Generating VisIt Visualization Data Files in SAMRAI

Peter L. Williams

1 Introduction

VisIt is a distributed, parallel, visualization tool for visualizing data defined on two- and three-dimensional structured and unstructured meshes, including data defined on structured AMR patch hierarchies and deformed structured AMR meshes, such as moving Lagrangian meshes. Practical details on obtaining and using VisIt are given in Section 7. A slide presentation on VisIt and the SAMRAI VisItDataWriter is available in the SAMRAI distribution at *docs/userdocs/VisIt-writer-slides.pdf*.

The purpose of the VisIt data writer is to dump SAMRAI data to files that can be read-in and processed by VisIt. These files have traditionally been called "plot" files, but are also referred to herein as VisIt data files or dump files. As described in Section 7, these dump files may be viewed using VisIt either after the simulation has completed, or during the simulation, thus allowing the simulation to be monitored visually. In addition, as described in Section 7.11, the dumped data can be viewed remotely. If the simulation was run on a remote distributed machine, the data can be left there and viewed on a local workstation using VisIt. VisIt may then be run in distributed mode on the remote machine so that the visualization compute engine can run in parallel.

In the next section, we give a summary of the differences between the VisIt and Vizamrai data writers. Section 3 describes the VisIt data writer class and gives step by step instructions for creating and using an object of this class. In Section 4, code fragments are given for a typical call sequence to a VisIt data writer object from an example SAMRAI application. Then in Section 5, we discuss materials and how they may be added to a plot file. In Section 6, we discuss how ghost data can be dumped for viewing with VisIt. Finally in Section 7, we give a brief introduction to using VisIt with SAMRAI plot files.

2 Differences Between VisIt and Vizamrai Data Writers

This section may be skipped by users who have not used Vizamrai in the past. (Vizamrai was a previous, more limited visualization system for SAMRAI data).

The VisIt data writer is very similar to the Vizamrai data writer, however, there are some significant differences which are listed next. Items number 1 through 6 indicate differences that may require changes to existing application code to convert from the use of the Vizamrai data writer to the VisIt data writer. The remaining items, 7 through 14, are more informational, mainly indicating new or modified features.

1. The method `setFinestLevelToPlot()`, available for Vizamrai, is not implemented for VisIt.
2. The constructor for the VisIt data writer, in addition to the optional argument setting $number\_procs\_per\_file$, described in Item 8, has a new required argument, the directory name where the dumped data will be stored, The separate method `setDirectoryName()` is no longer used for VisIt. For Vizamrai setting a directory name was optional.
3. The method `setRatioToCoarserLevel()` used by Vizamrai, is not used by the VisIt data writer.
4. The `writePlotData()` method for VisIt requires a time step number; and it does not take a file name. Whereas the Vizamrai data writer requires a file name; and the time step number is optional.

5. The method `setDerivedDataWriter()` used by Vizamrai, is renamed to `setDefaultDerivedDataWriter()` for VisIt.

6. With the VisIt data writer, only float data is actually written to the dump files. However, the data passed from SAMRAI to the VisIt data writer may be integer, float or double. This data is stored internally by the data writer in a double precision buffer so it may be properly scaled if a scale factor is specified. Just before writing the data to disk, the data is converted to floating point format. Thus the methods `setPlotDataToDouble()` and `setPlotDataToFloat()`, used by Vizamrai, are not used by the VisIt data writer.

7. The VisIt data writer dumps data to HDF files, whereas Vizamrai writes binary files. Thus SAMRAI must be compiled with the HDF5 library.

8. For parallel runs, the VisIt data writer allows subsets of processors to dump their data to a single file thus reducing parallel I/O contention. The size of the subset, $number\_procs\_per\_file$, is an optional argument to the VisIt data writer constructor. This feature is not available with Vizamrai. With VisIt, data from parallel runs is ready for immediate visualization, without any assembly step as was required with Vizamrai.

9. The VisIt data writer can handle both node-centered as well as cell-centered data. The Vizamrai data writer only deals with cell-centered data.

10. The VisIt data writer accepts data defined on ghost cells or nodes. This was not possible with Vizamrai.

11. With VisIt, vector data with NDIM components, as well as 2nd order tensor data with (NDIM * NDIM) components, are supported. With Vizamrai only vector data with NDIM components are supported. The methods `register[Derived]PlotTensor()` have been introduced in the VisIt Data Writer, and various optional parameters have been added to all data registration methods to deal with centering and ghost data.

12. The VisIt data writer can accept data defined on deformed structured AMR meshes, such as moving Lagrangian meshes. The Vizamrai data writer does not handle this type of data.

13. For the VisIt data writer, a variable's data values do not need to exist on all patches, nor on all levels. For the Vizamrai data writer, a variable is expected to appear on all patches.

14. Material fractions, species fractions, and state variables defined on a material, may be dumped using the VisIt data writer. This capability was not possible with Vizamrai.

3 How to Generate VisIt Data Files in SAMRAI

In Section 3.1, we present background information that is essential to know before using the VisIt data writer. In Section 3.2 we give the detailed steps to create and use a VisIt data writer object for dumping SAMRAI state variable data and derived data. Then in Section 3.3 we give the additional steps necessary if materials are to be added to the dump files. In Section 3.4 we give the additional steps required to deal with deformed AMR grids. Section 6 describes how to dump ghost cell data so it can be viewed in VisIt.

3.1 Background Information

An object of the SAMRAI class `appu_VisItDataWriterX` is used to generate VisIt data files. We refer to this object as a VisIt data writer object. This class applies when the underlying mesh geometry is Cartesian, that is, managed by a `geom_CartesianGridGeometryX` object, or the mesh itself is stored as a state variable to allow moving deformed AMR grids. The VisIt data writer requires compilation of SAMRAI with the HDF5 library and will deal with either 2D or 3D data ($NDIM = 2$ or $NDIM = 3$). The types of data that the data writer will accept from SAMRAI are integer, float and double. Scalar data, vector data with NDIM components, and 2nd order tensor data with (NDIM * NDIM) components may be dumped.

The VisIt data writer class supports the dumping of two kinds of data: data that resides on an AMR patch hierarchy at the time the dump file is written, and *derived data*, data that does not live on the patch hierarchy, but that can be computed from data which does live on the patch hierarchy, e.g. total energy calculated from velocity, density and pressure. Derived data requires the user to implement a method in a concrete class derived from the `appu_VisDerivedDataStrategyX` abstract class. In that method, the user computes and packs into a buffer the specified derived data quantity over a given box region on a patch. This method may be implemented so as to calculate one of several derived data quantities, with the choice being determined by the *variable_name* argument. This is discussed in detail in Section 4.4. Writing derived data requires no user intervention when generating a dump file. The VisIt system also provides a limited capability for deriving new variables from the original data in the dump file.

