURED/ umx, umy, umz, connectivity
ccc local variables
integer m

m = VISIT_ERROR
if(visitstrcmp(name, lname, "unstructured3d", 14).eq.0) then
c Create an unstructured mesh here
m = visitmeshunstructured(handle, 3, NNODES, NZONES, 0,
. NZONES-1, umx, umy, umz, LCONN, connectivity)
endif
visitgetmesh = m
end

```

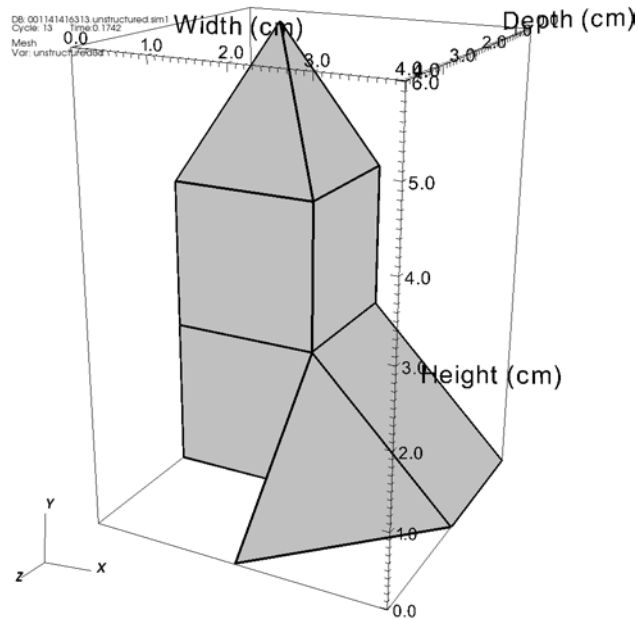


Figure 5-52: 3D unstructured mesh returned by the previous code examples

4.7 Data access function for scalars

This chapter has so far shown how to instrument a simulation code so VisIt can connect to it and read out meshes so they can be plotted. This section will illustrate how to add a data access function that lets VisIt access your simulation's scalar data. Reading scalar data requires a new data access function. Adding a new data access function means that you will be adding a new function pointer to the `visitCallbacks` object. If your simulation is written in Fortran, you must implement the `visitgetscalar` function to return your simulation's scalar data.

The data access function for scalars returns a `Visit_ScalarData` object. The `Visit_ScalarData` object is a simple structure, defined in `VisitDataInterface_V1.h`, consisting of little more than a data array containing the scalar values. This section will show how to return your simulation's scalar data so they can be visualized in VisIt.

4.7.1 Adding a scalar data access function

Adding a scalar data access function means that you have to first write a function and set the `visitCallbacks` object's `GetScalar` member so it points to your function. The scalar data access function takes 2 arguments if you program in C. The first argument is a domain number, which you can use to return scalar data for a smaller piece of the whole mesh. The second argument is the name of the scalar that VisIt wants to read. The scalar

name will be one of the scalars that you added to the metadata. The basic procedure involved in writing a scalar data access function is to first check the incoming name against the names of the scalars that your simulation is prepared to return and when one is found, return it to VisIt in a `Visit_ScalarData` object. If your scalar data access routine does not recognize the name of the scalar then you can return `NULL` instead of returning a `Visit_ScalarData` object.

Listing 5-53: scalar.c: C-Language example for installing a scalar data access function.

```

Visit_ScalarData *VisitGetScalar(int domain, const char *name)
{
    size_t sz = sizeof(Visit_ScalarData);
    Visit_ScalarData *scalar = (Visit_ScalarData*)malloc(sz);
    memset(scalar, 0, sz);

    if(strcmp(name, "zonal") == 0)
        /* Make scalar return the zonal array. */
    else if(strcmp(name, "nodal") == 0)
        /* Make scalar return the nodal array. */
    else
    {
        free(scalar);
        scalar = NULL;
    }
    return scalar;
}

