EBChombo Software Package for Cartesian Grid, Embedded Boundary Applications

P. Colella
D. T. Graves
T. J. Ligocki
G. Miller
D. Modiano
P. O. Schwartz
B. Van Straalen
J. Pilliod
D. Trebotich
M. Barad
B. Keen
A. Nonaka
C. Shen

Applied Numerical Algorithms Group
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA

Feb. 2, 2000

Disclaimer

Contents

Chapter 1

Layer 1–EBTools: Single Level Tools

1.1 Introduction

This document is to describe the EBTools component of the EBChombo distribution. This infrastructure is based upon the Chombo infrastructure developed by the Applied Numerical Algorithms Group at Lawrence Berkeley National Laboratory [4]. EBTools is meant to be an infrastructure for Cartesian grid embedded boundary algorithms. The goal of this software support is to provide a relatively compact set of abstractions in which the Cartesian grid embedded boundary algorithms we are developing can be expressed and implemented. The particular design we are proposing here is motivated by the following observations. First, the dependent variables in a finite difference method are represented as arrays defined on subsets of an index space. Second, the transformations on arrays can be expressed as combinations of pointwise operations on the arrays, and of sums over nearby points of arrays, *i.e.*, stencil operations. For standard finite difference methods on rectangular grids, the index space is the d-dimensional rectangular lattice of d-tuples of integers, where d is the spatial dimension of the problem. For multigrid or AMR methods, the index space is the hierarchy of d-dimensional rectangular lattices, where the successive members of the hierarchy are related to one another by coarsening and refinement operations. In both of these cases, the stencil operations can be expressed formally as a loop over stencil locations. In the AMR case, both the stencil locations and the locations where the stencil operations are applied are computed using a set calculus on the index space. If one fully exploits this picture to derive a set of abstractions for expressing these algorithms, it leads to a very concise implementation of the algorithms in these two domains.

The above characterization of finite difference methods holds for the EB algorithms as well, with the critical difference that the index space is no longer a rectangular lattice, but a more complicated object. In the case of a non-hierarchical grid representation, the index space is a combination of a rectangular lattice (the Cartesian grid part) and a graph representing the irregular cell fragments that abut the irregular boundary. For a hierarchical method, we have one such index space for each level of refinement, related to one another by coarsening and refinement operations. In addition, we want to support the overall implementation strategy that the bulk of the calculations (corresponding to data defined on the rectangular lattice) are performed using rectangular array representations, thus restricting the irregular array accesses and computations to a set of codimension one. Finally, we wish to appropriately integrate the AMR implementation strategies for block-structured refinement with the EB algorithms.

1.2 Overview of Embedded Boundary Description

Cartesian grids with embedded boundaries are useful to describe volume-of-fluid representations of irregular boundaries. In this description, geometry is represented by volumes (Λh^d) and apertures ($\vec{A}^\alpha h^{d-1}$). See figure 1.1 for an illustration. In the figure, the grey area represents the region excluded from the solution domain and the arrows represent fluxes. A conservative, "finite volume" discretization of a flux divergence $\nabla \cdot \vec{F}$ is of the form:

$$\nabla \cdot \vec{F} \approx \frac{1}{\Lambda h} \sum \vec{F}^\alpha \cdot \vec{A}^\alpha \tag{1.1}$$

This is useful for many important partial differential equations. Consider Poisson's equation with Neumann boundary conditions

$$\nabla \cdot \vec{F} = \Delta \phi = \rho \text{ on } \Omega \ . \tag{1.2}$$

$$\frac{\partial \phi}{\partial n} = 0 \text{ on } \partial\Omega \tag{1.3}$$

The volume-of fluid description reduces the problem to finding sufficiently accurate gradients at the apertures. See Johansen and Colella [9] for a complete description of solving Poisson's equation on embedded boundaries. Hyperbolic conservation laws can be solved using similar divergence examples. See Modiano and Colella [12] for such an algorithm. Gueyffier, et al. [7] use a similar approach for their volume-of-fluid application. The only geometric information required for the algorithms described above are:

- Volume fractions
- Area fractions
- Centers of volume, area.

The problem with this description of the geometry is it can create multiply-valued cells and non-rectangular connectivity. Refer to figure 1.2. The interior of the cartoon airfoil represents the area excluded from the domain and the lines connecting the cell centers represent the connectivity of the discrete domain This very simple geometry produces a graph which is more complex than a rectangular lattice simply because the grid which surrounds it cannot resolve the geometry. The lack of resolution is fundamental to many geometries of interest (trailing edges of airfoils, infinitely thin shells). Algorithms which

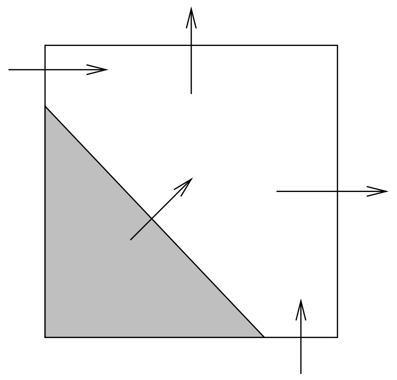

Figure 1.1: Embedded boundary cell. The grey area represents the region excluded from the solution domain and the arrows represent fluxes.

require grid coarsening (multigrid, for example) also produce grids where such complex connectivity occurs. The connectivity can become arbitrarily complex (see figure 1.3) in general, so the software infrastructure must support abstractions which can express this complexity.

Our solution to this abstraction problem is to define the embedded boundary grid as a graph. The irregular part of the index space can be represented by a graph $G = \{N, E\}$, where N is the set of all nodes in the graph, and E the set of all edges of the graph connecting various pairs of nodes. Geometrically, the nodes correspond to irregular control volumes (cell fragments) cut out by the intersection of the body with the rectangular mesh, and the edges correspond to the parts of cell faces that abut a pair of irregular cell fragments. The remaining parts of space are indexed using elements of Z^d, or are covered by the body, and not indexed into at all. However, it is possible to think of the entire index space (both the regular and irregular parts) as a graph: in the regular part of the index space, the nodes are just elements of Z^d, and the edges are the cell faces that separate pair of successive cells along the coordinate directions. If we used this representation for the entire calculation, the method would correspond to a unstructured grid method. We will use this specification of the entire index space as a convenient uniform interface to both the structured and unstructured parts of the index space.

We discretize a complex problem domain as a background Cartesian grid with an embedded boundary representing the irregular domain region. See figure 1.4. We recognize three types of grid cells or faces: a cell or `Face` that the embedded boundary intersects is *irregular*. A cell or `Face` in the irregular problem domain which the boundary does not intersect is *regular*. A cell or face outside the problem domain is *covered*. The boundary of a cell is considered to be part of the cell, so that cells A, B and C in figure 1.5 are irregular.

An irregular cell is formed from the intersection of a grid cell and the irregular problem domain. We represent the segment of the embedded boundary as a single flat segment. Quantities located at the irregular boundary are given the superscript B. Depending on

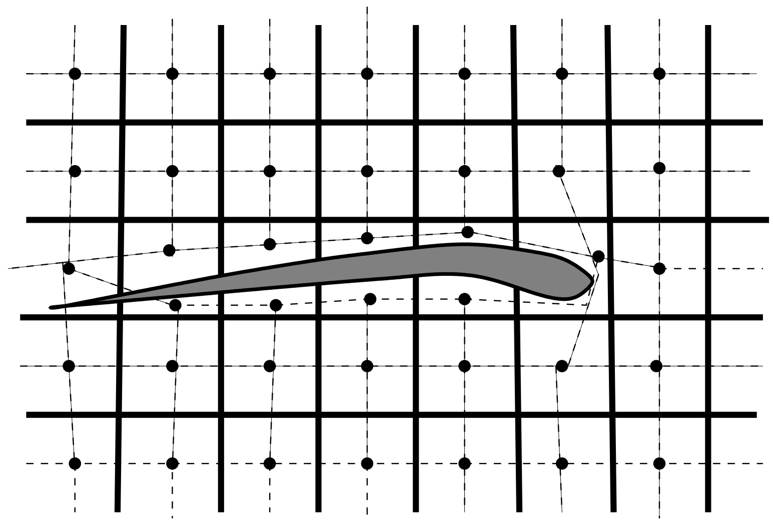

Figure 1.2: Example of embedded boundary description of an interface. The interior of the cartoon airfoil represents the area excluded from the domain and the lines connecting the cell centers represent the graph connectivity of the discrete domain.

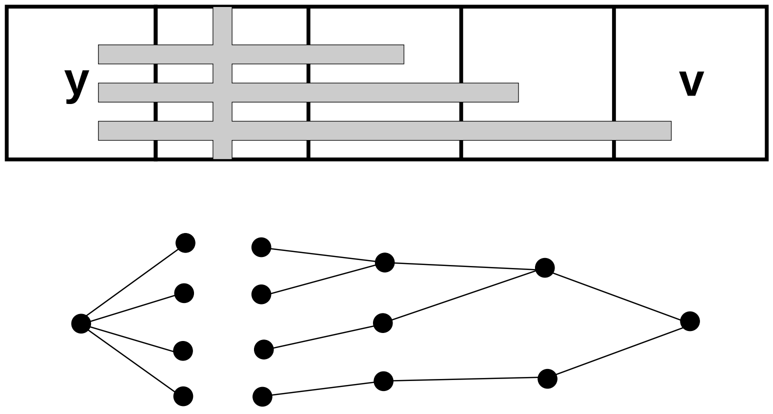

Figure 1.3: Example of embedded boundary description of an interface and its corresponding graph description. The graph can be almost arbitrarily complex when the grid is underresolved.

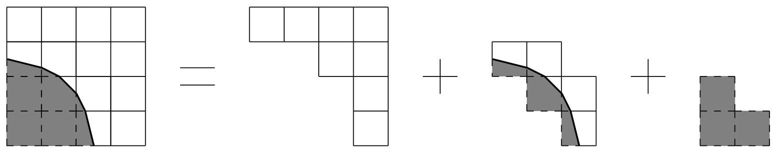

Figure 1.4: Decomposition of the grid into regular, irregular, and covered cells. The gray regions are outside the solution domain.

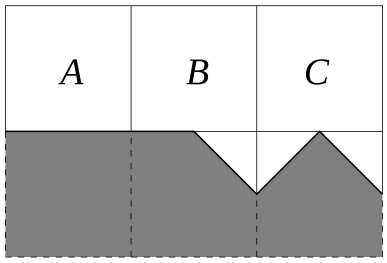

Figure 1.5: Cells with unit volume that are irregular.

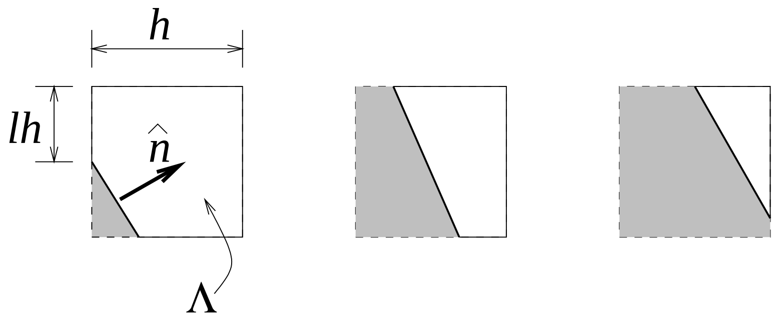

Figure 1.6: Representable irregular cell geometry. The gray regions are outside the solution domain.

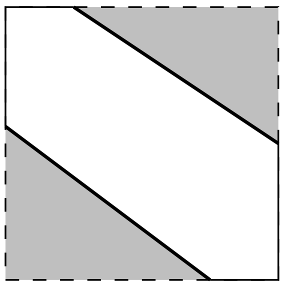

Figure 1.7: Unrepresentable irregular cell geometry. The gray regions are outside the solution domain.

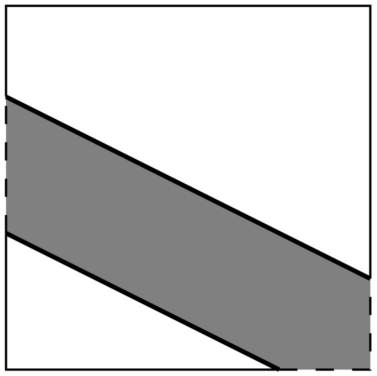

Figure 1.8: Multiple irregular VoFs sharing a grid cell. The left face of the grid cell is also multi-valued. The gray region is outside the solution domain.

which grid faces the embedded boundary intersects, the irregular cell can be a pentagon, a trapezoid, or a triangle, as shown in figure 1.6. A cell has a volume Λh^2, where Λ is its volume fraction. A face has an area ℓh, where ℓ is its area fraction. The polygonal representation is reconstructed from the volume and area fractions under the assumption that the cell has one of the shapes above. Since the boundary segment is reconstructed solely from data local to the cell, it will typically not be continuous with the boundary segment in neighboring cells. We also derive the normal to the embedded boundary face $\hat{n}$ and the area of that face $\ell^B h$.

We do not represent irregular cells such as shown in figure 1.7, in which the embedded boundary has two disjoint segments in the cell. If such a cell is present, it will be reconstructed incorrectly. The mathematical formulation and its implementation allow multiple irregular cells in one grid cell, such as seen in figure 1.8.

1.3 Derived Quantities

We derive all our discrete geometric information from only the volume fraction and area fraction data. To do this we often use a discrete form of the divergence theorem. Analytically, given a vector field $\vec{B}$ on a finite domain Ω (with some constraints on both):

$$\int_\Omega \nabla \cdot \vec{B} dV = \int_{\partial\Omega} \vec{B} \cdot \hat{n} dA \tag{1.4}$$

where $\hat{n}$ is the unit normal vector to the boundary of the domain. We discretize this equation so that a given volume of fluid (VoF) is the domain Ω. Given V_v is the volume of a VoF and A_f is the area of a face f, we discretize equation 1.4 as follows:

$$V_v \nabla \cdot \vec{B} = \sum_f \vec{B} \cdot \hat{n} A_f. \tag{1.5}$$

By cleverly picking $\vec{B}$, we can derive many of the geometric quantities that we need. Telescoping sums force the discrete constraint to be enforced over the entire computational domain.

1.3.1 Interface Normal and Area

Suppose $\vec{B} = \hat{e}_x$, the unit vector in the x direction. Equation 1.4 becomes

$$\int_{\partial\Omega} n_x = 0, \tag{1.6}$$

where n_{x0} is the component of the normal to $\partial\Omega$ in the $x0$ direction. Define $\hat{n}^I$ to be the normal to the embedded face and A_I to be the area of the irregular face. Equation 1.5 becomes

$$A_{x0,h} - A_{x0,l} = n^I_{x0} A_I \tag{1.7}$$

where $A_{x0,h,l}$ are the areas on the high and low side of the VoF in the $x0$ direction. Because $\hat{n}^I$ is a unit vector, $|\hat{n}^I| = 1$ and the area of the irregular boundary is given by

$$A_I = (\sum_{i=0}^{D-1} (A_{xi,h} - A_{xi,l})^2)^{\frac{1}{2}} \tag{1.8}$$

and the normal to the face in the $x0$ direction is given by

$$n^I_{x0} = \frac{A_{x0,h} - A_{x0,l}}{A_I}. \tag{1.9}$$

For VoFs with multiple faces in a particular direction, we use the sum of the face areas in equations 1.8 and 1.9.

1.4 Overview of API Design

The pieces of the graph of the discrete space are represented by the classes `VolIndex` and `FaceIndex`. `VolIndex` is an abstract index into cell-centered locations corresponding to the nodes of the graph (VoFs). `FaceIndex` is an abstract index into edge-centered locations (locations between VoFs). The class `EBIndexSpace` is a container for geometric information at all levels of refinement. The class `EBISLevel` contains the geometric information for a given level of refinement. `EBISLevel` is not part of the public API and is considered internal to `EBIndexSpace`. `EBISBox` represents the intersection between an `EBISLevel` and a `Box` and is used for aggregate access of geometric information. `EBISLayout` is a set of `EBISBox`es corresponding to the boxes in a `DisjointBoxLayout` grown by a specified number of ghost cells.

Concept	Chombo	EBChombo
Z^D	—-	EBIndexSpace
point in Z^D	IntVect	VoF
region over Z^D	Box	EBISBox
Union of Rectangles in Z^D	BoxLayout	EBISLayout
data over region Z^D	BaseFab	BaseEBCellFAB, BaseEBFaceFAB
iterator over points	BoxIterator	VoFIterator, FaceIterator

Table 1.1: The concepts represented in Chombo and EBChombo.

1.5 Data Structures for Graph Representation

The class `VolIndex` is an abstract index into cell-centered locations corresponding to the nodes of the graph. The class `FaceIndex` is an abstract index into edge-centered locations (locations between VoFs). It is characterized by the pair of `VolIndexes` that are connected by the `FaceIndex`. The possible range of values that can be taken on by a `VolIndex` or a `FaceIndex` is determined by the index space containing the `VolIndex`. `FaceIndexes` always live at cell faces (there can be no `FaceIndex` interior to a cell). The entire graph is represented in the class `EBIndexSpace`. It stores all the connectivity of the graph and other geometric information (volume fractions, area fractions, etc). `EBISBox` represents a subset of the `EBIndexSpace` at a particular refinement and over a particular box in the space. `EBISLayout` is a collection of `EBISBoxes` distributed over processors associated with an input `DisjointBoxLayout`.

1.5.1 Class `EBIndexSpace`

The entire graph description of the geometry is represented in the class `EBIndexSpace`. It stores all the connectivity of the graph and other geometric information (volume fractions, area fractions, etc). The important member functions of `EBIndexSpace` are as follows.

-
  ```
  void define(const Box& BoundingBox,
              const RealVect& origin,
              const Real& dx,
              const GeometryService& geometry,
              int ncellmax, int nmaxcoarsen);
  ```

  Define data sizes. `BoundingBox` is the `Box` which defines the domain of the EBIndexSpace at its finest resolution. The arguments `origin` and `dx` specify the location of the lower-left corner of the domain and the grid spacing in each coordinate direction at the finest resolution. The `geometry` argument is the service class which tells the `EBIndexSpace` how to build itself. See section 1.5.2 for a description of the `GeometryService` interface class. Coarser resolutions of the EBIndexSpace are also generated in the initialization process. ncellmax is the maximum box size in the database. A smaller box size means that the the EBIndexSpace will be able to distribute its data among more processors. nmaxcoarsen says now many coarsenings will be represented. If either of these integers is -1, a reasonable default box size and full coarsening are set.

-
  ```
  void fillEBISLayout(EBISLayout& ebisLayout,
                      const DisjointBoxLayout& dbl,
                      const Box& domain,
                      const int& nGhost);
  ```

  Define an EBISLayout for each box in the input layout grown by the input ghost cells. The input domain defines the refinement level at which the layout exists. The argument dbl is the layout over which the data is distributed. If every box does not lie within the input domain, a runtime error occurs. The domain argument is the problem domain at the refinement of the layout. If the refinement does not exist within the EBIndexSpace, a runtime error occurs. The nghost argument defines the number of ghost cells in each coordinate direction.

- `int numLevels() const;`

  Return the number of levels of refinement represented in the EBIndexSpace

- `int getLevel(const Box& a_domain) const;`

  Return level index of domain. Return -1 if a_domain does not correspond to any refinement of the EBIndexSpace.

EBIndexSpace can only be accessed through the the Chombo_EBIS singleton class. The usage pattern follows this model. At some point, one defines the GeometryService object one wants to use (in the example we use a SlabService) and defines the singleton as follows:

```
  SlabService slab(coveredBox);
  EBIndexSpace* ebisPtr = Chombo_EBIS::instance();
  ebisPtr->define(domain, origin, dx, slab);
```

Whenever one needs to define an EBISLayout, the usage is as follows:

```
void makeEBISL(EBISLayout& a_ebisl,
               const DisjointBoxLayout& a_grids,
               const Box& a_domain,
               const int& a_nghost)
{
  const EBIndexSpace* const ebisPtr = Chombo_EBIS::instance();
  assert(ebisPtr->isDefined());
  ebisPtr->fillEBISLayout(a_ebisl, a_grids, a_domain, a_nghost);
}
```

1.5.2 Class GeometryService

The GeometryService class defines an interface that EBIndexSpace uses for geometry generation. EBIndexSpace builds an adaptive hierarchy of its geometry information. It queries the input GeometryService with a two pass algorithm. First EBIndexSpace resolves which regions of the space are wholly regular, which are wholly covered, and which contain irregular cells. Then EBIndexSpace loops through the regions which contain irregular cells and sends these regions (in the EBISBox form to the GeometryService to be filled. The interface of GeometryService is

-
  ```
  virtual bool isRegular(const Box& region, const Box& domain,
                         const RealVect& origin, const Real& dx)=0;
  virtual bool isCovered(const Box& region, const Box& domain,
                         const RealVect& origin, const Real& dx)=0;
  ```

  Return true if *every* cell in the input region is regular or covered. Argument region is the subset of the domain. The domain argument specifies the span of the solution index space. The origin argument specifies the location of the lower-left corner (the zero node) of the solution domain and the dx argument specifies the grid spacing.

-
  ```
  virtual void fillEBISBox(EBISBox& ebisRegion,
                           const Box& region,
                           const Box& domain,
                           const RealVect& origin,
                           const Real& dx)=0;
  ```

  Fill the geometry of ebisRegion. The region argument specifies the subset of the domain over which the EBISBox will live. The domain argument specifies the span of the solution index space. The origin argument specifies the location of the lower-left corner (the zero node) of the solution domain and the dx argument specifies the grid spacing. EBIndexSpace checks that ebisRegion covers the region on output. In this function, the GeometryService must correctly fill all of the internal data in the EBISBox class (we enumerate this data in section 1.5.3. This function is only called if isRegular and isCovered return false for the input region. The steps for filling this data are as follows:

  - Set ebisRegion.m_type=EBISBoxImplem::HasIrregular.
  - Set ebisRegion.m_box=region.
  - Resize and set ebisRegion.m_typeID. On covered cells you set this to -2, on regular cells, you set it to -1 and on irregular cells you set it to 0 or higher, corresponding to the cell's index into ebisRegion.irregVols.
  - Set the volumes in ebisRegion.m_irregVols. At each cell, create a vector of volumes whose length is the number of VoFs in the cell. The internal

class `Volume` contains all the auxiliary VoF information which is not absolutely necessary for indexing. For each `Volume vol` the `GeometryService` must set

  * `vol.m_index`, the `VolIndex` of the volume.
  * `m_volFrac`, the volume fraction of the volume.
  * `m_loFaces`, the low faces of the volume in each direction.
  * `m_hiFaces`, the high faces of the volume in each direction.
  * `m_loAreaFrac`, the low area fractions of the volume in each direction.
  * `m_hiAreaFrac`, the high area fractions of the volume in each direction.

For a `GeometryService` to fill an `EBISBox`, it must extract the internal data of the `EBISBox` and fill it. The internal data of `EBISBox` is described in section 1.5.3.

`GeometryService` is a friend class to `EBISBox` and has access to its internal data. Not all compilers respect that classes which derive from friend classes are also friends. Therefore the internal data should be accessed through these `GeometryService` functions which are designed to get around this compiler feature:

- `Box& getEBISBoxRegion(EBISBox& a_ebisBox) const`

  This returns a reference to the region that the `EBISBox` covers. This needs to be set in all cases.

- `EBISBoxImplem::TAG& getEBISBoxEnum(EBISBox& a_ebisBox) const`

  This returns a reference to the tag that marks whether the `EBISBox` is all regular, all covered, or has irregular cells. This needs to be set in all cases.

- `Vector<Vector<Vol> >& getEBISBoxIrregVols(EBISBox& a_ebisBox) const`

  This returns the list of irregular VoF representations. This must only be filled if the this `EBISBox` is tagged to have irregular cells.

- `BaseFab<int>& getEBISBoxTypeID(EBISBox& a_ebisBox) const`

  Return the flags for each cell in the `EBISBox`. This must only be filled if the this `EBISBox` is tagged to have irregular cells. In this case, covered cells are to be tagged with -2, regular cells are to be tagged with -1 and irregular VoFs are tagged with the index into the vector of irregular volumes which corresponds to this particular VoF.

- `IntVectSet& getEBISBoxMultiCells(EBISBox& a_ebisBox) const`

  Returns a reference to the multiply-valued cells in the `EBISBox`. This must only be filled if the this `EBISBox` is tagged to have irregular cells.

- `IntVectSet& getEBISBoxIrregCells(EBISBox& a_ebisBox) const`

  Return a reference to the set of irregular cells in the `EBISBox`. This must only be filled if the this `EBISBox` is tagged to have irregular cells.

An example of a `GeometryService` class is given in section 1.9.1.

1.5.3 Class EBISBox

EBISBox represents the geometric information of the domain at a given refinement within the boundaries of a particular box. EBISBox can only be accessed by using the the EBISLayout interface. EBISBox has as member data

```
class EBISBox{
...
protected:
  Tag m_type;            //all reg, all covered, or has irregular
  BaseFab<int> m_typeID; //-2 covered,-1 regular, 0 or higher irreg
  Box m_box;             //region
  Vector< Vector< Volume > > irregVols;
```

where the internal class Volume contains all the auxiliary VoF information which is not absolutely necessary for indexing. Volume has the form

```
struct Vol
{
  //this stuff gets filled in the finest level
  //by geometry service
  VolIndex m_index;
  Real m_volFrac;
  Tuple<Vector<FaceIndex>, SpaceDim> m_loFaces;
  Tuple<Vector<FaceIndex>, SpaceDim> m_hiFaces;
  Tuple<Vector<Real>, SpaceDim> m_loAreaFrac;
  Tuple<Vector<Real>, SpaceDim> m_hiAreaFrac;

  //this stuff gets managed by ebindexspace
  Vector<VolIndex> m_finerVoFs;
  VolIndex  m_coarserVoF;
};
```

The integers stored in m_typeid double as the indices into the the vectors of VoF information. The important public member functions of EBISBox are as follows:

- `IntVectSet getMultiCells(const Box& subbox) const;`

  Returns a list all multi-valued cells at the given level of refinement within the input `Box subbox`.

- `IntVectSet getIrregIVS(const Box& boxin) const;`

  Returns the irregular cells of the EBISBox that are within the input subbox.

