Generating Vizamrai Data Files in SAMRAI

Introduction

Vizamrai is a fairly extensive visualization and analysis tool for data generated on a structured AMR patch hierarchy. It is distributed as part of the SAMRAI library. This document describes how to use the Vizamrai data writer capabilities in a SAMRAI application. Experienced SAMRAI users should note that in SAMRAI version 1.3.1, capabilities were added to Vizamrai and the SAMRAI Vizamrai data writer classes to accommodate vector quantities. The description in this document supercedes any previous versions.

Questions and requests for additional information should be sent via email to `samrai@llnl.gov`.

Generating Vizamrai Files in SAMRAI

The primary class used in SAMRAI for Vizamrai data file generation is called `appu_CartesianVizamraiDataWriterX`. This class applies when the underlying mesh geometry is Cartesian; that is, it is managed by a `geom_CartesianGridGeometryX` object. This class supports two kinds of data quantities. The first is data that resides on an AMR patch hierarchy at the point that Vizamrai files are written. The second is *derived* data that can be computed using existing data on an AMR patch hierarchy. Writing data of the type requires no user intervention when generating a plot file. The second data type requires that a user implement a concrete class derived from the `appu_VizamraiDerivedDataStrategyX` abstract interface. In an overloaded function of that class, a user computes the derived data quantity over a given box region on a patch and writes it to a specified file stream.

Both derived and regular plot quantities can be either vectors or scalars.

Before a Vizamrai data writer object can be used to generate Vizamrai data files, it must be constructed and initialized. Initialization involves things like setting information about the levels in the AMR hierarchy that will be plotted, setting the type of the data to write out (e.g., double or float), setting the directory into which data files are written, and registering variable quantities to send to the plot file. After initialization, the object can be used to generate a series of data files during the execution of some simulation code.

Typical usage of the Vizamrai data writer object involves several steps among which there are slight variations. Here, we list the basic steps. Then, we remark on some of the variations that are allowed.

1. Create a `appu_CartesianVizamraiDataWriterX` object.
2. Set the finest level to plot with the member function `setFinestLevelToPlot()`. This is the finest level allowable *in any* subsequent plot file. Note that only levels existing in the hierarchy when a data file is generated and which are coarsen than the finest level will be written.

3. For each level numbered one to the finest level number to plot, specify the ratio between the index space of the level and the next coarser level in the AMR hierarchy. This information is needed to scale the AMR levels properly in Vizamrai. The member function to use is `setRatioToCoarserLevel()`.
4. Set the type of data to put in the Vizamrai file, using either function `setPlotDataToDouble()` or `setPlotDataToFloat()`.
5. Set the directory for the plot files, if desired, using the function `setDirectoryName()`.
6. If any *derived* quantities will be generated, one can set the concrete derived data writer object using function `setDerivedDataWriter()`. Alternatively, one may specify a derived data writer when registering each derived data quantity. This provides some flexibility in using different derived data writer objects for different quantities if desired.
7. The `resetLevelPlotVariable()` member function is provided to allow a plot quantity to live at different patch data indices on different levels. Calling this routine redefines the patch data index for a given variable on a given level that will be written to a plot file. Before this function is called, the variable must be registered using the `registerPlotVariable()` function as described earlier. **NOTE: This functionality was added in SAMRAI version 1.2.1.**
8. Register data quantities to plot using functions `registerPlotScalar()`, `registerPlotVector()`, `registerDerivedPlotScalar()`, or `registerDerivedPlotVector()`. The `registerDerivedPlot...` functions require a variable name string identifier and a derived data writer if not specified earlier (see step 5). A derived plot vector also requires the number of vector components to be written. The `registerPlot...` functions require a variable name string identifier and an integer patch data index on the AMR hierarchy. Other arguments may include an integer depth index, and a double scale factor. When a scalar plot data item corresponds to a patch data array object with depth greater than one, the depth index must be specified. When a vector quantity is registered all data components will be written out. When a scale factor is given, each data value will be multiplied by this factor before it is written to the file.

After all necessary data writer parameters are initialized, the writer will generate a Vizamrai data file when the member function `writePlotData()` is called. Minimally, a hierarchy and a file name must be passed to this routine. One can also supply an integer file extension and a plot time, both of which are useful when generating visualization files for time-dependent applications. When an integer file extension is given, the data file name format will be "filename.extension". If no extension is given, only the file name is used. Thus, to avoid overwriting files when generating a series of files, either the file name or the extension must be different for each call to `writePlotData()`.