Visit_SimulationCallback visitCallbacks =
{
    &VisitGetMetaData,
    &VisitGetMesh,
    NULL, /* GetMaterial */
    NULL, /* GetSpecies */
    &VisitGetScalar, /* GetScalar */
    NULL, /* GetCurve */
    NULL, /* GetMixedScalar */
    NULL /* GetDomainList */
};

```

The Fortran interface does not require you to install functions into the `visitCallbacks` structure because that is taken care for you by the `libsimg` library. In order to return scalar data from a Fortran simulation, you must implement the `visitgetscalar` function. The `visitgetscalar` function is the Fortran interface's scalar data access function. The function takes four arguments. The first argument is a handle to a scalar data object. The second argument is the domain number for which VisIt wants the specified scalar. The domain argument can be ignored if your simulation only has one domain per processor. The third and fourth arguments are the name of the scalar to return and the length of that name string, respectively. As in the C interface, the `visitgetscalar` function must check the incoming names against the

names of the scalars that the simulation has exposed to VisIt via metadata. You can use the `visitstrcmp` function to match the incoming name against the names of the known scalars.

Listing 5-54: `fscalar.f`: Fortran language example of a scalar data access function.

```

-----
c visitgetscalar
-----
      integer function visitgetscalar(handle, domain, name, lname)
      implicit none
      character*8 name
      integer      handle, domain, lname
      include "visitfortransiminterface.inc"
      integer m
      m = VISIT_ERROR
      if(visitstrcmp(name, lname, "zonal", 5).eq.0) then
c Pass the scalar to VisIt, setting m
      elseif(visitstrcmp(name, lname, "nodal", 5).eq.0) then
c Pass the scalar to VisIt, setting m
      endif
      visitgetscalar = m
      end

```

4.7.2 Passing a simulation's data array

The `VisIt_ScalarData` object contains only two items: the length of the data array, and a data array object that points to the scalar data. Both the C and Fortran interfaces provide multiple functions for passing a simulation's data into the `VisIt_ScalarData` object's data array. There are multiple functions to account for multiple variable types since a `VisIt_ScalarData` object's data array can contain char, int, float, and double scalar arrays. The C interface provides the `VisIt_CreateDataArrayFromChar`, `VisIt_CreateDataArrayFromInt`, `VisIt_CreateDataArrayFromFloat`, and `VisIt_CreateDataArrayFromDouble` functions to pass simulation data into the `VisIt_ScalarData` object. Each of the functions accepts two arguments: an owner and a pointer to the scalar data. The owner flag indicates whether or not VisIt will be responsible for freeing the scalar data array when it is no longer needed. If you pass `VISIT_OWNER_SIM` then VisIt will never free the data because the simulation owns the scalar's memory. If you pass `VISIT_OWNER_VISIT` then VisIt will free the scalar's memory when the scalar is no longer needed.

Listing 5-55: `scalar.c`: C-Language example for returning a scalar variable.

```

int   rmesh_dims[] = {4, 5, 1};
float zonal[]     = {1.,2.,3.,4.,5.,6.,7.,8.,9.,10.,11.,12.};
int   cmesh_dims[] = {4, 3, 2};
double nodal[2][3][4] = {

```

```

    {{1.,2.,3.,4.},{5.,6.,7.,8.},{9.,10.,11.,12}},
    {{13.,14.,15.,16.},{17.,18.,19.,20.},{21.,22.,23.,24.}}
};
VisIt_ScalarData *VisItGetScalar(int domain, const char *name)
{
    size_t sz = sizeof(VisIt_ScalarData);
    VisIt_ScalarData *scalar = (VisIt_ScalarData*)malloc(sz);
    memset(scalar, 0, sz);

    if(strcmp(name, "zonal") == 0)
    {
        scalar->len = (rmesh_dims[0]-1) * (rmesh_dims[1]-1);
        scalar->data = VisIt_CreateDataArrayFromFloat(
            VISIT_OWNER_SIM, zonal);
    }
    else if(strcmp(name, "nodal") == 0)
    {
        scalar->len = cmesh_dims[0] * cmesh_dims[1] *
            cmesh_dims[2];
        scalar->data = VisIt_CreateDataArrayFromDouble(
            VISIT_OWNER_SIM, (double*)nodal);
    }
    else
    {
        free(scalar);
        scalar = NULL;
    }
    return scalar;
}

```

The Fortran interface provides the `visitscalarsetdatac`, `visitscalarsetdatai`, `visitscalarsetdataf`, and `visitscalarsetdatad` functions for passing your simulation's scalar data back to VisIt. The functions allow you to pass back char, integer, real, and double precision data, respectively. Each of the functions takes four arguments. The first argument is the handle that VisIt passed into the `visitgetscalar` function. The second argument is the actual array that contains the scalar. Be sure that you use the appropriate function for the type of array that you are passing or you will experience runtime problems. The third argument is a 3 element integer array indicating the dimensions of the array that you're providing. The dimensions should match the number of nodes in the mesh for nodal variables. If your mesh is 2D then set the third array element to one. If you are providing a zonal scalar variable then the array elements should contain your mesh's number of zones in each dimension. If you are passing data for an unstructured mesh then you should put either the number of zones or nodes in the first element and add ones for the next two array elements. If you adhere to the guidelines that have been given then you can provide 3 for

the number of array dimensions; otherwise the number of dimensions should match the number of mesh dimensions for structured meshes or use one for unstructured meshes.

Listing 5-56: fscalar.f: Fortran language example for returning a scalar variable.