- `Vector<VolIndex> getVoFs(const IntVect& iv);`

  Gets all the VoFs in a particular cell.

- `int numVoFs(const IntVect& iv) const;`

  Returns the number of VoFs in a particular cell.

-
  ```
  Vector<FaceIndex> getFaces(const VolIndex& vof,
                             int idir, Side::LoHiSide sd);
  ```

  Gets all faces at the specified side and direction of the VoF.

- `bool isRegular(const IntVect& iv) const;`

  Returns true if the input cell is a regular VoF.

- `bool isRegular(const Box& box) const;`

  Returns true if every cell in the input Box is a regular VoF.

- `bool isCovered(const IntVect& iv) const;`

  Returns true if the input cell is a covered cell.

- `bool isCovered(const Box& box) const;`

  Returns true if every cell in the input box is a covered cell.

- `bool isIrregular(const IntVect& iv) const;`

  Returns true if the input cell is an irregular cell.

-
  ```
  int numFaces(const VolIndex& vofin,
          int dir, Side::LoHiSide sd) const;
  ```

  Returns the number of faces the input VoF has in the given direction and side. Returns zero if the VoF has no faces in the given orientation.

- `Real volFrac(const VolIndex& vofin) const;`

  Returns the volume fraction of the input VoF.

-
  ```
  bool isConnected(const VolIndex& vof1,
                   const VolIndex& vof2) const;
  ```

  Return true if the two input VoFs are connected by a face.

- `bool isAllCovered();`

  Return true if every cell in the `EBISBox` is covered.

- `bool isAllRegular();`

  Return true if every cell in the `EBISBox` is regular.

- `RealVect normal(const VolIndex& vofin) const;`

  Returns the normal to the body at the input VoF. Return the zero vector if the answer is undefined (for example, if the VoF is regular or covered).

- `RealVect centroid(const VolIndex& vofin) const;`

  Returns the centroid of the VoF. Returns the zero vector if the VoF is regular or covered. The answer is given as a normalized (by grid spacing) offset from the center of the cell (all numbers range from -0.5 to 0.5).

- `RealVect  centroid(const FaceIndex& facein) const;`

  Return centroid of input face as a RealVect whose component in the uninteresting direction normal to the face is undefined. In the (one or two) interesting directions returns the centroid of the input VoF. Return the zero vector if the face is covered or regular. The answer is given as a normalized (by grid spacing) offset from the center of the cell face (all numbers range from -0.5 to 0.5).

- `Real areaFrac(const FaceIndex& a_vof1);`

  Return the area fraction of the face. Returns zero if the two VoFs in the face are not actually connected.

- `Vector<VolIndex> refine(const VolIndex& coarseVoF) const;`

  Returns the corresponding set of VoFs from the next finer `EBISLevel` (factor of two refinement). The result is only defined if this `EBISBox` was defined by coarsening.

- `VolIndex coarsen(const VolIndex& vofin);`

  Returns the corresponding VoF from the next coarser `EBISLevel` (same solution location, different index space, factor of two refinement ratio).

-
  ```
  void copy(const Box&     a_regionFrom, const Interval& Cd,
            const Box&     a_regionTo,
            const EBISBox& a_source, const Interval& Cs);
  ```

  Copy the information from `a_source` over box `a_regionFrom`, to the `a_regionTo` box of the current `EBISBox`. The interval arguments are ignored. This function is required by the `LevelData` template class.

`GeometryService` is a friend class to `EBISBox` so it can manipulate the internal data of `EBISBox` to create the geometric description.

1.5.4 Class `EBISLayout`

`EBISLayout` is a collection of `EBISBoxes` distributed across processors and associated with a `DisjointBoxLayout` and a number of ghost cells. In a parallel context, `EBISLayout` is the way the user can create parallel, distributed data. `EBISLayouts` are null-constructed and are defined by sending them to the `fillEBISLayout(...)` function of `EBIndexSpace`. `EBISLayout` is constructed around a reference-counted pointer of an `EBISLayoutImplem` object so copying `EBISLayouts` is inexpensive and follows the reference-counted pointer semantic (changing the copied-to object changes the copied-from object). Recall that one can coarsen and refine only by a factor of two using the `EBISBox` class directly. Because `EBISBox` archives the information to do this, it is an inexpensive operation. Coarsening and refinement using larger factors of refinement must be done through `EBISLayout` and it can be expensive, especially in terms of memory usage. When one sets the maximum levels of refinement and coarsening, `EBISLayout` creates mirrors of itself at all intermediate levels of refinement and holds those new `EBISLayouts` as member data. Refinement and coarsening is done by threading through these intermediate levels. The important functions of `EBISLayout` follow.

- `const EBISBox& operator[] (const DataIndex& a_datInd) const;`

  Access the `EBISBox` associated with the input `DataIndex`. Only constant access is permitted.

- `void setMaxRefinementRatio(const int& a_maxRefine);`

  Sets the maximum level of refinement that this `EBISLayout` will have to perform. Creates and holds new `EBISLayouts` at intermediate levels of refinement. Default is one (no refinement done).

- `setMaxCoarseningRatio(const int& a_maxCoarsen);`

  Sets the maximum level of coarsening that this `EBISLayout` will have to perform. Creates and holds new `EBISLayouts` at intermediate levels of coarsening. Default is one (no coarsening done).

-
  ```
  VolIndex coarsen(const VolIndex& a_vof,
                   const int& a_ratio,
                   const DataIndex& a_datInd) const;
  ```

  Returns the index of the VoF corresponding to coarsening the input VoF by the input ratio. It is an error if the ratio is greater than the maximum coarsening ratio or if the VoF does not exist at the input data index.

-
  ```
  Vector<VolIndex> refine(const VolIndex& a_vof,
                          const int& a_ratio,
                          const DataIndex& a_datInd) const;
  ```

  Returns the indices of the VoFs corresponding to refining the input VoF by the input ratio. It is an error if the ratio is greater than the maximum refinement ratio or if the VoF does not exist at the input data index.

- `const BoxLayout& getLayout() const`

  Return the ghosted layout that underlies the `EBISLayout`

1.5.5 Class `VolIndex`

The class `VolIndex` is an abstract index into cell-centered locations which corresponds to the nodes of the computational graph. Every VoF has an associated volume fraction that can be between zero and one. A VoF with zero volume fraction has no volume inside the solution domain. A VoF with unity volume fraction has no covered region. The types of VoF are listed below:

- Regular: VoF has unit volume fraction and has exactly 2*D Faces, each of unit area fraction.
- Covered: VoF has zero volume fraction and no faces.
- Irregular: Any other valid VoF. These are VoFs which either intersect the embedded boundary or border a covered cell.
- Invalid: The VoF is incompletely defined. The default when you create a VoF, and used as the out-of-domain VoF of a boundary Face.

Since we anticipate storing them in very large numbers, we design the class `VolIndex` to be a very small object in terms of memory. Its only member data is an `IntVect` to identify its cell and an integer identifier.

```
class VolIndex{
...
protected:
  IntVect m_cell; // which cell am i in
  int     m_ident;
```

The integer identifier is used to find all the geometric information stored in its `EBISBox`. The class `VolIndex` contains the following important member functions:

- `IntVect gridIndex() const` Returns the `IntVect` of the VoF.
- `int cellIndex() const` Returns the cell identifier of the VoF.

1.5.6 Class `FaceIndex`

The class `FaceIndex` is an abstract index into locations centered on the edges of the graph. A `FaceIndex` exists between two VoFs and is defined by those VoFs. Every `FaceIndex` has an associated area fraction that can be between zero and one. A `FaceIndex` with zero area fraction has no flow area. A `FaceIndex` with unity area fraction has no covered area. It should be noted that a `FaceIndex` knows whether it is a boundary face or an interior face by which constructor was used to define it. Only friend classes (`EBISBox, GeometryService, EBIndexSpace...`) may call the defining constructors. Only the null constructor of `FaceIndex` should be used by users. The internal data of the `FaceIndex` class is as follows:

```
int m_idir;
bool m_isBoundary;
int m_ivoflo;
int m_ivofhi;
IntVect m_ivhi;
IntVect m_ivlo;
```

The cell locations (the `IntVects`) can lie outside the domain if the `FaceIndex` is on the boundary of the domain. The important member functions of this class are:

- `const IntVect& gridIndex(Side::LoHiSide sd) const`

  Return the cell of the `VolIndex` on the `sd` side of the face.

- `const int& cellIndex(Side::LoHiSide sd) const`

  Return the cell index of the `VolIndex` on the `sd` side of the face. Returns -1 if that `VolIndex` is outside the domain of computation.

- `VolIndex getVoF(Side::LoHiSide sd) const`

  Get the VoF at the given side of the face. Will return a VoF with a negative cell index if the IntVect of that VoF is outside the domain.

- `int direction() const;`

  Returns direction of the face. The direction of a `FaceIndex` is the integer coordinate direction (0...D-1) whose unit vector is normal to the face.

- `bool isBoundary() const`

  Returns true if the face is on the boundary of the domain.

1.6 Data Holders for Embedded Boundary Applications

A `BaseIVFAB<T>` is an array of data defined in an irregular region of space. The irregular region is specified by the `VolIndexs` of an `IntVectSet`. Multiple data components per `VolIndex` may be specified in the `BaseIVFAB` definition.

A `BaseIFFAB<T>` is an array of data defined in an irregular region of space. The irregular region is specified by the faces of an `IntVectSet`. All the faces in a `BaseIFFAB` must have the same spatial orientation, which is specified in the `BaseIFFAB` definition. Multiple data components per face may be specified in the definition. `BaseEBCellFAB` is a templated class which holds cell-centered data over a region which is described by a rectangular subset of an embedded boundary. `BaseEBFaceFAB` is a templated class which holds face-centered data over a similar region.

1.6.1 Class `BaseIFFAB<T>`

A `BaseIFFAB<T>` is a templated array of data defined in an irregular region of space. The irregular region is specified by the faces of an `IntVectSet`. All the faces in a `BaseIFFAB` must have the same spatial orientation, which is specified in the `BaseIFFAB` definition. Multiple data components per face may be specified in the definition. The important functions of `BaseIFFAB` follow.

-
  ```
  BaseIFFAB(const IntVectSet& iggeom_in,
            const EBISBox& a_ebisBox,
            int dirin, int nvarin,
            bool interiorOnly=false);
  ```
  Defining constructor. The arguments specify the valid domain in the form of an IntVectSet, the spatial orientation of the faces, and the number of data components per face. The contents are uninitialized. The `interiorOnly` argument specifies whether the data holder will span either the surrounding faces of the set or the interior faces of the set.

- `void setVal(T value);`

  Set a value everywhere. Every data location in this `BaseIFFAB` is set.

-
  ```
  void copy(const Box& a_intBox, const Interval& Cd,
            const Box& a_toBox
            const BaseIFFAB<T>& a_source, const Interval& Cs);
  ```
  Copy the contents of another BaseIFFAB into this BaseIFFAB over the specified regions and intervals.

- `int nComp() const;`

  Return the number of data components of this BaseIFFAB.

- `int direction() const;`

  Return the direction of the faces of this BaseIFFAB.

- `T& operator() (const FaceIndex& edin, int varlocin);`

  Indexing operator. Return a reference to the contents of this BaseIFFAB, at the specified face and data component. The first component is zero, the last is *nvar-1*. The returned object is a modifiable lvalue.

1.6.2 Class `BaseIVFAB<T>`

A `BaseIVFAB<T>` is a templated array of data defined in an irregular region of space. The irregular region is specified by the `VolIndex`s of an `IntVectSet`. Multiple data components per `VolIndex` may be specified in the `BaseIVFAB` definition. The important member functions of `BaseIVFAB` follow.

-
  ```
   BaseIVFAB(const IntVectSet& iggeom_in,
             const EBISBox& a_ebisBox,
             int nvarin = 1);
  ```

  Defining constructor. Specifies the valid domain in the form of an IntVectSet and the number of data components per VoF. The contents are uninitialized.

- `void setVal(T value);`

  Set a value everywhere. Every data location in this `BaseIVFAB` is set to the input value.

-
  ```
  void copy(const Box& a_fromBox, const Interval& destInterval,
            const Box& a_toBox,
            const BaseIVFAB<T>& src, const Interval& srcInterval);
  ```

  Copy the contents of another `BaseIVFAB` into this `BaseIVFAB`. over the specified regions and intervals.

- `T& operator() (const VolIndex& ndin, int varlocin);`

  Indexing operator. Return a reference to the contents of this `BaseIVFAB`, at the specified VoF and data component. The first component is zero, the last is *nvar-1*. The returned object is a modifiable lvalue.

1.6.3 Class `BaseEBCellFAB<T>`

A `BaseEBCellFAB<T>` is a templated holder for cell-centered data over a region which consists of the intersection of a cell-centered box and an `EBIndexSpace`. At every uncovered VoF in this intersection, the `BaseEBCellFAB` contains a specified number of data values. At singly valued cells, the data is stored internally in a `BaseFab<T>`. At multiply-valued cells, the data is stored internally in a `BaseIVFAB`. `BaseEBCellFAB` provides indexing by VoF and access to the regular data's pointer for passage to FORTRAN subroutines. This class does not provide a copy constructor or assignment operator.

The important functions for the class `BaseEBCellFAB` is defined as follows.

- `void define(const EBISBox a_ebis,const Box& a_region,int a_nVar);`

  Full define function. Defines the domain of the `BaseEBCellFAB` to be the intersection of the input `Box` and the domain of the input `EBISBox`. Creates the space for data at every VoF in this intersection.

- `void setVal(T a_value);`

  Set the value of all data in the container to a_value.

-
  ```
  void copy(const Box& a_RegionFrom, const Interval& destInt,
            const Box& a_RegionTo,
  ```

```
            const BaseEBCellFAB<T>& a_srcFab,
            const Interval& srcInt);
```

Copy the data from a_srcFab into the current BaseEBCellFAB regions and intervals specified.

- `T& operator()(const VolIndex& a_vof, int a_nVarLoc);`

  Returns the data at VoF a_vof for variable number a_nVarLoc. Returns a modifiable l value.

- `BaseFab<T>& getSingleValuedFAB();`

  Returns the single-valued data holder. This holds all data which is single-valued (regular and irregular). This is useful so that the data can be passed to Fortran using the BaseFab interface.

- `BaseIVFAB<T>& getMultiValuedFAB();`

  Returns the multi-valued data holder.

- `const IntVectSet& getMultiCells() const;`

  Returns the IntVectSet of all the multiply-valued cells.

1.6.4 Class EBCellFAB

An EBCellFAB is a holder for cell-centered floating–point data over a region which consists of the intersection of a cell-centered box and an EBIndexSpace. It is an extension of a BaseEBCellFAB<Real> which includes arithmetic functions. The data is stored internally in a FArrayBox. At multiply-valued cells, the data is stored internally in a BaseIVFAB<Real>. EBCellFAB provides indexing by VoF and access to the regular data's pointer for passage to FORTRAN subroutines. This class does not provide a copy constructor or assignment operator. EBCellFAB has all the functions of BaseEBCellFAB<Real> and the following extra functions:

- `FArrayBox& getSingleValuedFAB();`

  Returns the regular data holder. This is useful so that the data can be passed to Fortran using the BaseFab interface.

-
  ```
  EBCellFAB& operator+=(const Real& a_valin);
  EBCellFAB& operator-=(const Real& a_valin);
  EBCellFAB& operator*=(const Real& a_valin);
  EBCellFAB& operator/=(const Real& a_valin);
  ```

  Add (or subtract or multiply or divide a_valin to (or from or by or into) every data value in the holder.

-
  ```
  EBCellFAB& operator+=(const EBCellFAB& a_fabin);
  EBCellFAB& operator-=(const EBCellFAB& a_fabin);
  EBCellFAB& operator*=(const EBCellFAB& a_fabin);
  EBCellFAB& operator/=(const EBCellFAB& a_fabin);
  ```

  Add (or subtract or multiply or divide) the internal values to (or from or by or into) the values in `fabin` over the intersection of the domains of the two holders and put the result in the current holder. It is an error if the two holders do not contain the same number of variables.

1.6.5 Class `BaseEBFaceFAB<T>`

A `BaseEBFaceFAB<T>` is a templated holder for face-centered data over a region which consists of the intersection of a cell-centered box and an `EBIndexSpace`. At every uncovered face in this intersection, the `BaseEBFaceFAB` contains a specified number of data values. At singly valued faces, the data is stored internally in a `BaseFab<T>`. At multiply-valued cells, the data is stored internally in a `BaseIFFAB`. `BaseEBFaceFAB` provides indexing by face and access to the regular data's pointer for passage to FORTRAN subroutines. This class does not provide a copy constructor or assignment operator. The important functions for the class `BaseEBFaceFAB` are defined as follows.

-
  ```
  void define(const EBISBox& a_ebis,
              const Box& a_region, int a_idir, int a_nVar,
              bool interiorOnly = false);
  ```

  Full define function. Defines the domain of the `BaseEBFaceFAB` to be the intersection of the input `Box` and the faces of the input `EBISBox` in the given direction. Creates the space for data at every face in this intersection. The `interiorOnly` argument specifies whether the data holder will span either the surrounding faces of the set or the interior faces of the set.

-
  ```
  void setVal(T a_value);
  ```

  Set the value of all data in the container to `a_value`.

-
  ```
  T& operator()(const FaceIndex& a_face, int a_nVarLoc);
  ```

  Returns the data at face `a_face` for variable number `a_nVarLoc`. Returns a modifiable l value.

-
  ```
  void copy(const Box& a_RegionFrom, const Interval& a_destInt,
            const Box& a_RegionTo,
            const EBFaceFAB<T>& a_source,
            const Interval& a_srcInt);
  ```

  Copy the data from `a_source` into the current `BaseEBFaceFAB` over regions and intervals specified.

- `BaseFab<T>& getSingleValuedFAB();`

  Returns the regular data holder. This is useful so that the data can be passed to Fortran using the `BaseFab` interface.

- `const IntVectSet& getMultiCells() const;`

  Returns the `IntVectSet` of all the multiply-valued cells.

1.6.6 Class `EBFaceFAB`

An `EBFaceFAB` is a holder for face-centered floating-point data over a region which consists of the intersection of a face-centered box and an `EBIndexSpace`. It is an extension of a `BaseEBFaceFAB<Real>` which includes arithmetic functions. At single-valued cells, the data is stored internally in a `BaseFab<Real>`. At multiply-valued faces, the data is stored internally in a `BaseIFFAB<Real>`. `EBFaceFAB` has all the functions of `BaseEBFaceFAB<Real>` and the following extra functions:

- `FArrayBox& getSingleValuedFAB();`

  Returns the regular data holder. This is useful so that the data can be passed to Fortran using the `BaseFab` interface.

- `EBFaceFAB& operator+=(const EBFaceFAB& fabin);`
  `EBFaceFAB& operator-=(const EBFaceFAB& fabin);`
  `EBFaceFAB& operator*=(const EBFaceFAB& fabin);`
  `EBFaceFAB& operator/=(const EBFaceFAB& fabin);`

  Add (or subtract or multiply or divide) the values in `a_fabin` to (or from or by or into) the internal values over the intersection of the domains of the two holders and put the result in the current holder. It is an error if the two holders do not contain the same number of variables. It is an error if the two holders have different face directions.

- `EBFaceFAB& operator+=(const Real& a_valin);`
  `EBFaceFAB& operator-=(const Real& a_valin);`
  `EBFaceFAB& operator*=(const Real& a_valin);`
  `EBFaceFAB& operator/=(const Real& a_valin);`

  Add (or subtract or multiply or divide) `a_valin` to (or from or by or into) every data value in the holder.

1.7 Data Structures for Pointwise Iteration

EBChombo contains two classes which facilitate pointwise iteration, `VoFIterator` and `FaceIterator`. `VoFIterator` is used to iterate over every point in an `IntVectSet`. `FaceIterator` iterates over faces in an `IntVectSet` in a particular direction.

1.7.1 Class `VoFIterator`

VoFIterator iterates over every uncovered VoF in an `IntVectSet` inside an `EBISBox`. Its important functions are as follows

-
  ```
  VoFIterator(const IntVectSet& a_ivs,
              const EBGraph& a_ebisBox);
  void define(const IntVectSet& a_ivs,
              const EBGraph& a_ebisBox);
  ```

  Define the `VoFIterator` with the input `IntVectSet` and the `EBISBox`. The `IntVectSet` defines the points that will be iterated over and should be contained within the region of `EBISBox`. Calls `reset()` after construction.

- `void reset();`

  Rewind the iterator to its beginning.

- `void operator++();`

  Advance the iterator to its next VoF.

- `bool ok() const;`

  Return true if there are more unvisited VoFs for the iterator to cover.

- `const VolIndex& operator() () const;`

  Return the current VoF.

The following routine sets the 0th component of the data holder to a constant value at each point in the input set.

```
/*****************/
void setPhiToValue(EBCellFAB& a_phi,
                   const IntVectSet& a_ivs,
                   const EBISBox& a_ebisBox,
                   const Real& a_value)
{
   VoFIterator vofit(a_ivs, a_ebisBox.getEBGraph());
   for(vofit.reset(); vofit.ok(); ++vofit)
     {
        const VolIndex& vof = vofit();
        a_phi(vof, 0) = a_value;
     }
}
/*****************/
```

The call to `reset()` in the above code is unnecessary in this case. One only needs to call `reset()` if an iterator is used multiple times.

1.7.1.1 Performance Note

VoFIterator caches all its VolIndexes into a Vector on construction. In this way, VoFIterator is designed to be fast in iteration but not necessarily fast in construction. If one were to find VoFIterator construction to be a significant performance issue in a class, one might consider caching the VoFIterators one needs in the member data of said class.

1.7.2 Class FaceIterator

The FaceIterator class is used to iterate over faces of a particular direction in an IntVectSet. First we must define FaceStop, the enumeration class which distinguishes which faces at which a given FaceIterator will stop. The entirety of the FaceStop class is given below.

```
class FaceStop
{
public:
  enum WhichFaces{Invalid=-1,
                  SurroundingWithBoundary=0, HiWithBoundary, LoWithBoundary,
                  SurroundingNoBoundary    , HiNoBoundary  , LoNoBoundary,
                  NUMTYPES};
};
```

The enumeratives are described as follows:

- SurroundingWithBoundary means stop at all faces on the high and low sides of IntVectSet cells.
- SurroundingNoBoundary means stop at all faces on the high and low sides of IntVectSet cells, excluding faces on the domain boundary.
- LoWithBoundary means stop at all faces on the low side of IntVectSet cells.
- LoNoBoundary means stop at all faces on the low side of IntVectSet cells, excluding faces on the domain boundary.
- HiWithBoundary means stop at all faces on the high side of IntVectSet cells.
- HiNoBoundary means stop at all faces on the high side of IntVectSet cells, excluding faces on the domain boundary.

Now we may define the important interface of FaceIterator:

-
  ```
  FaceIterator(const IntVectSet& a_ivs,
               const EBGraph& a_ebisBox,
               const int& a_direction,
  ```

```
                 const FaceStop::WhichFaces& a_location);
   void define(const IntVectSet& a_ivs,
               const EBGraph& a_ebisBox,
               const int& a_direction,
               const FaceStop::WhichFaces& a_location);
```

Defining constructor.

- `void reset();`

  Rewind the iterator to its beginning.

- `void operator++();`

  Advance the iterator to its next face.

- `bool ok() const;`

  Return true if there are more unvisited faces for the iterator to cover.

- `const FaceIndex& operator() () const;`

  Return the current face.

The following routine sets the 0th component of the data holder to a constant value at each face in the input set, including boundary faces.

```
/******************/
void setFacePhiToValue(EBFaceFAB& a_phi,
                       const IntVectSet& a_ivs,
                       const EBISBox& a_ebisBox,
                       const Real& a_value)
{
   int direction = a_phi.direction();
   FaceIterator faceit(a_ivs, a_ebisBox.getEBGraph(), direction,
                       FaceStop::SurroundingWithBoundary);
   for(faceit.reset(); faceit.ok(); ++faceit)
     {
        const FaceIndex& face = faceit();
        a_phi(face, 0) = a_value;
     }
}
/******************/
```

The call to `reset()` in the above code is unnecessary in this case. One only needs to call `reset()` if an iterator is used multiple times.

1.7.2.1 Performance Note

`FaceIterator` caches all its `FaceIndexes` into a `Vector` on construction. In this way, `FaceIterator` is designed to be fast in iteration but not necessarily fast in construction. If one were to find `FaceIterator` construction to be a significant performance issue in a class, one might consider caching the `FaceIterators` one needs in the member data of said class.

1.8 Input/Output

1.8.1 Design Considerations

Goals of Input/Output API are:

- Binary portable files
- Completeness:
  - An EBChombo data file should contain sufficient information to reconstruct both the `EBIndexSpace` object, as well as the appropriate data type. In our case we have broken these two functions into separate files.
- Efficiency. An EBChombo data file should strive to be at least as space efficient as the in-memory representation.

As of this time HDF5 still does not support asynchronous DATASET or GROUP creation. This means that to achieve parallel performance we still cannot move to a protocol that maps each `EBISBox` to an HDF5 DATASET.

1.8.2 HDF5 Data File Description

EBChombo uses two independent files. The first file is an output of the `EBIndexSpace` object (a *serialization*). It outputs the EBIndexSpace at it's finest resolution. At the finest grid resolution there are no grid cells that contain multiple control volumes. This is called the `EBFile`. We will strive to consistently use the file extension `.eb` for this file.

The second file is our `EBData` file. It handles EBChombo in a similar manner to EBChombo: regular array data over most of the domain, unstructured data representation where the embedded boundary intersects the regular Cartesian. It is not a complete geometry representation as it only stores geometric information at the AMR level of refinement. We will use the file extension `.ebd` for these files.

VisIt can utilize the `EBData` file.

EBChombo can read and write both `EBFile` and `EBData` files. On read EBChombo only extracts the non-geometric data from an `EBData` file. Geometric information is sourced from the `EBFile`. A future optimization for EBChombo I/O would be an option to turn off the writing of geometric information into the EBData file when VisIt will not be used, or when just regular cell visualization is adequate. This can wait until profiling.

