PeleAnalysis Documentation

Release 2018.10

J.B. Bell, M.S. Day, D. Graves, M. Henry de Frahan, R.W. Grout

Jul 13, 2023

Contents:

CHAPTER 1

Data Structures

There are a number of data structure formats used within the `PeleAnalysis` suite of codes. Some are basic AMReX types, such as `MultiFab` or `plotfiles`, while others are specific to the diagnostics and cannot be represented naturally in AMReX's containers that were generally created for block-structured data. Here, we provide an overview of the data structures used by the various tools, both how they are (currently) written to disk and how they are used in the analysis tools.

1.1 Plotfiles

Plotfiles are the standard format for reading data from a Pele simulation. Their format is discussed in the AMReX documentation. For a multi-level AMR calculation, a plotfile contains an ASCII `Header` file and one subfolder for each refinement level. There may also be a folder containing multilevel particle data, and other application-specific files (build information, typical state values, run input parameters, etc). Each refinement level contains one or more `MultiFab` file sets (including an ASCII header file and a number of data files, numbered sequentially). The main `Header` file includes a list of variable names written to the structure, some details about the domain and refinement data and specific run status, and a mapping for which `Multifab` files in the level subfolders contain each of the plotfile variables.

The analysis codes read plotfiles from disk into memory using one of two AMReX-provided C++ classes: `PlotFileData` or `AmrData`. The former is the more recently developed of the two, and has a simple interface for reading and interpolating data from the plotfiles into local temporaries used to build diagnostics. The `AmrData` object is somewhat more limited in terms of interface to interpolations, etc, but has the significant advantage of allowing for demand-driven data reads - that is, only the plotfile metadata is read from disk when the `AmrData` object is instantiated; the data is read only as needed to satisfy a `FillVar` request - and then it is read at the granularity of component and box. A `FlushGrids` operation is available to dynamically manage memory used by the `AmrData` object. This functionality can be critical when processing extremely large datasets on massively parallel machines. In the set of analysis codes in this repository, either may be used depending on the application for which the tool was developed. They can also be rather trivially converted, as needed.

Note that many of the processing tools read plotfiles as input, derive fields that live on the grid structure of the plotfile, and write results out as new plotfiles. There are a number of tools one can use to subset plotfiles in space and component indices and to join together multiple plotfiles (with some limitations on the compatibility of the level and grid structures).

1.2 FArrayBox and MultiFab

All of the usual AMReX data structures are, of course, available in the analysis tools, and are used extensively. These structures are documented in the AMReX documentation. Normally, these structures are used as recommended in the AMReX documentation and application codes. However, as discussed below, there are operations that these structures support which we have "hijacked" for our own purposes. In particular, `FArrayBox` and `MultiFab` can be read and written to disk in a format that is somewhat "self-describing". Floating point data is managed in a portable way and the commands to interact with the IO are particularly convenient/simple. Thus, when we need to read or write data, and it happens to be structured in a way that makes use of these containers, we use them... even if we have to "cheat" to do it!