```

c-----
c visitgetscalar
c-----
      integer function visitgetscalar(handle, domain, name, lname)
      implicit none
      character*8 name
      integer      handle, domain, lname
      include "visitfortransiminterface.inc"
ccc  RECTMESH data
      integer NX, NY
      parameter (NX = 4)
      parameter (NY = 5)
      integer rmdims(3)
      real zonal(NX-1,NY-1)
ccc  CURVMESH data
      integer CNX, CNY, CNZ
      parameter (CNX = 4)
      parameter (CNY = 3)
      parameter (CNZ = 2)
      integer cmdims(3)
      double precision nodal(CNX,CNY,CNZ)
ccc  local vars
      integer m, sdims(3)
ccc  Data
      data rmdims /4, 5, 1/
      data zonal/1.,2.,3.,4.,5.,6.,7.,8.,9.,10.,11.,12./
      data cmdims/CNX,CNY,CNZ/
      data nodal/1.,2.,3.,4.,5.,6.,7.,8.,9.,10.,11.,12.,13.,14.,15.,
. 16.,17.,18.,19.,20.,21.,22.,23.,24./

      m = VISIT_ERROR
      if(visitstrcmp(name, lname, "zonal", 5).eq.0) then
c A zonal variable has 1 less value in each dimension as there
c are nodes. Send back REAL data.
          sdims(1) = rmdims(1)-1
          sdims(2) = rmdims(2)-1
          sdims(3) = rmdims(3)-1
          m = visitsetdataf(handle, zonal, sdims, rmdims)
      elseif(visitstrcmp(name, lname, "nodal", 5).eq.0) then
c A nodal variable has the same number values in each dimension
c as there are nodes. Send back DOUBLE PRECISION data.
          m = visitsetdatad(handle, nodal, cmdims, cmdims)
      endif
      visitgetscalar = m
      end

```

The scalar data access functions in the previous examples build on some of the earlier mesh data access function examples, specifically the examples that returned rectilinear and curvilinear meshes. The zonal variable returned in the examples in this section return data defined on the “mesh2d” rectilinear mesh. The nodal variable returned in the examples in this section return data on the “mesh3d” curvilinear mesh. Examples of both scalar variables are shown in Figure 5-57.

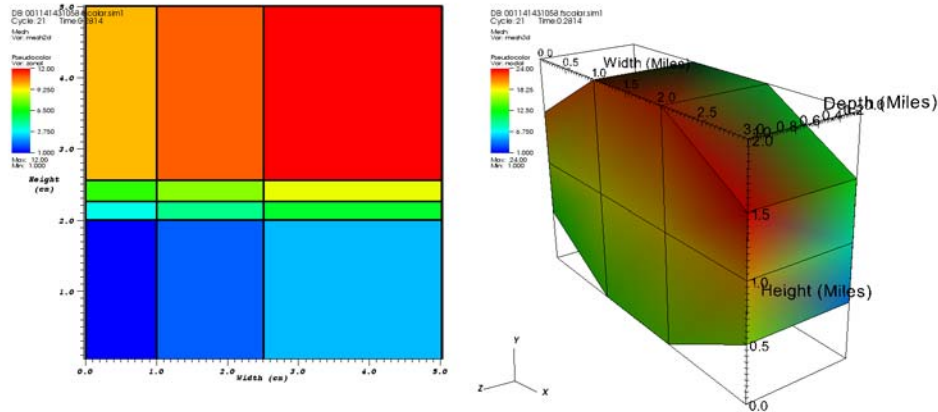


Figure 5-57: Examples of scalar variables returned by a scalar data access function.

4.8 Data access function for curves

This section illustrates how to add a data access function that lets VisIt access your simulation’s curve data. Reading curve data requires a new data access function, requiring you to add a new function pointer to the `visitCallbacks` object. If your simulation is written in Fortran, you must implement the `visitgetcurve` function to return your simulation’s scalar data.

The data access function for scalars returns a `Visit_CurveData` object. The `Visit_CurveData` object is a simple structure, defined in `VisitDataInterface_V1.h`, consisting of two arrays to contain the curve’s X,Y coordinate pairs. This section shows how to create a data access function for curves so your simulation’s curve data are available in VisIt.

4.8.1 Adding a curve data access function

Adding a curve data access function means that you have to first write a function and set the `visitCallbacks` object’s `GetCurve` member so it points to your function. If you program in C, the curve data access function takes the name of a curve object as an argument. The basic procedure for returning curve data is to first check the incoming name against the names of the curves that your simulation is prepared to return and when one is found, return it to VisIt in a `Visit_CurveData` object. If your curve data access

routine does not recognize the name of the curve then you can return NULL instead of returning a `Visit_CurveData` object.

Listing 5-58: `curve.c`: C-Language example for installing a curve data access function.