The VisIt data writer only writes float data to the dump files since VisIt uses only float data internally. However, the data passed from SAMRAI to the VisIt data writer may be integer, float or double. All such data is stored internally by the data writer in a double precision buffer so it may be properly scaled if a scale factor is specified. This scaling feature may be used to bring double precision data into a range suitable for visualization as floats. After scaling, and just before writing the data to disk, the data is converted to floating point format. Any data value, after scaling, that exceeds the maximum representable floating point value MAX_FLT is clamped to MAX_FLT; in addition, a warning is written to the log file for each offending variable for each time step indicating the number of values clamped and the maximum double precision value that was clamped. This information allows the application to properly set the scale factor for future runs. The number of underflows are also recorded in the log file. An example log file entry is:

```
variable: Density had 3 floating point POSITIVE OVERFLOWS,
MAX POSITIVE OVERFLOW: 7.5e+50
All positive overflows clamped to 3.40282e+38

variable: Density had 2 floating point NEGATIVE OVERFLOWS,
MAX NEGATIVE OVERFLOW: -4.31e+55
All negative overflows clamped to -3.40282e+38
```

In simulations where cells contain fractional volumetric amounts of material compounds, data relating to these materials may be dumped. The concept of materials is explained in detail in Section 5.1; this section is essential reading if you plan to dump materials.

Before a VisIt data writer object can be used to generate VisIt data files, it must first be constructed and initialized. Initialization involves registering which scalar and vector data, derived data quantities, and material-related data are to be dumped. After initialization, the VisIt data writer object may be used to generate a series of visualization dump files during the execution of an application.

Complete documentation on each method in the VisIt data writer class and the VisIt derived data and materials data strategy classes can be found in the Application Utilities Section of the SAMRAI Reference Manual in the SAMRAI/docs directory.

3.2 Basic Steps to Create and Use a VisIt Data Writer Object

These are the basic steps to create and use a VisIt data writer object for dumping SAMRAI state variable data and derived data.

1. Create an `appu_VisItDataWriterX` object. The *object_name* argument is used primarily for error reporting. The *dump_directory_name* argument is required, and is the directory where the VisIt data writer will create the subdirectory structure used to hold all the dumps. If the specified directory does not exist, it will be created. The directory name may include a path, if so, any intermediate directories in the path will be created if they do not already exist.

   An optional argument to the constructor, *number_procs_per_file*, applicable to parallel runs, sets the number of processors that share a common dump file. The purpose of this is to reduce parallel I/O contention and improve I/O efficiency. The default value of *number_procs_per_file* is 1. If the specified value is greater than the number of processors, then all processors share a single dump file.

When starting from a restart file, it is essential that the value of the argument *number_procs_per_file* be the same as was used to generate the restart file.

2. If any derived quantities will be generated, a default concrete `appu_VisDerivedDataStrategyX` object may be registered using the method `setDefaultDerivedDataWriter()`. When registering each derived data quantity, a concrete derived data strategy object may be specified to use instead of the default strategy object. This provides some flexibility in using different derived data strategy objects for different quantities if desired. For each concrete derived data strategy object, the method `packDerivedDataIntoDoubleBuffer()` must be implemented which calculates the desired derived data quantity or quantities, (several different derived data quantities may be calculated in that method.) Details on the implementation of the concrete derived data strategy object are given in Section 4.4.

3. Register variable data fields using either `registerPlotScalar()`, `registerPlotVector()`, `registerPlotTensor()`, `registerDerivedPlotScalar()`, `registerDerivedPlotVector()`, or `registerDerivedPlotTensor()`. All variables require a unique string identifier and, for the non-derived data, an index into the patch data array on the AMR hierarchy. Data is not required to exist on all patches nor on all levels. If a variable was previously registered with the same string identifier, an error results. A scale factor may also be specified; if so, each data value is multiplied by this factor before being written to the dump file. Scaling is performed in double precision. For derived data, a derived data writer object may be specified, otherwise the default derived data writer object will be used. In addition, for derived data, the user is responsible for registering the correct *centering*, (by default, derived data is assumed to be cell-centered).

   The optional depth index argument of `registerPlotScalar()` may be specified to allow a single component of a data object with multiple components to be dumped. For non-derived vector or tensor data to be dumped as a vector or tensor, use `registerPlotVector()` or `registerPlotTensor()`, optionally setting a *start_depth_index*. Each component of the vector/tensor starting from *start_depth_index* will be registered for writing to the plot file. If no *start_depth_index* is specified, a *start_depth_index* of 0 is used. For vector data, NDIM components will be written to the plot file. For tensor data, (NDIM * NDIM) components will be dumped.

4. If a variable lives at different patch data array indices on different hierarchy levels, after first registering that variable with `registerPlotScalar()`, `registerPlotVector()`, or `registerPlotTensor()`, invoke `resetLevelPlotScalar()`, `resetLevelPlotVector()`, or `resetLevelPlotTensor()` to redefine the patch data array index. For scalar variables, the depth index may also be modified; for vector or tensor variables, the start depth index may be modified. This change redefines the patch data objects written to the plot file on the specified level to the data at the new patch data array index / depth index. For example, suppose a scalar variable lives at a patch data array index on every level except the finest hierarchy level, where it lives at a different index. First, the scalar variable must be registered using `registerPlotScalar()`. Second, the patch data array index for the finest hierarchy level is reset using `resetLevelPlotScalar()`. When the data is plotted, it will appear on all levels in the hierarchy. The data corresponding to the new patch data array index must have the same centering and data type as the data for which the variable was originally registered.

5. Finally, invoke `writePlotData()` at the appropriate time to generate a VisIt dump of all registered items, specifying a hierarchy, time-step number, and optionally a plot time. The time step number must be non-negative and greater than the previous time step number, if any. New variables may be registered at future time steps, but there is no provision for de-registering variables.

3.3 Adding Material Data to the Dump File

Material-related data, (materials, species, and their related state variables), are discussed in detail in Section 5. If the user is going to dump material-related data, it is essential to first carefully read Section 5.1. In the present section, we assume the user is familiar with the information in Section 5 and give only a terse, but nevertheless complete, summary of the steps required (in addition to those given in the previous section) if material-related data are to be dumped.

1. First invoke `registerMaterialNames()` with an array of strings, the names of the materials.

2. Then if species or material state variables are to be dumped, the methods `registerSpeciesNames()`, `registerMaterialStateVariable()`, and/or `registerSpeciesStateVariable()` may be invoked. For each material that has species, invoke the method `registerSpeciesNames()` with the names of the species as well as the name of the material. If material state variables are to be dumped, use the method `registerMaterialStateVariable()` to specify the name of the variable, and its depth (1 for a scalar, NDIM for a vector, (NDIM * NDIM) for a tensor), and optionally a scale factor. The state variable name must not have been registered as a variable name either for a regular variable or a derived variable. If species state variables are to be viewed, use the method `registerSpeciesStateVariable()` to specify the name of the variable; this variable name must have already been registered using `registerPlotScalar()` or `registerPlotVector()`, or their derived variable counterparts.

3. Finally, if dumping any material-related data, in addition to registering the data, a concrete materials data writer object, derived from `appu_VisMaterialsDataStrategyX`, must be registered using the method `setMaterialsDataWriter()`, and the relevant methods for packing the materials data into a buffer (see Section 5) must be implemented in the concrete material data writer object.

3.4 Deformed Structured AMR Grids

For simulations defined on deformed moving grids, in addition to the steps given in Section 3.2, it is necessary to:

1. Register the node coordinates. To do this, the method `registerNodeCoordinate()` is called once for each of the NDIM dimensions. The *coordinate_number* argument is set to tell VisIt which coordinate is being registered. So, if data is registered with *coordinate_number* = 0, VisIt will use this data as the x coordinate. If the patch data array index refers to a vector, the optional *depth_index* argument specifies the component of that vector. The *scale_factor* may be different in each call.