1.3 MEF - Marc's Element Format

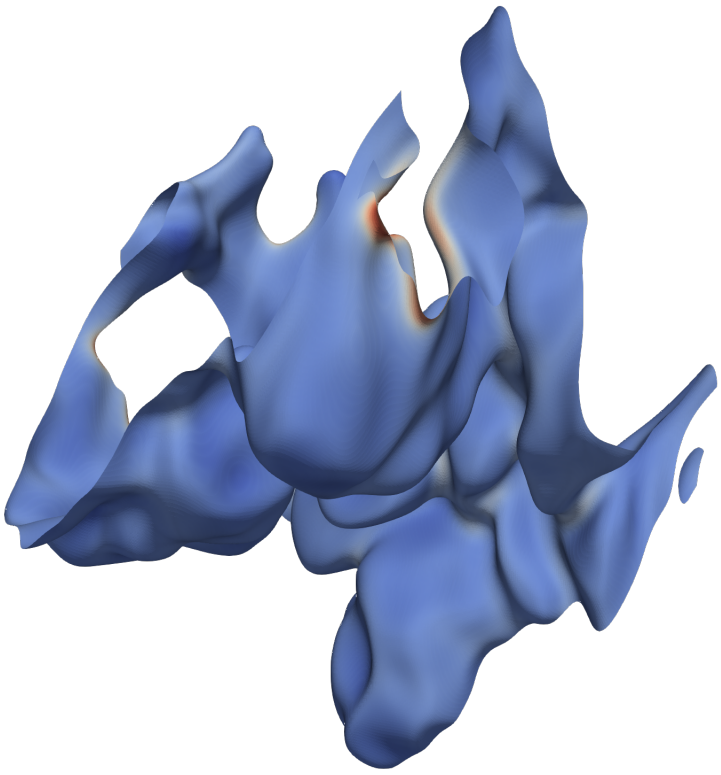
An `MEF` file is an example of (ab)using the AMReX IO functions for the `FArrayBox`. An `MEF` file is the on-disk representation of an unstructured data set, which is inherently not supported in the native AMReX data structures. An unstructured dataset contains a number of named components at a set of implicitly numbered "nodes", and then a set of integer sets that identify an oriented list of nodes to connect for each "element". An `MEF` file makes use of the IO operations of `FArrayBox` in order to collect together into a single file all the data associated with the unstructured data. Examples where an `MEF` structure is used include triangulated surfaces (such as those that result from the isovalue contouring operations in 3D) and polylines (or segment sets that represent contours of 2D data). A limitation of the `MEF` format is that all elements must have the same number of nodes, and all nodes must have the same number of components (in the same ordering). Also, we typically assume the first `AMREX_SPACEDIM` components contain the spatial coordinates. On disk, the MEF file is a concatenated set of information containing: a label, the variable names, the number of nodes per element, the number of elements, followed by a dump of the `FArrayBox` used to store the nodes, followed by an ASCII write of the elements. At the moment, the IO of the unstructured datasets is done explicitly by each tool that needs them (we should probably encapsulate this into a class that manages this part... TODO). Two other notes are that the node numbering in an MEF file starts at 1 (rather than 0) by convention, and the `FArrayBox` used to manage the nodes is built on an AMReX `Box` object that has its first component running from 0 to `Nnodes-1`, and its 1:`AMREX_SPACEIM-1` extents running from 0 to 0. Thus, one does not want to treat this special `Box` or `FArrayBox` in the normal AMReX way (intersections in index space will NOT give the user what is expected!). Visualization of these data structures using standard AMReX tools will also give perhaps unexpected results (there are conversion tools to generate VTK VTP-format files from `MEF` files for reading into Paraview, e.g. See the figure below). The special `Box` and `FArrayBox` objects used with `MEF` files are typically created on-the-fly for the sole purpsoe of IO operations. Outside of IO, the data is typically moved into structures that more clearly indicate usage.

Many of the analysis routines that interact with the unstructured datasets need to work in parallel (with MPI). Typically, when that is done, each rank has a local node numbering. However, the `MEF` format does not have a parallel counterpart, so all IO is typically done via the IO rank, and thus requires an explicit aggregation and node number rationalization step (which, again, is done explicitly in the codes, and probably ought to be encapsulated into a class or something to ease use - TODO).

Several of the diagnostic tools interact with `MEF` structures. They are used to represent isosurfaces of 3D data, polylines that are isolines of 2D plotfile data, and isolines of 3D surface data. Although they are generally used for node-centered data, the structures can be (ab)used to represent data that is element-centered. This is done by duplicating the data at the nodes of each triangle, including its position, but setting the values of the other components for all nodes in the element to the elemental value. This is the strategy used, for example, to represent an element-averaged value of a quantity (or even values of integrated quantities over stream tubes - discussed below). The (brute force) writing of an `MEF` to a `std::ofstream os` looks like this:

```
os << label << std::endl;
os << vars << std::endl;
```

(continues on next page)



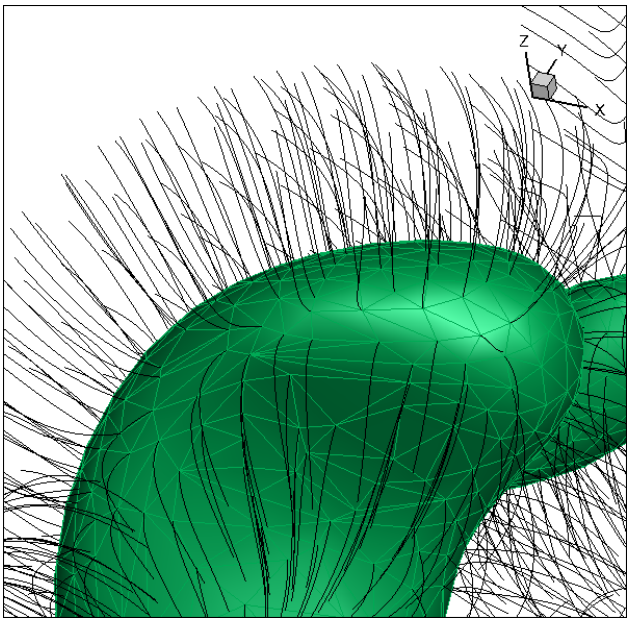
1.1: : An isotherm of a flame-in-a-box case, where the surface is colored by the concentration of a flame intermediate species. This was created as an `MEF` file from the `isosurface.cpp` tool, then converted from `MEF` to `DAT` (Tecplot ASCII data), then from `DAT` to `VTP` using the included python script, *datToVTP.py*, and imported into Paraview).