```

Visit_CurveData *VisitGetCurve(const char *name)
{
    size_t sz = sizeof(Visit_CurveData);
    Visit_CurveData *curve = (Visit_CurveData*)malloc(sz);
    memset(curve, 0, sz);
    if(strcmp(name, "sine") == 0)
    {
        /* Populate curve object's data. */
    }
    else
    {
        free(curve);
        curve = NULL;
    }

    return curve;
}

Visit_SimulationCallback visitCallbacks =
{
    &VisitGetMetaData,
    &VisitGetMesh,
    NULL, /* GetMaterial */
    NULL, /* GetSpecies */
    &VisitGetScalar,
    &VisitGetCurve,
    NULL, /* GetMixedScalar */
    NULL /* GetDomainList */
};

```

The Fortran interface does not require you to install functions into the `visitCallbacks` structure because that is taken care for you in the `libsimg` library. In order to return curve data from a Fortran simulation, you must implement the `visitgetcurve` function. The `visitgetcurve` function is the Fortran interface's curve data access function. The function takes three arguments. The first argument is a handle to a curve data object. The second and third arguments are the name of the curve to return and the length of that name string, respectively. As in the C interface, the `visitgetcurve` function must check the incoming names against the names of the scalars that the simulation has exposed to Visit via metadata. You can use the `visitstrcmp` function to match the incoming name against the names of the known curves.

Listing 5-59: `fcurve.f`: Fortran language example of a curve data access function.


```

-----
c visitgetcurve
-----
integer function visitgetcurve(handle, name, lname)
implicit none
character*8 name
integer handle, lname
include "visitfortransiminterface.inc"
integer m
m = VISIT_ERROR
if(visitstrcmp(name, lname, "sine", 4).eq.0) then
c Pass the curve to VisIt, setting m
endif
visitgetcurve = m
end

```

4.8.2 Passing curve data to VisIt

Now that you know how to install a curve data access function, you need to know how to actually pass curve data back to VisIt. If you program in C, this means filling in the data members of the `VisIt_CurveData` structure. The `VisIt_CurveData` structure has three members. First of all, it contains a number of points that make up the curve data. Next, it contains x and y data arrays that contain the x and y coordinates of the points that make up your curve object. The x and y arrays should have the same number of entries and that number should match the length that you've indicated in the `VisIt_CurveData` structure.

Listing 5-60: curve.c: C-Language example for passing curve data back to VisIt.

```

VisIt_CurveData *VisItGetCurve(const char *name)
{
    size_t sz = sizeof(VisIt_CurveData);
    VisIt_CurveData *curve = (VisIt_CurveData*)malloc(sz);
    memset(curve, 0, sz);
    if(strcmp(name, "sine") == 0)
    {
        int i;
        /* Create a sine curve with 200 points. */
        float *x = NULL, *y = NULL;
        x = (float*)malloc(200 * sizeof(float));
        y = (float*)malloc(200 * sizeof(float));
        for(i = 0; i < 200; ++i)
        {
            x[i] = ((float)i / (float)(200-1)) * 4. * 3.14159;
            y[i] = sin(x[i]);
        }