3.5 Dumping Ghost Data

Please see Section 6 which describes how to dump ghost data so it can be viewed in VisIt.

4 Example of VisIt Data Writer Basic Usage: Writing SAMRAI State Variable Data and Derived Data

Here, we present code fragments from an application code for an example that will create a VisItDataWriter object, initialize it, and then generate dump files of various SAMRAI state variables and derived data. For derived data, a concrete data strategy object must also be implemented. For simplicity of presentation, we defer the coverage of materials-related data and ghost cell data until Sections 5 and 6.

First, we set up the example. Then in Section 4.1 we give an excerpt from a typical application input script file specifying the relevant VisIt dump parameters. In Section 4.2, we give code fragments from the main.C file, some of which depend on the VisIt dump parameters from the input script file. In Section 4.3, we give code fragments from the application class, and in Section 4.4, we describe the strategy object needed for derived data and give an implementation.

In our example, we will give a typical call sequence to the VisIt data writer from a 3 dimensional ($NDIM = 3$) application named *Applic*. In this sequence, we will dump five non-derived variables: *density*, *pressure* scaled by a factor of 10.0, the z component of velocity *z_velocity*, and the *stress* vector; and finally, the *velocity* vector which appears in the hierarchy at depth indices 2, 3, and 4 of a state space vector. We will reset *z_velocity* on level 6 to be at depth index 4 of the state space variable. We also dump 2 derived variables: *total_energy*, and the vector field *momentum*; and, we set the default derived data writer object.

4.1 Excerpt from a Typical Application *.input* File

In this excerpt from a typical application *.input* file, we specify the parameters relevant to a VisIt dump. These parameters will be referred to by the code in the following section.

```
// excerpt from example3d.input file

   . . .

Main {
// log file parameters

   . . .

// visit dump parameters
   visit_dump_interval      = 1 // zero turns off visit dumps
   visit_dump_dir_name      = "visit_example3d"
   visit_number_procs_per_file = 3

// restart dump parameters

   . . .

}

   . . .
```

4.2 Code Fragments from the *main.C* File

Here we give the code fragments from the main.C file which create a new VisIt data writer object, set the default derived data writer, and invoke the `writePlotData()` method.

```
// main.C

  . . .

// create application object

   Applic* applic_object = new Applic("Applic",
                                      input_db->getDatabase("Applic"),
                                      grid_geometry);

// Create & initialize data writer

   Pointer<VisItDataWriter> d_visit_data_writer;
   int visit_dump_interval = 0;
   int visit_number_procs_per_file = 1;
   bool use_visit_data_writer = false;
   string visit_dump_dir_name;

   if (main_db->keyExists("visit_dump_interval")){
      visit_dump_interval = main_db->getInteger("visit_dump_interval");
   }
   use_visit_data_writer == (visit_dump_interval > 0);
   if (use_visit_data_writer) {
      if (main_db->keyExists("visit_dump_dir_name")) {
         visit_dump_dir_name = main_db->getString("visit_dump_dir_name");
      }
      if (main_db->keyExists("visit_number_procs_per_file")) {
         visit_number_procs_per_file = main_db->getString("visit_number_procs_per_file'');
      }
      // create visit dataWriter object:
      d_visit_data_writer = new VisItDataWriter("Applic VisIt Writer",
                                                visit_dump_dir_name,
                                                visit_number_procs_per_file);
   }

   . . .

   if (use_visit_data_writer) {
      // register data writer with application model object
      applic_object->registerVisItDataWriter(d_visit_data_writer);

      // if derived data to be used, optionally set a default
      // derived data writer object;
      d_visit_data_writer->setDefaultDerivedDataWriter(applic_object);
   }

   . . .

// Time Step Loop

   double loop_time = time_integrator->getIntegratorTime();
   double loop_time_end = time_integrator->getEndTime();
   while ((loop_time < loop_time_end) && time_integrator->stepsRemaining()) {
      int iteration_num = time_integrator->getIntegratorStep() + 1;

      /*
       * Advance solution data.
       */

      . . .

      /*
       * At specified intervals, write visualization dump files
       */

      if ((iteration_num % visit_dump_interval) == 0 ) {
         // generate the dump files now:
         d_visit_data_writer->writePlotData(patch_hierarchy,
                                            iteration_number,
                                            loop_time);
      }

   }
```

4.3 Code Fragments from the Application Class File

We now consider the application model object defined in the file *Applic.C*. It should have a method such as `registerVisItDataWriter()`, shown below, to register a pointer to the VisIt data writer object. The application object will then use that pointer to register variables with the VisIt data writer. The application routine, `registerModelVariables()`, contains the registration of variables with the VisIt data writer. The routine `mapVariableAndContextToIndex()` returns the patch data array index of the variable. Note that when using derived data, as shown in the beginning of the code example in Section 4.4, *Applic* derives from `appu_VisDerivedDataStrategyX`.

```
// Applic.C

// initialize Applic member variables
. . .
Pointer<VisItDataWriter> d_visit_data_writer = NULL;

void Applic::registerVisItDataWriter(Pointer<VisItDataWriter> visit_data_writer)
{
   d_visit_data_writer = visit_data_writer;
}

void Applic::registerModelVariables(HyperbolicLevelIntegrator* integrator)
{
   . . .

   d_plot_context = integrator->getPlotContext();
   VariableDatabase* vardb = VariableDatabase::getDatabase();

   if (!d_visit_data_writer.isNull()) {
      // register non-derived variables for dumping:

      // register scalar variable "density"
      d_visit_data_writer->registerPlotScalar("density",
                                              vardb->mapVariableAndContextToIndex(
                                                          d_density, d_plot_context));

      // register scalar variable "pressure", scaling it by a factor of 10.0.  Since
      // the scale_factor is the last of 2 optional args, both optional args need to appear.
      d_visit_data_writer->registerPlotScalar("pressure",
                                              vardb->mapVariableAndContextToIndex(
                                                         d_pressure, d_plot_context),
                                              0,     // depth_index
                                              10.0); // scale_factor

      // register scalar variable "z_velocity", which appears at a depth index of 2
      // in a data object with multiple components called d_velocity.
      d_visit_data_writer->registerPlotScalar("z_velocity",
                                              vardb->mapVariableAndContextToIndex(
                                                         d_velocity, d_plot_context),
                                              2);    // depth_index

      // register vector variable "stress"
      d_visit_data_writer->registerPlotVector("stress",
                                              vardb->mapVariableAndContextToIndex(
                                                           d_stress, d_plot_context));
      // register vector variable "velocity", which appears in a multi-component variable
      // d_state_space at depth indices 2, 3, and 4.  The last two args are optional,
      // hence the scale_factor must be specified.
      d_visit_data_writer->registerPlotVector("velocity",
                                              vardb->mapVariableAndContextToIndex(
                                                        d_state_space, d_plot_context));
                                              1.0,  // scale_factor
                                              2);   // start_depth_index

      // reset plot levels:

      // reset "z_velocity" on level 6 to be at depth index of 4 of the state space vector.
      d_visit_data_writer->resetLevelPlotScalar("z_velocity",
                                                6,  // level number
                                                vardb->mapVariableAndContextToIndex(
                                                        d_state_space, d_plot_context),
                                                4); // depth index

      // register derived variables for dumping (here we assume the use of the  default
      // registered above in main.C

      d_visit_data_writer->registerDerivedPlotScalar("total_energy");
      d_visit_data_writer->registerDerivedPlotVector("momentum");

   }
}
```

For a complete implementation, see the source code in the example applications, such as Euler. When testing using a file in one of the *sample_input* directories, note that the input script for specifying plot file parameters is more complex than shown above in Section 4.1 to allow for the use of either Vizamrai or VisIt data writers. In these input files, set *viz_writer* to "VisIt" if you want to create VisIt plot files.