(continued from previous page)

```
os << nElts << " " << nodesPerElt << std::endl;
nodefab.writeOn(*os);
os.write((char*)eltVec.dataPtr(),sizeof(int)*eltVec.size());
```

1.4 StreamData

StreamData is a class whose design sorta follows the MEF ideas. The format was generated to represent "streamline data" in 2- and 3-space, and is fundamentally unstructured in nature. Imagine you have a cloud of point locations that you would like to use to seed the generation of integral paths along a vector field. For example, imagine the points are the nodes of an isotherm that defines a "flame", and that from each node we construct a path along the integral curves of the temperature gradient, into both the hot and cold sides of the surface. The connectivity of the triangles on the seed surface can be used to define a connectivity of prism-shaped elements that tile a subregion of the domain between the hot and cold ends of the integral curves.



1.2: : An isotherm from a flame calculation, where the triangles defining the surface are visible. The black paths follow integral paths from the surface nodes. Note that the integral curves do not cross. As they are constructed from the 3D plotfile, any quantity defined in the plotfile can be interpolated to these paths. Also, a number of operations can be defined on the curved elements defined by extension of the surface triangles along these paths - e.g., these can define local "flamelets".

The representation of this data in memory is quite complex. Each streamline consists of a set of points, and each point has a location and any number of quantities that have been interpolated from the source plotfile data. Additionally, we want to preserve the connectivity of the surface that implies the connectivity of the curved prism-shaped elements that tile the volume of space surrounding the seed surface.

Because stream data can be quite large, the structures are inherently parallel, and make extensive use the MultiFab and its parallel IO capabilies. Each line contains the original seed point, which falls into the valid ragion of a box from the finest level of the plotfile that contains that point. The Box associated with this region of space and refinement level is deemed the "owner" of the stream line. The data associated with the stream is stored in an FArrayBox associated with the MultiFab at the level of the owning Box. The special Box of the owning FArrayBox is created over the bounds (0:Nlocal,-Npts:Npts,0), where Nlocal is the number of seed points owned by this box, and Npts

is the number of points on the stream line towards each direction from the surface (`j=0` is on the seed surface). The `FArrayBox` created on this `Box` object has `Ncomp` components, including position coordinates and any number of fields interpolated from the source plotfile. The data is distributed with the same distribution map used to distribute the field data when the plotfile is read (determined by the analysis code, NOT the original simulation). Any `Box` in the `BoxArray` at each level in the stream data that do not contain stream lines are set to a default (invalid) size, marking to the analysis code that there are no stream lines there to process. Much like the temporaries used in IO of `MEF` data, the `MultiFab` structures associated with stream data should not be treated like normal AMReX data structures - visualization and manipulation of the data requires detailed knowledge of their layout.

On disk, the `StreamData` object looks much like a plotfile. There is an ASCII `Header` file, and subfolders for each AMR level. Within the subfolders, there are `MultiFab` files associated with the stream line data, possibly written in parallel across multiple data files, etc. Additionally, there is a text file that specifies the connectivity of the elements. Presently, these structures are written, brute force, by the analysis codes (see the function `write_ml_streamline_data` in `stream.cpp`, for example). The functionality has been lifted in to a `StreamData` class, but the analysis tools haven't yet been ported to use these class - TODO.

1.5 N-dimensional bins

Many of the analysis tools generate bins of data. These bins typically are used to create joint probability density functions (jPDFs) in 1, 2 or higher dimensions. They are also used to condition statistics as an intermediate step to generating jPDFs. 2D jPDFs are somewhat special in that we typically assume constant bin widths in each coordinate so that an `FArrayBox` is a natural container to use to hold the result. Also because of the IO capabilities of this class, it is a natural candidate for a format on disk. However, an `FArrayBox` is a simple container, and has no notion of axes labels, variable names, bin sizes, etc. Thus, whenever we are generating this type of data, there is an inherent complexity in how to represent the final output data to enable plotting and interpretation of the results. Note that the analysis tools here DO NOT include plotting routines, so there has to be an understanding about how to communicate all these details to the end user (such as `xmgrace` or `matlab`, etc). To date, we have not come up with a sufficiently flexible, self-describing way to convey all this information, so the tools typically dump everything one needs and the person orchestrating the plots must manually assemble the necessary information.