        /* Use VISIT_OWNER_VISIT and VisIt will free the arrays. */
        curve->len = 200;
        curve->x=VisIt_CreateDataArrayFromFloat(VISIT_OWNER_VISIT, x);
    }
}

```

```

        curve->y=Visit_CreateDataArrayFromFloat(VISIT_OWNER_VISIT, y);
    }
    else
    {
        free(curve);
        curve = NULL;
    }

    return curve;
}

```

If you use the Fortran interface to return curve data then you will implement the `visitgetcurve` function. When the `visitgetcurve` function is passed the name of a valid curve then you must call the `visitcurvesetdataf` if your curve contains single-precision data or the `visitcurvesetdatad` function if your curve contains double-precision floating point data. Both functions accept four arguments. The first argument is the integer handle passed to your `visitgetcurve` function. The second and third arguments are the arrays containing the x and y coordinates, respectively. The final argument is an integer containing the number of points that make up your curve.

Listing 5-61: `fcurve.f`: Fortran language example for passing curve data back to Visit.

```

-----
c visitgetcurve
-----
    integer function visitgetcurve(handle, name, lname)
    implicit none
    character*8 name
    integer      handle, lname, m, NPTS, i
    parameter    (NPTS = 200)
    real         x(NPTS), y(NPTS), t
    include "visitfortransiminterface.inc"
    integer m
    m = VISIT_ERROR
    if(visitstrcmp(name, lname, "sine", 4).eq.0) then
        do 10000 i=1,NPTS
            t = float(i-1) / float(NPTS-1)
            x(i) = t * 4. * 3.14159
            y(i) = sin(x(i))
10000 continue
            m = visitcurvesetdataf(handle, x, y, NPTS)
        endif
        visitgetcurve = m
    end

```

Both of the code examples for returning curve data produce a sine curve, shown in Figure 5-62.

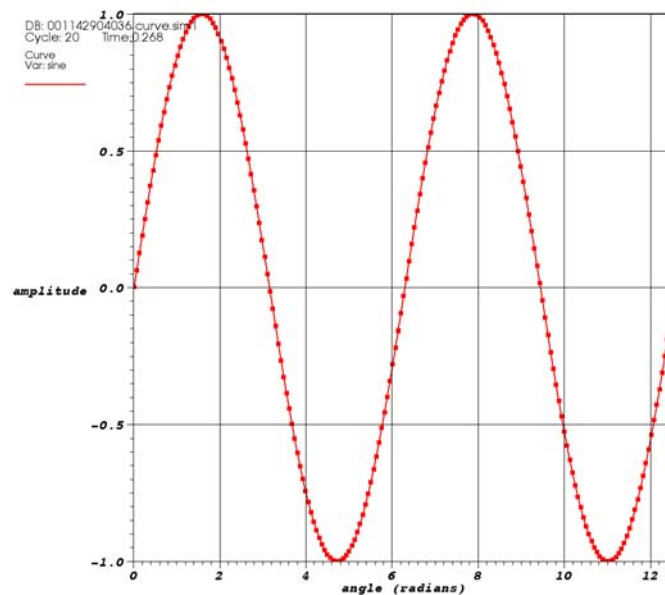


Figure 5-62: Sine curve produced by the curve data access function example programs.

4.9 Data access function for the domain list

The domain list is an object that tells VisIt how many domains there are in your simulation and to which processors they belong. Domain lists are used by VisIt's load balancer to assign work to various processors when running in parallel. Since most parallel simulations only ever process a single domain's worth of data, the domain list will almost always contain a single domain, though the total number of domains is free to change. Note that you must provide a domain list when you run a parallel simulation so VisIt's load balancer can retrieve domains from the appropriate simulation processors.

Installing a domain list data access function is done using the same procedure as for installing other types of data access functions. If you program in C then you must create a new function and add it to the `visitCallbacks` object by setting the `GetDomainList` member.

Listing 5-63: `curve.c`: C-Language example for returning a domain list.