1.8.2.1 EBData file

We introduce four new H5T compound data types: H5T_VOF2D, H5T_VOF3D H5T_FACE2D, H5T_FACE3D:

```
DATATYPE  H5T_COMPOUND H5T_VOF*D {
           H5T_REAL "volFrac";
           H5T_REAL "bndryArea";
           H5T_REALVECT "normal";
           H5T_REALVECT "centroid";
           H5T_INTVECT "cell";
}

DATATYPE  H5T_COMPOUND H5T_FACE*D {
           H5T_REAL "areaFrac";
           H5T_REALVECT "centroid";
           H5T_UINT "hiVol"; // used to index into an H5T_VOF*D array
           H5T_UINT "loVol"; // any face only connects two VOFs
}
```

Header information (as HDF5 Scalar ATTRIBUTE unless otherwise specified):

- "SpaceDim" { DATATYPE H5T_INTEGER }
- "Filetype" { DATATYPE H5T_STRING }
- "AspectRatio" { H5T_REALVECT }
- "NumLevels" { H5T_INTEGER : number of AMR data AND LevelSet levels
- DATASET "RefRatios" { DATATYPE H5T_INTEGER }
- "ProblemDomain" { DATATYPE H5T_BOX} coarsest grid index space
- "DX" { DATATYPE H5T_REAL }
  - coarse grid spacing
- ATTRIBUTE "Ghost" { DATATYPE H5T_INTVECT }

  - not positive how to handle ghosting actually. do we allow the face and node centered data to have a different amount of ghost cells ? What would the LevelSet and Mask need to look like in this case ? Currently this is not going to be allowed. All data sets in an EBData file have the same amount of ghost cells.
  - Box + Ghost are used to determine the sizei, sizej and sizek values. sizei=box.len(0)+2*ghost[0].

- `"CellCenteredComponents" { GROUP}`
  - `"NumC" { DATATYPE H5T_INTEGER }`
  - `"ComponentN" { DATATYPE H5T_STRING }` n'th cell-centered component name
- `"XFaceCenteredComponents" { GROUP }`
  - `"NumX" { DATATYPE H5T_INTEGER }`
  - `"ComponentN" { DATATYPE H5T_STRING }` n'th xface-centered component name
- `"YFaceCenteredComponents" { GROUP }`
  - `"NumY" { DATATYPE H5T_INTEGER }`
  - `"ComponentN" { DATATYPE H5T_STRING }` n'th yface-centered component name
- `"ZFaceCenteredComponents" { GROUP }`
  - `"NumZ" { DATATYPE H5T_INTEGER }`
  - `"ComponentN" { DATATYPE H5T_STRING }` n'th zface-centered component name
- `"NodeCenteredComponents" { GROUP }`
  - `"NumN" { DATATYPE H5T_INTEGER }`
  - `"ComponentN" { DATATYPE H5T_STRING }` n'th node-centered component name

for each AMR level there is an HDF5 GROUP named `level_n`. For each AMR Box at this level of refinement we have regular data, and a set of irregular data. The naming convention has moved to uppercase-contiguous naming instead of lowercase-underscore naming

- `DATASET "Boxes" { DATATYPE H5T_BOX }`
  - Same meaning as in regular Chombo IO.

- `DATASET "Processor" { DATATYPE H5T_INTEGER }`
  - Processor assignment of "Boxes" at the time of writing this file
- `DATASET "Mask" { DATATYPE H5T_CHAR }`
  - Dataset defined at every point of "Boxes" same as Chombo regular data
  - contains the number volumes in this grid cell.
  - not sure if "mask" should be combined with the LevelSet material. the mask is cell-centered, whereas the LevelSet information is node-centered.
  - set to -1 for cells outside problem domain
- `DATASET "MOffsets" { DATATYPE H5T_LLONG }`
- `DATASET "Levelset" { DATATYPE H5T_REAL }`
  - node-centered real-valued data set defined over "Boxes"
  - function is negative valued on covered nodes, positive on uncovered nodes.
  - I do not know yet if this will be an actual *distance function*.
  - *currently unimplemented*
- `DATASET "COffsets" { DATATYPE H5T_LLONG }`
- `DATASET "CRegular" { DATATYPE H5T_REAL }`
  - Regular cell-centered data over "Boxes"
  - Indexing the n'th component at position [i,j,k] in the b'th box:
  - val = CRegular[COffsets[b]+n*(sizei*sizej*sizek)+k*(sizej*sizei)+j*sizei+i]
- `DATASET "NOffsets" { DATATYPE H5T_LLONG }`
- `DATASET "NRegular" { DATATYPE H5T_REAL }`
  - Regular node-centered data over "Boxes"
  - Indexing the n'th component at node [i,j,k] in the b'th box:
  - val = NRegular[NOffsets[b]+n*((sizei+1)*(sizej+1)*(sizek+1)) +k*((sizej+1)*(sizei+1))+j*(si
- `DATASET "VOffsets" { DATATYPE H5T_UINT }`
  - Offset into `DATASET "VOFs"` and `DATASET "CIrregular"` per box
- `DATASET "VOFs" { DATATYPE H5T_VOF*D }`
  - All VOFs stored in box-contiguous form

  - This dataset always has a single layer of ghost cells.

- DATASET "CIrregular" { DATATYPE H5T_REAL }
  - contains NumC data values per irregular VOF
  - indexing the i'th component of the j'th irregular VOF of the b'th Box
  - val = irregular[NumC*(VOffsets[b]+j) + i]
- DATASET "FOffsets" { DATATYPE H5T_LLONG }
- DATASET "XRegular" { DATATYPE H5T_REAL }
  - Regular x-face centered data over "Boxes"
  - Indexing the n'th component at face [i,j,k] in the b'th box:
  - val = XRegular[FOffsets[b]+n*((sizei+1)*sizej*sizek)+k*((sizej)* (size1+1))+j*(sizei+1)+i]
- DATASET "XFaceOffsets" { DATATYPE H5T_UINT }
  - Offset into DATASET "XFaces" and "XIrregular"
- DATASET "XFaces" { DATATYPE H5T_FACE*D }
  - "hiVol" and "loVol" are local to this box. To index the n'th component of "hiVol" associated with i'th xface of the b'th box:
  - val = CIrregular[NumC*(VOffsets[b] + (XFaceOffsets[b] + i).hiVol) + n]
- DATASET "XIrregular" { DATATYPE H5T_REAL }
  - to access the n'th data component associated with the i'th xface of the b'th box
  - val = XIrregular[(XFaceOffsets[b]+i)*NumX + n]
- repeat DATASETs "XRegular" "XFaceOffsets" "XFaces" "XIrregular" for Y and Z faces
  - indexing requires the extra +1 for the direction in question

1.8.3 I/O API

1.9 Usage Patterns

Here we present the usage patterns of the concepts presented in section 1.4. We present an initialization pattern and a calculation pattern along with an example of a GeometryService class.

1.9.1 Creating a GeometryService Object

We show the important `SlabService` class functions. This class specifies that a `Box` in the domain is covered and all other cells are full. It has one data member, `Box m_coveredRegion`, which specifies the covered region of the domain.

```
/******************/
bool
SlabService::isRegular(const Box& a_region,
                       const Box& domain,
                       const RealVect& a_origin,
                       const Real& a_dx) const
{
  Box interBox = m_coveredRegion & a_region;
  return (interBox.isEmpty());
}
/******************/
/******************/
bool
SlabService::isCovered(const Box& a_region,
                       const Box& domain,
                       const RealVect& a_origin,
                       const Real& a_dx) const
{
  return (m_coveredRegion.contains(a_region));
}
/******************/
/******************/
void
SlabService::fillEBISBox(EBISBox& a_ebisRegion,
                         const Box& a_region,
                         const Box& a_domain,
                         const RealVect& a_origin,
                         const Real& a_dx) const
{
  //for some reason, g++ is not letting classes derived
  //from friends be friends so I have to use the end-around
  ebisBoxClear(a_ebisRegion);
  Box&                  implem_region    = getEBISBoxRegion(a_ebisRegion);
  Box&                  implem_domain    = getEBISBoxDomain(a_ebisRegion);
  EBISBoxImplem::TAG&   implem_tag       = getEBISBoxEnum(a_ebisRegion);
  Vector<Vector<Vol> >& implem_irregVols = getEBISBoxIrregVols(a_ebisRegion);
  IntVectSet&           implem_irregCells= getEBISBoxIrregCells(a_ebisRegion);
  BaseFab<int>&         implem_typeID    = getEBISBoxTypeID(a_ebisRegion);
  //don't need this one---no multiply valued cells here.
  IntVectSet&           implem_multiCells= getEBISBoxMultiCells(a_ebisRegion);
```

```
implem_multiCells.makeEmpty();
implem_region  = a_region;
implem_domain  = a_domain;

Box interBox = m_coveredRegion & a_region;
if(interBox.isEmpty())
  {
    implem_tag = EBISBoxImplem::AllRegular;
  }
else if(m_coveredRegion.contains(a_region))
  {
    implem_tag = EBISBoxImplem::AllCovered;
  }
else
  {
    implem_tag = EBISBoxImplem::HasIrregular;
    implem_typeID.resize(a_region, 1);
    //set all cells to regular
    implem_typeID.setVal(-1);
    //set to covered over intersection of two boxes.
    implem_typeID.setVal(-2, interBox, 0, 1);
    //set cells next to the covered region to irregular
    for(int idir = 0; idir < SpaceDim; idir++)
      {
        Box loSideBox = adjCellLo(m_coveredRegion, idir);
        Box hiSideBox = adjCellHi(m_coveredRegion, idir);
        Vector<Box> boxesToDo(2);
        boxesToDo[0] = loSideBox;
        boxesToDo[1] = hiSideBox;
        for(int ibox = 0; ibox < boxesToDo.size(); ibox++)
          {
            const Box& thisBox = boxesToDo[ibox];
            Box iterBox = (thisBox & a_region);
            if(!iterBox.isEmpty())
              {
                BoxIterator bit(iterBox);
                for(bit.reset(); bit.ok(); ++bit)
                  {
                    const IntVect& iv =bit();
                    Vol newVol;
                    newVol.m_volFrac = 1.0;
                    //all irregular cells have only one vof in this EBIS
                    VolIndex thisVoF= getVolIndex(iv, 0);
                    newVol.m_index = thisVoF;
                    //loop through face directions
                    for(int jdir = 0;jdir < SpaceDim; jdir++)
                      {
                        //only add faces in the directions
                        //that are not covered.
                        // all areafracs are unity
                        IntVect loiv = iv - BASISV(jdir);
                        IntVect hiiv = iv + BASISV(jdir);
                        Real areaFrac = 1.0;
                        if(!m_coveredRegion.contains(loiv))
                          {
                            VolIndex loVoF= getVolIndex(loiv, 0);
                            FaceIndex loface;
                            if(a_domain.contains(loiv))
                              {
                                loface=getFaceIndex(loVoF, thisVoF,jdir);
                              }
                            else
                              {
                                loface=getFaceIndex(thisVoF, jdir, Side::Lo);
                              }
                            newVol.m_loFaces[jdir].push_back(loface);
                            newVol.m_loAreaFrac[jdir].push_back(areaFrac);
                          }
                        if(!m_coveredRegion.contains(hiiv))
                          {
                            VolIndex hiVoF= getVolIndex(hiiv, 0);
                            FaceIndex hiface;
                            if(a_domain.contains(hiiv))
                              {
                                hiface=getFaceIndex(hiVoF, thisVoF,jdir);
                              }
                            else
                              {
                                hiface=getFaceIndex(thisVoF, jdir, Side::Hi);
                              }
                            newVol.m_hiFaces[jdir].push_back(hiface);
                            newVol.m_hiAreaFrac[jdir].push_back(areaFrac);
                          }
                      }//end inner loop over face directions
                    implem_irregCells |= iv;
                    //trick.standard.
                    implem_typeID(iv, 0) = implem_irregVols.size();
                    //add the new volume to the ebis
                    implem_irregVols.push_back(Vector<Vol>(1,newVol));
                  }//end loop over cells of box
```

```
                } //end (is the edge box in a_region)
            }//end loop over boxes on the outside of covered box in dir
        } // end loop over directions
    } //end if(a_region intersects covered region)
}
```

1.9.2 Creating Data Holders and Geometric Information

To start a calculation, first the EBIndexSpace is created and the geometric description is fixed. The DisjointBoxLayouts are then created for each level and the corresponding EBISBoxes are then generated. Data holders over the levels are created using a factory class.

```
int NFine; //finest grid size
int NLevels; // number of refinement levels
...
Box domain(IntVect::Zero, (NFine-1)*IntVect::Unit);
createMyGeometry(ebis);
Vector<DisjointBoxLayout> allGrids;
Vector<int> refRatios;
Vector<Box> domains;
Real dxfine;
createMyGrids(NLevels,refRatios,allGrids, domains, dxfine );

EBIndexSpace ebis(domain);
Vector<EBISLayout*> vec_ebislayout(NLevels);
//maximum number of ghost cells I will ever use (this includes
//temporary arrays).
int maxghost = 4;
EBIndexSpace* ebisPtr = Chombo_EBIS::instance();
RealVect origin = RealVect::Zero;
MyGeometryService mygeom;
ebisPtr->define(domain, origin, dxfine, mygeom);

for(int ilevel = 0; ilevel < NLevels; ilevel++)
{
  //domain used to match correct level of refinement
  //for the ebis.   The layout box grown by the number
  //of ghost cells determines how large each EBISBox in
  //the EBISLayout is.

  vec_ebislayout[ilevel] = new EBISLayout();
  ebisPtr->fillEBISLayout(*vec_ebislayout[ilevel],
                          allGrids[ilevel],
                          domains[ilevel], maxghost);
}
//now define the data in all its LevelData splendor
Vector<LevelData<EBCellFAB>* > allDataPtrs(NLevels, NULL);
int nVar = 10;
for(int ilevel = 0; ilevel < NLevels; ilevel++)
{
   const EBISLayout& levelEBIS = vec_ebislayout[ilevel];
   const DisjointBoxLayout& levelGrids = allGrids[ilevel];
   EBCellFABFactory ebfact(levelEBIS);
   allDataPtrs[ilevel] =
        new LevelData<EBCellFAB>(levelGrids,
                                 nVar, maxghost*IntVect::Unit, ebfact);
   defineMyInitialData(*allDataPtrs[ilevel], domains[ilevel]);
}
```

1.9.3 Finite Difference Calculations using EBChombo

Here we present our calculation usage pattern with EBChombo. The regular part of the data holder is extracted and sent to a Fortran routine using Chombo Fortran macros. In the second step, we do the irregular VoFs pointwise.

```
/***********************/
/***********************/
void
EBPoissonOp::applyOp(LevelData<EBCellFAB >& a_lofPhi,
                     LevelData<EBCellFAB >& a_phi,
                     const bool& a_isHomogeneous)
{
  a_phi.exchange(a_phi.interval());
  //loop over grids.
  for(DataIterator dit = a_phi.dataIterator(); dit.ok(); ++dit)
    {
      applyOpGrid(a_lofPhi[dit()], a_phi[dit()], dit(), a_isHomogeneous);
    } //end loop over grids
}

/***********************/
/***********************/
void
EBPoissonOp::applyOpGrid(EBCellFAB& a_lofPhi,
                         const EBCellFAB& a_phi,
                         const DataIndex& a_datInd,
                         bool a_isHomogeneous)
{
  //set value of lphi to zero then loop through
```

```
//directions, adding the 1-D divergence of the
//flux in each direction on each pass.
a_lofPhi.setVal(0.);
const EBISBox& ebisBox = m_ebisl[a_datInd];

for(int idir = 0; idir < SpaceDim; idir++)
  {
    const BaseFab<Real>& regPhi = a_phi.getRegFAB();
    BaseFab<Real>& regLPhi = a_lofPhi.getRegFAB();
    const Box& regBox = m_grids.get(a_datInd);
    assert(regPhi.box().contains(regBox));
    assert(regLPhi.box().contains(regBox));
    Box interiorBox = m_domain;

    interiorBox.grow(idir, -1);
    Box calcBox = (regBox & interiorBox);
    FORT_INCREMENTLAP(CHF_FRA(regLPhi),
                      CHF_CONST_FRA(regPhi),
                      CHF_BOX(calcBox),
                      CHF_CONST_INT(idir),
                      CHF_CONST_REAL(m_dxLevel));


    SideIterator sit;
    for(sit.reset(); sit.ok(); ++sit)
      {
        Box bndrybox, cellbox;
        bool isboundary = false;
        int iside = sign(sit());
        if(sit() == Side::Lo)
          {
            isboundary = (regBox.smallEnd(idir) ==
                          m_domain.smallEnd(idir));
            bndrybox = bdryLo(regBox, idir);
            cellbox = adjCellLo(regBox, idir);
            cellbox.shift(idir, 1);
          }
        else
          {
            isboundary = (regBox.bigEnd(idir) ==
                          m_domain.bigEnd(idir));
            bndrybox = bdryHi(regBox, idir);
            cellbox = adjCellHi(regBox, idir);
            cellbox.shift(idir, -1);
          }
        if(isboundary)
          {
            //now the flux is CELL centered
            BaseFab<Real> flux(cellbox, 1);
            for(BoxIterator bit(cellbox); bit.ok(); ++bit)
              {
                const IntVect& iv = bit();
                Vector<VolIndex> vofs = ebisBox.getVoFs(bit());
                Real fluxval = 0.0;
                for(int ivof = 0; ivof < vofs.size(); ivof++)
                  {
                    const VolIndex& vof = vofs[ivof];
                    const BaseFunc& bdata =
                      getDomBndryData(idir, sit(), a_datInd);
                    const FluxBC& fluxbc = m_domfluxbc(idir,sit());
                    //domfluxbc stuff is already multiplied
                    //by face area*areafrac
                    fluxval =fluxbc.applyFluxBC(vof, 0, ebisBox, a_phi,
                                                bdata, a_isHomogeneous);
                  }
                flux(iv, 0) = fluxval;
              } //end loop over boundary box
            //this makes the flux face centered
            flux.shiftHalf(idir, iside);

            FORT_INCRLINELAP(CHF_FRA(regLPhi),
                             CHF_CONST_FRA(regPhi),
                             CHF_BOX(cellbox),
                             CHF_CONST_INT(idir),
                             CHF_CONST_INT(iside),
                             CHF_CONST_REAL(m_dxLevel));

            FORT_BOUNDARYLAP(CHF_FRA(regLPhi),
                             CHF_CONST_FRA(flux),
                             CHF_CONST_FRA(regPhi),
                             CHF_BOX(bndrybox),
                             CHF_CONST_INT(idir),
                             CHF_CONST_INT(iside),
                             CHF_CONST_REAL(m_dxLevel));
          }//end is boundary
      }//end loop over sides
  }//end loop over directions

//do irregular cells. this includes boundary conditions
//also redo cells next to boundary
IntVectSet ivsIrreg = m_irregRegions[a_datInd];
for(VoFIterator vofit(m_irregRegions[a_datInd], ebisBox);
    vofit.ok(); ++vofit)
  {
    a_lofPhi(vofit(), 0) =  applyOpVoF(vofit(), a_phi, a_datInd,
                                       a_isHomogeneous);
  }
}
```

1.10 Landmines

This section is intended to point out some of the uses of EBChombo that will result in errors that can be difficult to detect.

1.10.1 Data Holder Architecture

For performance reasons, `BaseEBFaceFAB` and `BaseEBCellFAB` both hold all their *single-valued* data in dense arrays and *multi-valued* data in irregular arrays. Note that this is distinct from regular and irregular cells. This makes data access much faster but it also provides (at current count) three traps for the unwary.

1.10.1.1 Update-in-Place difficulties

If one naively follows the standard EBChombo usage pattern for updating a quantity in place, one will probably

- Update the regular data in Fortran.
- Update irregular data in C++
- Figure out much later that the single-valued irregular cells have been updated twice.

To avoid this, one can store her state before the update starts and use this stored state to update the irregular cells properly.

1.10.1.2 Norms and other Agglomerations of Data

Say one wants to compute a maximum of the wave speed of her data over a particular box. The naive implementation that simply calls Fortran for all single-valued calls and then loops over all multivalued cells in C++ can have undefined behavior. Any cell in the `BaseFab` that underlies a multivalued cell has undefined values. We recommend that such an operation be done pointwise in C++.

1.10.1.3 Stencil Size and Data Holders

By fiat we have defined that regular cells are those cells who have unit area fractions and unit volume fraction. We also define to be irregular any full cell that borders a multivalued cell. This allows stencils that extend only one extra cell (in each direction) in Fortran. If one uses a wider stencil, she risks updating in Fortran valid regular data with invalid data that underlies multivalued cells.

1.10.2 Sending Irregular Data to Fortran

If one intends to send irregular data (`BaseIFFAB` or `BaseIVFAB`) to Fortran, she must understand that the `Box` arguments that have been sent to Fortran are artificial. The `Box` is just a construct to provide Fortran with the correct size of the data. The actual indices of the data `in no way correspond` to the data locations on the grid. This has two very important implications.

- Irregular data holders of different sizes will not be able to interact in Fortran. The indices of data in the same VoF `will not` be the same for the two data holders.
- Only pointwise operations on data are well-defined. Any kind of finite difference-type operation in Fortran for irregular data holders will result in undefined behavior.

Chapter 2

Layer 2: Workshop: Using The Divergence Theorem for Geometry Generation

2.1 Introduction

For some computations, it is possible to specify the domain of computation based upon functional description of the boundary interface. The workshop package provides a systematic algorithm and API that will produce an embedded boundary description of the interface which converges to second order with grid refinement to the functional description of the interface. Gueyffier, et. al. [7] solve a similar problem in their volume-of-fluid method in three dimensions. We have used some of their ideas about renormalization in this work.

This document specifies the workshop algorithm and the associated API.

2.2 Algorithm

The workshop algorithm provides a numerically convergent way to convert a functional description of an interface to an embedded boundary description. Any given cell is either entirely inside the covered region (covered), entirely outside the covered region (regular), or intersected the cell by the interface (irregular). A workshop class must be able to identify cells as such. Once a cell is identified to be irregular, the workshop class then defines which direction is "up". This determines which coordinate direction of the surface is treated as the dependent variable in the cell ($y = \phi(x, z)$, for example). The workshop class must then provide the intersection of the surface with coordinate lines that vary only in the "up" direction.

2.2.1 Calculation of Edge Intersections

Given a surface, S, we have assumed that S is specified locally as a function. To fix notation, suppose that at a given n-dimensional cell,E, a coordinate direction x_i has been specified as well as a the local function $\phi : \mathbf{R}^{n-1} \to \mathbf{R}$, which describes S. For any $\mathbf{x} \in \partial E$, the intersection of S with the line in the x_i direction that passes through $\mathbf{x}$ may be calculated by evaluating $\phi(x_1, x_2, \cdots x_{i-1}, x_{i+1}, \cdots x_n)$.

For the edges of E that don't lie in the $\mathbf{e}_i$ direction an iterative method such as Brent's Algorithm may be used to calculate the intersection or observe the non-intersection of S with the edge. However for the special case of the discrete Divergence operator a simpler method suffices. We now address this special case.

Denote the endpoints of such an edge by $\mathbf{a}'$ and $\mathbf{c}'$. Let $\mathbf{a} = a_1, a_2, \cdots a_{i-1}, a_{i+1}, \cdots a_n)$. That is, $\mathbf{a}$ denotes the projection of $\mathbf{a}'$ onto the space spanned by the $n-1$ vectors $\mathbf{e}_j$ for $j \neq i$. Similarly, $\mathbf{c}$ denotes the projection of $\mathbf{c}'$ on the same space.

If $A = \phi(\mathbf{a}) > a_i$ and $C = \phi(\mathbf{c}) > c_i$ or $A = \phi(\mathbf{a}) < a_i$ and $C = \phi(\mathbf{c}) < c_i$, we conclude that S does not intersect the edge. If $A = \phi(\mathbf{a}) = a_i$ or $C = \phi(\mathbf{c}) = c_i$, then S intersects an endpoint of the segment. If $(A - a_i)(C - c_i) < 0$ then there is an intersection in the interior, which we estimate by linear interpolation.

2.2.2 Algorithm for d-dimensional cube sliced by a co-dimension 1 surface

Let Ω denote a polytope formed by the intersection of an d-dimensional cube with a codimension one surface. Let Θ denote the portion of the surface that forms one of the faces of Ω. Define B by $B = \partial\Omega - \Theta$. B comprises the faces of Ω that lie in coordinate directions.

We consider the problem of computing moments over Ω, assuming that the intersections of Θ with the one-dimensional edges of Ω have been calculated.

Let $\mathbf{k}$ denote a multiindex of whole numbers, $\mathbf{k} = k_1, \cdots k_n$. Let $\mathbf{x}^{\mathbf{k}}$ denote a monomial of order $|\mathbf{k}| = k_1 + \cdots + k_n$. That is,$\mathbf{x}^{\mathbf{k}} = x_1^{k_1} \cdots x_n^{k_n}$. Finally, let $\boldsymbol{e}_i$ denote a unit vector in the i direction.

Fix a whole number $p > 0$. For each coordinate direction, $\boldsymbol{e}_i$, for each multi-index, $\mathbf{k}$, of order $p+1$, we construct a vector field, $\boldsymbol{F} = \boldsymbol{F}(\mathbf{k}, i) = \mathbf{x}^{\mathbf{k}}\boldsymbol{e}_i$. Note that the divergence of $\boldsymbol{F}$ is either zero or a monomial of order p. Furthermore, every monomial of order p may be obtained as the divergence of such an $\boldsymbol{F}$. For example, let $\mathbf{j}$ denote a multi-index of order p, for each i, $\boldsymbol{F} = \mathbf{x}^{\mathbf{j}} x_i \boldsymbol{e}_i$ satisfies the condition $\frac{1}{j_i+1}\nabla \cdot \boldsymbol{F} = \mathbf{x}^{\mathbf{j}}$.

Therefore, using the divergence theorem we may write:

$$C \int_{\Omega} \mathbf{x}^{\mathbf{j}} d\Omega = \int_{\Omega} \nabla \cdot \boldsymbol{F} d\Omega = \int_{\partial\Omega} \boldsymbol{F} \cdot \boldsymbol{n} \, d(\partial\Omega) \tag{2.1}$$

where $\boldsymbol{n}$ denotes the outward unit normal and $C = j_i + 1$ is a constant.