A particularly noteworthy case is the `binMEF` tool, which bins the data in an arbitrary number of coordinates. For each coordinate, the user determines the min, max and number of bins, and the input data `MEF` file that represents a surface to be chopped up. The code proceeds through each coordinate, and each bit of area landing in a particular bin for coordinate `n`, is then chopped up into bins of coordinate `n+1`. This can be used to generate an area-weighted jPDF in multiple coordinates, but can also be used as a conditioning tool to exclude parts of the surface satisfying certain criteria (falling outside the bins defined for that coordinate). Given the array of bins, the result can be represented as a floating point number (the area) and an array of integers, one for each of the binning coordinates. Just like the simpler 2D jPDF example above, the end user plotting or analyzing the results of this tool must assemble all the bin info in their plotting package of choice. For the N-dimensional case however, it is rarely useful to store the data as a dense N-space container. The results are written to the screen in their naturally sparse format. We haven't yet developed a standardized way to communicate all these details, so the process can be tedious, but it is unavoidable.

CHAPTER 2

buildPMF - Create Fortran 1D interpolator

Given a text file consisting of an array of states over a 1D set of points, create a fortran function that interpolates the states by computing the average of each state between two locations. Typically, this is used to create a function for interpolating a 1D premixed flame solution from PREMIX or Cantera that can be linked with another code in 1-3 spatial dimensions for initializing data.

` Usage: `

Example:

CHAPTER 3

conditionalMean - Create data for conditional mean plots

Given a plotfile, a conditioning variable, and a list of variables, create data files, each containing conditional means and standard deviations in each of a set of bins of the conditioning variable. Use *plotJpdf.m* from the *Scripts* folder to plot the resulting data in matlab/octave, using a line for the mean, and a gray shaded region for the values within a standard-deviation of the mean.

```
Usage:
```

Example:

CHAPTER 4

averagePlotfile - Create a pltfile by averaging existing pltfiles

Two tools exist for the purpose of averaging plotfiles. *averagePlotfile* assumes all plotfiles to be averaged have the same underlying BoxArrays, or the output is only given at the base level. *averagePlotfileFlexible* relaxes this assumption: the output file will be refined anywhere *any* of the input files are refined (coarse data is interpolated to the finer levels in each file as needed before averaging to obtain this result). Both tools require that all files have the same domain and base grid, and by default the same variables. *averagePlotfileFlexible* adds the capability to optionally select a specifc list of variables, in which case that list must be present in all input files but otherwise the input files may contain different sets of variables.

‘‘‘ Usage:

```
./avgPlotfilesFlexible2d.gnu.MPI.ex infiles=$(ls -d plt*) [options]
```

‘‘‘

Example:

CHAPTER 5

combinePlts - Combine plotfiles

Create a new plotfile that is composed of a set of components taken from each of two existing plotfiles. The input plotfiles *must* have the same AMR hierarchy, up to the finest level requested for the output.

` Usage: `

Example:

CHAPTER 6

curvature - Curvature of plotfile scalar

Given a plotfile that contains a scalar quantity, compute the local value of the mean and Gaussian curvature of isopleths of that scalar at all cells in the solution. Either create a new plotfile containing only these computed quantities, or append the fields to the existing plotfile.

` Usage: `

Example:

CHAPTER 7

grad - Gradient of plotfile scalar

Given a plotfile that contains a scalar quantity, compute the components of the gradient, and its magnitude, of that scalar at all cells in the solution. Either create a new plotfile containing only these computed quantities, or append the fields to the existing plotfile.

` Usage: `

Example:

CHAPTER 8

isosurface - Isosurface of plotfile scalar

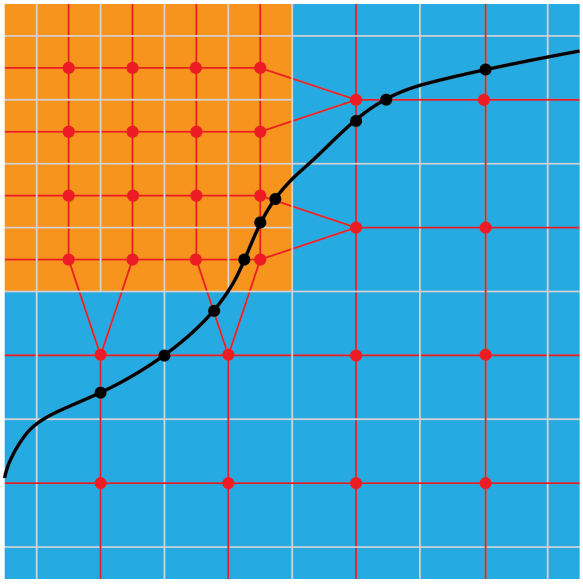
Create isosurfaces (contours) from a component in a plotfile, given the name of the variable and a value. The result is written as an MEF (Marc's element file). In 2D, the contour segments are connected up into a minimal set of polylines before writing. Optionally, map other variables to surface, and optionally compute signed distance from this surface onto the cell-centered mesh of the plotfile.

```
Usage:
  ./isosurface2d.gnu.MPI.ex inputs infile=<s> isoCompName=<s> isoVal=<v> [options]
      Options:
           infile=<s> where <s> is a pltfile
           isoCompName=<s> where <s> is the quantity being contoured
           isoVal=<v> where <v> is an isopleth value
           Choosing quantities to interp to surface:
             comps=int comp list [overrides sComp/nComp]
             sComp=start comp[DEF->0]
             nComp=number of comps[DEF->all]
           finestLevel=<n> finest level to use in pltfile[DEF->all]
           writeSurf=<1,0> output surface in binary MEF format [DEF->1]
           outfile=<s> name of tecplot output file [DEF->gen'd]
           build_distance_function=<t,f> create cc signed distance function [DEF->f]
           rm_external_elements=<t,f> remove elts beyond what is needed for␣
↪watertight surface [DEF->t]
```

8.1 Details

Computing the isosurface requires building a "flattened" representation of the AMR data structure that ignores covered data, and properly connects cells across coarse-fine interfaces. Once we have a watertight representation of the surface, we make use of the marching squares (in 2D) or marching cubes (3D) strategy to build a watertight contour based on intersections of the surface with the dual grid (see Figure 8.1).

The isosurface code contains the following steps:



8.1: In 2D, the isocontour (in black) is computed from the dual grid created by connecting cell centers. Even with multiple AMR levels the dual grid is composed on quads (hexes in 3D), but at the coarse-fine interfaces, some of the nodes are degenerate. Intersections between the isocontour and the quad element boundaries are computed by linear interpolation between the scalar values of the nodes at either end of the segment. The contour segments (or triangles) are obtained by connecting these intersections across the elements.

1. Use AMReX-provided class `PlotFileData` to load *isoCompName* field data, and `FillPatchTwoLevels` to fill grow cells, including those across periodic boundaries.
2. The location of each valid cell center is computed using Δx at that level. The grow cells are set to have all their locations at the center of the coarse cell outside, and the grow data is filled with piecewise constant interpolation. The effect is that the coarse and fine levels are joined in a water-tight way using degenerate quads/hexes, that actually look in 2D either like a triangle (a degenerate quad) or a trapezoid (depending on whether the two grow cells are in the same coarse cell or adjacent ones (see Figure 8.1).
3. Intersections of the isosurface with these element constructs is computed by searching each pair of adjacent nodes around the outside of the polygon for a sign change of (value - isoVal). When such a condition is found, the location of the intersection is computed by linear interpolation, and the desired additional variables are mapped to the same point. The result is stored as a pair<Edge,Point>, where Edge is a pair of IntVects defining the endpoints of the segement that is intersected and their associated Amr level ID, and the Point contains the coordinates first, then any other mapped/interpolated variables at that node. The intersection structs are stored in a map of these pairs, allowing exact comparisons for unique nodes with no floating point comparison issues.
4. Elements on the surface are described as an ordered vector of these iterators into a master map for this rank.
5. Portions of the surface intersecting each valid grid are computed in parallel, and collated onto a single MPI rank. Multiply defined nodes (those with the same pair of IntVect/AmrID) are eliminated and a global numbering of the unique set is created; the element list is merged and the numbering is reconciled.
6. The map representation of the surface is converted to a simple surface representation and written to disk by a single processor.

8.2 Signed distance function

Optionally, the code can also return a field containing the signed distance to the isosurface from each valid point in the plotfile structure. The shape of this data will correspond to that of the original plotfile. The signed distance is computed, up to a maximum, on all points in the domain.