4.4 Implementing the Derived Data Strategy Object

When writing derived data, in addition to registering the derived data as shown in Section 4.3, and optionally setting the default *DerivedDataWriter* object as shown in the code in Section 4.2, the application class member `packDerivedDataIntoDoubleBuffer()` needs to be implemented. The interface for this method is defined in **appu_VisDerivedDataStrategyX**; and the application class needs to inherit from this strategy class.

This packing method computes the derived data for a given patch over a given box and packs it into a double precision buffer. The data should be packed into the buffer in column major order, the ordering used by SAMRAI, i.e. for 3D data, $(f(i_0, j_0, k_0), f(i_1, j_0, k_0), f(i_2, j_0, k_0), ...)$, where $f(i, j, k)$ is the data value at index (i, j, k) on the patch data box. If the derived data was registered as node-centered, then `packDerivedDataIntoDoubleBuffer()` must return a buffer of node-centered data. Derived data need not be defined on all patches. It is the responsibility of the application to determine if the data exists on a patch and set the return value of packDerivedDataIntoDoubleBuffer() appropriately: true if the data exists on the patch, false otherwise. The buffer will already have been allocated to the correct size, taking into consideration whether the data is registered as node- or cell-centered. For vector or tensor data, the packing routine will be called once for each component of the vector or tensor, with the $depth_index$ argument indicating the component to pack.

```
// Applic.h
#include "VisDerivedDataStrategy.h"

class Applic :
   public VisDerivedDataStrategy



// Applic.C

bool Applic::packDerivedDataIntoDoubleBuffer(
      double *dbuffer,
      const Patch& patch,
      const Box& region,
      const string& variable_name,
      int   depth_index)
{
   if (variable_name == "total_energy") {

      /*
       * Compute total energy on the box region using the data
       * on the patch and pack it into dbuffer. If data is
       * defined on patch, set data_on_patch to true, else false.
       */

   } else if (variable_name == "momentum") {

      /*
       * Compute the momentum for the given depth index on the box
       * region using the data on the patch and pack it into dbuffer.
       * This method will be called once for each component of this
       * vector variable.  If data is defined on patch, set
       * data_on_patch to true, else false.
       */

   } else {
      TBOX_ERROR("Applic::packDerivedDataIntoDoubleBuffer"
         << "\n    application with name " << d_object_name
         << "\n    unknown variable_name " << variable_name << "\n");
   }

   return data_on_patch;
}
```

The VisIt data writer class in conjunction with the derived data strategy class give a very flexible and powerful mechanism for creating and dumping derived data. However, there may be times when it may be wise to utilize VisIt's own derived data mechanism. Provided the expression to calculate the derived data is relatively simple, as for example $\mathit{momentum} = \mathit{density} * \mathit{velocity}$, it will save disk space not to dump this derived data quantity, but instead to define it while using VisIt. The downside of using VisIt to calculate derived data is that the derived data is cached only while it is currently plotted on the screen, and that the derived data expression needs to be keyed in.

5 Writing Material Data

We start with a brief but essential conceptual overview of materials and related terminology as used by Visit. Then in Section 5.2 we discuss the details of using material fractions. In Section 5.3 we cover the use of material state variables. Then in Section 5.4, we cover the use of species and in Section 5.5 species state variables. Finally in Section 5.6 we discuss what to do if material-related data is not cell-centered, and in Section 5.7 we discuss the special case when one might want to use only species, with no materials.

5.1 Conceptual Overview of Materials

In simulations where cells contain fractional volumetric amounts of material compounds, data relating to these materials may be dumped by the VisIt data writer. A material may have subcomponents called *species*. So for example if a simulation has 4 materials, say *copper*, *gold*, *liquid* and *gas*, then the material *gas*, for example, may have subcomponents, e.g. *oxygen*, *methane* and *nitrogen*. In this example, the material *gas* has 3 species, and we say *methane* is a *species* of *gas*. Each material may have its own set of species.

When a mesh has materials defined over it, every cell contains a fractional amount *mf* ($0 <= mf <= 1.0$) of every material, called a *material fraction* or *volume fraction*. VisIt assumes that the sum of all material fractions for every cell is 1.0. Similarly, for each species, there is a *species fraction*, *sf*, (also known as a *mass fraction*), ($0 <= sf <= 1.0$) for each cell. The sum of the species fractions for all species of a given material m must equal 1.0 in every cell in which m appears.

There is an even more important difference between materials and species other than the fact that species are subcomponents of a material. A set of materials signifies a heterogeneous mixture of substances where each material has a distinct boundary, such as in concrete, or granite. Whereas, a species signify a homogeneous mixture of substances with no defined boundaries, such as seawater, Coke, or air.

Scalar or vector data may be defined over each material. These data are referred to as *material state variables* or *intensive variables*. So in the above example, each of the four materials may have a different temperature associated with it on a cell by cell basis. In this example, *temperature* is a material state variable, and there is a separate temperature field over each patch for each of the four materials, provided the patch contains some of the material. Every material state variable must have a value for each cell, for each material, if that material has a non-zero fraction in that cell.

Species too may have state variables; however they are treated differently than material state variables. Any SAMRAI state variable or derived variable registered with the VisIt data writer may be designated as a species state variable. A species state variable is unique in that VisIt treats it in a special way. When VisIt displays a species state variable, the value of the variable at a cell is multiplied by the sum of the species fractions for that cell, for all the species that are currently selected. (In VisIt, there is a *Subset Window* which shows all materials and species, and which allows the user to *select* (turn on/off) individual materials and species.) Consider an example where the variable *pressure* is being visualized, and *pressure* has been registered as a species state variable. If a material has 3 species, and the pressure for a cell c is 100, and only one species has been selected, say *nitrogen*, and its species fraction for c is 0.45, then the partial pressure for c will be 45. If two species were selected, say *nitrogen* with a species fraction of 0.45, and *methane* with a fraction of 0.25, then the partial pressure for c would be 70. Thus in the first example where only *nitrogen* was selected, a *partial* pressure field for *nitrogen* could be displayed, (e.g. using a pseudocolor plot). The user may think of other ways to make use of this feature of VisIt which multiplies the sum of the fractions of the currently selected species by a selected state variable.

Another use for species fractions is that they can be treated as a scalar field and the usual scalar plot operations applied to show *concentrations*. So for example if the material *seawater* has a species *salt*, then the *salinity* of the water can be displayed as a pseudocolor plot of the species fractions of *salt*.

Returning now to material fractions, VisIt uses these values to reconstruct (interpolate) material boundaries within cells which have multiple materials. More specifically, VisIt finds a crack-free piecewise two-manifold separating surface approximating the boundary surface between the materials. Therefore VisIt can display materials as multiple colored contiguous 3D regions with intra-cell boundaries. In addition, VisIt can intersect this volume field with a plane and show intra-cell 2D boundaries. VisIt can reconstruct material boundaries for either 2D or 3D data. If desired, the material fractions for a material can be treated as a scalar field and the usual scalar field plot tools, such as pseudocolor and contour, can be applied.