```

VisIt_DomainList *VisItGetDomainList(void)
{
    int np = 1, rank = 0;
    size_t sz = sizeof(VisIt_DomainList);

```

```

    VisIt_DomainList *dl = (VisIt_DomainList*)malloc(sz);
    memset(dl, 0, sz);

#ifdef PARALLEL
    /* Get number of processors and rank from MPI. */
    /* Set np, rank using those values. */
#endif

    dl->nTotalDomains = np;
    dl->nMyDomains = 1;
    dl->myDomains = VisIt_CreateDataArrayFromInt(
        VISIT_OWNER_SIM, &rank);
    return dl;
}

VisIt_SimulationCallback visitCallbacks =
{
    &VisItGetMetaData,
    &VisItGetMesh,
    NULL, /* GetMaterial */
    NULL, /* GetSpecies */
    &VisItGetScalar,
    &VisItGetCurve,
    NULL, /* GetMixedScalar */
    &VisItGetDomainList
};

```

If you use the Fortran interface then you must implement the `visitgetdomainlist` function. The `visitgetdomainlist` function is called when VisIt needs the number and distribution of the domains in use by your simulation. You can provide this information by calling the `visitsetdomainlist` function. The `visitsetdomainlist` function takes four arguments. The first argument is a handle to the `VisIt_DomainList` object that was passed into your `visitgetdomainlist` function. The second argument is an integer containing the total number of domains in your simulation. The total number of domains is almost always equal to the number of processors used by your simulation. The third argument is a list of domain id's. If your simulation assigns 1 domain per processor then you can use the processor's rank for the single value in the list of domain id's. The final argument is an integer containing the number of domains in the domain list.

Listing 5-64: `fcurve.f`: Fortran language example for returning a domain list.

```

c-----
c visitgetdomainlist
c-----
    integer function visitgetdomainlist(handle)
    implicit none
    integer handle
    include "visitfortransiminterface.inc"

```