CHAPTER 9

jpdf - Create data for joint PDF files

Given a plotfile and a subset of the variable names that it contains, bin data for every unique pair of variables, making n*(n-1)/2 sets. The results are written to separate files inside the plotfile folder and can be written as tecplot, matlab, gnuplot, fab, or plotfiles.

```
Usage:
```

Example:

CHAPTER 10

stream - Streamlines of plotfile vector

Given a plotfile containing a vector field and an MEF file containing a collection of "seed" points, create "streamlines" eminating from the seed points that are locally parallel to the vector field. The resulting streamlines will be of a fixed length going both directions along the vector field from the seed point. Results will be written in a custom plotfile-like data folder, which is discussed in the data section. These files cannot be directly visualized with standard plotfile tools.

```
Usage:
   ./stream2d.gnu.MPI.ex plotfile=<string> [options]
     Options:
          isoFile=<string>  OR  seedLoc=<real real [real]
          streamFile=<string>  OR  outFile=<string>
          is_per=<int int int> (DEF=1 1 1)
          finestLevel=<int> (DEF=finest level in plotfile)
          progressName=<string> (DEF=temp)
          traceAlongV=<bool> (DEF=0)
          buildAltSurf=<bool> (DEF=0)
              (if true, requires altVal=<real>, also takes dt=<real> (DEF=0) and␣
↪altIsoFile=<string>)
          nRKsteps=<int> (DEF=51)
          hRK=<real> (DEF=.1 (*dx_finest in plotfile)
          nGrow=<int> (DEF=4)
          bounds=<float * 4> (DEF=NULL)
```

10.1 Options

10.1.1 Seed points

Streamlines are computed to emanate from seed points specified by the user. The seed points can be defined as the nodes of a triangulated surface (in 3D) or a "polyline" (2D), or can be specified directly in the ParmParsed input.

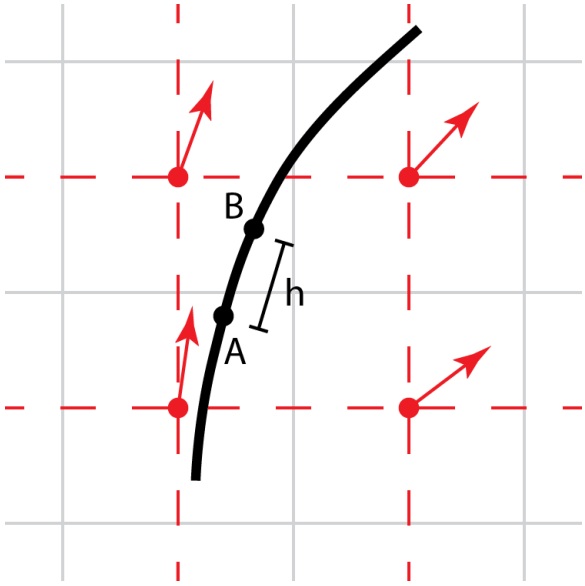
If a triangulated surface or polyline is used (by passing the name of the MEF file via the *isoFile* keyword), the resulting streamlines retain the connectivity inferred by the input structure. And since the steamlines will not, in general, cross they will bound a triangular-prism shaped volume extending a distance from the surface on either side. The union

of these volumes tile a layer around the original triagulated surface. In 2D, the streamlines will bound a polygonal structure that similarly tiles the region around the original polyline.

If the seed points are specified directly in the input, no connectivity information is inferred. Currently, the only option for this mode is accessible via the *seedLoc* keyword, which specifies the coordinates of a single seed point.

10.1.2 Integration options

Figure 10.1 illustrates a streamline (in black) that is computed from a vector field, whose components are specified on cell centers. The paths are integrated in both directions from the seed point, as depicted in 10.1. The user specifies the interval, h, as a fraction, *hRK*, of the grid spacing at the finest level of vector field used, as well as the total number of such intervals, *nRK*.



10.1: Streamlines (in black) are computed by integrating the vector field from a seed point in intervals of h, e.g., from point A to B, using the RK4 scheme. The vector field components are defined at the nodes of the dual grid connecting the cell centers and are linearly interpolated.

10.1.3 Vector field

Currently, the vector field can be constructed to align with the gradient of a scalar field, or with the flow velocity (if the option *traceAlongV* = *t*). If the the velocity field is not used, the required components of the gradient vector field (identified via the keyword, *progressName*) are computed on the fly with second-order centered differences.

10.1.4 Alt surface

If the keyword *buildAltSurf* = *t*, a new triangulated surface is constructed after the streamlines are generated. This surface will be created where the scalar identified as *progressName* takes the value specified by the keyword, *altVal* along the streamlines. The connectivity of this surface will be identical to the connectivity of the original surface (specified with the keyword, *isoFile*). The new surface is written to the file indicated by the keyword, *altIsoFile*.