If we let μ_i^+ and μ_i^- denote the two faces of Ω with outward normal $\boldsymbol{e}_i$ and $-\boldsymbol{e}_i$, respectively, we can rearrange the terms of last integral to observe:

$$C\int_{\Omega} \mathbf{x}^{\mathbf{j}}\, d\Omega - \int_{\Theta} \mathbf{x}^{\mathbf{j}} x_i \boldsymbol{n}_i^{\theta} d\theta \tag{2.2}$$

$$= \sum_{\mu \in M} \int_{\mu} \boldsymbol{F} \cdot \boldsymbol{n}^{\mu} d\mu \tag{2.3}$$

$$= \int_{\mu_i^+} \mathbf{x}^{\mathbf{k}}\, d\mu_i^+ - \int_{\mu_i^-} \mathbf{x}^{\mathbf{k}}\, d\mu_i^-. \tag{2.4}$$

where $\boldsymbol{n}^{\theta}$ denotes the normal to Θ and similarly for $\boldsymbol{n}^{\mu}$. μ_i^+ and μ_i^- are polytopes of dimension $n-1$.

We make the approximation that $\boldsymbol{n}_i^{\theta}$ is constant.

$$\int_{\Theta} \mathbf{x}^{\mathbf{j}} x_i \boldsymbol{n}_i^{\theta} d\theta \approx \boldsymbol{n}_i^{\theta} \int_{\Theta} \mathbf{x}^{\mathbf{j}} x_i d\theta \tag{2.5}$$

Using this approximation, we write an approximate divergence theorem expression:

$$C\int_{\Omega} \mathbf{x}^{\mathbf{j}}\, d\Omega - \boldsymbol{n}_i^{\theta} \int_{\Theta} \mathbf{x}^{\mathbf{j}} x_i d\theta \tag{2.6}$$

$$\approx \sum_{\mu \in M} \int_{\mu} \boldsymbol{F} \cdot \boldsymbol{n}^{\mu} d\mu \tag{2.7}$$

$$= \int_{\mu_i^+} \mathbf{x}^{\mathbf{k}}\, d\mu_i^+ - \int_{\mu_i^-} \mathbf{x}^{\mathbf{k}}\, d\mu_i^-. \tag{2.8}$$

We consider the the terms on the left as unknowns, and for the moment we assume that the terms on the right are known as well as the normal to Θ, $\boldsymbol{n}^{\theta}$. We replace $\approx$ by $=$ and form the linear system of all such equations that can be obtained by varying i and $\mathbf{j}$ in the construction of $\boldsymbol{F}(\mathbf{j}, i)$. (Recall that $|\mathbf{j}| = p + 1$.) Each monomial of order p contributes $n+1$ unknowns to the systems. While each monomial of order $p+1$ contributes n equations to the system. Since the number of monmomials of order (p+1) is always at least n more than the number of monomials of order p, the system is formally overdetermined. In the event that the surface S is planar, our approximation is exact and the system is consistent. In the event that S is not planar there may still be a plane that passes through the edge intersections. Here again the system is consistent; the algorithm recovers this plane. However, whether the system is consistent or not, approximate or not, the column space of the associated matrix has full rank, and the system has a least squares solution which we find.

Still assuming that the right hand side is known, $\boldsymbol{n}^{\theta}$ may be approximated by taking $\boldsymbol{F}$ to be a constant vector field ($p = 0$). In this case the linear system is exactly determined and we solve it exactly. Therefore the algorithm will be complete provided the right hand sides are known. However, the right hand sides are moments over $n-1$ dimensional cubes sliced by codimension 1 surfaces. Hence after $(n-2)$ further iterations of the algorithm we may express the right hand sides as $(p+n-1)$th moments over a set of one dimensional line segments, which may be conveniently calculated directly.

2.3 API

2.3.1 Design

`Workshop` is a class which inherits from `GeometryService` and implements the `GeometryService` interface using the workshop algorithm. `Workshop` contains a `BaseLocalGeometry` class. This base class is an interface which encapsulates the steps to the workshop algorithm of reducing a surface that is locally defined as a function to an embedded boundary description. The `Workshop` class is defined by its only constructor.

- `Workshop(const BaseLocalGeometry& localGeometry)`

  Define the Workshop with the input local geometry description.

A `BaseLocalGeometry`-derived class must be able to answer the following questions:

- Is the box in question regular, or covered (or neither)? A regular box in this context is a cell which is entirely outside the covered region. A covered box is entirely inside the covered region.
- If the cell is irregular, which signed coordinate direction is “up” in the cell? The up direction corresponds to the coordinate direction that corresponds to the largest component of the normal to the boundary at this cell.
- For an irregular cell, given the independent variables, return the dependent variable. Consider again figure **??**. In this example, the y direction is the “up” direction and the workshop-derived class must be able to return values of y in the cell given x and z.

Given this, the `BaseLocalGeometry` class has the following pure virtual functions in its interface:

-
  ```
  virtual bool isRegular(const Box& region, const Box& domain,
                         const RealVect& origin, const Real& dx)
                         const =0;
  virtual bool isCovered(const Box& region, const Box& domain,
                         const RealVect& origin, const Real& dx)
                         const =0;
  ```

  Return true if *every* cell in the input region is regular or covered. Argument `region` is the subset of the domain. The `domain` argument specifies is the span of the solution index space. The origin argument specifies the location of the lower-left corner (the zero node) of the solution domain and the dx argument specifies the grid spacing.

-
  ```
  virtual
  int upDirection(const IntVect& iv,
                  const Box& domain,
                  const RealVect& a_origin,
                  const Real& a_dx)  const = 0;
  ```
  This returns the signed integer which most closely represents the normal direction of the interface at an irregular cell (which coordinate direction has the largest normal component). This will only be called if the cell is irregular.

-
  ```
  virtual
  Real localFuncValue(const RealVect& independentCoords,
                      const int& upDirection,
                      const IntVect& a_iv,
                      const Box& a_domain,
                      const RealVect& a_origin,
                      const Real& a_dx) const = 0;
  ```
  Return the value at the dependent coordinate given the independent coordinates. The upDirection argument defines which coordinate is the dependent one.

-
  ```
  virtual
  BaseLocalGeometry* new_baseLocalGeometry() const = 0;
  ```
  Return a newly allocated derived class. The responsibility for deleting the memory is left to the calling function.

2.3.2 Example

The Workshop class has the following usage pattern. The local geometry description is defined and used to define the Workshop class. The Workshop is then used to define the global EBIndexSpace.

```
class MyGeometry: public BaseLocalGeometry
{
....
};

void defineMyEBIS(const Box& domain,
                  const RealVect& origin,
                  const Real& dx)
{
  MyGeometry myLocalGeom;
```

```
  Workshop myWorkshop(myLocalGeom);
  EBIndexSpace* ebisPtr = Chombo_EBIS::instance();
  ebisPtr->define(domain, origin, dx, myWorkshop);
}
```

Chapter 3

Layer 3–EBAMRTools: Multi-Level Tools

3.1 Introduction

This document is meant to discuss the different components of the EBAMRTools component of the EBChombo infrastructure for embedded boundary, block-structured adaptive mesh applications. The principal operations that these tools execute are as follows:

- Average a level's worth of data onto the next coarser level.
- Interpolate in a piecewise-linear fashion data from a coarser level to a finer level.
- Fill ghost cells at a coarse-fine interface with a second-order interpolation between the coarse and fine data.
- Fill ghost cells at a coarse-fine interface with data interpolated using a bilinear interpolation.
- Preserve multi-level conservation using refluxing.
- Redistribute mass differences between stable and conservative schemes.

After a discourse on the notational difficulties of embedded boundaries, we will discuss our algorithm for each of these tasks.

3.2 Notation

All these operations take place in a very similar context to that presented in [4]. For non-embedded boundary notation, refer to that document. The standard (i, j, k) is not sufficient here to denote a computational cell as there can be multiple VoFs per cell. We define $\boldsymbol{v}$ to be the notation for a VoF and $\boldsymbol{f}$ to be a face. The function $ind(\boldsymbol{v})$ produces the cell which the VoF lives in. We define $\boldsymbol{v}^+(f)$ to be the VoF on the high side of face f; $\boldsymbol{v}^-(f)$ is the VoF on the low side of face $\boldsymbol{f}$; $\boldsymbol{f}_d^+(v)$ is the set of faces on the high side of VoF v; $f_d^-(v)$ is the set of faces on the low side of VoF $\boldsymbol{v}$, where $d \in \{x, y, z\}$ is a coordinate direction (the number of directions is D). Also, we compose these operators to represent the set of VoFs directly connected to a given VoF: $v_d^+(v) = \boldsymbol{v}^+(f_d^+(\boldsymbol{v}))$ and $\boldsymbol{v}_d^-(v) = v^-(f_d^-(\boldsymbol{v}))$. The $<<$ operator shifts data in the direction of the right hand argument:

$$(\phi << \boldsymbol{e}^d)_{\boldsymbol{v}} = \phi_{v_d^+(v)} \tag{3.1}$$

We follow the same approach in the EB case in defining multilevel data and operators as we did for ordinary AMR. Given an AMR mesh hierarchy $\{\Omega^l\}_{l=0}^{lmax}$, we define the valid VoFs on level l to be

$$\mathcal{V}_{valid}^l = ind^{-1}(\Omega_{valid}^l) \tag{3.2}$$

and composite cell-centered data

$$\varphi^{comp} = \{\varphi^{l,valid}\}_{l=0}^{lmax}, \varphi^{l,valid} : \mathcal{V}_{valid}^l \to \mathbb{R}^m \tag{3.3}$$

For face-centered data,

$$\begin{gathered}\mathcal{F}_{valid}^{l,d} = ind^{-1}(\Omega_{valid}^{l,\boldsymbol{e}^d}) \\ \vec{F}^{l,valid} = (F_0^{l,valid}, \dots, F_{D-1}^{l,valid}) \\ F_d^{l,valid} : \mathcal{F}_{valid}^{l,d} \to \mathbb{R}^m\end{gathered} \tag{3.4}$$

3.3 Conservative Averaging

Assume that there are two levels of grids Ω^c, Ω^f, with data defined on the fine grid and on the valid region of the coarse grid

$$\varphi^f : ind^{-1}(\Omega^f) \to \mathbb{R}\varphi^{c,valid} : ind^{-1}(\Omega_{valid}^c) \to \mathbb{R} \tag{3.5}$$

We assume that $\mathcal{C}_r(\tilde{\Omega}^f) \cap \Gamma^c \subset \Omega^c$. We want to replace the coarse data which is covered by fine data with the volume-weighted average of the fine data. This operator is used to average from finer levels on to coarser levels, or for constructing averaged residuals in multigrid iteration. We define the volume weighted average

$$\begin{gathered}\varphi_{\boldsymbol{v_c}}^c = Av(\varphi^f, n_{ref})_{\boldsymbol{v_c}} \\ Av(\varphi^f) = \frac{1}{V^c} \sum_{\boldsymbol{v_f} \in \mathcal{F}} V^f \varphi_{\boldsymbol{v_f}} \\ \mathcal{F} = \mathcal{C}_{n_{ref}}^{-1}(\boldsymbol{v_c})\end{gathered} \tag{3.6}$$

3.4 Interpolation Operations

3.4.1 Piecewise Linear Interpolation

This method is primarily used to initialize fine grid data after regridding. Given a level array φ^c on Ω^c, we want to compute $I_{pwl}(\varphi)$ defined on an Ω^f properly nested in Ω^c. For the values on $C(\tilde{\Omega}^f)$, interpolate in a piecewise-linear fashion in space, using the values $\tilde{\varphi}^c$ (we assume that the coarse data already contains the average of the fine data as discussed in the last section).

$$\begin{gathered}\varphi^f_{\boldsymbol{v_f}} = \tilde{\varphi}^c_{\boldsymbol{v_c}} + \sum_{d=0}^{D-1} (\tfrac{(ind(\boldsymbol{v_f})_d + \frac{1}{2})}{n_{ref}} - ind(\boldsymbol{v_c}) + \tfrac{1}{2}))\Delta^d \cdot \varphi^c_{\boldsymbol{v_c}} \\ \text{where } \boldsymbol{v_c} \in ind^{-1}(\tilde{\Omega}^f - \Omega^f) \\ \boldsymbol{v_c} = \mathcal{C}_{n_{ref}}(\boldsymbol{v_f}).\end{gathered} \tag{3.7}$$

The slopes Δ^d are computed using minmod limiting as shown below:

$$\begin{gathered}\Delta^d W_{\boldsymbol{v_c}} = \delta^{minmod}(W_{\boldsymbol{v_c}}) | \delta^L(W_{\boldsymbol{v_c}}) | \delta^R(W_{\boldsymbol{v_c}}) | 0 \\ \delta^L(W_{\boldsymbol{v_c}}) = W_{\boldsymbol{v_c}} - (W^n_{\boldsymbol{v}<<-\boldsymbol{e}^d}) \\ \delta^R(W_{\boldsymbol{v_c}}) = (W^n_{\boldsymbol{v}<<\boldsymbol{e}^d}) - W_{\boldsymbol{v_c}}\end{gathered} \tag{3.8}$$

$$\delta^{minmod} = \left\{ \begin{array}{ll} min(|\delta_L|, |\delta_R|) \cdot sign(\delta_L + \delta_R) & \text{if } \delta_L \cdot \delta_R > 0 \\ 0 & \text{otherwise} \end{array} \right\} \tag{3.9}$$

The shift operator (denoted by $<<$) is defined using a simple average of connected values.

3.4.2 Piecewise-Linear Coarse-Fine Boundary Interpolation

In the next algorithm, we use the same linear interpolant but we also interpolate in time between levels of time. We have the solution on the coarser level of refinement at two time levels, t_{Cold} and t_{Cnew}. We want to compute an extension $\tilde{\varphi}^f$ of φ^f on $\tilde{\Omega}^f = \mathcal{G}(\Omega^f, p) \cap \Gamma^f, p > 0$ that exists at time level t_F where $t_{Cold} < t_f < t_{Cnew}$. We assume that $\mathcal{C}_r(\tilde{\Omega}^f) \cap \Gamma^c C \Omega^c$. Extend $\varphi^{c,valid}$ to φ^c, defined on all of $ind^{-1}(\Omega^c)$.

$$\varphi^c_{\boldsymbol{v_c}} = Av(\varphi^f, n_{ref})_{\boldsymbol{v_c}}, \boldsymbol{v_c} \in ind^{-1}\mathcal{C}_{n_{ref}}(\Omega^f) \tag{3.10}$$

At both t_{Cold} and t_{Cnew}, for the values on $\tilde{\Omega}^f - \Omega^f$ compute a piecewise linear interpolant, using the values $\tilde{\varphi}^c$.

$$\begin{gathered}\tilde{\varphi}^f_{\boldsymbol{v_f}} = \tilde{\varphi}^f_{\boldsymbol{v_c}} + \sum_{d=0}^{D-1} (\tfrac{(ind(\boldsymbol{v_f})_d + \frac{1}{2})}{n_{ref}} - (ind(\boldsymbol{v_c}) + \tfrac{1}{2}))\Delta^d \cdot \varphi^c_{\boldsymbol{v_c}} \\ \text{where } \boldsymbol{v_c} \in ind^{-1}(\tilde{\Omega}^f - \Omega^f), \\ \boldsymbol{v_c} = \mathcal{C}_{n_{ref}}(\boldsymbol{v_f}).\end{gathered} \tag{3.11}$$

The slopes Δ^d are computed using minmod limiting as shown in equation 3.9. We then interpolate in time between the new and old interpolated values.

$$\varphi^f_{\boldsymbol{v_f},t_F} = \tilde{\varphi}^f_{\boldsymbol{v_f},t_{Cold}} + \frac{t_F - t_{Cold}}{t_{Cnew} - t_{Cold}}(\tilde{\varphi}^f_{\boldsymbol{v_f},t_{Cnew}} - \tilde{\varphi}^f_{\boldsymbol{v_f},t_{Cold}}) \qquad (3.12)$$

This process should produce an interpolated value which has second-order error in both time and space.

3.4.3 Quadratic Coarse-Fine Boundary Interpolation

Away from locations where the embedded boundary crosses the coarse-fine interface, we use the algorithm in [4].

To proceed from here we need to define the corner ghost cells region. Say we have two levels of refinement. Define Ω^f as that region covered by the finer level. Define Λ^f to be the problem domain at the finest refinement. Define G^d to be the grow operation that only grows a region in the coordinate direction d. Define the grow operator G to be the operator which grows a region by one cell in all coordinate directions. In three dimensions, this is

$$G(\Omega, 1) \equiv G^1(G^2(G^3(\Omega, 1), 1), 1)$$

The coarse-fine layer of ghost Ω^{cf} cells is defined to be

$$\Omega^{cf,f} = (G(\Omega^f, 1) - \Omega^f) \cap \Lambda^f$$

The ghost cells which are not corners Ω^e can be obtained by shifting Ω^f along coordinate directions:

$$\Omega e, f = (\bigcup_{d=1}^{D} G^d(\Omega^f, 1) - \Omega^f) \cap \Lambda^f$$

The corner ghost cells Ω^p are defined to be

$$\Omega^p = \Omega^{cf} - \Omega^e$$

Define C to be the pointwise coarsening operation and r to be the refinement ratio. We define the coarse-fine interface set on the coarse level $\Omega^{cf,c}$ to be the coarse cells which underly the fine ghost cells.

$$\Omega^{cf,c} = C(\Omega^{cf,f}, r)$$

Because of proper nesting requirements, we claim that $\Omega^{e,c} = C(\Omega^{e,f}, r)$ and $\Omega^{p,c} = C(\Omega^{c,f}, r)$ do not intersect and $\Omega^{cf,c} = \Omega^{e,c} \cap \Omega^{p,c}$.

We use the Johansen stencil for Dirichlet EB boundary conditions away from the coarse-fine boundary. When any VoF of the Johansen stencil is within a ghost cell on the coarse-fine interface, we drop to the least-squares stencil which has a smaller. The least-squares stencil still requires corner ghost cells be filled. In the absence of coarse-fine interfaces intersecting embedded boundaries, we only need to fill ghost cells which were not corners Ω^e. Since we are proposing to allow this intersection, we must interpolate to all of Ω^{cf}.

3.4.3.1 Interpolation to non-corner ghost cells $\Omega^{e,f}$

Away from points where the coarse-fine interface, we interpolate using QuadCFInterp. It uses one-sided differences to avoid using coarse data under finer data. See the AMRTools section of the Chombo design document for details. Though that is somewhat ideologically inconsistent with the following strategy near embedded boundaries, we feel that, as a proven technology, QuadCFInterp should be left alone.

This section deals only with quadratic interpolation near where the coarse-fine interface crosses the embedded boundary design document. The functional change from QuadCFInterp here is that we only need to do one-sided difference when covered cells are near. Because we are doing higher-order averaging to fill coarse data that is under finer data (see section 3.7.0.1), we can allow the coarse-fine interpolation stencil to reach under finer grids.

We present the natural extension of the regular grid description to embedded boundaries of quadratic coarse-fine interpolation. For each coordinate direction d, we compute values of ϕ in the set $\Omega^{e,f}_d = (G^d(\Omega^f,1) - \Omega^f) \cap \Lambda^f$. Define the "valid" parts of the domain to be the parts of the domain whose volume fractions are greater than zero.

$$\Omega^{c,valid}_{\boldsymbol{i}} = \{\boldsymbol{i} : \boldsymbol{i} \in \Omega^c \text{ and } \kappa_{\boldsymbol{i}} > 0\}$$

To perform this interpolation, we first observe that, given $\boldsymbol{i} \in \tilde{\Omega}^f_k - \Omega^f$, there is a unique choice of $\pm$ and d, such that $\boldsymbol{i} \mp \boldsymbol{e}^d \in \Omega^f_k$. Having specified that choice, the interpolant is constructed in two steps

(i) Interpolation in the direction orthogonal to $\boldsymbol{e}^d$. We compute

$$\boldsymbol{x} = \frac{\boldsymbol{i} + \frac{1}{2}\boldsymbol{u}}{r} - (\boldsymbol{i}^c + \frac{1}{2}\boldsymbol{u})$$

where $\boldsymbol{i}^c = \mathcal{C}_r(\boldsymbol{i})$. The real-valued vector $\boldsymbol{x}$ is the displacement of the cell center $\boldsymbol{i}$ on the fine grid from the cell center at $\boldsymbol{i}^c$ on the coarse grid, scaled by h^c.

$$\hat{\varphi}_{\boldsymbol{i}} = \varphi^c_{\boldsymbol{i}^c} + \sum_{d' \neq d} \Big[\big(x_{d'} (D^{1,d'} \varphi^c)_{\boldsymbol{i}^c} + \frac{1}{2} (x_{d'})^2 (D^{2,d'} \varphi^c)_{\boldsymbol{i}^c} \big) + \sum_{d'' \neq d, d'' \neq d'} x_{d'} x_{d''} (D^{d'd''} \varphi^c)_{\boldsymbol{i}^c} \Big]$$

The second sum has only one term if $\mathbf{D} = 3$, and no terms if $\mathbf{D} = 2$.

(ii) Interpolation in the normal direction.

$$\tilde{\varphi}_{\boldsymbol{i}} = I^B_q(\varphi^f, \varphi^{c,valid}) \equiv 4a + 2b + c \ , \ \tilde{x_d} = x_d - \frac{1}{2}(r+3)1$$

where a, b, c are computed to interpolate between the collinear data

$$((\boldsymbol{i} \pm \frac{1}{2}(n^l_{ref} - 1)\boldsymbol{e}^d)h, \hat{\varphi}_{\boldsymbol{i}}),$$
$$((\boldsymbol{i} \mp \boldsymbol{e}^d)h, \varphi^l_{\boldsymbol{i} \mp \boldsymbol{e}_d}),$$
$$((\boldsymbol{i} \mp 2\boldsymbol{e}^d)h, \varphi^l_{\boldsymbol{i} \mp 2\boldsymbol{e}_d)}$$

In (i), the quantities $D^{1,d'}\varphi^c, D^{2,d'}\varphi^c$ and $D^{d'd''}\varphi^c$ are difference approximations to $\frac{\partial}{\partial x_{d'}}, \frac{\partial^2}{\partial x_{d'}^2}$, and $\frac{\partial^2}{\partial x_{d'}\partial x_{d''}}$, respectively. $D^{1,d}\varphi$ must be accurate to $O(h^2)$, while the other two quantities need only be $O(h)$. The strategy for computing these quantities is to use only values in Ω^c_{valid} to compute these difference approximations. For the case of $D^{1,d'}\varphi, D^{2,d'}\varphi$, we use 3-point stencils, centered if possible, or shifted as required to consist of points on Ω^c_{valid}.

$$(D^{1,d'}\varphi)_{\boldsymbol{i}} = \begin{cases} \frac{1}{2}(\varphi^c_{\boldsymbol{i}+\boldsymbol{e}^{d'}} - \varphi^c_{\boldsymbol{i}-\boldsymbol{e}^{d'}}) & \text{if both } \boldsymbol{i} \pm \boldsymbol{e}^{d'} \in \Omega^c_{valid} \\ \pm\frac{3}{2}(\varphi^c_{\boldsymbol{i}\pm\boldsymbol{e}^{d'}} - \varphi^c_{\boldsymbol{i}}) \mp \frac{1}{2}(\varphi^c_{\boldsymbol{i}\pm 2\boldsymbol{e}^{d'}} - \varphi^c_{\boldsymbol{i}\pm\boldsymbol{e}^{d'}}) & \text{if } \boldsymbol{i} \pm \boldsymbol{e}^{d'} \in \Omega^c_{valid},\ \boldsymbol{i} \mp \boldsymbol{e}^{d'} \notin \Omega^c_{valid} \\ 0 & \text{otherwise} \end{cases}$$

$$(D^{2,d'}\varphi)_{\boldsymbol{i}} = \begin{cases} \varphi^c_{\boldsymbol{i}+\boldsymbol{e}^{d'}} - 2\varphi^c_{\boldsymbol{i}} + \varphi^c_{\boldsymbol{i}-\boldsymbol{e}^{d'}} & \text{if both } \boldsymbol{i} \pm \boldsymbol{e}^{d'} \in \Omega^c_{valid} \\ \varphi^c_{\boldsymbol{i}} - 2\varphi^c_{\boldsymbol{i}\pm\boldsymbol{e}^{d'}} + \varphi^c_{\boldsymbol{i}\pm 2\boldsymbol{e}^{d'}} & \text{if } \boldsymbol{i} \pm \boldsymbol{e}^{d'} \in \Omega^c_{valid},\ \boldsymbol{i} \mp \boldsymbol{e}^{d'} \notin \Omega^c_{valid} \\ 0 & \text{otherwise} \end{cases}$$

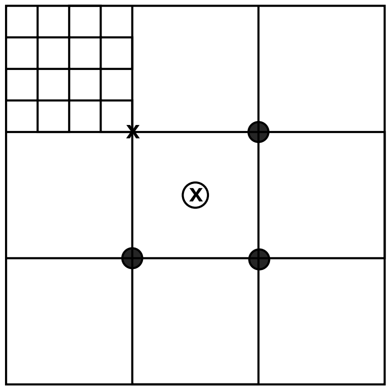

Figure 3.1: Mixed-derivative approximation illustration. The upper-left corner is covered by a finer level so the mixed derivative in the upper left (the uncircled x) has a stencil which extends into the finer level. We therefore average the mixed derivatives centered on the other corners (the filled circles) to approximate the mixed derivatives for coarse-fine interpolation in three dimensions.