5.2 Steps to Add Material Volume Fraction Data

We will start with the steps to register materials which have volume fractions; later we will add the steps needed if one wishes to add species and/or material state variables.

First, as part of the initialization process, i.e. steps 2 through 4 of Section 3.2, invoke the method `registerMaterialNames()` with an array of strings, the names of all the materials used in the simulation. It is a requirement that `registerMaterialNames()` be invoked at most once.

It is also necessary to create and register a *MaterialsDataWriter* object. To do this, the application class member needs to inherit from `appu_VisMaterialsDataStrategyX`. This is an abstract base class which defines an interface for writing out the various material-related fields. A concrete object of this strategy must be registered with `setMaterialsDataWriter()`.

Before we discuss the implementation of this strategy, we show the two required initialization invocations. For our example, we will assume there are four materials in our simulation: *copper*, *gold*, *liquid*, and *gas*.

First, in the main.C code fragment given in Section 4.2, find the call:

```
d_visit_data_writer->setDefaultDerivedDataWriter(applic_object);
```

Now add the following call immediately after it:

```
d_visit_data_writer->setMaterialsDataWriter(applic_object);
```

Second, in the code for the application class file *Applic.C* shown in Section 4.3, find the call:

```
d_visit_data_writer->registerDerivedPlotVector("momentum");
```

Now add the following after that:

```
// register all material names
tbox_Array<string> name_array(4);
name_array[0] = "copper";
name_array[1] = "gold";
name_array[2] = "liquid";
name_array[3] = "gas";
d_visit_writer->registerMaterialNames(name_array);
```

We now return to the implementation of the concrete materials data writer strategy object. This concrete object is responsible for providing an implementation of the method
`packMaterialFractionsIntoDoubleBuffer()`. This method packs the material fractions for the specified material into an already allocated buffer, over the specified patch and region. The data is packed in column major order (see Section 4.4). If material data is used, then it is assumed that material data is defined on all patches. All material data must be cell-centered.

The return value of `packMaterialFractionsIntoDoubleBuffer()` is important. If there is none of the specified material M in any of the cells of the specified patch P, i.e. material_fraction(M) = 0.0 for all cells on P, then return the enumeration constant *VisMaterialsDataStrategy::ALL_ZEROS*. If the material_fraction(M) = 1.0 for all cells on P, return *VisMaterialsDataStrategy::ALL_ONES*. Otherwise, return *VisMaterialsDataStrategy::SOME*. These return values are used to compress the materials data dumped to disk, dramatically economizing disk space. For material data this is extremely important!! When a packing method returns *VisMaterialsDataStrategy::ALL_ONES* or *VisMaterialsDataStrategy::ALL_ZEROS*, the data in the buffer is ignored and the VisIt data writer writes a single integer code to disk for that material and patch rather than the entire buffer.

The code fragment from the application *Applic* shown next implements the concrete materials data strategy object. Note that *Applic* inherits from `VisMaterialsDataStrategy`.

```
// Applic.h

#include "VisMaterialsDataStrategy.h"

class Applic :
   public VisMaterialsDataStrategy

 . . .

// Applic.C

int Applic::packMaterialFractionsIntoDoubleBuffer(
      double *dbuffer,
      const hier_PatchX& patch,
      const hier_BoxX& region,
      const string& material_name)
{
   if (material_name == "gold") {

      /*
       * Pack the material fractions for "gold" for each cell in
       * this box region into dbuffer. If there is no "gold" in a
       * cell, enter 0 for that cell.
       *
       * If there is no "gold" in any of the cells of this patch,
       *     set return_value = VisMaterialsDataStrategy::ALL_ZEROS
       * If "gold = 1.0" for all cells on this patch,
       *     set return_value = VisMaterialsDataStrategy::ALL_ONES
       * Otherwise
       *     set return_value = VisMaterialsDataStrategy::SOME
       *
       * If a non-zero ghost cell width vector was specified when
       * registerMaterialNames() was invoked, then ghost data must also
       * be packed into dbuffer.
       */

      return return_value;

   } else if (material_name == "copper") {

      /*
       * As above, but for "copper".
       */

      return return_value;
   }

   .  .  .

   } else {
      TBOX_ERROR("Applic::packMaterialFractionsIntoDoubleBuffer"
         << "\n    application with name " << d_object_name
         << "\n    unknown material_name " << material_name << "\n");
   }
```

At the time of each dump, this packing method will be called once for each patch for each material.

In certain simulations, such as air flow around a solid object (such as an air foil or a building), one may be tempted to register just one material — the solid object. However this is not correct since the volume fractions in every cell must add up to 1.0. Nevertheless, to support this special case, the VisIt data reader detects if only one material M is registered and automatically creates a second dummy material called "void" whose material fractions mf are $mf(\text{"void"}) = 1 - mf(M)$. This feature of VisIt only works when a single material is registered. For all other cases the total of the material fractions for every cell must equal 1.0.

5.3 Steps to Add Material State Variable Data

If you are not completely clear on what a material state variable is, please read Section 5.1 before continuing. To add a material state variable, invoke the method `registerMaterialStateVariable()`. This method specifies the name of the variable, and its depth (NDIM, or NDIM*NDIM) if it is a vector or tensor, and optionally a scale factor. The variable name must not have been already registered as a variable name for any other variable. It is a requirement that `registerMaterialNames()` be invoked before `registerMaterialStateVariable()`. (Note this method is used only to register state variables related to materials, not for state variables related to species; species state variables are described in Section 5.5.)

Before we describe the packing method for material state variables, we show an example registration of a single material state variable: *temperature*. Note that this registration, shown next, actually sets the stage for 4 temperature fields, one for each of the four materials (*Copper*, *Gold*, *Liquid* and *Gas*). The following invocation is placed in the application class file *Applic.C* immediately after the call to `registerMaterialNames()` shown above in Section 5.2.

```
      // register a material state variable
      d_visit_writer->registerMaterialStateVariable("temperature");
```

In Section 5.2, we also created and registered a *MaterialsDataWriter* object, and implemented the method `packMaterialFractionsIntoDoubleBuffer()`. For material state variables, we need to add the implementation of the method `packMaterialStateVariableIntoDoubleBuffer()`. A code fragment from the application file *Applic.C*, shown next, implements this packing method.

```
// Applic.C

void Applic::packMaterialStateVariableIntoDoubleBuffer(
      double *dbuffer,
      const hier_PatchX& patch,
      const hier_BoxX& region,
      const string& material_name,
      const string& state_variable_name,
      const int depth_index)
{
   if ((state_variable_name == "temperature") && (material_name == "gold")) {

      /*
       * Pack the "temperature" for "gold" for each cell in this
       * box region into dbuffer. If there is no "gold" in a cell,
       * some value must be placed in dbuffer for that cell
       * to serve as a place holder, however since that value will
       * be skipped over during visualization, its value does not
       * matter.  The buffer must contain a value for every cell in
       * the patch.
       *
```

```
        * This method will not be called if the patch does not contain
        * any "gold". Therefore this method has no return value and is
        * declared as void.
        *
        * If a non-zero ghost cell width vector was specified when
        * registerMaterialNames() was invoked, then ghost data must be
        * packed into dbuffer.
        */

   } else if ((state_variable_name == "temperature") &&
                              (material_name == "copper")) {
      /*
       * As above, but for "copper".
       */
   }

   .  .  .

   } else {
      TBOX_ERROR("Applic::packMaterialStateVariableIntoDoubleBuffer"
         << "\n    application with name " << d_object_name
         << "\n    unknown material_name, state_variable_name pair "
         <<        material_name << "," << state_variable_name << endl;
   }
}
```

At the time each dump is created, this packing method will be invoked once for each patch for each (*material*, *state_variable*) pair. If the state variable is registered as a vector or tensor, this method will be called once for each component of the vector or tensor with the argument *depth_index* set to indicate which component to pack. In cells where a specific material does not exist, i.e. has a zero fraction, the state variable for that material will not be accessed by VisIt, therefore it does not matter what value is written for such cells. However, a value must be packed into the buffer for every cell in the patch. This packing method will not be called if the entire patch does not contain any of the specified material.