Remarks:

- Since Vizamrai can only handle cell-centered data, the Vizamrai data write class only applies to cell-centered data.
- In actuality, Vizamrai processes plot data in "float" format. Thus, setting the plot type to "double" increases the size of Vizamrai data files and adds nothing to the plotting capabilities. The ability to save data in double format is included here for future data post-processing capabilities.

- Like other classes, “noprefix” header files are available so that may use the name `CartesianVizamraiDataWriter` instead to avoid the SAMRAI prefix and dimension qualifiers.
- Step 2 above in which the finest plot level is set is optional. If the function `setFinestLevelToPlot()` is not used, the finest level will be set to the finest level specified in any call to the function `setRatioToCoarserLevel()`.
- Step 3 above in which the ratio between the index space on each level and the next coarser level is optional. If the function `setRatioToCoarserLevel()` is not used, then the finest level and ratio information will be determined by the state of the hierarchy that is passed to the first call to the function `writePlotData()`.
- If no directory is specified (step 5 above), files will be written into the current directory.
- Registering more than one variable with the same string name identifier is not allowed. If this is attempted, an error message results and the program will abort.

Example of Vizamrai Data File Writer Usage

We conclude with a brief illustration of how the Vizamrai data writer capabilities are used in the SAMRAI Euler example application. The relevant portions of the main program are as follows:

```
/*
 * Parse input file.  Get Vizamrai file dump interval,
 * file name, and directory name.
 */

Pointer<Database> input_db = new tbox_InputDatabase("input_db");
InputManager::getManager()->parseInputFile(input_filename,
                                           input_db);

Pointer<Database> main_db = input_db->getDatabase("Main");

. . .

int viz_dump_interval = 0;
if (main_db->keyExists("viz_dump_interval")){
   viz_dump_interval = main_db->getInteger("viz_dump_interval");
}

string viz_dump_filename;
string viz_dump_dirname;
if ( viz_dump_interval > 0 ) {
   if (main_db->keyExists("viz_dump_filename")) {
      viz_dump_filename = main_db->getString("viz_dump_filename");
   }
   if (main_db->keyExists("viz_dump_dirname")) {
      viz_dump_dirname = main_db->getString("viz_dump_dirname");
   }
}

const bool viz_dump_data = (viz_dump_interval > 0)
                           && !(viz_dump_filename.empty());
```

```
. . .

/*
 * Create relevant algorithm objects used in simulation.
 */

Pointer<CartesianGridGeometry> grid_geometry = . . .

Pointer<PatchHierarchy> patch_hierarchy = . . .

Pointer<CartesianVizamraiDataWriter> viz_data_writer =
   new CartesianVizamraiDataWriter("Euler Viz Writer");

Euler* euler_model = new Euler("Euler",
                               input_db->getDatabase("Euler"),
                               grid_geometry,
                               viz_data_writer);

. . .

Pointer<GriddingAlgorithm> gridding_algorithm = . . .

. . .

/*
 * Set up Vizamrai plot file writer.
 */

if (viz_dump_data) {
   viz_data_writer->setDirectoryName(viz_dump_dirname);
   viz_data_writer->setPlotDataToFloat();
   viz_data_writer->setFinestLevelToPlot(
                    gridding_algorithm->getMaxLevels()-1);
   for (int ln = 1; ln < gridding_algorithm->getMaxLevels(); ln++) {
      const IntVector& lratio =
         gridding_algorithm->getRatioToCoarserLevel(ln);
      viz_data_writer->setRatioToCoarsestLevel(ln, lratio);
   }

   viz_data_writer->setDerivedDataWriter(euler_model);
}

. . .

/*
 * 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 out data files for plotting.
      */

     if (viz_dump_data && (iteration_num % viz_dump_interval) == 0) {

        viz_data_writer->writePlotData(patch_hierarchy,
                                       viz_dump_filename,
                                       iteration_num,
                                       loop_time);
     }

  }
```

In this example, we first parse the input file and read information defining the plot file sequence and data file names from the input file. Second, we create the objects needed to run our simulation. Note that we pass a pointer to the `CartesianVizamraiDataWriter` object to the Euler class. We cache this pointer so that we can register plot variables with the writer at the same time we register variables with the integrator class. This is described later. Third, we initialize the state of the Vizamrai data writer object. The operations presented illustrate the use of all functions in the Vizamrai data writer interface. Note that not all of these calls are necessary in every circumstance. Please see the remark section above for more information. Fourth, we loop over timesteps in the simulation and write out plot files at the desired intervals. We pass the step iteration count to the `writePlotData()` routine so that each plot file has a suffix associated with the proper timestep number.