```
ccc  local variables
      integer totaldomains, domainids(1), ndomids

      totaldomains = 1
      domainids(1) = 0
      ndomids = 1
      visitgetdomainlist = visitsetdomainlist(handle, totaldomains,
      . domainids, ndomids)
      end
```

Index

A

avtMaterial 134

B

BOV file format 9
BOV header file 10
Brick of Floats 10
Brick of Values 10

C

Command line argument -debug 101
Command line argument -debug 5 100
Creating a new Silo file 15
Curve file format 11
Cycle 17
Cycles 126

D

Data extents 70, 131
Dealing with time 16
Debugging logs 100
Debugging your plugin 100
dlopen 143
Double precision 45
Dynamic load balancing 138

E

EMPTY keyword 52
export-dynamic linker flag 164

G

Ghost zones 74, 136, 137

I

Inspecting Silo files 14

L

LD_LIBRARY_PATH 161
LDFLAGS 164
libsim - VisItAttemptToCompleteConnection 150
libsim - VisItControlInterface_V1.h 142
libsim - VisItDetectInput 149
libsim - VisItDisconnect 150
libsim - visitfortransiminterface.inc 142
libsim - VisItInitializeSocketAndDumpSim-File 144, 161
libsim - VisItProcessEngineCommand 150
libsim - VisItSetBroadcastIntFunction 145
libsim - VisItSetBroadcastStringFunction 145
libsim - VisItSetCommandCallback 174
libsim - VisItSetParallel 145
libsim - VisItSetParallelRank 145
libsim - VisItSetupEnvironment 144

M

Materials 81, 82, 83, 84, 134
MPI 137

O

Option lists 17

P

Plugin development - ActivateTimestep 138
Plugin development - Auxiliary data 130
Plugin development - avtDatabaseMetaData 103
Plugin development - Curvilinear meshes 113
Plugin development - expression metadata 108
Plugin development - GetAuxiliaryData 130, 131
Plugin development - GetMesh 99, 110, 111, 112, 114, 115, 116, 118, 119
Plugin development - GetVar 99, 122, 123
Plugin development - GetVectorVar 99, 123
Plugin development - libE 89, 137
Plugin development - libI 89
Plugin development - libM 89
Plugin development - material metadata 108
Plugin development - mesh metadata 103
Plugin development - MTMD 90
Plugin development - MTSD 90
Plugin development - Parallelizing your reader 137
Plugin development - Point meshes 115
Plugin development - PopulateDatabaseMetaData 99, 103, 122, 123
Plugin development - Rectilinear meshes 111
Plugin development - Returning a mesh 109
Plugin development - Returning a scalar variable 122
Plugin development - Returning a vector variable 123
Plugin development - Returning cycles and times 126
Plugin development - Returning ghost zones 136
Plugin development - Returning materials 134
Plugin development - scalar metadata 106
Plugin development - STMD 90
Plugin development - STSD 90
Plugin development - Unstructured meshes 118
Plugin development - Using a VTK reader class 125

Plugin development - vector metadata 107
Plugin development - xml2info 94
Plugin development - xml2makefile 94, 96, 97
Plugin development - xml2plugin 94
Plugin development - XMLEdit 90

S

Silo 9
Silo - browser 14
Silo - DB_CHAR 38
Silo - DB_F77NULL 20
Silo - DB_FLOAT 37
Silo - DB_HDF5 15
Silo - DB_NODECENT 38
Silo - DB_NONCOLLINEAR 22
Silo - DB_PDB 15
Silo - DB_ZONECENT 38
Silo - DBAddOption 17
Silo - DBCreate 15
Silo - DBFreeOptlist 17, 32, 44
Silo - DBMakeOptlist 17, 32, 44
Silo - DBOPT_UNITS 44
Silo - DBPutdefvars 45
Silo - dbputdefvars 46
Silo - dbputmat 85
Silo - DBPutMaterial 84
Silo - dbputmmesh 49
Silo - DBPutMultimesh 48
Silo - DBPutMultivar 50, 72
Silo - dbputpm 26
Silo - DBPutPointmesh 25
Silo - DBPutPointVar1 41
Silo - dbputqm 20, 21, 23, 24
Silo - DBPutQuadmesh 18, 20, 22, 81
Silo - DBPutQuadvar1 35, 37, 38, 44
Silo - dbputqv1 38
Silo - DBPutUcdmesh 28
Silo - DBPutUcdvar1 43, 44
Silo - dbputuv1 43

Silo - DBPutZonelist 28
Silo - dbset2dstrlen 46
Silo - header files 12
Silo - linking with 13
SimV1 database reader plugin 163, 164
Spatial extents 73, 132
Static load balancing 138
Strategies 2

T

Time 17
Times 126
topological dimension 104

U

Units 44

V

VCEL 141
VisIt Compute Engine Library 141
VisIt_CreateDataArrayFromChar 196
VisIt_CreateDataArrayFromDouble 196
VisIt_CreateDataArrayFromFloat 180, 187,
196
VisIt_CreateDataArrayFromInt 191, 196
VisIt_CurveData 199

VisIt_MaterialMetaData 172
VisIt_MeshData 177
VisIt_MeshMetaData 168
VISIT_OWNER_SIM 180, 196
VISIT_OWNER_VISIT 180, 196
VisIt_PointMesh 186
VisIt_ScalarData 194, 196
VisIt_ScalarMetaData 170
VisIt_SimulationMetaData 166
visit_writer - write_curvilinear_mesh 58
visit_writer - write_point_mesh 61
visit_writer - write_regular_mesh 54
visit_writer - write_unstructured_mesh 62
visitaddsimcommand 176
visitbroadcastintfunction 159, 160
visitbroadcaststringfunction 159, 160
visitCallbacks 162, 163, 164
visitcommandcallback 176
visitdetectinput 157
visitgetcurve 199, 200
visitgetmesh 177, 192
visitgetscalar 194, 195
visitinitializesim 156
visitmdcurvecreate 172
visitmdcurvesetlabels 172
visitmdcurvesetunits 172
visitmdexpressioncreate 173
visitmdmaterialadd 172
visitmdmaterialcreate 172
visitmdmeshcreate 169
visitmdmeshsetblockpiename 170
visitmdmeshsetblocktitle 170
visitmdmeshsetlabels 169
visitmdmeshsetunits 169
visitmdscalarcreate 170
visitmdsetcycletime 166, 167
visitmdsetrunning 166, 167
visitmeshcurvilinear 185
visitmeshrectilinear 181
visitmeshunstructured 192
VISITPLUGINDIR 161
visitprocessenginecommand 157
visitscalarsetdatac 197
visitscalarsetdatad 197
visitscalarsetdataf 197
visitscalarsetdatai 197

visitsetparallel 156
visitsetparallelrank 156
visitslaveprocesscallback 159, 160
visitstrcmp 176, 196
VTK 9, 125
vtkFloatArray 122, 123
vtkRectilinearGrid 111
vtkStructuredGrid 113
vtkUnstructuredGrid 115, 118, 122

X

X-Y plots 11