As with the packing method for material fractions, the buffer will have already been allocated and the data is to be packed in column major order. All material state variable data must be cell-centered.

5.4 Steps to Add Species

It is important to have carefully read Section 5.1 before continuing.

To add species for a particular material, invoke the method `registerSpeciesNames()`. This method requires two arguments: an array of strings, the names of the species, and the name of the material to which the species belong. It is a requirement that `registerMaterialNames()` must be invoked before `registerSpeciesNames()`. First, we show an example of species registration, following that we describe the packing method for species fractions.

In our example which we started earlier, we will give the material *gas* three species: *oxygen*, *nitrogen*, and *methane*. The material *liquid* will have two species: *salt* and *h2o*. The following code fragment appears in the application class file after the call to `registerMaterialNames()` shown above in Section 5.2.

```
      // register the three species of the material gas:
      name_array.resizeArray(3);
      name_array[0] = "oxygen";
      name_array[1] = "methane";
      name_array[2] = "nitrogen";
      d_visit_writer->registerSpeciesNames("gas",name_array);
      // register the two species of the material liquid
      name_array.resizeArray(2);
      name_array[0] = "salt";
      name_array[1] = "h2o";
      d_visit_writer->registerSpeciesNames("liquid",name_array);
```

In Section 5.2, we created and registered a *MaterialsDataWriter* object, and implemented the method `packMaterialFractionsIntoDoubleBuffer()`. For species, we need to add the implementation of the method `packSpeciesFractionsIntoDoubleBuffer()`. A code fragment from the application file *Applic.C*, shown next, implements this packing method.

```
int Applic::packSpeciesFractionsIntoDoubleBuffer(
      double *dbuffer,
      const hier_PatchX& patch,
      const hier_BoxX& region,
      const string& material_name,
      const string& species_name)
{
   if ((species_name == "methane") && (material_name == gas)) {

      /*
       * Pack the species fractions for species "methane" of material
       * "gas" for each cell in this box region into dbuffer. If
       * there is no "gas" in a cell enter 0 for that cell.
       *
       * If there is no "methane" in any of the cells of this patch,
       *     set return_value = VisMaterialsDataStrategy::ALL_ZEROS
       * If "methane = 1.0" for all cells on this patch,
       *     set return_value = VisMaterialsDataStrategy::ALL_ONES
       * Otherwise
       *     set return_value = VisMaterialsDataStrategy::SOME
       *
       * If a non-zero ghost cell width vector was specified when
       * registerMaterialNames() was invoked, then ghost data must also
       * be packed into dbuffer.
       */

      return return_value;

   } else if ((species_name == "oxygen") && (material_name == gas)) {

//       As above, but for "oxygen".

      return return_value;
   }

   .  .  .

   } else {
      TBOX_ERROR("Applic::packSpeciesFractionsIntoDoubleBuffer"
         << "\n    application with name " << d_object_name
         << "\n    unknown material_name, species_name pair "
         <<        material_name << "," << species_name << endl;
   }
```

At the time each dump is created, this packing method will be invoked once for each patch for each species for each material that has species. As with the packing method for material fractions, the buffer will have already been allocated and the data is to be packed in column major order. All species fraction data must be cell-centered.

The return values for `packSpeciesFractionsIntoDoubleBuffer()` are the same as for material fractions described in Section 5.2 and are described in the comment in the code fragment for packing just above.

5.5 Steps to Add Species State Variables

Species state variables are treated in a rather unique way by VisIt, so it is important to have carefully read Section 5.1 before continuing.

To add a species state variable for a particular material, invoke the method `registerSpeciesStateVariable()`. This method requires one argument, the name of variable. This variable name must have been registered previously using `registerPlotScalar()`, `registerPlotVector()`, or `registerPlotTensor()`, or their derived variable counterparts.

Note the difference from a material state variable: Registering a material state variable means a separate variable field must be dumped for each material, whereas registering a species state variable is purely informational — it just informs VisIt that an already existing variable can be treated as a species state variable with the significance described in Section 5.1.

Continuing our ongoing materials example, we will register one species state variable: *pressure*. The following registration goes in the *Applic.C* file following the registration of the species names.

```
// register a species state variable
d_visit_writer->registerSpeciesStateVariable("pressure");
```

5.6 Special Topic: Node-Centered Materials-Related Data

The VisIt system currently requires that material fractions, species fractions, and material variables be cell-centered. In simulations, material fractions usually are cell-centered so this is not a problem. However, there may be simulations where species fractions or material variables may be node-centered. For those cases, since VisIt requires such data to be cell-centered, the application will need to interpolate the node-centered species fractions to cell-centered values in the `packSpeciesFractionsIntoDoubleBuffer()` method. Similarly for material state variables in the `packMaterialStateVariableIntoDoubleBuffer()` method. There is no limitation on the centering of species state variables; VisIt will do its best to intelligently interpolate these values appropriately.

5.7 Special Topic: Using Species Only

Given the difference between materials and species, i.e. the heterogeneity or homogeneity of the mixture referred to in Section 5.1, it might be appropriate for an application to only have species, and no materials, e.g. a simulation whose domain is entirely submerged in a homogeneous fluid. In order to satisfy VisIt's requirement that species be subcomponents of some material, the work-around is to just declare some material, e.g. gas, and then just set the material fractions for gas to 1.0 everywhere. This can easily be done, as described in Section 5.2, by setting the return value of `packMaterialFractionsIntoDoubleBuffer()` to the *VisMaterialsDataStrategy::ALL_ONES*. In this case, it is not necessary to actually pack any data into the buffer, since the VisIt data writer ignores the contents of the buffer for this particular return value. There is no significant storage penalty for defining a material in this instance, since the VisIt data writer compresses such data.

6 Writing Ghost Data

Visualizing ghost data may be helpful in debugging: for example to check the validity of boundary conditions, to check the correctness of interpolation at boundaries between patches at different patch levels, and for examining how deformed moving grids match up at patch boundaries.

For state variable data defined on the SAMRAI hierarchy, the data writer automatically detects the presence of ghost data and includes ghost data in the dump if it is available. This behavior can be overridden by setting the optional *omit_ghost_data* argument to $TRUE$ when the data is registered. The setting of the *omit_ghost_data* argument does not have to be the same for all variables.