The Euler routine, `registerModelVariables()`, contains the registration of plot variables:

```
  void Euler::registerModelVariables(
      HyperbolicLevelIntegrator* integrator)
  {

     /*
      * Register variables with the Hyperbolic Level integrator.
      */

     . . .

     Pointer<VariableContext> plot_context =
        integrator->getPlotContext();

     VariableDatabase* vardb = VariableDatabase::getDatabase();

     d_viz_writer->registerPlotScalar(
                 "Density",
                  vardb->mapVariableAndContextToIndex(
                         d_density, plot_context));
```

```
      d_viz_writer->registerPlotVector(
                    “Velocity",
                    vardb->mapVariableAndContextToIndex(
                    d_velocity, plot_context));

      d_viz_writer->registerPlotVariable(
                    "Pressure",
                    vardb->mapVariableAndContextToIndex(
                           d_pressure, plot_context));

      d_viz_writer->registerDerivedPlotScalar("Total Energy");

      d_viz_writer->registerDerivedPlotVector("Momentum");

   }
```

In this routine, we register the Euler state variables, density, velocity, and pressure, for plotting. These data quantities exist on the hierarchy at all times. So we give their name and patch data index. The index is determined by mapping the variable and plot context to the index in the variable database. Density and pressure are scalar quantities and velocity is a vector. Since these variables exist on the AMR patch hierarchy when a data file is written. So no additional user interaction is required to generate the Vizamrai data files. Lastly, we register “Total Energy” and “Momentum” as derived quantitities. Since these quantities do not reside on the AMR patch hierarchy all the time, the Euler class must supply the routine `writeDerivedDataToStream()` that computes the total energy and the momentum and writes them to the given file stream. For more details, please consult the source code in the Euler sample application.

```
   void Euler::writeDerivedDataToStream(
      AbstractStream& stream,
      const Patch& patch,
      const Box& region,
      const string& variable_name,
      int data_depth,
      int plot_type)
   {

      if (variable_name == “Total Energy”) {

         /*
          * Compute total energy on the box “region” using the Euler
          * state data on the patch and write it to the given stream.
          */

          . . .

      }

      if (variable_name == “Momentum”) {

         /*
          * Compute each component of momentum on the box “region”
using
          * the Euler state data on the patch and write it to the given
```

```
         * stream.
         */

         . . .
      }


   }
```

We end with one final note about Vizamrai data files generated in parallel. When running an application in parallel, a different Vizamrai file will be generated for each processor. Each file will contain the data from the patches that are local to the processor. Each file will have a suffix ".xxxx" indicating the MPI process number. Before the data can be viewed using Vizamrai, the data files need to be concatenated into one file. There are scripts for doing this in the directory /SAMRAI/tools/scripts. That directory also contains a short README file describing how to use the scripts.

Changes to Vizamrai file generation after SAMRAI Version 1.2

The member function `resetLevelPlotVariable()` was added to allow a single plot quantity to live at different locations in the patch data array on different levels in and AMR patch hierarchy. Calling this routine redefines the patch data index for a given variable on a given level. Before this function is called, the variable must be registered using the `registerPlotVariable()` function.

Although it is not used in the Euler example application, we can illustrate how this new function is used by recalling the Euler routine, `registerModelVariables()`:

```
void Euler::registerModelVariables(
    HyperbolicLevelIntegrator* integrator)
{
   . . .

   Pointer<VariableContext> plot_context =
      integrator->getPlotContext();

   VariableDatabase* vardb = VariableDatabase::getDatabase();

   d_viz_writer->registerPlotScalar(
                "Density",
                 vardb->mapVariableAndContextToIndex(
                        d_density, plot_context));

   /*
    * Reset density plot data index on level 5.
    */
   d_viz_writer->resetLevelPlotVariable(
                "Density",
                5,
                vardb->mapVariableAndContextToIndex(
                       d_density, new_plot_context);
```

```
    . . .
}
```

Here, we changed the data index for plotting density on level 5 to that which corresponds to some new plot context. On all other levels, the plotted density will correspond to the previously-registered index.