In the case of $D^{d'd''}\varphi^c$, we use an average of all of the four-point difference approximations $\frac{\partial^2}{\partial x_{d'}\partial x_{d''}}$ centered at d', d'' corners adjacent to $\boldsymbol{i}$ such that all four points in the stencil are in Ω^c_{valid} (Figure 3.1)

$$(D^{d'd''}_{corner}\varphi^c)_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d'}+\frac{1}{2}\boldsymbol{e}^{d''}} = \begin{cases} \frac{1}{h^2}(\varphi_{\boldsymbol{i}+\boldsymbol{e}^{d'}+\boldsymbol{e}^{d''}} + \varphi_{\boldsymbol{i}} - \varphi_{\boldsymbol{i}+\boldsymbol{e}^{d'}} - \varphi_{\boldsymbol{i}+\boldsymbol{e}^{d''}}) & \text{if } [\boldsymbol{i}, \boldsymbol{i}+\boldsymbol{e}^{d'}+\boldsymbol{e}^{d''}] \subset \Omega^c_{valid} \\ 0 & \text{otherwise} \end{cases}$$

$$(D^{2,d'd''}\varphi^c)_{\boldsymbol{i}} = \begin{cases} \frac{1}{N_{valid}} \sum_{s'=\pm1} \sum_{s''=\pm1} (D^{d'd''}\varphi^c)_{\boldsymbol{i}+\frac{1}{2}s'\boldsymbol{e}^{d'}+\frac{1}{2}s''\boldsymbol{e}^{d''}} & \text{if } N_{valid} > 0 \\ 0 & \text{otherwise} \end{cases}$$

where N_{valid} is the number of nonzero summands. To compute (ii), we need to compute the interpolation coefficients a b, and c.

$$a = \frac{\hat{\varphi} - (r \cdot |x_d| + 2)\varphi_{\boldsymbol{i}\mp\boldsymbol{e}^d} + (r \cdot |x_d| + 1)\varphi_{\boldsymbol{i}\mp2\boldsymbol{e}^d}}{(r \cdot |x_d| + 2)(r \cdot |x_d| + 1)}$$
$$b = \varphi_{\boldsymbol{i}\mp\boldsymbol{e}^d} - \varphi_{\boldsymbol{i}\mp2\boldsymbol{e}^d} - a$$
$$c = \varphi_{\boldsymbol{i}\mp2\boldsymbol{e}^d}$$

3.4.3.2 Interpolation to corner ghost cells

We now discuss how we fill data on $\Omega^{p,f}$ the ghost cells over the coarse fine interface which cannot be reached from a single move in a coordinate direction. Define $\mathcal{D}$ to be the set of directions which have the requisite number of uncovered, single-valued cells from a corner cell $\boldsymbol{i}$. It is clear from the location of the corner which direction one needs to extrapolate from.

$$\mathcal{D}_{\boldsymbol{i}} = \{d : \kappa_{\boldsymbol{i}\pm\boldsymbol{e}} > 0 \text{ and } \kappa_{\boldsymbol{i}\pm2\boldsymbol{e}} > 0 \text{ and } \kappa_{\boldsymbol{i}\pm3\boldsymbol{e}} > 0\}$$

We also exclude from $\mathcal{D}$ any directions with a multivalued cell in the stencil. We define N_D to be the number of directions contained in $\mathcal{D}$.

$$\phi_{\boldsymbol{i}} = \frac{1}{N_D} \sum_{d\in\mathcal{D}} 3(\phi_{\boldsymbol{i}\pm\boldsymbol{e}^d} - \phi_{\boldsymbol{i}\pm2\boldsymbol{e}^d}) + \phi_{\boldsymbol{i}\pm3\boldsymbol{e}^d}$$

We must exercise some care here to ensure that our algorithm is independent of how we divide our region into rectangles. For this reason, after we do the above extrapolation, we do a cornerCopier exchange operation to fill corner cells that are covered by ghost cells of a neighboring fine grid. Finally we do an ordinary exchange operation to fill any ghost cells which are covered by the valid fine grid.

3.5 Redistribution

To preserve stability and conservation in embedded boundary calculations, we must redistribute a quantity of mass δM (the difference between stable and conservative updates) from irregular VoFs to their neighbors. This mass is normalized by h^D where h is the grid spacing on the level. We define $\eta_{\boldsymbol{v}}$ to be the set of neighbors (no farther away than the redistribution radius) which can be reached by a monotonic path. We then assign normalized weights to each of the neighbors $\boldsymbol{v}^{'}$ and divide the mass accordingly:

$$\delta M_{\boldsymbol{v}} = \sum_{\boldsymbol{v}^{'}\in\eta_{\boldsymbol{v}}} w_{\boldsymbol{v},'}\kappa_{\boldsymbol{v}'}\delta M_{\boldsymbol{v}} \tag{3.13}$$

where

$$\sum_{\boldsymbol{v}' \in \eta_{\boldsymbol{v}}} w_{\boldsymbol{v},\boldsymbol{v}'} \kappa_{\boldsymbol{v}'} = 1 \tag{3.14}$$

We then update the solution U at the neighboring cells $\boldsymbol{v}^{'}$

$$U^l_{\boldsymbol{v}'} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'} \delta M^l_{\boldsymbol{v}}. \tag{3.15}$$

This operation occurs at all $\boldsymbol{v} \in ind^{-1}(\Omega^l)$ without regard to valid or invalid regions. If the irregular cell is within the redistribution radius of a coarse-fine interface, we must account for mass that is redistributed across the interface.

3.5.1 Multilevel Redistribution Summary

We begin with $\delta M^l_{\boldsymbol{v}}, \boldsymbol{v} \in ind^{-1}\Omega^l$, the redistribution mass for level l.

Define the redistribution radius to be R^r. We define the coarsening operator to be $C_{N_{ref}}$ and the refinement operator to be $C^{-1}_{N_{ref}}$. We define the "growth" operator to be G. The operator which produces the Z^D index of a vof is ind and the operator to produces the VoFs for points in Z^D is ind^{-1}.

If $\boldsymbol{v}$ is part of the valid region, the redistribution mass is divided into three parts,

$$\begin{gathered} \delta M^l_{\boldsymbol{v}} = \delta^1 M^l_{\boldsymbol{v}} + \delta^2 M^{l,l+1}_{\boldsymbol{v}} + \delta^2 M^{l,l-1}_{\boldsymbol{v}}, \\ \boldsymbol{v} \in ind^{-1}(\Omega^{l,valid}). \end{gathered} \tag{3.16}$$

$\delta^1 M^l_{\boldsymbol{v}}$ is the part of the mass which is put onto the $\Omega^{l,valid}$. $\delta^2 M^{l,l+1}_{\boldsymbol{v}}$ is the part of the mass which is redistributed to $\Omega^l \cap C_{N_{ref}}(\Omega^{l+1})$ (the part of the level covered by the next finer level). $\delta M^{l,l-1}_{\boldsymbol{v}}$ is the part of the mass which is redistributed off level l.

If $\boldsymbol{v}$ is not part of the valid region, the redistribution mass is divided into two parts,

$$\begin{gathered} \delta M^l_{\boldsymbol{v}} = \delta^I M^l_{\boldsymbol{v}} + \delta M^{l,l}_{\boldsymbol{v}} \\ \boldsymbol{v} \in ind^{-1}(\Omega - \Omega^{l,valid}). \end{gathered} \tag{3.17}$$

$\delta^I M^l_{\boldsymbol{v}}$ is the portion of $\delta^l M^l_{\boldsymbol{v}}$ which is redistributed to other invalid VoFs of level l. $\delta^I M^P l, l\boldsymbol{v}$ is the portion of $\delta^l M^l_{\boldsymbol{v}}$ which is redistributed to valid VoFs of level l and must be removed later from the solution.

We must account for $\delta M^{l,l-1}_{\boldsymbol{v}}$, $\delta^2 M^{l,l+1}_{\boldsymbol{v}}$ and $\delta^3 M^{l,l}_{\boldsymbol{v}}$ to preserve conservation. $\delta^2 M^{l,l+1}_{\boldsymbol{v}}$ is added to the level $l+1$ solution. $\delta^2 M^{l,l-1}_{\boldsymbol{v}}$ is added to the level $l-1$ solution. $\delta^3 M^{l,l}_{\boldsymbol{v}}$ is removed from the level l solution.

3.5.2 Coarse to Fine Redistribution

The mass going from coarse to fine is accounted for as follows. Recall that the mass we store is normalized by h_c^D where h_c is the grid spacing of the level of the source. Define h_f

to be the grid spacing of the destination. For all VoFs $\boldsymbol{v_c} \in ind^{-1}(C_{N_{ref}}(G(\Omega^{l+1}, R^r) - \Omega^{l+1}))$, we define the coarse-to-fine redistribution mass $\delta^2 M^{l,l+1}$ to be

$$\begin{aligned} \delta^2 M^{l,l+1}_{\boldsymbol{v_c}} &= \sum_{\boldsymbol{v'_c} \in S(\boldsymbol{v_c})} \delta M^l_{\boldsymbol{v_c}} w_{\boldsymbol{v_c},\boldsymbol{v'_c}} \kappa_{\boldsymbol{v'_c}} \\ S(\boldsymbol{v_c}) &= \eta_{\boldsymbol{v_c}} \cap ind^{-1}(C_{N_{ref}}(\Omega^{l+1})). \end{aligned} \tag{3.18}$$

Define $\zeta^2_{\boldsymbol{v'_c}}$ to be the unnormalized mass that goes to VoF $\boldsymbol{v'_c}$. We distribute this mass to the VoFs $\boldsymbol{v'_f}$ that cover $\boldsymbol{v'_c}$ ($\boldsymbol{v'_f} \in C^{-1}_{N_{ref}}(\boldsymbol{v'_c})$) in a volume-weighted fashion.

$$\begin{aligned} \zeta^2_{\boldsymbol{v'_c}} &= h_c^D w_{\boldsymbol{v_c},\boldsymbol{v'_c}} \kappa_{\boldsymbol{v'_c}} \delta M^l_{\boldsymbol{v_c}} \\ \zeta^2_{\boldsymbol{v'_f}} &= \frac{\kappa_{\boldsymbol{v'_f}} h_f^D}{\kappa_c h_c^D} \zeta^2_{\boldsymbol{v'_c}} \\ \zeta^2_{\boldsymbol{v'_f}} &= \kappa_{\boldsymbol{v'_f}} h_f^D w_{\boldsymbol{v_c},\boldsymbol{v'_c}} \delta M^l_{\boldsymbol{v_c}} \end{aligned} \tag{3.19}$$

The change in the fine solution is the given by

$$\begin{aligned} \delta U^{l+1}_{\boldsymbol{v'_f}} &= \frac{\zeta^2_{\boldsymbol{v'_f}}}{\kappa_{\boldsymbol{v'_f}} h_f^D} = \delta M^l_{\boldsymbol{v_c}} w_{\boldsymbol{v_c},\boldsymbol{v'_c}} \\ U^{l+1}_{\boldsymbol{v'_f}} &\mathrel{+}= \delta M^l_{\boldsymbol{v_c}} w_{\boldsymbol{v_c},\boldsymbol{v'_c}} \\ \boldsymbol{v_c} &\in ind^{-1}(C_{N_{ref}}(G(\Omega^{l+1}, R^r) - \Omega^{l+1})) \\ \boldsymbol{v'_c} &= \eta_{\boldsymbol{v_c}} \cap ind^{-1}(C_{N_{ref}}(\Omega^{l+1})) \\ \boldsymbol{v'_f} &\in C^{-1}_{N_{ref}}(\boldsymbol{v'_c}) \end{aligned} \tag{3.20}$$

This can be interpreted as a piecewise-constant interpolation of the solution density.

3.5.3 Fine to Coarse Redistribution

The mass going from fine to coarse is accounted for as follows. Recall that the mass we store is normalized by h_f^D where h_f is the grid spacing of the level of the source. Define h_c to be the grid spacing of the destination. For all VoFs $\boldsymbol{v_f} \in ind^{-1}(\Omega^l - G(\Omega^l, -R^r))$, we define the fine-to-coarse redistribution mass $\delta^2 M^{l,l-1}$ to be

$$\begin{aligned} \delta^2 M^{l,l-1}_{\boldsymbol{v_f}} &= \sum_{\boldsymbol{v'_f} \in Q(\boldsymbol{v_f})} \delta M^l_{\boldsymbol{v_f}} w_{\boldsymbol{v_f},\boldsymbol{v'_f}} \kappa_{\boldsymbol{v'_f}} \\ Q(\boldsymbol{v_f}) &= \eta_{\boldsymbol{v_f}} \cap ind^{-1}(C^{-1}_{N_{ref}}(\Omega^{l-1}) - \Omega^l). \end{aligned} \tag{3.21}$$

For all VoFs $\boldsymbol{v_c} \in ind^{-1}(C_{N_{ref}}(G(\Omega^{l+1}, R^r) - \Omega^{l+1}))$, we define the coarse-to-fine redistribution mass $\delta^2 M^{l,l+1}$ to be

$$\begin{aligned} \delta^2 M^{l,l+1}_{\boldsymbol{v_c}} &= \sum_{\boldsymbol{v'_c} \in S(\boldsymbol{v_c})} \delta M^l_{\boldsymbol{v_c}} w_{\boldsymbol{v_c},\boldsymbol{v'_c}} \kappa_{\boldsymbol{v'_c}} \\ S(\boldsymbol{v_c}) &= \eta_{\boldsymbol{v_c}} \cap ind^{-1}(C_{N_{ref}}(\Omega^{l+1})). \end{aligned} \tag{3.22}$$

Define $\zeta^2_{\boldsymbol{v'_f}}$ to be the unnormalized mass that goes to VoF $\boldsymbol{v'_f}$. We distribute this mass to the VoF $\boldsymbol{v'_c} = C_{N_{ref}}(\boldsymbol{v'_f})$.

$$\zeta^2_{\boldsymbol{v'_f}} = \zeta^2_{\boldsymbol{v'_c}} = h_f^D w_{\boldsymbol{v_f},\boldsymbol{v'_f}} \kappa_{\boldsymbol{v'_f}} \delta M^l_{\boldsymbol{v_f}} \tag{3.23}$$

We define $\delta U^{l-1}_{\boldsymbol{v'_c}}$ to be the change in the coarse solution density due to $\delta^w M_{\boldsymbol{v_f},\boldsymbol{v'_f}}$:

$$\delta U^{l-1}_{\boldsymbol{v'_c}} = \frac{\zeta^2_{\boldsymbol{v'_f}}}{\kappa_{\boldsymbol{v'_c}} h_c^D} \tag{3.24}$$

Substituting from above, we increment the coarse solution as follows

$$\begin{gathered} U^{l-1}_{\boldsymbol{v'_c}} \mathrel{+}= \frac{\kappa_{\boldsymbol{v'_f}}}{\kappa_{\boldsymbol{v'_c}} N^D_{ref}} \delta M^l_{\boldsymbol{v_f}} w_{\boldsymbol{v_f},\boldsymbol{v'_f}} \\ \boldsymbol{v_f} \in ind^{-1}(\Omega^l - G(\Omega^l, -R^r)), \\ \boldsymbol{v'_f} \in \eta_{\boldsymbol{v_f}} \cap ind^{-1}(C^{-1}_{N_{ref}}(\Omega^{l-1}) - \Omega^l) \\ \boldsymbol{v'_c} = C_{N_{ref}}(\boldsymbol{v'_f}) \end{gathered} \tag{3.25}$$

3.5.4 Coarse to Coarse Redistribution

The re-redistribution algorithm proceeds as follows. Given $\boldsymbol{v} \in ind^{-1}(C_{N_{ref}}(\Omega^{l+1})$, we define the re-redistribution mass $\delta^3 Ml, l$ to be

$$\begin{gathered} \delta^3 M^{l,l}_{\boldsymbol{v}} = \sum_{\boldsymbol{v'} \in T(\boldsymbol{v})} \delta M^l_{\boldsymbol{v}} w_{\boldsymbol{v},\boldsymbol{v'}} \kappa_{\boldsymbol{v'}} \\ T(\boldsymbol{v}) = \eta_{\boldsymbol{v}} \cap ind^{-1}(\Omega^l). \end{gathered} \tag{3.26}$$

In the level redistribution step, we have added this mass to the solution density using equation 3.15. Re-redistribution is the process of removing it so that the solution is not modified by invalid regions

$$\begin{gathered} U^l_{\boldsymbol{v'}} \mathrel{-}= \delta M^l_{\boldsymbol{v}} w_{\boldsymbol{v},\boldsymbol{v'}} \\ \boldsymbol{v} \in ind^{-1}(C_{N_{ref}}(\Omega^{l+1})) \end{gathered} \tag{3.27}$$

3.6 Refluxing

First we describe the refluxing algorithm which, along with redistribution, preserves conservation at coarse-fine interfaces. The standard refluxing algorithm Given a level vector field F on Ω, we define a discrete divergence operator D as follows:

$$\begin{gathered} \kappa_{\boldsymbol{v}}(D \cdot \vec{F}) = \frac{1}{h}(\sum_{d=0}^{D-1}(\sum_{\boldsymbol{f} \in \mathcal{F}^+_d(\boldsymbol{v})} \alpha_{\boldsymbol{f}} \tilde{F}_{\boldsymbol{f}} - \sum_{\boldsymbol{f} \in \mathcal{F}^-_d(\boldsymbol{v})} \alpha_{\boldsymbol{f}} \tilde{F}_{\boldsymbol{f}}) + \alpha^B_{\boldsymbol{v}} F^B_{\boldsymbol{v}}) \\ \tilde{F}_{\boldsymbol{f}} = F_{\boldsymbol{f}} + \sum_{d: d \neq dir(\boldsymbol{f})} |x_{\boldsymbol{f},d}|(F_{\boldsymbol{f} << sign(x_{\boldsymbol{f},d})\boldsymbol{e}^d} - F_{\boldsymbol{f}}), \end{gathered} \tag{3.28}$$

where $\kappa_{\boldsymbol{v}}$ is the volume fraction of VoF $\boldsymbol{v}$ and $\alpha_{\boldsymbol{f}}$ is the area fraction of face $\boldsymbol{f}$. Equation 3.28 consists of a summation of interpolated fluxes and a boundary flux. The flux interpolation is described in [9]. Let $\vec{F}^{comp} = \{\vec{F}^f, \vec{F}^{c,valid}\}$ be a two-level composite vector field. We want to define a composite divergence $D^{comp}(\vec{F}^f, \vec{F}^{c,valid})_{\boldsymbol{v}}$, for $\boldsymbol{v} \in V^c_{valid}$. We do this by extending $F^{c,valid}$ to the faces adjacent to $\boldsymbol{v} \in V^c_{valid}$, but are covered by $\mathcal{F}^f_{valid}$.

$$\begin{gathered} < F^f_d >_{\boldsymbol{f}^c} = \frac{1}{(n_{ref})^{(\mathbf{D}-1)}} \sum_{\boldsymbol{f} \in \mathcal{C}^{-1}_{n_{ref}}(\boldsymbol{f}^c)} \alpha_{\boldsymbol{f}} F^f_d \\ \boldsymbol{f}^c \in ind^{-1}(\boldsymbol{i} + \tfrac{1}{2}\boldsymbol{e}^d), \boldsymbol{i} + \tfrac{1}{2}\boldsymbol{e}^d \in \zeta^f_{d,+} \cup \zeta^f_{d,-} \\ \zeta^f_{d,\pm} = \{\boldsymbol{i} \pm \tfrac{1}{2}\boldsymbol{e}^d : \boldsymbol{i} \pm \boldsymbol{e}^d \in \Omega^c_{valid}, \boldsymbol{i} \in \mathcal{C}_{n_{ref}}(\Omega^f)\} \end{gathered} \tag{3.29}$$

Then we can define $(D \cdot \vec{F})_{\boldsymbol{v}}, \boldsymbol{v} \in \mathcal{V}^c_{valid}$, using the expression above, with $\tilde{F}_{\boldsymbol{f}} =< F^f_d >$ on faces covered by $\mathcal{F}^f$. We can express the composite divergence in terms of a level divergence, plus a correction. We define a flux register $\delta\vec{F}^f$, associated with the fine level

$$\begin{gathered} \delta\vec{F}^f = (\delta F^f_{0,\ldots}\delta F^f_{D-1}) \\ \delta F^f_d : ind^{-1}(\zeta^f_{d,+} \cup \zeta^f_{d,-}) \rightarrow \mathbb{R}^m \end{gathered} \tag{3.30}$$

If $\vec{F}^c$ is any coarse level vector field that extends $\vec{F}^{c,valid}$, i.e. $F^c_d = F^{c,valid}_d$ on $\mathcal{F}^{c,d}_{valid}$ then for $\boldsymbol{v} \in \mathcal{V}^c_{valid}$

$$D^{comp}(\vec{F}^f, \vec{F}^{c,valid})_{\boldsymbol{v}} = (D\vec{F}^c)_{\boldsymbol{v}} + D_R(\delta\vec{F}^c)_{\boldsymbol{v}} \tag{3.31}$$

Here $\delta\vec{F}^f$ is a flux register, set to be

$$\delta F^f_d =< F^f_d > -\alpha_{\boldsymbol{f}^c} F^c_d \text{ on } ind^{-1}(\zeta^c_{d,+} \cup \zeta^c_{d,-}) \tag{3.32}$$

D_R is the reflux divergence operator. For valid coarse vofs adjacent to Ω^f it is given by

$$\kappa_{\boldsymbol{v}}(D_R\delta\vec{F}^f)_{\boldsymbol{v}} = \sum_{d=0}^{D-1}\Big(\sum_{\boldsymbol{f}:\boldsymbol{v}=\boldsymbol{v}^+(\boldsymbol{f})} \delta F^f_{d,\boldsymbol{f}} - \sum_{\boldsymbol{f}:\boldsymbol{v}=\boldsymbol{v}^-(\boldsymbol{f})} \delta F^f_{d,\boldsymbol{f}}\Big) \tag{3.33}$$

For the remaining vofs in $\mathcal{V}^f_{valid}$,

$$(D_R\delta\vec{F}^f) \equiv 0 \tag{3.34}$$

We then add the reflux divergence to adjust the coarse solution U^c to preserve conservation.

$$U^c_{\boldsymbol{v}} += \kappa_{\boldsymbol{v}}(D_R(\delta F))_{\boldsymbol{v}} \tag{3.35}$$

At coarse cells which are also irregular, this leaves unaccounted-for the quantity of mass δM^{Ref} given by

$$\delta M^{Ref} = (1 - \kappa_{\boldsymbol{v}})(D_R(\delta F))_{\boldsymbol{v}} \tag{3.36}$$

This mass must be redistributed to preserve conservation:

$$\delta M^{Ref,c}_{\boldsymbol{v}} = \sum_{\boldsymbol{v}' \in \eta_{\boldsymbol{v}} - C(\mathcal{V}^{l,valid})} \kappa_{\boldsymbol{v}'} w_{\boldsymbol{v},\boldsymbol{v}'} \delta M^{Ref,c}_{\boldsymbol{v}} \tag{3.37}$$

We increment the solution in the neighboring VoFs with their portion of δM^{Ref}:

$$\begin{gathered} U^c_{\boldsymbol{v}'} \mathrel{+}= \kappa_{\boldsymbol{v}'} w_{\boldsymbol{v},\boldsymbol{v}'} \delta M^{Ref,c}_{\boldsymbol{v}} \\ \boldsymbol{v}' \in \eta_{\boldsymbol{v}} - C(\mathcal{V}^{f,valid}) \end{gathered} \tag{3.38}$$

Time steps and other factors have been absorbed into the definition of δM. Unfortunately, we are not finished. In equation 3.38, some of the mass will be going back onto the fine grid

$$\delta M^{RR,c} \mathrel{+}= \delta M^{Ref} \sum_{\boldsymbol{v}' \in \eta_{\boldsymbol{v}} - \mathcal{V}^{c,valid}} \kappa_{\boldsymbol{v}} w_{\boldsymbol{v},\boldsymbol{v}'} \tag{3.39}$$

This mass must be accumulated at each fine time step. When the fine level has caught up with the coarse level in time, we adjust the fine solution to account for this mass:

$$\begin{gathered} U^f_{C^{-1}(\boldsymbol{v}')} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'} \delta M^{RR,c}_{\boldsymbol{v}} \\ \boldsymbol{v}' \in \eta_{\boldsymbol{v}} - \mathcal{V}^{f,valid} \end{gathered} \tag{3.40}$$

3.7 Subcycling in time with embedded boundaries

We use the subcycling-in-time algorithm specified by Berger and Oliger [2] to advance an AMR solution in time. Embedded boundary synchronization substantially complicates Berger-Oliger timestepping. Here we present an overview of Berger-Oliger subcycling in time for adaptive mesh refinement in the context of embedded boundaries. Say we are solving the hyperbolic system of equations

$$\frac{\partial U}{\partial t} + \nabla \cdot F = 0 \tag{3.41}$$

in a domain discretized as described above. Here is an outline of the Berger-Oliger algorithm for this equation. First we perform the steps required to preserve stability and conservation in the presence of embedded boundaries.

- Compute fluxes F^l on $\mathcal{F}$.
- Compute the conservative and non-conservative solution updates ($D^C(F^l)$ and $D^{NCC}(F^l)$).
- Update the solution on the level:

$$U^{new,l}_{\boldsymbol{v}} = U^{old,l}_{\boldsymbol{v}} - \Delta t(\kappa D^{NC}(F^l)_{\boldsymbol{v}} + (1-\kappa) D^C(F^l)_{\boldsymbol{v}}), \quad \boldsymbol{v} \in ind^{-1}(\Omega^l) \tag{3.42}$$

- Initialize redistribution mass δM^l to be the mass left out in the previous step.

$$\begin{gathered} \delta M^l_{\boldsymbol{v}} = \Delta t \kappa_{\boldsymbol{v}} (1-\kappa_{\boldsymbol{v}})(D^{NC}(F^l)_{\boldsymbol{v}} - D^C(F^l)_{\boldsymbol{v}}) \\ \boldsymbol{v} \in ind^{-1}\mathcal{I}^l \end{gathered} \tag{3.43}$$

- Perform level redistribution of δM^l:

$$\begin{gathered} U^{new,l}_{\boldsymbol{v}'} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'}\delta M^l_v \\ \boldsymbol{v}' \in \{\eta_{\boldsymbol{v}} \cap ind^{-1}(\Omega^l)\} \\ \sum_{\boldsymbol{v}' \in \eta_v} w_{\boldsymbol{v},\boldsymbol{v}'}\kappa_{\boldsymbol{v}'} = 1 \end{gathered} \tag{3.44}$$

Second we perform the steps required to preserve conservation across coarse-fine interfaces. We define δF to be flux registers and $\delta^2 M$ to be redistribution registers.