In order for the VisIt data writer to be able to access ghost data, it is necessary (1) that the call to `writePlotData()` be placed in the appropriate context (a context where ghost data exists) in the application code, and (2) that the data writer registration calls (such as `registerPlotScalar()` for example) pass an index into the patch data array on the AMR hierarchy in which ghost data exist.

The method `registerNodeCoordinate()` for deformed data does not take an *omit_ghost_data* argument since VisIt requires coordinates of ghost cells if they exist.

For derived data, it is the user's responsibility to advise the VisIt data writer whether or not ghost data will be dumped. This is done in the registration call for derived data by setting the optional ghost_cell_width argument to the proper ghost cell width vector. By default, this vector is the zero vector.

If ghost data is to be dumped for material data, it is necessary to specify the ghost cell width vector in the call to `registerMaterialNames()`. By default, this vector argument is the zero vector. The ghost cell width so specified will then be used for **all** material-related data.

When implementing the various packing routines for the derived and material data strategy objects, it is important to keep in mind that if ghost cell data has been specified in the relevant registration calls, then ghost data must be packed in the packing routines.

If ghost data is to be visualized, it is important that the application specify data values for **all** ghost cells or nodes on a patch. So, for example, an application should avoid only specifying initial values for ghost cells or nodes on the faces of the box region. The SAMRAI function `fillAll()` is useful in this regard.

As described in Section 7.8, while visualizing data using VisIt, ghost data (if available) may be turned on or off interactively on a variable by variable basis.

7 Using VisIt with SAMRAI Data Files

After a brief description in Section 7.1 of how to access VisIt and use it to look at SAMRAI data, we discuss a number of helpful topics including slicing and browsing, the subset dialog box, materials visualization, picking, and looking at ghosts. Then in Section 7.10, we explain the rudiments of making a movie, and in Section 7.11 we describe how to run VisIt remotely and/or in parallel. For more information on using VisIt please see the VisIt web site: *http://www.llnl.gov/visit*. For problems or help, contact *visit-help@llnl.gov*.

7.1 Finding and Starting Up VisIt

The VisIt system is described in detail in the VisIt User's Manual; in addition, there is a VisIt Getting Started Manual. Both of these manuals as well as other useful information on VisIt are available at *http://www.llnl.gov/visit*. However, you may find the information given below, tailored to a SAMRAI user, to be helpful in getting started.

At CASC, on the *tux* machines, the path to VisIt is */usr/apps/visit/bin/visit*. On the main LC machines at LLNL, VisIt is installed under */usr/gapps/visit/bin/visit*. The VisIt team will be happy to install the binaries on any other LLNL machine. For other locations that may not have VisIt, the VisIt source or executables can be downloaded from the above web site.

Start VisIt by invoking *visit* at the command line with no arguments. Once VisIt starts up, from the *File* menu, choose *Select file*, and then navigate to the directory specified in the application code when the VisIt data writer object was created (see Section 3.2, step 1). Click *Remove all*, highlight the file named *dumps.visit* in the *Files* list, and click *Select*. Now *dumps.visit* will appear in the *Selected files* list as the only file. Click *OK*. Now you will be back in the main GUI. Double click on the file that appears in the *Selected files* list in the upper left. After a slight delay as VisIt reads in the summary information, all the time steps will appear just below where you just clicked. Then after another slight pause – be patient – a view of the patch boundaries will appear in the main window. With this image, try rotating, zooming and panning (shift left mouse). Now try swapping the foreground and background colors (top menu bar, button with black and white triangles and 2 green arrows). Turn on the *toggle bounding box* button (the one just to left of the *swap background* button) and try rotating again. Pull down the *File* menu and select *Save Window*, do it again, and then look in your directory.

Now select any time step and then choose one of the *Plots*, say *Contour: Density* (i.e. display a contour of density). Click the *Draw* button and you should see a density contour plot appear. Now click the VCR "forward" button to start an animation over all time steps. Click the VCR "stop" button to stop the animation. To adjust the opacities of the contours, click the *PlotAtts* button and select *Contour*. To select specific contours, choose *Select by Value* on the plot attributes menu, and then enter values separated by spaces, adjusting the opacities if necessary. (The VCR needs to be stopped in order to make these modifications.)

After rotating, panning and/or zooming the image, if you want to restore the image to its original orientation, pull down the *Controls* menu and select *View*; then on the resulting dialog box, select the *Advanced* tab and then click on the Reset view button.

7.2 Changing Data Sets, Adding New Variables and Time Steps

To remove a data set from VisIt, from the *File* menu, choose *Select file*, click *Remove all*, and then *OK*.

If your simulation registered a new variable after the first dump, you will not see it in VisIt unless you do the following. Say you registered "pressure" after time step 5, then when you start up VisIt, "pressure" will not appear in any of the pull down menus that list variables for any time step. Here is what you need to do. Double click on a time step where "pressure" is in fact in existence, for example the very last time step, then press the ReOpen button. Now "pressure" will appear on the pull down lists of variables, even for time steps where it does not exist. VisIt will crash if you generate a video animation of a variable that does not exist on all time steps, so be cautious. The VisIt team has been notified about this problem and it should be fixed soon.

If VisIt is started up on dumps from a running SAMRAI application, new time steps can be added to VisIt as they are created by clicking the *ReOpen* button, thus allowing the progress of the simulation to be monitored. If you are running a video animation, you need to stop the animation before clicking the *ReOpen* button.

7.3 Getting Information about a Dump File

Useful information about each dump can be found by choosing *File Information ...* from the *File* menu. This information includes: the time and date of the dump, the simulation time, the spatial extents of the data set, names of all variables, and material-related data, and for each variable: its scale factor, extents, ghost cell information, centering, etc.

7.4 Slicing and Browsing

When looking at 3D data it is often very helpful to look at a 2D slice of the data. Use the *Slice* operator to do so. After applying the slice operator, it is important to go to the Ópatts dialog box for *Slice* and unclick the *Project to 2D* button. To move the position of the slice, use the *Plane Tool* which you get by selecting the icon on the lower top row of the graphics window. Use the help "balloon" for an icon to see which one is the *Plane Tool*. To move the plane, click the mouse on the **origin** of the plane tool arrows, and drag the plane.

Another very useful slicing tool is the *ThreeSlice* operator, which you can apply to a pseudocolor or contour plot for example. Use the *Point Tool* to drag the "three-slice", not the *Plane Tool*.

7.5 The Subset Dialog Box

The Subset Dialog Box is extremely helpful. In the *Controls* Menu at the top of the main VisIt window, select *Subset*. In the resulting dialog box, you can click on patches or levels. (If you are using materials, you will also see materials there too. We discuss this is Section 7.6.) This will bring up a new set of selections where you can turn on or off all or some levels or patches. These selections apply to whatever you may wish to visualize, the mesh, a pseudocolor plot, etc. The levels or patches that are currently selected are said to be the “active” levels or patches.

In addition to seeing the mesh on certain patches/levels, you may select either the *Boundary* or *Filled Boundary* Plot of levels or patches. The *Boundary* plot shows the boundaries between active patches.

When you select a subset of the patches for viewing, the image of the subset may expand to fill the entire window. To avoid this and keep the viewing scale constant, click the radio button *view* under *Maintain limits*.