10.1.5 Algorithm details

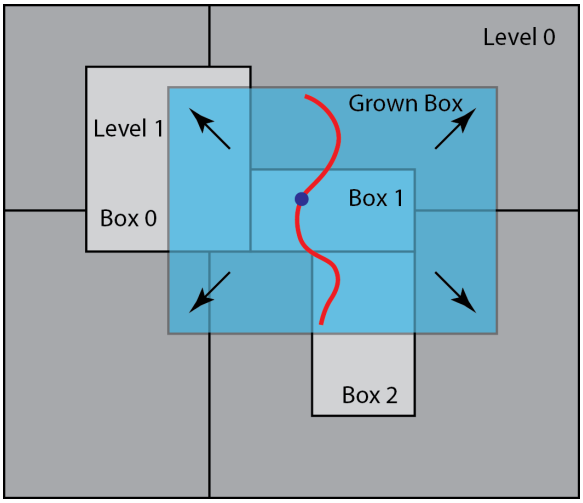
The algorithm starts by determining the finest AMR level box in the plotfile (indicated by the keyword, *plotfile*) that contains the physical location of each seed point (up to and including the level indicated by the keyword, *finestLevel*). Then, as the required plotfile data is read (in parallel), a distribution map will be created for each level, and we use this to assign the processor that will be responsible for computing the streamline associated with that point.

The RK4 scheme is used to integrate the vector field, u, along streamline for a distance h from A to B (see Figure 10.1):

$$
\begin{aligned}
x_B &= x_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\
k_1 &= h\,u(x_A), \quad x_1 = x_A + 0.5k_1 \\
k_2 &= h\,u(x_1), \quad x_2 = x_A + 0.5k_2 \\
k_3 &= h\,u(x_2), \quad x_3 = x_A + k_3 \\
k_4 &= h\,u(x_3)
\end{aligned}
$$

The vector field u is defined at cell-centers and we need to construct a function that, given the vector field data at nodes, is able to linearly interpolate these components as needed to evaluate the above expressions. A simple way to orchestrate this interpolater is to base it on source data that lives on a logically rectangular, uniformly space grid, as this allows simple/fast "mod" operations to locate the specific source data indices for the interpolation.

However, if the seed point starts off, for example, near the boundary of the owning box, it is possible that the integration will eventually step off the grid, and possibly across AMR levels, before reaching the required path length, and thus attempt to access data that is unavailable to this processor. A simple solution follows the usual AMReX approach in these situations - grow cells. Given the *hRK* and *nRK* parameters, we can compute the size of a grow region buffer that is guaranteed to fully contain the path - even if it is rather large - see Figure 10.2. And given the standard AMReX fill-patching infrastructure, we can fill the required data locally from the plotfile classes, being careful to account for periodic and physical domain boundaries.



10.2: A streamline (red) is generated from the seed point (blue), which is owned by Box 1 in the finest level here, Level 1. The streamline goes beyond the valid region of Box 1. Data to fill the grown box is copied from neighboring grids at the same refinement level, and interpolated from coarse levels where needed.

Note that because the size of the grow region needed depends on the maximum length of the streamlines, these patches can be quite large, particularly in 3D. However, this approach is far simpler than any method that might move between levels and/or processors whenever boundaries are crossed. In order to manage very large datasets, this tool has been

written to run in parallel with MPI. For maximum flexibility, there is also a separate tool that can read the streamline generated with the above strategy, and interpolate a set of fields onto the streamlines.

CHAPTER 11

sampleStreamlines - Sample a plotfile to streamlines

Given a set of streamlines, and a plotfile, sample the specified field quantities to the streamlines, and write out the new lines with all quantities. This routine is specifically set up to work with limited memory. For example, the list of components to be interpolated can be broken into groups to minimize the memory required.