- We increment the flux register between this level and the next coarser level.

$$\begin{gathered} \delta F^{l,l-1}_{\boldsymbol{f}} \mathrel{+}= < F^l >_{\boldsymbol{f}} \Delta t^l \\ \boldsymbol{f} \in \partial(C(\mathcal{F}^{l-1})) \end{gathered} \tag{3.45}$$

- We initialize the flux register between this level and the next finer level.

$$\begin{gathered} \delta F^{l+1,l}_{\boldsymbol{f}} =< F^l >_{\boldsymbol{f}} \Delta t^l \\ \boldsymbol{f} \in \partial(\mathcal{F}^{l+1}) \end{gathered} \tag{3.46}$$

- Increment redistribution registers between this level and the next coarser level.

$$\delta^2 M^{l,l-1}_{\boldsymbol{v}} = \delta M^l_{\boldsymbol{v}} \boldsymbol{v} \in ind^{-1}(\mathcal{I}^l) \tag{3.47}$$

- Initialize redistribution registers with next finer level and the coarse-coarse ("re-redistribution") registers. for $\boldsymbol{v} \in ind^{-1}(\mathcal{I})^l$

$$\begin{gathered} \delta^2 M^{l,l+1}_{\boldsymbol{v}} = \delta M^l_{\boldsymbol{v}} \\ \delta^2 M^{l,l}_{\boldsymbol{v}} = -\delta M^l_{\boldsymbol{v}} \\ \delta^2 M^{l+1,l}_{\boldsymbol{v}} = 0 \end{gathered} \tag{3.48}$$

- Advance level $l+1$ solution to time $t^{new,l}$ (requires a minimum of n_{ref} time steps.

- Reflux a portion of the flux difference in equation 3.46 and save the extra mass into the appropriate redistribution register.

$$\begin{gathered} U^{new,l}_{\boldsymbol{v}} \mathrel{+}= \kappa D_R(\delta F^{l+1})_{\boldsymbol{v}} \\ \delta^2 M^{l,l+1}_{\boldsymbol{v}} \mathrel{+}= \kappa_{\boldsymbol{v}}(1-\kappa_{\boldsymbol{v}})D_R(\delta F^{l+1})_{\boldsymbol{v}} \\ \delta^3 M^{l,l}_{\boldsymbol{v}} \mathrel{+}= \kappa_{\boldsymbol{v}}(1-\kappa_{\boldsymbol{v}})D_R(\delta F^{l+1})_{\boldsymbol{v}} \end{gathered} \tag{3.49}$$

- Redistribute mass that was redistributed (in both directions) across coarse-fine interfaces.

$$\begin{gathered} U^{l+1}_{\boldsymbol{v'_f}} \mathrel{+}= \delta^2 M^{l,l+1}_{\boldsymbol{v_c}} w_{\boldsymbol{v_c},\boldsymbol{v'_c}} \\ \boldsymbol{v_c} \in ind^{-1}(C_{N_{ref}}(G(\Omega^{l+1}, R^r) - \Omega^{l+1})) \\ \boldsymbol{v'_c} = \eta_{\boldsymbol{v_c}} \cap ind^{-1}(C_{N_{ref}}(\Omega^{l+1})) \\ \boldsymbol{v'_f} \in C^{-1}_{N_{ref}}(\boldsymbol{v'_c}) \end{gathered} \tag{3.50}$$

$$\begin{gathered} U^{l-1}_{\boldsymbol{v'_c}} \mathrel{+}= \frac{\kappa_{\boldsymbol{v'_f}}}{\kappa_{\boldsymbol{v'_c}} N^D_{ref}} \delta^2 M^{l,l-1}_{\boldsymbol{v_f}} w_{\boldsymbol{v_f},\boldsymbol{v'_f}} \\ \boldsymbol{v_f} \in ind^{-1}(\Omega^l - G(\Omega^l, -R^r)), \\ \boldsymbol{v'_f} \in \eta_{\boldsymbol{v_f}} \cap ind^{-1}(C^{-1}_{N_{ref}}(\Omega^{l-1}) - \Omega^l) \\ \boldsymbol{v'_c} = C_{N_{ref}}(\boldsymbol{v'_f}) \end{gathered} \tag{3.51}$$

- Re-redistribute mass that was redistributed from invalid regions.

$$\begin{gathered} U^l_{\boldsymbol{v'}} \mathrel{-}= \delta^3 M^{l,l}_{\boldsymbol{v}} w_{\boldsymbol{v},\boldsymbol{v'}} \\ \boldsymbol{v} \in ind^{-1}(C_{N_{ref}}(\Omega^{l+1})) \end{gathered} \tag{3.52}$$

- Finally average down the finer solution where appropriate

$$U^{new,l}_{\boldsymbol{v}} = < U^{new,l+1} >, \quad \boldsymbol{v} \in ind^{-1} C_{N_{ref}}(\Omega^l + 1) \tag{3.53}$$

3.7.0.1 $O(h^3)$ Averaging

The stencil for Dirichlet EB boundary conditions on a coarse level can reach under the fine level. Because of this, we need to average ϕ from the finer level to the coarser level before evaluating $L\phi$. We use a higher-order ($O(h^3)$) averaging operator because we need a more accurate value at a coarse location than averaging the fine values which cover the coarse cell would produce. Martin and Cartwright discuss this in detail. The standard averaging operator is second order accurate and the truncation error analysis works such that to avoid making $O(1)$ errors in the Laplacian on coarse cells near the fine grid, we need a third order estimate of the solution on regions covered by finer grids. We therefore use a modified averaging operator in which we eliminate term of the truncation error of the standard averaging operator. Consider a coarse cell at $\vec{i}_c$. The coarse cell is covered by fine cells and the refinement ratio is two, the fine grid spacing is h_f and the coarse grid spacing is h_c. Suppose we have a smooth function ϕ^e which exists at all points in space. Away from coarse-fine interfaces, the Laplacian is discretized in the standard way. In two dimensions, this discretization is:

$$(L\phi)_{i,j} = \frac{1}{h^2}(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}) \tag{3.54}$$

and in three dimensions

$$(L\phi)_{i,j} = \frac{1}{h^2}(\phi_{i+1,j,k} + \phi_{i-1,j,k} + \phi_{i,j+1,k} + \phi_{i,j-1,k} + \phi_{i,j,k+1} + \phi_{i,j,k-1} - 6\phi_{i,j,k}) \tag{3.55}$$

At the coarse-fine interface, we interpolate values onto ghost cells which surround the union of rectangles that correspond to the level's domain and use equation 3.54 to calculate the Laplacian. We define the standard averaging operator A_S in two dimensions to be

$$(A_S(\phi^e))(h_c\vec{i}_c) = \frac{1}{4}\left(\begin{array}{l} \phi^e(h_f i_f, h_f j_f) + \\ \phi^e(h_f(i_f+1), h_f j_f) + \\ \phi^e(h_f i_f, h_f(j_f+1)) + \\ \phi^e(h_f(i_f+1), h_f(j_f+1)) \end{array}\right) \tag{3.56}$$

and in three dimensions to be

$$(A_S(\phi^e))(h_c\vec{i}_c) = \frac{1}{8}\left(\begin{array}{l} \phi^e(h_f i_f, h_f j_f, h_f k_f)+ \\ \phi^e(h_f(i_f+1), h_f j_f, h_f k_f)+ \\ \phi^e(h_f i_f, h_f(j_f+1,), h_f k_f)+ \\ \phi^e(h_f(i_f+1), h_f(j_f+1), h_f k_f)+ \\ \phi^e(h_f i_f, h_f j_f, h_f(k_f+1))+ \\ \phi^e(h_f(i_f+1), h_f j_f, h_f(k_f+1))+ \\ \phi^e(h_f i_f, h_f(j_f+1), h_f(k_f+1))+ \\ \phi^e(h_f(i_f+1), h_f(j_f+1), h_f(k_f+1)) \end{array}\right) \tag{3.57}$$

where $\vec{i}_f = 2\vec{i}_c$. The truncation error τ of A_S is given by

$$\tau = \phi^e(h_c\vec{i}_c) - (A_S(\phi^e))(h_c\vec{i}_c) = \frac{h_f^2}{2}\nabla^2\phi_e(h_c\vec{i}_c) + O(h_f^3) \tag{3.58}$$

Away from the embedded boundary, we define the modified averaging operator A_M to be A_S with the leading order in the truncation error subtracted off:

$$(A_M(\phi_f))_{\vec{i}_c} = A_S(\phi_f)_{\vec{i}_c} - \frac{h_f^2}{2}L(\phi_f)_{\vec{i}_c} \tag{3.59}$$

Near the embedded boundary, we extrapolate to $O(h^2)$ from fine cells to the coarse cell and average the result.

3.8 EBAMRTools User Interface

This section describes the various classes which implement the various algorithms described in the above section.

3.8.1 Classes EBCoarseAverage/EBCoarsen

The EBCoarseAverage class is used to average from finer levels on to coarser levels, or for constructing averaged residuals in multigrid iteration. It averages fine data to coarse in a volume-weighted way (see equation 3.6). This class uses copying from one layout to another for communication. This class has as data a scratch copy of the data at the coarse level. The averaging operator is blocking due to the copy. EBCoarsen does the same thing with the same interface, but averages to $O(h^3)$ and is not conservative. The important functions of the EBCoarseAverage/EBCoarsen classes are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
  ```

```
            const int& nref,
            const int& nvar);
```

Define the stencils and internal data of the class. This must be called before the average function will work.

- dblFine, dblCoar: The fine and coarse layouts of the data.
- ebislFine, ebislCoar: The fine and coarse layouts of the geometric description.
- nref: The refinement ratio between the two levels.
- nvar: The number of variables contained in the data at each VoF.

• ```
void average(LevelData<EBCellFAB>& coarData,
             const LevelData<EBCellFAB>& fineData,
             const Interval& variables);
```

Average the fine data onto the coarse data over the intersection of the coarse layout with the coarsened fine layout.

- coarData: The data over the coarse layout.
- fineData: The data over the fine layout. Fine and coarse data must have the same number of variables.
- variables: The variables to average. Those not in this range will be left alone. This range of variables must be in both the coarse and fine data.

### 3.8.2 Class EBPWLFineInterp

The EBPWLFineInterp class is used to interpolate in a piecewise-linear fashion coarse data onto fine layouts (see equation 3.7). This is primarily a useful class for regridding. It contains stencils and slopes over the coarse level and uses copy for communication. This makes its interpolate function blocking. The important functions of EBPWLFineInterp are as follows:

• ```
void define(const DisjointBoxLayout& dblFine,
            const DisjointBoxLayout& dblCoar,
            const EBISLayout& ebislFine,
            const EBISLayout& ebislCoar,
            const Box& domainCoar,
            const int& nref,
            const int& nvar);
```

Define the stencils and internal data of the class. This must be called before the interpolate function will work.

- dblFine, dblCoar: The fine and coarse layouts of the data.

  - `ebislFine, ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void interpolate(LevelData<EBCellFAB>& fineData,
                   const LevelData<EBCellFAB>& coarData,
                   const Interval& variables);
  ```

  Interpolate the fine data from the coarse data over the intersection of the fine layout with the refined coarse layout.

  - `fineData`: The data over the fine layout.
  - `coarData`: The data over the coarse layout.
  - `variables`: The variables to interpolate. Those not in this range will be left alone. This range of variables must be in both the coarse and fine data.

3.8.3 Class `EBPWLFillPatch`

Given coarse data at old and new times, during subcycling in time, we need to interpolated ghost data onto a fine data set at a time between the old and new coarse times. The `EBPWLFillPatch` class is used to interpolate fine data over the ghost region that is not covered by other fine grids. Data is simply copied from other fine grids where it is available. Only one layer of ghost cells is filled.

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the stencils and internal data of the class. This must be called before the `interpolate` function will work.

  - `dblFine, dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine, ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void interpolate(LevelData<EBCellFAB>& fineData,
                   const LevelData<EBCellFAB>& coarDataOld,
                   const LevelData<EBCellFAB>& coarDataNew,
                   const Real& coarTimeOld,
                   const Real& coarTimeNew,
                   const Real& fineTime,
                   const Interval& variables);
  ```

  Interpolate the indicated fine data variables from the coarse data on ghost cells which overlay a coarse-fine interface. Copy fine data onto ghost cells where appropriate (using `LevelData::exchange`). Only one layer of ghost cells is filled.

  - `fineData`: The data over the fine layout.
  - `coarDataOld, coarDataNew`: The data over the coarse layout at the old and new times. Fine and coarse data must have the same number of variables.
  - `coarTimeOld, coarTimeNew`: The values of the old and new time of the coarse data. The old time must be smaller than the new time.
  - `fineTime`: The time at which the fine data exists. This time must be between the old and new coarse time.

3.8.4 Class `RedistStencil`

The `RedistStencil` class holds the stencil at every irregular VoF in a layout. The default weights that the stencil holds are volume weights. The class does allow the flexibility to redefine these weights. The weights correspond to $w_{\boldsymbol{v},\boldsymbol{v}'}$ in equations 3.37 and 3.44.

-
  ```
  void define(const DisjointBoxLayout& dbl,
              const EBISLayout& ebisl,
              const Box& domain,
              const int& redistRadius);
  ```

  Define the internals of the `RedistStencil` class.

  - `dbl`: The layout of the data.
  - `ebisl`: The layout of the geometric description.
  - `domain`: The computational domain at this level of refinement.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```

  Modify the weights in the stencil by multiplying by the inputs in a normalized way.

  - `weights`: Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

-
  ```
  const BaseIVFAB<VoFStencil>&
  operator[] (const DataIndex& datInd) const
  ```

  Returns the redistribution stencil at every irregular point in input `Box` associated with this `DataIndex`.

3.8.5 Class `EBLevelRedist`

The `EBLevelRedist` class performs mass redistribution in an embedded boundary context. The algorithm for this is described in section 3.5. At irregular cells in a level described by a union of rectangles, mass to be redistributed is stored incrementally (one `Box` at a time, with a ghost width equal to the redistribution radius). `EBLevelRedist` is then used to increment a solution by the stored redistribution mass. The redistribution radius is a constant static member of the class. The important functions of `EBLevelRedist` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dbl,
              const EBISLayout& ebisl,
              const Box& domain,
              const int& nvar)
  ```

  Define the internals of the `EBLevelRedist` class. Buffers are made at every irregular cell including ghost buffers at a width of the redistribution radius. Sets values at all buffers to zero.

  - `dbl`: The layout of the data.
  - `ebisl`: The layout of the geometric description.
  - `domain`: The computational domain at this level of refinement.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```

  Modify the weights in the stencil by multiplying by the inputs in a normalized way.

  - `weights`: Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

-
  ```
  void storeMass(const BaseIVFAB<Real>& massDiff,
                 const DataIndex& datInd,
                 const Interval& variables);
  ```

Store the input mass difference in the internal buffers of the class by incrementing the buffer with the mass difference.

  - massDiff: Conserved values to store in registers.
  - datInd: The index of the Box in the input DisjointBoxLayout to which massDiff corresponds].
  - variables: The variables to store. These must fit within zero and the number of variables input to the define function.

- `void setToZero();`

  Set the internal buffer to zero.

-
  ```
  void redistribute(LevelData<EBCellFAB>& solution,
                    const Interval& variables);
  ```

  Redistribute the data contained in the internal buffers $U_{\boldsymbol{v}'} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'} \delta M_{\boldsymbol{v}}$.

  - solution: Solution to increment.
  - variables: The variables to increment.

3.8.6 Class EBFluxRegister

The EBFluxRegister class performs refluxing in an embedded boundary context. The algorithm for this is described in section 3.6. The important functions of EBFluxRegister are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the internals of the EBFluxRegister class. Buffers are made at every irregular cell including ghost buffers at a width of the redistribution radius. Sets values at all buffers to zero.

  - dblFine, dblCoar: The fine and coarse layouts of the data.
  - ebislFine, ebislCoar: The fine and coarse layouts of the geometric description.
  - nref: The refinement ratio between the two levels.
  - nvar: The number of variables contained in the data at each VoF.

- `void setToZero();`

  Set the registers to zero.

-
  ```
  void incrementCoarseRegular(
                  const EBFaceFAB& coarseFlux,
                  const Real& scale,
                  const DataIndex& coarsePatchIndex,
                  const Interval& variables,
                  const int& dir);
  void incrementCoarseIrregular(
                  const BaseIFFAB<Real>& coarseFlux,
                  const Real& scale,
                  const DataIndex& coarsePatchIndex,
                  const Interval& variables,
                  const int& dir);
  ```

  Increments the register with data from `coarseFlux`, multiplied by `scale` (α): $\delta F^f_d += \alpha F^c_d$, for all of the d-faces where the input flux (defined on a single rectangle) coincide with the d-faces on which the flux register is defined. `coarseFlux` contains fluxes in the `dir` direction for the grid `dblCoar[coarsePatchIndex]`. Only the registers corresponding to the low faces of `dblCoarse[coarsePatchIndex]` in the `dir` direction are incremented (this avoids double-counting at coarse-coarse interfaces. of the flux register.

  - `coarseFlux` : Flux to put into the flux register. This is not `const` because its box is shifted back and forth - no net change occurs.
  - `scale` : Factor by which to multiply `coarseFlux` in flux register.
  - `coarsePatchIndex` : Index which corresponds to the box in the coarse solution from which `coarseFlux` was calculated.
  - `variables` : The components to put into the flux register.
  - `dir` : Direction of the faces upon which the fluxes live.

-
  ```
  void incrementFineRegular(
                  const EBFaceFAB& fineFlux,
                  const Real& scale,
                  const DataIndex& finePatchIndex,
                  const Interval& variables,
                  const int& dir,
                  const Side::LoHiSide& sd);
  void incrementFineIrregular(
                  const BaseIFFAB<Real>& fineFlux,
                  const Real& scale,
                  const DataIndex& finePatchIndex,
                  const Interval& variables,
  ```

```
                    const int& dir,
                    const Side::LoHiSide& sd);
```

Increments the register with the average over each face of data from `fineFlux`, scaled by `scale` (α): $\delta F^f_d \mathrel{+}= \alpha < F^f_d >$, for all of the d-faces where the input flux (defined on a single rectangle) cover the d-faces on which the flux register is defined. `fineFlux` contains fluxes in the `dir` direction for the grid `dbl[finePatchIndex]`. Only the register corresponding to the direction `dir` and the side `sd` is initialized. `srcInterval` and `dstInterval` are as above.

- `fineFlux` : Flux to put into the flux register. This is not `const` because its box is shifted back and forth - no net change occurs.
- `scale` : Factor by which to multiply `fineFlux` in flux register.
- `finePatchIndex` : Index which corresponds to which box in the `LevelData<FArrayBox>` solution from which `fineFlux` was calculated.
- `variables` : The `Interval` of components of the flux register into which the flux data is put.
- `dir` : Direction of faces upon which fluxes live.
- `sd` : Side of the fine face where coarse-fine interface lies.

•
```
void reflux(LevelData<EBCellFAB>& uCoarse,
            const Interval& variables,
            const Real& scale);
```

Increments `uCoarse` with the reflux divergence of the contents of the flux register, scaled by `scale` (α): $U^c \mathrel{+}= \alpha D_R(\delta \vec{F})$.

- `uCoarse` : The solution that gets modified by refluxing.
- `variables`: gives the `Interval` of components of the flux register that correspond to the components of `uCoarse`.
- `scale` : Factor by which to scale the flux register.

•
```
void incrementRedistRegister(EBCoarToFineRedist& register,
                             const Interval& variables);
```

Increments redistribution register with left-over mass from reflux divergence as in equation 3.49: $\delta^2 M^{l,l+1}_{\boldsymbol{v}} \mathrel{+}= \kappa_{\boldsymbol{v}}(1-\kappa_{\boldsymbol{v}}) D_R(\delta F^{l+1})_{\boldsymbol{v}}$.

- `register`: Coarse to fine register that must be incremented ($\delta^2 M^{l,l+1}$).
- `variables`: Array indices to be incremented.

3.8.7 Class `EBCoarToFineRedist`

The `EBCoarToFineRedist` class stores and redistributes mass that must move from the coarse solution to the fine solution The important functions of `EBCoarToFineRedist` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the internals of the class.

  - `dblFine, dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine, ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.
  - `weightModifier`: Multiplier to stencil weights (density if you want mass weighting). If this is NULL, use volume weights.
  - `weightModVar` Variable number of weight modifier.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```

  Modify the weights in the stencil by multiplying by the inputs in a normalized way.

  - `weights:` Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

-
  ```
  void setToZero();
  ```

  Set the registers to zero.

-
  ```
  void increment(BaseIVFAB<Real>& coarMass,
                 const DataIndex& coarPatchIndex,
                 const Interval& variables);
  ```

  Increment the registers by the mass difference in coarMass as shown in the second part equation 3.49.

  - `coarMass:` The mass difference to add to the register.
  - `coarPatchIndex:` The index to the box on the coarse grid.

    - `variables`: The variables in the register to increment.

-
  ```
  void redistribute(LevelData<EBCellFAB>& fineSolution,
                    const Interval& variables);
  ```

  Redistribute the data contained in the internal buffers $U^{new,l+1}_{\boldsymbol{v}^f} {+}{=} w_{\boldsymbol{v},\boldsymbol{v}'}\delta^2 M^{l,l+1}_{\boldsymbol{v}}, \ \boldsymbol{v}^f \in C^{-1}_{nref}(\boldsymbol{v})$

    - `fineSolution`: Solution to increment.
    - variables: The variables to increment.

3.8.8 Class `EBFineToCoarRedist`

The `EBFineToCoarToRedist` class stores and redistributes mass that must go from the fine to the coarse grid. The important functions of `EBFineToCoarRedist` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the internals of the class.

    - `dblFine, dblCoar`: The fine and coarse layouts of the data.
    - `ebislFine, ebislCoar`: The fine and coarse layouts of the geometric description.
    - `nref`: The refinement ratio between the two levels.
    - `nvar`: The number of variables contained in the data at each VoF.
    - `weightModifier`: Multiplier to stencil weights (density if you want mass weighting). If this is NULL, use volume weights.
    - `weightModVar` Variable number of weight modifier.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```

  Modify the weights in the stencil by multiplying by the inputs in a normalized way.

    - `weights`: Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

-
  ```
  void setToZero();
  ```

  Set the registers to zero.

-
  ```
  void increment(BaseIVFAB<Real>& fineMass,
                 const DataIndex& finePatchIndex,
                 const Interval& variables);
  ```

  Increment the registers by the mass difference in fineMass as shown in equation 3.49.

  - `fineMass`: The mass difference to add to the register.
  - `finePatchIndex`: The index to the box on the fine grid.
  - `variables`: The variables in the register to increment.

-
  ```
  void redistribute(LevelData<EBCellFAB>& coarSolution,
                    const Interval& variables);
  ```

  Redistribute the data contained in the internal buffers $U_{\boldsymbol{v}'}^{new,l} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'}^{fc}\delta^2 M_{\boldsymbol{v}}^{l+1,l}$

  - `fineSolution`: Solution to increment.
  - variables: The variables to increment.

3.8.9 Class `EBCoarToCoarRedist`

The `EBCoarToCoarToRedist` class stores and redistributes mass that was redistributed to the coarse grid that is covered by the fine grid and now must be corrected. This is the notorious "re-redistribution" process. The important functions of `EBCoarToCoarRedist` are as follows:

-
  ```
  void define(const DisjointBoxLayout& dblFine,
              const DisjointBoxLayout& dblCoar,
              const EBISLayout& ebislFine,
              const EBISLayout& ebislCoar,
              const Box& domainCoar,
              const int& nref,
              const int& nvar);
  ```

  Define the internals of the class.

  - `dblFine, dblCoar`: The fine and coarse layouts of the data.
  - `ebislFine, ebislCoar`: The fine and coarse layouts of the geometric description.
  - `nref`: The refinement ratio between the two levels.
  - `nvar`: The number of variables contained in the data at each VoF.

-
  ```
  void resetWeights(const LevelData<EBCellFAB>& modifier,
                    const int& ivar)
  ```

  Modify the weights in the stencil by multiplying by the inputs in a normalized way.

- `weights`: Relative weights at each VoF in the stencil. For instance, if one were to want to set the weighting to be mass weighting then `modifier(v, ivar)` would contain the density at VoF `v`.

-
  ```
  void setToZero();
  ```
  Set the registers to zero.

-
  ```
  void increment(BaseIVFAB<Real>& coarMass,
                 const DataIndex& finePatchIndex,
                 const Interval& variables);
  ```
  Increment the registers by the mass difference in coarMass as shown in equation 3.49.
  - `coarMass`: The mass difference to add to the register.
  - `coarPatchIndex`: The index to the box on the fine grid.
  - `variables`: The variables in the register to increment.

-
  ```
  void redistribute(LevelData<EBCellFAB>& coarSolution,
                    const Interval& variables);
  ```
  Redistribute the data contained in the internal buffers $U^{new,l}_{\boldsymbol{v}'} \mathrel{+}= w_{\boldsymbol{v},\boldsymbol{v}'}\delta^2 M^{l,l}_{\boldsymbol{v}}$
  - `coarSolution`: Solution to increment.
  - variables: The variables to increment.

3.8.10 Class `EBQuadCFInterp`

This class interpolates to ghost cells over the coarse-fine interface with $O(h^3)$ error.

-
  ```
  EBQuadCFInterp(const DisjointBoxLayout&         a_dblFine,
                 const DisjointBoxLayout&         a_dblCoar,
                 const EBISLayout&                a_ebislFine,
                 const EBISLayout&                a_ebislCoar,
                 const ProblemDomain&             a_domainCoar,
                 const int&                       a_nref,
                 const int&                       a_nvar,
                 const LayoutData<IntVectSet>&    a_cfivs);
  ```
  Define the interpolation object.

-
  ```
  void
  interpolate(LevelData<EBCellFAB>&       a_fineData,
              const LevelData<EBCellFAB>& a_coarData,
              const Interval&             a_variables);
  ```
  Interpolate to the ghost cells of the fine data to $O(h^3)$.