7.6 Materials Visualization

Bring up the Subset Dialog Box as described in Section 7.5. Click on “materials” or “species”. You should now see a list of the materials or species that are available for visualization. To look at the material data, use *Filled Boundary* plot and select “materials”. Whatever materials you have selected in the Subset Dialog Box (i.e. the “active” materials or species) will be displayed. You may want to do a 2D slice of 3D data and browse it as described in Section 7.4, or you may choose to make the plot semitransparent using the *PlotAtts* dialog box for *Filled Boundary*.

The *Boundary* plot will show you the boundaries between materials. The legend is a bit confusing as it arbitrarily chooses one of the two materials which define the boundary to label the boundary; so if you have 4 materials, only 3 may appear on the legend. For species, you may do a pseudocolor, contour or volume plot. Again only the active species will be used.

More information on viewing materials is given in Section 7.7 on picking.

7.7 Picking

Picking refers to clicking the mouse on a VisIt plot and determining information about the point clicked on. This information includes the spatial coordinates of the point, the i, j, k indices of the enclosing cell, and the node indices of the nodes that define the cell. In addition, the level and patch number of the pick point are given. If a variable plot is selected (highlighted) in the *Active Plots* wiindow, then the value of the variable at that point is given. If a plot involving materials is highlighted, pick will tell you the percentage of each material in the picked cell. If species are selected, you will get information on the different species in the picked cell.

To do picking here is what you need to do. First, go to the large graphics window and find the compass-like icon on the far left of the second row of icons at the top of the window. When this button is selected, you are in the normal mouse-driven navigational mode. To go *Pick* mode, select either of the two icon buttons next to the compass-like icon. The one with *Z* is for cell-picking mode, and the one with *N* for node-picking mode. Now you need to do one more thing. Pull down the *Controls* menu and select *Pick*. On the resulting dialog box, under **both** *Display for Nodes:* **and** *Display for Zones:*, unclick the *ID* buttons and click “on” the *Domain-Logical Coords* buttons. Then click *Apply*. Now the next mouse pick will show the information described above. For easier picking, you may want to use the *Slice* operator on 3D data, and/or the*Subset* dialog box to turn on only selected levels and or patches. After you are done picking, select the compass-like icon again to enable mouse-driven navigation again.

7.8 Looking at Ghosts

To look at ghosts in VisIt, assuming you have dumped ghost data, use the Operator *Inverse Ghost Zone* on the *Operators* pull down menu. (In VisIt the term “zone” refers to what is called a “cell” in SAMRAI.) If this operator does not appear in the *Operators* pull down menu, pull down the *Options* menu at the top of the control GUI, and choose *Plugin Manager*. In the *Plugin Manager* dialog window, select the *Operators* tab in the middle and click on the *Inverse Ghost Zone* box. Now the *Inverse Ghost Zone* operator will appear in your regular *Operators* menu. If you want it to remain there for future uses of VisIt, go back to the *Options* menu at the top and select *Save Settings*. This will store a configuration file in *.visit* in your home directory.

Now bring up the *OpAtts* menu and select *Inverse Ghost Zone*. In the dialog box that results, you will see that you may select to see either only the ghost zones or both real and ghost zones. Remember the

Inverse Ghost Zone operator only works when it has been applied to some variable or the mesh, just like the *Slice* operator. So if you don't want to see any ghost zones, then do not apply the *Inverse Ghost Zone* operator to that data. This capability mean you may turn on or off ghost data interactively on a variable by variable basis.

To check if data in a ghost cell of one patch is consistent with the corresponding data in a different patch, try this. Using the *Subset* dialog box, select only these two patches, and then use *Inverse Ghost Zones* to turn on "only ghost cells" for one patch; do not use ghost cells at all for the other patch. Here are two ways to make the comparison. Turn on *toggle bounding box* so the system redraws the screen continuously while the image is rotated. Now rotate the image and check the area of overlap between ghost cells and non-ghost cells to see if you get a "flashing" effect. If so, this may indicate that the data in the cells that flash are not identical. You can also check the data in these cells by "picking".

At present, the VisIt team is upgrading the ghost data visualization capabilities of VisIt and we can expect more enhancements. You can contact *visit-help@llnl.gov* for the current status and also make a request for a feature you would like.

7.9 Problems?

If at any time, the VisIt graphical interface looks strange or is not working, as sometimes occurs, one thing to try is to move the *.visit* directory in your home directory to some other name and then restart VisIt. VisIt will run fine even if there is no *.visit* directory. VisIt stores various preferences in the *.visit* directory and you may inadvertently have caused VisIt to store some preference which is now causing you a problem.

If you were viewing a 2D data set and then decide to load a 3D data set, or vice versa, you will need to *delete* the existing plots before loading the new data set. If this does not work, try quitting VisIt and then restarting it with the new data set by itself.

If all else fails, contact *visit-help@llnl.gov.*

7.10 Making a Movie

To make a movie, set up your plot just like you want it to appear in the movie, select time step 0 or whatever time step you want the movie to start from, and then from the *File* menu, select *Save session.* The saved session file will appear in a directory named *.visit* in your home directory. At the command line in the *.visit* directory, type:
visit -movie -geometry 800x800 -sessionfile sessionfile_name -format mpeg -output mymovie
Other output file options such as *tif*, etc are available. To see all the options, type *visit -movie.* This simple method also allows you to save movies that involve keyframing. A more powerful approach is to write a Python script that tells VisIt everything that you want to do for your movie. This allows very sophisticated movies including fly-by sequences. If you write a movie script called *movie.py*, you make VisIt run it by the following command:
visit -cli -geometry 1024x1024 -nowin -s movie.py
A manual on making VisIt movies with Python is available on request from visit-help@llnl.gov.

7.11 Running VisIt Remotely and/or in Parallel

If your simulation runs on a distributed machine, you may leave your data there and visualize it on your desktop workstation. Not only that, you may run VisIt in distributed mode so the visualization compute engine runs in parallel remotely, while the images are displayed on your desktop. Here is how to do that.

First be sure the path to VisIt is included in your path environment variable on the **remote** machine. On the Livermore Computing (LC) distributed machines, the path is slightly different than at CASC; the path to use on the LC machines is */usr/gapps/visit/bin.*

Now on your **local** desktop, start up VisIt. From the *File* menu, choose *Select file.* In the window that pops up, enter the distributed machine name in the *Host* text field and press *Enter* — after a pause you will be prompted for a password (unless you are password-less on that machine). After another pause, your files and directories on the remote machine will appear in the File Selection window. Now proceed, as described in Section 7.1, to select the file you wish to view. When you double click on the time step in the main GUI, a dialog window will pop up for you to choose a *Host profile.* This window will list several choices, e.g. on *frost* you can choose *serial, parallel pdebug,* or *parallel pbatch*, (batch mode is useful if you want to create a movie.) You can also select the number of processors you wish to run on. (You will have a default profile which specifies number processors, etc. for each machine, but you may change any of the preferences at this point.) VisIt will automatically distribute the SAMRAI patch data over the number of processors that you choose. So there is no point in running with more processors than patches.

(You can find out which machines VisIt will run on in parallel by choosing *Host profiles* from the *Options* menu. Currently at CASC, the following machines are listed: *blue, frost, gps, ilx, mcr, pengra, pvc, riptide,* and *snow.*)

Once you have completed the dialog window for the *Host profile*, shortly thereafter an image of the patch boundaries will come up. Now you may use the various plots and operators described in Section 7.1. Remote visualization using VisIt is described in more detail in the VisIt User's Manual.

8 Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract no. W-7405-Eng-48.