` Usage: `

Example:

CHAPTER 12

streamSub - Subset streamlines

Extract a subset of streamlines from a streamline file, based on one of a set of user-selectable criteria (random sampling, physical location of the streamline seed point, etc). This process will discard the connectivity info of the streamlines. Typically, this tool is used to extract a manageable number of lines for quick plotting in order to get a feel, or gather statistics, for the data contained in the full set.

```
` Usage:  `
```

Example:

CHAPTER 13

streamTubeStats - Stream tube statistics

Given a set of streamlines that represents the bounds of triangular-wedge shaped volumes, gather statistics of the volumes. Output the results as a triangulated surface MEF file, where for each triangle, the values at all three nodes are equal and represent the quanity computed for the entire wedge volume. Thus, the MEF format is overloaded here so that each node is multiply-defined, based on the number of triangles is it part of. The resulting MEF file will also contain the area of the triangle from the original tesselation that created the seed points for the streamline, with the intention that the results here can be used to construct statistics weighted by this area. Typically, this processing is used to gather statistics associated with a isosurface (such as one that represents a flame).

```
` Usage: `
```

Example:

CHAPTER 14

subPlt - Subset plotfile

Create a new plotfile from an original one by subsetting in space and/or component. The spatial subsetting is specified by giving a bounding box in integer coordinates at the coarsest AMR level in the file (the default is the entire domain). The component subset is specifeid as an integer list of components (the default is all components).

```
` Usage: `
```

Example:

CHAPTER 15

surfMEF Tools - Manipulate MEF files

The MEF (Marc's element format) file format is hacked up data structure-on-disk, primarily intended to store a triangulated surface. The data at the nodes (including the position, but also potentially other field quantities) is written using AMReX's FabIO functions so that the floating point data is portable. The triangles are specified as a set of integer triples, where each integer is the (zero-based) id of the node. The first four lines in an MEF file are ASCII, adn include a label, the list of variables, the integer number of elements, and then the FAB header info. The binary FAB info is then concatenated, and is then followed by the element triples, one element per one, written in ASCII.

The surfMEF conversion tools are used to transform to and from the MEF format, and to do simple arithmetic operations on the data.

- combineMEF: Combine the components of two MEF files, assuming they have the same node positions and connectivity
- mergeMEF: Merge the triangles of two different MEF files
- multMEF: Multiply specific components of MEF files together
- scaleMEF: Scale specific components of the MEF by constants
- sliceMEF: Compute a contour on an MEF surface
- smoothMEF: Smooth an MEF surface
- surfDATtoMEF: Convert a Tecplot-formatted ASCII triangulated surface file into an MEF file
- surfMEFtoDAT: Convert an MEF-file into a Tecplot-formatted ASCII triangulated surface file
- trimMEFgen: Do an area-weighted binning of an MEF surface file, by assuming linear variatoin of the field on each triagle and slicing the triangles into bits at the bin boundaries

```
` Usage: `
```

Example:

CHAPTER 16

README.rst:

CHAPTER 17

PeleAnalysis

This repository contains a collection of standalone routines for processing plotfiles created with the AMReX software framework for block-structured adaptive mesh refinement simulations. Documentation is under development, but is available at

```
https://peleanalysis.readthedocs.io/en/latest/
```

AMReX is required for these tools, and is available https://github.com/AMReX-Codes/amrex

In order to build these processing tools, you should clone or fork the amrex repository, and set the environment variable `AMREX_HOME` to point to the local folder where that is placed. Then clone this repository, `cd Src` and edit the `GNUmakefile` to select which tool to build. If AMReX is configured properly, a stand-alone executable will be built locally, based on the selected options, including spatial dimension (2 or 3), compiler choices, whether to build with MPI and/or OpenMP enabled, and whether to build a debugging or optimized version. Note that some of the tools require building a companion f90 source file - you must manually set the flag in the `GNUmakefile` accordingly. More extensive documentation is available (see building instructions below).

To add a new feature to PeleAnalysis, the procedure is:

1. Create a branch for the new feature (locally)

```
git checkout -b AmazingNewFeature
```

2. Develop the feature, merging changes often from the *master* branch into your AmazingNewFeature branch

```
git commit -m "Developed AmazingNewFeature"
git checkout master
git pull                      [fix any identified conflicts between local and␣
↪remote branches of "master"]
git checkout AmazingNewFeature
git merge master         [fix any identified conflicts between "master" and
↪"AmazingNewFeature"]
```

3. Push feature branch to PeleAnalysis repository (if you have write access, otherwise fork the repo and push the new branch to your fork):

```
git push -u origin AmazingNewFeature [Note: -u option required only for the first␣
↪push of new branch]
```

4. Submit a merge request through the github project page - be sure you are requesting to merge your branch to the master branch.

Documentation for the analysis routines exists in the Docs directory. To build the documentation:

```
cd Docs
make html
```

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration – responsible for the planning and preparation of a capable exascale ecosystem – including software, applications, hardware, advanced system engineering, and early testbed platforms – to support the nation's exascale computing imperative.

CHAPTER 18

Indices and tables

- genindex
- search