Chapter 4

Layer 4—EBAMRElliptic: Tools for Elliptic problems

4.1 Introduction

This document is to briefly explain the workings of the EBAMRElliptic implementation. First, the algorithm is explained. Here we extend the AMR multigrid algorithm of Martin and Cartwright [11] to embedded boundaries using the results of Johansen and Colella [9], [10]. Second, we document the implementation of the EBAMRElliptic software in detail, trying wherever possible to draw the connections from the software to the algorithm specification. We shall describe the algorithm in both two and three dimensions.

We are going to present two examples. For both, we are given a bounded domain Ω and a charge density distribution ρ which exists over all of Ω and we are given a boundary condition for the solution is given on $\partial\Omega$. We discretize the domain with a block-structured, adaptive mesh.

In the first example, we solve Poisson's equation

$$\nabla \cdot (\nabla \phi) = \rho \tag{4.1}$$

for ϕ. In the second example, we present

4.2 Poisson's equation

This section describes the method for solving the elliptic partial differential equation

$$L(\phi(\vec{x})) = \rho(\vec{x}) \tag{4.2}$$

on a Cartesian grid embedded boundary mesh, for the special case of Poisson's equation, in which

$$L(\phi(\vec{x})) = \nabla^2 \phi(\vec{x}) \tag{4.3}$$

is the Laplacian. This algorithm is largely an extension of that developed by Johansen and Colella [9] combined with the AMR multigrid algorithm of Martin and Cartwright [11].

4.2.1 Operator Discretization

4.2.1.1 Notation

To suppress the use (i, j, k) notation, we define: $v^+(f)$ to be the VoF on the high side of face f; $v^-(f)$ to be the VoF on the low side of face f; $f_d^+(v)$ to be the set of faces on the high side of VoF v; $f_d^-(v)$ to be the set of faces on the low side of VoF v, where $d \in \{x, y, z\}$ is a coordinate direction (the number of directions is $\mathbf{D}$). Also, we compose these operators to obtain the set of VoFs directly connected to a given VoF: $v_d^+(v) = v^+(f_d^+(v))$ and $v_d^-(v) = v^-(f_d^-(v))$.

Barred variables, such as $\bar{x}_v$ or $\bar{x}_f$, are distances from the center of the grid cell containing v or of the grid face containing f, respectively, that have been normalized by the grid spacing h. Typically, $-\frac{1}{2} \leq \bar{(\cdot)} \leq \frac{1}{2}$.

4.2.1.2 Interior Method

The Laplacian of ϕ is defined in three stages: compute the grid-centered gradient of ϕ, recenter the gradient, and compute the divergence of recentered gradient. The face-centered gradient of ϕ is defined as

$$\widetilde{g}_f^d = \frac{1}{h}\left(\phi_{v^+(f)} - \phi_{v^-(f)}\right) \tag{4.4}$$

The gradients at the irregular face centroids g_f^d are computed by interpolation using a modification of the Johansen-Colella method. Interpolation is done in the $\mathbf{D} - 1$ dimensional, linear subspace which contains the irregular face. In 2D, this is a line and, in 3D, this is a plane. If possible, multilinear interpolation is done using the face-centered gradients whose locations bound the centroid of the irregular face. Multilinear interpolation is possible if all the face-centered gradients needed can be used (see below for the definition of "can be used" in this context). If multilinear interpolation is not possible then the $\widetilde{g}_f^d$ is used at the irregular face centroid, i.e., piecewise constant interpolation.

By the divergence theorem, the integral of the Laplacian of ϕ over a VoF is equal to the integral around the boundary of the VoF of the gradient of ϕ. Discretizing the integral with the midpoint rule yields the approximation

$$L_v(\phi) = \frac{1}{\kappa_v h}\left(\sum_{f \in f_d^+(v)} \alpha_f g_f^d - \sum_{f \in f_d^-(v)} \alpha_f g_f^d \; - \; \alpha_v^{EB}\left(\vec{g}_v^{EB} \cdot \widehat{n}_v^{EB}\right)\right) \tag{4.5}$$

where κ_v is the volume fraction of a VoF v, α_f is the area fraction of face f, and α_v^{EB} is the area fraction of the embedded boundary of the VoF. The superscript EB, in general, refers to quantities associated with the segment of the embedded boundary within a VoF. The calculation of $(\vec{g}_v^B \cdot \widehat{n}_v^B)$, the normal gradient of ϕ at the boundary, is described in section 4.2.1.3. In regions of the grid where all VoFs and faces involved are regular, no recentering of the gradient is required and there is no contribution from the embedded

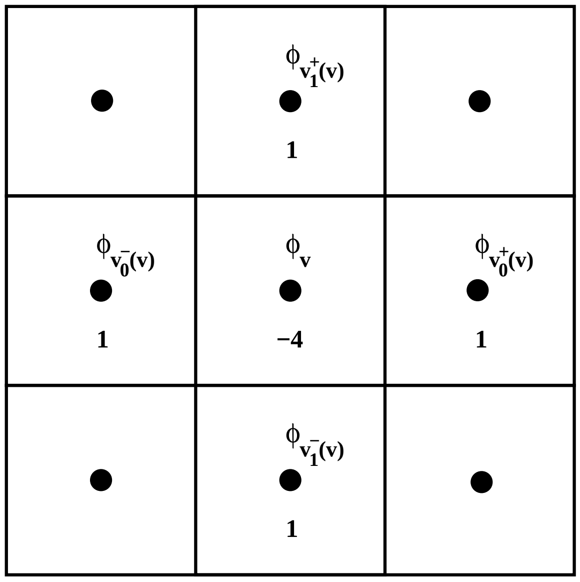

Figure 4.1: Illustration of the 5-point Laplacian stencil in two dimensions.

boundary. In this case equation 4.4 gives the gradient at the VoF face and this method reduces to the familiar star-shaped direction-split stencil for the Laplacian:

$$L_v(\phi) = \frac{1}{h^2}\left(\sum_{d=0}^{\mathbf{D}-1} \phi_{v_d^+(v)} - 2\phi_v + \phi_{v_d^-(v)}\right). \tag{4.6}$$

See figure 4.1 for a graphical version of the stencil in two dimensions.

Now we define when a face-centered gradient "can be used" in the context of computing the gradient at the centroid of an irregular face. For each direction $d' \neq d$, we define two sets of VoFs,

$$\begin{aligned} v_{d'}^- &= v^\pm(f_{d'}^\pm((v^-(f))) \\ v_{d'}^+ &= v^\pm(f_{d'}^\pm((v^+(f))) \end{aligned} \tag{4.7}$$

where the choice of sign is the sign of $\bar{x}_f^{d'}$, the normalized centroid of f in the d' direction. Basically, we take the VoF on each side of f (in the d direction), find all faces connected to that VoF in the low or high, $\pm$, d' direction, and then collect all the VoFs connected to the other side of these faces.

Now, construct the set of faces that are shared by a VoF in $v_{d'}^-$ and a VoF in $v_{d'}^+$. If there is one such face, it is $f'(d')$. If there are no faces or more than one face then $g_f^d = \widetilde{g}_f^d$, i.e., drop order.

4.2.1.3 Boundary Conditions

There are two distinct type of boundaries: faces which lie on the boundary of the solution domain, and embedded boundary segments which are contained within a VoF. See fig-

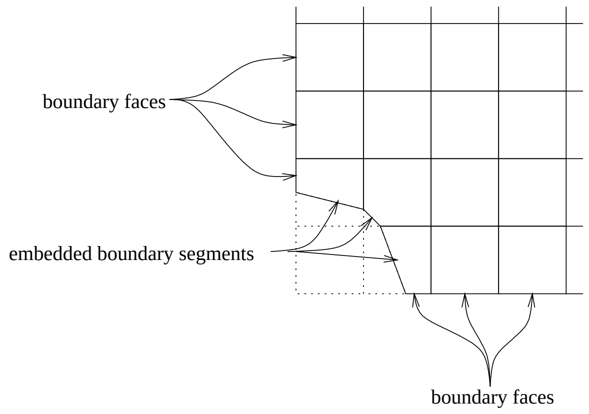

Figure 4.2: Boundary faces and embedded boundary segments.

ure 4.2. Discretization of homogeneous Dirichlet and Neumann boundary conditions are described for each type of boundary face. Homogeneous Neumann boundary conditions are defined by setting the appropriate gradient to zero. Homogeneous Dirichlet boundary conditions are more involved.

4.2.1.4 Homogeneous Dirichlet Boundary Condition at Faces

For a boundary face f normal to d, the normal gradient depends on whether the solution domain is on the high side $(+)$ of f or on the low side $(-)$ of f. The gradient is

$$\widetilde{g}_f^d = \pm\frac{1}{h}\left(3\phi_v - \phi_1/3\right) \tag{4.8}$$

where

$$v = v^{\pm}(f) \tag{4.9}$$

is the first VoF in the solution domain, and

$$\phi_1 = \frac{\displaystyle\sum_{f' \in f_d^{\pm}(v)} \ell_{f'}\phi_{v^{\pm}(f')}}{\displaystyle\sum_{f' \in f_d^{\pm}(v)} \ell_{f'}} \tag{4.10}$$

is the face-area-averaged value of the solution in the set of VoFs in the second cell inward in the solution domain that are directly connected to the VoF v. An example is shown in figure 4.3. In the figure, the solution domain is on the high side of the face f. The set of VoFs $v^+(f')$ is shaded gray. Note that the crosshatched VoF is in the same cell as $v^+(f')$, but does not participate in the average.

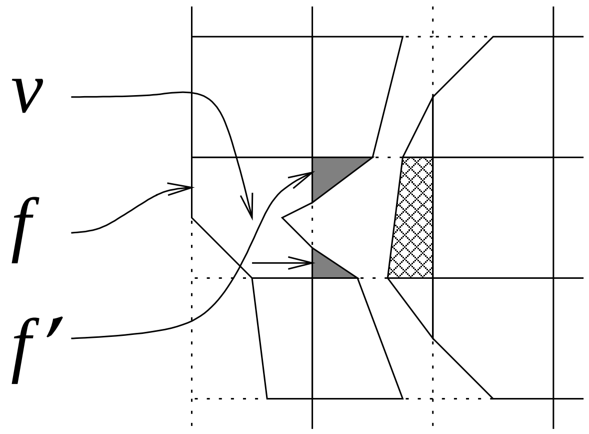

Figure 4.3: VoFs and faces for Dirichlet boundary condition at a boundary face.

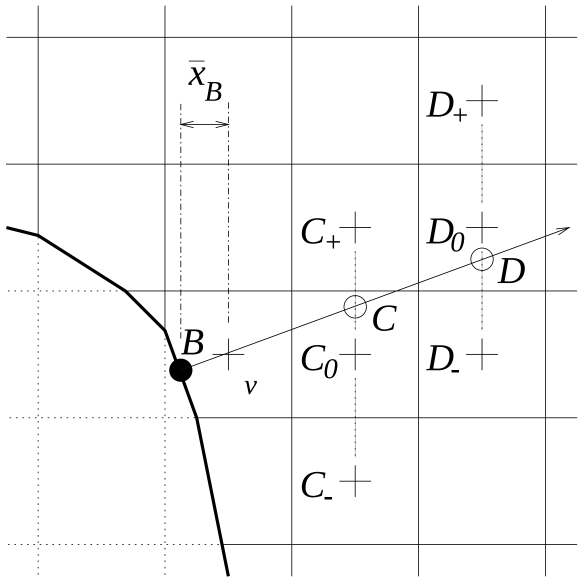

Figure 4.4: Dirichlet boundary condition at embedded boundary segment. Stencil for quadratic interpolation. Interpolate values from cell centers (+) to intersection points (○). Gradient at boundary segment centroid (●) is found by differencing values at the ○s.

4.2.1.5 Homogeneous Dirichlet Boundary Condition at Embedded Boundary Segments

For an embedded boundary segment, the gradient normal to the segment is found by casting a ray normal to the segment into the solution domain. See figure 4.4. The ray $\overrightarrow{BCD}$ is cast from the centroid of the embedded boundary face B in VoF v. Note that, in this example, $\widehat{n}^B_{v,x} \geq \widehat{n}^B_{v,y} \geq 0$. If this inequality does not hold, the problem is transformed into a different coordinate system in which it does hold by means of coordinate swaps and reflections. The direction that transforms to x is called the *major coordinate*. Unless otherwise specified, the rest of this discussion is in terms of the transformed coordinate system.

Planes are constructed normal to x through the centers of cells near v. We then find the intersection of the ray with the two planes that are closest to but do not intersect v. In the figure, the intersection points are C, the closer point, and D, the further point. We will first describe the method assuming all the cells necessary are regular except this one containing the embedded boundary. We locate the centers of the cells these intersections are within, C_0 and D_0, and the centers of the neighbor cells of each in the same intersection plane, C_+ and C_-, and D_+ and D_-, respectively (in three dimensions, there are eight neighbors each, C_{++}, C_{+0}, C_{+-}, *etc.*). Values at C and D are found by quadratic interpolation. In two dimensions,

$$\phi_C = N_+(\bar{y}_C)\phi_{C_+} + N_0(\bar{y}_C)\phi_{C_0} + N_-(\bar{y}_C)\phi_{C_-} \tag{4.11}$$

and similarly for ϕ_D, where

$$\bar{y}_C h = y_C - y_{C_0} \tag{4.12}$$

and the interpolation functions are

$$\begin{aligned} N_+(\xi) &= \tfrac{1}{2}\xi\,(\xi+1) \\ N_0(\xi) &= 1-\xi^2 \\ N_-(\xi) &= \tfrac{1}{2}\xi\,(\xi-1)\,. \end{aligned} \tag{4.13}$$

If the major coordinate were y, then y would be replaced by x in equation 4.11.

With the values ϕ_C and ϕ_D known, the gradient at B normal to the embedded boundary segment is

$$\left(\vec{g}^B_v \cdot \widehat{n}^B_v\right) = \frac{n^B_{v,x}}{h}\left(\frac{2-\bar{x}^B}{1-\bar{x}^B}\,\phi_C \;-\; \frac{1-\bar{x}^B}{2-\bar{x}^B}\,\phi_D\right) \tag{4.14}$$

where $\bar{x}^B h = x^B - i^B h$ is the x-component of the distance from the centroid of the embedded boundary face B to the center of the cell it is in. The terms $n^B_{v,x}$ and $\bar{x}^B$ are associated with the major coordinate.

Note that the value of the solution at a VoF v does not affect the value of the gradient at B, and therefore does not contribute to the Laplacian at v via this boundary condition.

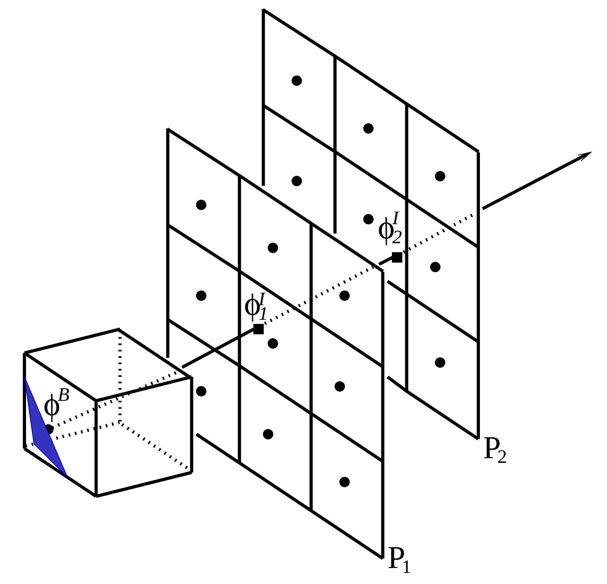

Figure 4.5: Quadratic approximation of C in three dimensions. We interpolate in each plane then interpolate along ray.

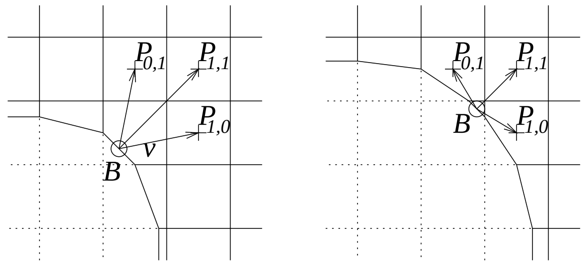

Figure 4.6: Dirichlet boundary condition at embedded boundary segment. Stencil for least-squares fit. Left:: typical situation. Right:: nearly degenerate situation.

The method just described can be extended to the case where the cells are not all regular. Define VoF_0 to be the VoF containing the current embedded boundary. Let VoF_1 be the set of VoFs connected to VoF_0 via the face the normal first intersects (what is done if the normal intersects an edge or corner is described below). Note, all the VoFs, VoF_1, will lie in one cell, $Cell_1$. Continuing in this fashion, let VoF_2 be the set of all the VoFs connected to a VoF in VoF_1 via the second face the normal intersects and they will all lie in $Cell_2$. Observe that one of the $Cell_i$ will be the cell that C lies in, C_0, and one will be the cell that D lies in, D_0. Call these cells $Cell_C$ and $Cell_D$, respectively. Now, the set of VoFs corresponding to these cells, VoF_C and VoF_D, may or may not be empty. If VoF_C is empty then we approximate the gradient at B normal to the embedded boundary by 0. If VoF_C is not empty but VoF_D is empty we will use linear interpolation to compute the value a C

If the cells containing C and D, described above, are not regular or are part of a coarse-fine interface, a different method is used to approximate the normal gradient. The gradient also can be found by a least-squares minimization method. See figure 4.6. Note that the components of the normal in this example are positive. If they are not, the problem is transformed via coordinate swaps into a coordinate system in which they are. We wish to find the normal gradient at B, the centroid of an embedded boundary face, in a VoF v. The details depend on the dimensionality of the problem. The two-dimensional case is described first.

In two dimensions, we select three cells adjacent to v's cell, two directly adjacent and one diagonally adjacent. The centers of these cells are the points $P_{1,0}$, $P_{0,1}$ and $P_{1,1}$. Note that the points $P_{1,0}$ *etc.* are always the centers of the cells, even if a cell is irregular. We then do a least-squares fit on the gradients in the directions $BP_{0,1}$, $BP_{1,0}$ and $BP_{1,1}$ (which are known) to determine the components of the full vector gradient (ϕ_x^B, ϕ_y^B).

Figure 4.6 also shows the need for a least-squares fit, rather than a coordinate transformation. On the left of the figure is the situation if the volume of v is not small. In this situation, the gradients in the directions $BP_{1,0}$ and $BP_{0,1}$ are linearly independent, so it is possible to compute the full gradient from them alone. On the right is the situation in the degenerate case in which the volume of v is small (and the normal $\widehat{n}_v^B$ is not aligned with the grid). The point B is almost at the corner of the grid cell, directly between $P_{1,0}$ and $P_{0,1}$. Thus, the gradients in the directions $BP_{1,0}$ and $BP_{0,1}$ are not linearly independent, and a full gradient cannot be computed from them alone.

The overdetermined system we need to approximate is

$$\begin{aligned}
\phi_{1,0} - \phi^B &= \left(x_{1,0} - x^B\right)\phi_x^B + \left(y_{1,0} - y^B\right)\phi_y^B \\
\phi_{0,1} - \phi^B &= \left(x_{0,1} - x^B\right)\phi_x^B + \left(y_{0,1} - y^B\right)\phi_y^B \\
\phi_{1,1} - \phi^B &= \left(x_{1,1} - x^B\right)\phi_x^B + \left(y_{1,1} - y^B\right)\phi_y^B
\end{aligned} \tag{4.15}$$

or, with $\bar{x}h = x - i^B h$, and replacing $\bar{x}_{1,0}$ *etc.* with the actual values (which are always either unity or zero because $P_{1,0}$ *etc.* are cell centers),

$$\mathbf{A}\, g = \Delta\Phi \tag{4.16}$$

where

$$\Delta\Phi = \left\{ \begin{array}{c} \phi_{1,0} - \phi^B \\ \phi_{0,1} - \phi^B \\ \phi_{1,1} - \phi^B \end{array} \right\}, \quad \mathbf{A} = \left[\begin{array}{cc} 1 - \bar{x}^B & -\bar{y}^B \\ -\bar{x}^B & 1 - \bar{y}^B \\ 1 - \bar{x}^B & 1 - \bar{y}^B \end{array} \right] h, \quad g = \left\{ \begin{array}{c} \phi_x^B \\ \phi_y^B \end{array} \right\}. \tag{4.17}$$

Note that $\phi^B = 0$. The least-squares approximation to equation 4.16 is the solution g to

$$\mathbf{M}\, g = b \tag{4.18}$$

where

$$\mathbf{M} = \mathbf{A}^{\mathbf{T}}\mathbf{A}, \quad b = \mathbf{A}^{\mathbf{T}}\Delta\Phi. \tag{4.19}$$

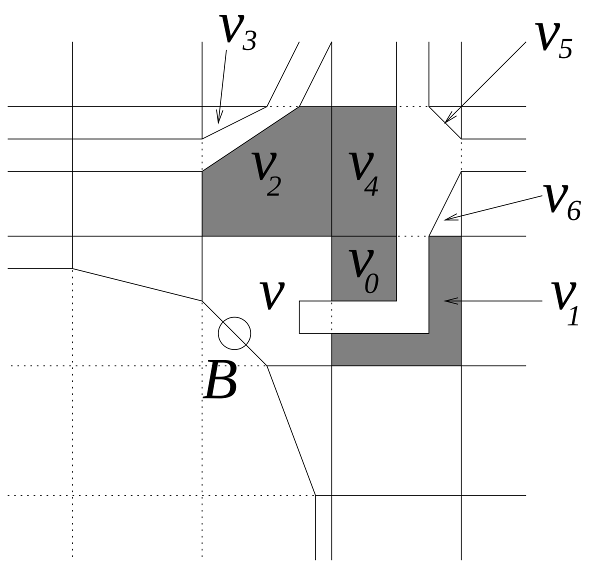

Figure 4.7: VoFs in the least-squares stencil. The VoFs v_0, v_1, v_2 and v_4 are in the stencil for the gradient at B.

The normal gradient is then

$$\left(\vec{g}_v^B \cdot \widehat{n}_v^B\right) \;=\; \widehat{n}_v^B \cdot g \;=\; n_{v,x}^B \phi_x^B \;+\; n_{v,y}^B \phi_y^B \tag{4.20}$$

If any of the cells containing a point $P_{1,0}$ *etc.* are irregular and contain multiple VoFs, we use for the value ϕ at that cell's center the volume-weighted average of the values of ϕ in the set of VoFs in that cell which are correctly connected to the VoF v. Note that the appropriate set of VoFs for the diagonal neighbor $P_{1,1}$ is the set of VoFs in that cell that are connected to the appropriate VoFs in both the cells of $P_{1,0}$ and $P_{0,1}$. See figure 4.7. In the figure, the VoFs v_0, v_1 and v_2 are in the stencil because they are directly connected to v. The VoF v_3 is in an adjacent cell, but is not connected to v. We call the set of VoFs $\{v_0, v_1\}$ the *x-neighbors* of v and the set of VoFs $\{v_2\}$ the *y-neighbors* of v. Of the VoFs in the diagonally adjacent cell, the VoF v_4 is in the stencil because it is connected both to an x-neighbor of v (namely v_0), and to a y-neighbor of v (namely v_2); the VoF v_5 is excluded because it is connected neither to an x-neighbor nor a y-neighbor of v; and the VoF v_6 is excluded because, although it is connected to an x-neighbor of v (namely v_1), is is not connected to any y-neighbor.

In three dimensions, we need four equations to form an overdetermined system for three components of the full vector gradient. We use the directional gradients from B to each of four cell centers. The cells are the three directly adjacent cells and one cell that is diagonally adjacent in the plane through v's cell that is normal to the direction in which the component of the normal $\widehat{n}_v^B$ is the least. See figure 4.8. In the figure, $n_{v,x}^B > n_{v,y}^B > n_{v,z}^B$, so that the fourth point is $P_{1,1,0}$, the center of the cell in the same xy-plane as the center of v's cell. We now have

$$\mathbf{A}\, g = \Delta\Phi \tag{4.21}$$

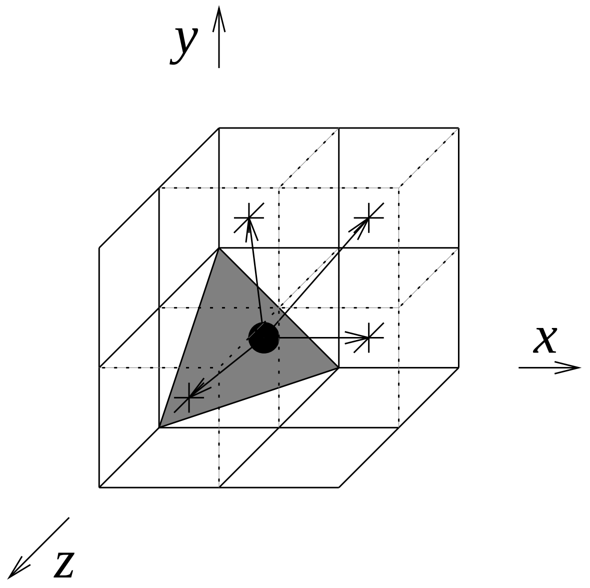

Figure 4.8: Least-squares stencil for three dimensions. The centroid of the embedded boundary face is the •. The centers of the neighbor cells for the least-squares approximation are the $+$s.

where

$$\Delta\Phi = \left\{ \begin{array}{c} \phi_{1,0,0} - \phi^B \\ \phi_{0,1,0} - \phi^B \\ \phi_{0,0,1} - \phi^B \\ \phi_{1,1,0} - \phi^B \end{array} \right\}, \quad \mathbf{A} = \left[\begin{array}{ccc} 1-\bar{x}^B & -\bar{y}^B & -\bar{z}^B \\ -\bar{x}^B & 1-\bar{y}^B & -\bar{z}^B \\ -\bar{x}^B & -\bar{y}^B & 1-\bar{z}^B \\ 1-\bar{x}^B & 1-\bar{y}^B & -\bar{z}^B \end{array} \right] h, \quad g = \left\{ \begin{array}{c} \phi^B_x \\ \phi^B_y \\ \phi^B_z \end{array} \right\} \tag{4.22}$$

which is approximated similarly to the two-dimensional case.

Again, the value of the solution at a VoF v does not affect the value of the gradient at B, and therefore does not contribute to the Laplacian at v via this boundary condition.

4.2.2 Relaxation Method

The relaxation method is based on Gauss-Seidel iteration with red-black ordering (GSRB) []. The VoFs are divided into three sets: irregular VoFs, and two sets of regular VoFs, red and black, such that every black VoF is adjacent only to red and irregular VoFs, and every red VoF is adjacent only to black and irregular VoFs. One iteration consists of the following steps:

- update the solution at the black VoFs,
- update the solution at the red VoFs,
- update the solution at the irregular VoFs N_{irreg} times, where $N_{\text{irreg}} > 0$ is an adjustable parameter.

The solution at a VoF is updated by incrementing it with

$$\delta\phi_v = \alpha_v \left(\rho_v - L_v\left(\phi\right)\right) \tag{4.23}$$

where α_v is a relaxation coefficient designed to annihilate the diagonal terms of the differential operator $L_v(\phi)$. It is constructed by applying the operator to a delta function and taking the inverse,

$$\frac{1}{\alpha_v} = L_v\left(\delta_v\right) \tag{4.24}$$

where

$$\delta_v\left(\vec{x}\right) = \begin{cases} 1 & \text{if } \vec{x} \text{ is within } v \\ 0 & \text{otherwise} \end{cases} \tag{4.25}$$

The operator L_v must include the face boundary conditions (see section 4.2.1.4), but does not require the embedded boundary segment boundary conditions (see section 4.2.1.5) because in the latter the contribution to the operator at the VoF v does not depend on the value of the solution at v.

4.2.3 Multigrid Algorithm

Multigrid is a method for the acceleration of convergence of iterative schemes for elliptic and parabolic equations. It involves the creation of a sequence of coarser grids on which a coarser problem is solved. A procedure is also specified for transferring solution data between grids of different resolution. For solving a linear problem, we use the residual-correction form of multigrid. One multigrid cycle consists of the following sequence of steps:

- perform N_{pre} iterations of the relaxation procedure
- restrict residual from this grid to the next coarser grid:

$$\rho^{2h} = I_h^{2h}\left(\rho^h - L^h(\phi^h)\right) \tag{4.26}$$

- perform N_{cycles} multigrid cycles on the coarser grid to solve

$$L^{2h}(\phi^{2h}) = \rho^{2h} \tag{4.27}$$

- interpolate correction from the next coarser grid to this grid:

$$e^h = I_{2h}^h\left(\phi^{2h}\right) \tag{4.28}$$

- increment solution on this grid with the correction e^h
- perform N_{post} iterations of the relaxation procedure

If $N_{cycles} = 1$, the method is called a V-cycle; if $N_{cycles} = 2$, the method is called a W-cycle; other values of N_{cycle} are unusual.

Something about bottom solves.

We use the sequence of coarse grids produced by `EBIndexSpace`'s coarsening algorithm. A grid has half the resolution of the finer grid from which it was created, and the volumes of VoFs and areas of faces are "conserved,"

$$\begin{aligned} \Lambda_v &= \frac{1}{2^D} \sum_{v' \in \text{refine}(v)} \Lambda_{v'} \\ \ell_f &= \frac{1}{2^{D-1}} \sum_{f' \in \text{refine}(f)} \Lambda_{f'}. \end{aligned} \tag{4.29}$$

Data is transferred to a coarser grid by a volume-weighted average restriction operator I_h^{2h}, defined as

$$\phi_v = \frac{1}{2^D \Lambda_v} \sum_{v' \in \text{refine}(v)} \Lambda_{v'} \phi_{v'}. \tag{4.30}$$

Data is transferred to a finer grid by piecewise-constant interpolation operator I_{2h}^h.

Because we are using the residual-correction form of multigrid, all boundary conditions on the coarser grids are homogeneous.

4.2.4 Viscous operator discretization

In this section, we define a Helmholtz operator and how we solve it. We are solving

$$(I + \mu L)\phi = rho \tag{4.31}$$

where μ is a constant. Just as we did for the MAC projection, we discretize $L \equiv DG^{mac}$ (see equation **??**). In this context, however, since the irregular boundary is a no-slip boundary, we must solve (4.31) with Dirichlet boundary conditions $\phi = 0$ on the irregular boundary. To do this, we must compute

$$F^B = \frac{\partial \phi}{\partial \hat{n}}$$

at the embedded boundary. We follow Schwartz, et. al [14] and compute this gradient by casting a ray into space, interpolating ϕ to points along the ray, and computing the normal gradient of phi by differencing the result. We cast a ray along the normal of the VoF from the centroid of area of the irregular face C. We find the closest points B and C where the ray intersects the planes formed by cell centered points. The axes of these planes d_1, d_2 will be the directions not equal to the largest direction of the normal. We use biquadratic interpolation to interpolate data from the nearest cell centers to the intersection points B and C. In two dimensions, we find the nearest lines of cell centers (instead of planes)

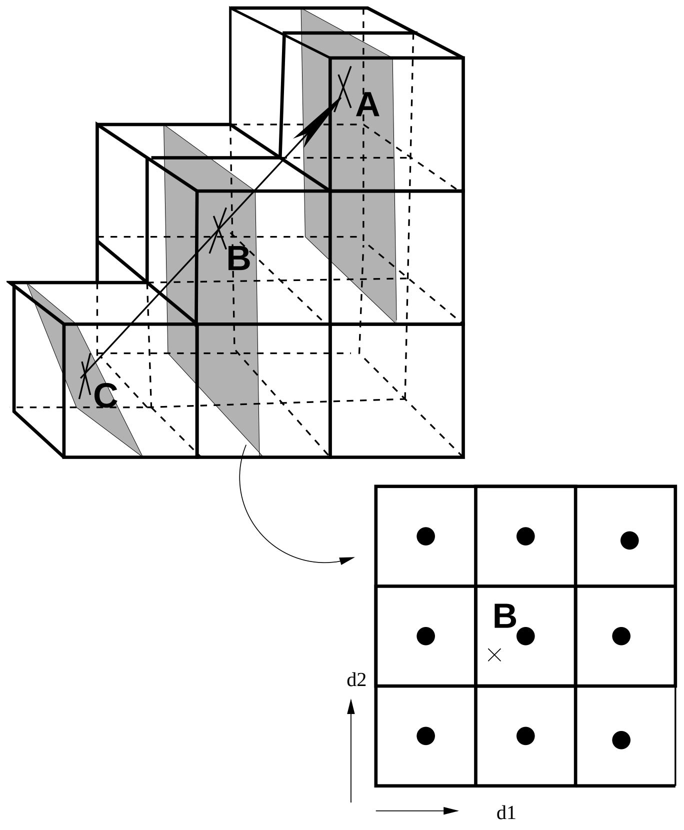

Figure 4.9: Ray casting to get fluxes for Dirichlet boundary conditions at the irregular boundary. A ray is cast along the normal from the centroid of the irregular area C and the points A and B are the places where this ray intersects the planes formed by cell centers. Data is interpolated in these planes to get values at the intersection points. That data is used to compute a normal gradient of the solution.
.

and the interpolation is quadratic. We then use this interpolated data to compute a $O(h^2$ approximation of $\frac{\partial \phi}{\partial \hat{n}}$. In the case where there are not enough cells to cast this ray, we use a least-squares approximation to $\frac{\partial \phi}{\partial \hat{n}}$ which is $O(h)$. As shown in [8], the modified equation analysis shows that, for Dirichlet boundary conditions, it is sufficient to have $O(1)$ boundary conditions to achieve second order solution error convergence for elliptic equations.

4.2.5 Slope Calculation

The notation

$$CC = A \mid B \mid C$$

means that the 3-point formula A is used for CC if all cell-centered values it uses are available, the 2-point formula B is used if the cell to the right (i.e. the high side) of the current cell is covered, and the 2-point formula C is used if the cell to the left (i.e. the low side) current cell is covered.

To compute the limited differences in the first step on the algorithm, we use the second-order slope calculation [1] with van Leer limiting.

$$\begin{aligned}
\Delta_2^d W_{\boldsymbol{i}} &= \Delta^{vL}(\Delta^C W_{\boldsymbol{i}}, \Delta^L W_{\boldsymbol{i}}, \Delta^R W_{\boldsymbol{i}}) \mid \Delta^{VLL} W_{\boldsymbol{i}} \mid \Delta^{VLR} W_{\boldsymbol{i}} \\
\Delta^B W_{\boldsymbol{i}} &= \tfrac{2}{3}((W - \tfrac{1}{4}\Delta_2^d W)_{\boldsymbol{i}+\boldsymbol{e}^d} - (W + \tfrac{1}{4}\Delta_2^d W)_{\boldsymbol{i}-\boldsymbol{e}^d}) \\
\Delta^C W_{\boldsymbol{i}} &= \tfrac{1}{2}(W^n_{\boldsymbol{i}+\boldsymbol{e}^d} - W^n_{\boldsymbol{i}-\boldsymbol{e}^d}) \\
\Delta^L W_{\boldsymbol{i}} &= W^n_{\boldsymbol{i}} - W^n_{\boldsymbol{i}-\boldsymbol{e}^d} \\
\Delta^R W_{\boldsymbol{i}} &= W^n_{\boldsymbol{i}+\boldsymbol{e}^d} - W^n_{\boldsymbol{i}} \\
\Delta^{3L} W_{\boldsymbol{i}} &= \tfrac{1}{2}(3W^n_{\boldsymbol{i}} - 4W^n_{\boldsymbol{i}-\boldsymbol{e}^d} + W^n_{\boldsymbol{i}-2\boldsymbol{e}^d}) \\
\Delta^{3R} W_{\boldsymbol{i}} &= \tfrac{1}{2}(-3W^n_{\boldsymbol{i}} + 4W^n_{\boldsymbol{i}+\boldsymbol{e}^d} - W^n_{\boldsymbol{i}+2\boldsymbol{e}^d})
\end{aligned}$$

$$\begin{aligned}
\Delta^{VLL} W_{\boldsymbol{i}} &= \min(\Delta^{3L} W_{\boldsymbol{i}}, \Delta^L W_{\boldsymbol{i}}) && \text{if } \Delta^{3L} W_{\boldsymbol{i}} \cdot \Delta^L W_{\boldsymbol{i}} > 0 \\
\Delta^{VLL} W_{\boldsymbol{i}} &= 0 && \text{otherwise} \\
\Delta^{VLR} W_{\boldsymbol{i}} &= \min(\Delta^{3R} W_{\boldsymbol{i}}, \Delta^R W_{\boldsymbol{i}}) && \text{if } \Delta^{3R} W_{\boldsymbol{i}} \cdot \Delta^R W_{\boldsymbol{i}} > 0 \\
\Delta^{VLR} W_{\boldsymbol{i}} &= 0 && \text{otherwise}
\end{aligned}$$

We apply the van Leer limiter component-wise to the differences.

4.3 Viscous Tensor Equation

This section describes the method for solving the elliptic partial differential equation

$$\kappa L\vec{v} = \kappa\alpha\vec{v} + \beta\nabla \cdot F = \kappa\rho.$$

α is a constant and $\beta = \beta(\vec{x})$. F is given by

$$F = \eta(\nabla\vec{v} + \nabla\vec{v}^T) + \lambda(I\nabla \cdot \vec{v}) \tag{4.32}$$

where I is the identity matrix, $\eta = \eta(\vec{x})$, and $\lambda = \lambda(\vec{x})$.

4.3.1 Discretization

We discretize normal components of the face-centered gradient using an average of cell-centered gradients for tangential components and a centered-difference approximation to the normal gradient.

$$(\nabla\vec{v})^{d'}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = \left(\begin{array}{c} \frac{1}{h}(\vec{v}_{\boldsymbol{i}+\boldsymbol{e}^d} - \vec{v}_{\boldsymbol{i}}) \text{ if } d = d' \\ \frac{1}{2}((\nabla\vec{v})^{d'}_{\boldsymbol{i}+\boldsymbol{e}^d} + (\nabla\vec{v})^{d'}_{\boldsymbol{i}}) \text{ if } d \neq d' \end{array} \right)$$

where

$$(\nabla\vec{v})^d_{\boldsymbol{i}} = \frac{1}{2h}(\vec{v}_{\boldsymbol{i}+\boldsymbol{e}^d} - \vec{v}_{\boldsymbol{i}-\boldsymbol{e}^d}).$$

We discretize the divergence as follows

$$(\kappa\nabla\cdot F)_{\boldsymbol{i}} = \sum_{d'=1}^{D}(\alpha\nabla F)^{d'}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d}} + \alpha_B F^B$$

where κ and α are the volume and area fractions.

We use equation 4.32 get the flux at cell face centers. We then interpolate the flux to face centroids. In two dimensions, this interpolation takes the form

$$\widetilde{F}^{n+\frac{1}{2}}_{\boldsymbol{f}} = F^{n+\frac{1}{2}}_{\boldsymbol{f}} + |\bar{x}|(F^{n+\frac{1}{2}}_{\boldsymbol{f}\ll sign(\bar{x})\boldsymbol{e}^{d}} - F^{n+\frac{1}{2}}_{\boldsymbol{f}}) \tag{4.33}$$

where $\bar{x}$ is the centroid in the direction d perpendicular to the face normal. In three dimensions, define $(\bar{x},\bar{y})$ to be the coordinates of the centroid in the plane (d^1, d^2) perpendicular to the face normal.

$$\widetilde{F}^{n+\frac{1}{2}}_{\boldsymbol{f}} = F^{n+\frac{1}{2}}_{\boldsymbol{f}}(1-\bar{x}\bar{y}+|\bar{x}\bar{y}|)+ \tag{4.34}$$

$$F^{n+\frac{1}{2}}_{\boldsymbol{f}\ll sign(\bar{x})\boldsymbol{e}^{d^1}}(|\bar{x}|-|\bar{x}\bar{y}|)+ \tag{4.35}$$

$$F^{n+\frac{1}{2}}_{\boldsymbol{f}\ll sign(\bar{x})\boldsymbol{e}^{d^2}}(|\bar{y}|-|\bar{x}\bar{y}|)+ \tag{4.36}$$

$$F^{n+\frac{1}{2}}_{\boldsymbol{f}<<sign(\bar{x})\boldsymbol{e}^{d^1}<<sign(\bar{x})\boldsymbol{e}^{d^2}}(|\bar{x}\bar{y}|) \tag{4.37}$$

Centroids in any dimension are normalized by Δx and centered at the cell center. This interpolation is only done if the shifts that are used in the interpolation are uniquely-defined and single-valued. We use a conservative discretization for the flux divergence.

$$\kappa_{\boldsymbol{v}}\nabla\cdot\vec{F} \equiv (D\cdot\vec{F}) = ((\sum_{d=0}^{\mathbf{D}-1}\sum_{\pm=+,-}\sum_{\boldsymbol{f}\in\mathcal{F}^{d,\pm}_{\boldsymbol{v}}} \pm\alpha_{\boldsymbol{f}}\widetilde{F}^{n+\frac{1}{2}}_{\boldsymbol{f}}) + \alpha^B_{\boldsymbol{v}}F^{B,n+\frac{1}{2}}_{\boldsymbol{v}}) \tag{4.38}$$

where where F^B is the flux at the irregular boundary, wherein lies most of the difficulty in this operator.

4.3.2 Flux at the Boundary

In all cases, we construct the gradient at the boundary and use equation 4.32 to construct the flux.

For Neumann boundary conditions, the gradient of the solution is specified at the boundary.

For Dirichlet boundary conditions, the gradient normal to the boundary is determined using the value at the boundary. The gradients tangential to the boundary are specified. For irregular boundaries, the procedure for calculating the gradient normal to the boundary is given in section 4.2.1.5. For domain boundaries, we construct a quadratic function with the value at the boundary and the two adjacent points along the normal to construct the gradient. For example, say we are at the low side of the domain with a value ϕ_0 at the boundary. The normal gradient is given by. that means that normal gradient is given by

$$(\nabla\phi)^x_{-\frac{1}{2},j,k} = \frac{9(\phi_{0,j,k} - \phi_0) - (\phi_{1,j,k} - \phi_0)}{3\Delta x}$$

4.4 Conductivity Equation

This section describes the method for solving the elliptic partial differential equation

$$\kappa L\phi = \kappa\alpha a\phi + \beta\nabla \cdot F = \kappa\rho.$$

α and $beta$ are constants, a is a function of space and F is given by

$$F = b\nabla\phi \tag{4.39}$$

The conductivity b is a function of space.

4.4.1 Discretization

We discretize the face-centered gradient for the flux using a centered-difference approximation.

$$(\nabla\phi)^d_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = \frac{1}{h}(phi_{\boldsymbol{i}+\boldsymbol{e}^d} - \phi_{\boldsymbol{i}})$$

We discretize the divergence as follows

$$(\kappa\nabla \cdot F)_{\boldsymbol{i}} = \sum_{d'=1}^{D}(\alpha\nabla F)^{d'}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} + \alpha_B F^B$$

where κ and α are the volume and area fractions. We use equation 4.39 get the flux at cell face centers. We then interpolate the flux to face centroids. In two dimensions, this interpolation takes the form

$$\widetilde{F}^{n+\frac{1}{2}}_{\boldsymbol{f}} = F^{n+\frac{1}{2}}_{\boldsymbol{f}} + |\bar{x}|(F^{n+\frac{1}{2}}_{\boldsymbol{f}\ll sign(\bar{x})\boldsymbol{e}^d} - F^{n+\frac{1}{2}}_{\boldsymbol{f}}) \tag{4.40}$$

where $\bar{x}$ is the centroid in the direction d perpendicular to the face normal. In three dimensions, define $(\bar{x}, \bar{y})$ to be the coordinates of the centroid in the plane (d^1, d^2) perpendicular to the face normal.

$$\widetilde{F}^{n+\frac{1}{2}}_{\boldsymbol{f}} = F^{n+\frac{1}{2}}_{\boldsymbol{f}}(1 - \bar{x}\bar{y} + |\bar{x}\bar{y}|)+ \tag{4.41}$$

$$F^{n+\frac{1}{2}}_{\boldsymbol{f}\ll sign(\bar{x})\boldsymbol{e}^{d^1}}(|\bar{x}| - |\bar{x}\bar{y}|)+ \tag{4.42}$$

$$F^{n+\frac{1}{2}}_{\boldsymbol{f}\ll sign(\bar{x})\boldsymbol{e}^{d^2}}(|\bar{y}| - |\bar{x}\bar{y}|)+ \tag{4.43}$$

$$F^{n+\frac{1}{2}}_{\boldsymbol{f}<<sign(\bar{x})\boldsymbol{e}^{d^1}<<sign(\bar{x})\boldsymbol{e}^{d^2}}(|\bar{x}\bar{y}|) \tag{4.44}$$

Centroids in any dimension are normalized by Δx and centered at the cell center. This interpolation is only done if the shifts that are used in the interpolation are uniquely-defined and single-valued. We use a conservative discretization for the flux divergence.

$$\kappa_{\boldsymbol{v}} \nabla \cdot \vec{F} \equiv (D \cdot \vec{F}) = ((\sum_{d=0}^{\mathbf{D}-1} \sum_{\pm=+,-} \sum_{\boldsymbol{f} \in \mathcal{F}_{\boldsymbol{v}}^{d,\pm}} \pm \alpha_{\boldsymbol{f}} \widetilde{F}_{\boldsymbol{f}}^{n+\frac{1}{2}}) + \alpha_{\boldsymbol{v}}^{B} F_{\boldsymbol{v}}^{B,n+\frac{1}{2}}) \tag{4.45}$$

where where F^B is the flux at the irregular boundary, wherein lies most of the difficulty in this operator.

4.4.2 Flux at the Boundary

In all cases, we construct the gradient at the boundary and use equation 4.39 to construct the flux.

For Neumann boundary conditions, the gradient of the solution is specified at the boundary.

For Dirichlet boundary conditions, the gradient normal to the boundary is determined using the value at the boundary. The gradients tangential to the boundary are specified. For irregular boundaries, the procedure for calculating the gradient normal to the boundary is given in section 4.2.1.5. For domain boundaries, we construct a quadratic function with the value at the boundary and the two adjacent points along the normal to construct the gradient. For example, say we are at the low side of the domain with a value ϕ_0 at the boundary. The normal gradient is given by. that means that normal gradient is given by

$$(\nabla \phi)^x_{-\frac{1}{2},j,k} = \frac{9(\phi_{0,j,k} - \phi_0) - (\phi_{1,j,k} - \phi_0)}{3\Delta x}$$

4.5 Overview

The principal `EBAMRElliptic` classes are:

- `EBPoissonOp` conforms to the `MGLevelOp` interface and is used to solve Poisson's equation over a single level.
- `EBAMRPoissonOp` conforms to the `AMRLevelOp` interface and is used to solve Poisson's (or Helmholtz's) equation over an AMR hierarchy with constant coefficients.
- `EBConductivityOp` conforms to the `AMRLevelOp` interface and is used to solve Poisson's (or Helmholtz's) equation over an AMR hierarchy with variable coefficients.

- EBViscousTensorOp conforms to the AMRLevelOp interface and is used to solve the viscous tensor equation over an AMR hierarchy with variable coefficients.
- EBAMRTGA advances a solution of the heat equation one step using the TGA [15] algorithm.

The first two, since their interface is well described in the Chombo Design document [4] will only be described through their factories, since the factories are the parts of the interface that the user actually has to use in order to use the class.

4.6 EBPoissonOpFactory

Factory to generate an operator to solve $(\alpha + \beta L)\phi = \rho$. This follows the MGLevelOp interface.

```
EBPoissonOpFactory(const EBLevelGrid&                            eblgs,
                   const RealVect&                               dx,
                   const RealVect&                               origin,
                   const int&                                    orderEB,
                   const int&                                    numPreCondIters,
                   const int&                                    relaxType,
                   RefCountedPtr<BaseDomainBCFactory>            domainBCFactory,
                   RefCountedPtr<BaseEBBCFactory>                ebBcFactory,
                   const Real&                                   alpha,
                   const Real&                                   beta,
                   const IntVect&                                ghostCellsPhi,
                   const IntVect&                                ghostCellsRhs);
```

- eblgs : layout of the level
- domainFactory : domain boundary conditions
- ebBCFactory: eb boundary conditions
- dxCoarse: grid spacing at coarsest level
- origin: offset to lowest corner of the domain
- refRatio: refinement ratios. refRatio[i] is between levels i and i+1
- preCondIters: number of iterations to do for pre-conditioning
- relaxType: 0 means point Jacobi, 1 is Gauss-Seidel, 2 is line solver.
- orderEB: 0 to not do flux interpolation at cut faces.

- `alpha`: coefficient of identity
- `beta`: coefficient of Laplacian.
- `ghostCellsPhi`: Number of ghost cells in phi, correction (typically one)
- `ghostCellsRhs`: Number of ghost cells in RHS, residual, lphi (typically zero) Ghost cell arguments are there for caching reasons. Once you set them, an error is thrown if you send in data that does not match.

4.7 EBAMRPoissonOpFactory

Factory to generate an operator to solve $(\alpha + \beta L)\phi = \rho$. This follows the AMRLevelOp interface.

```
EBAMRPoissonOpFactory(const Vector<EBLevelGrid>&                    eblgs,
                      const Vector<int>&                            refRatio,
                      const Vector<RefCountedPtr<EBQuadCFInterp> >& quadCFI,
                      const RealVect&                               dxCoarse,
                      const RealVect&                               origin,
                      const int&                                    orderEB,
                      const int&                                    numPreCondIters,
                      const int&                                    relaxType,
                      RefCountedPtr<BaseDomainBCFactory>            domainBCFactory,
                      RefCountedPtr<BaseEBBCFactory>                ebBcFactory,
                      const Real&                                   alpha,
                      const Real&                                   beta,
                      const Real&                                   time,
                      const IntVect&                                ghostCellsPhi,
                      const IntVect&                                ghostCellsRhs,
                      int numLevels = -1);
```

- `eblgs` : layouts at each AMR level
- `domainFactory` : domain boundary conditions
- `ebBCFactory`: eb boundary conditions
- `dxCoarse`: grid spacing at coarsest level
- `origin`: offset to lowest corner of the domain
- `refRatio`: refinement ratios. refRatio[i] is between levels i and i+1
- `preCondIters`: number of iterations to do for pre-conditioning
- `relaxType`: 0 means point Jacobi, 1 is Gauss-Seidel, 2 is line solver.

- `orderEB`: 0 to not do flux interpolation at cut faces.
- `alpha`: coefficient of identity
- `beta`: coefficient of Laplacian.
- `time`: time for boundary conditions
- `ghostCellsPhi`: Number of ghost cells in phi, correction (typically one)
- `ghostCellsRhs`: Number of ghost cells in RHS, residual, lphi (typically zero) Ghost cell arguments are there for caching reasons. Once you set them, an error is thrown if you send in data that does not match. Use numlevels = -1 if you want to use the size of the vectors for numlevels.

4.8 EBAMRTGA

EBAMR implementation of the TGA algorithm to solve the heat equation.

```
EBAMRTGA(const Vector<EBLevelGrid>&                    eblg,
         const Vector<int>&                            refRatio,
         const Vector<RefCountedPtr<EBQuadCFInterp> >& quadCFI,
         const RealVect&                               dxCoar,
         const RefCountedPtr<BaseDomainBCFactory>&     domainBCFactory,
         const RefCountedPtr<BaseEBBCFactory>&         ebBCFactory,
         const int&                                    numlevels,
         const RealVect&                               origin,
         const Real&                                   diffusionConst,
         const IntVect&                                ghostCellsPhi,
         const IntVect&                                ghostCellsRHS,
         const int&                                    numSmooths,
         const int&                                    iterMax,
         const int&                                    ODESolver,
         const int&                                    numMGCycles,
         const int&                                    numPreCondIters,
         const int&                                    relaxType,
         const int&                                    verbocity);
```

4.9 Example

4.10 Snippet to solve Poisson's equation

```
void solve(const PoissonParameters&  a_params,
  Vector<LevelData<EBCellFAB>* >& phi,
  Vector<LevelData<EBCellFAB>* >& rhs,
  Vector<DisjointBoxLayout>&      grids,
  Vector<EBISLayout>&             ebisl)
  )
{
  int nvar = 1;
  //create the solver
  AMRMultiGrid<LevelData<EBCellFAB> > solver;
  pout() << "defining  solver" << endl;

  BiCGStabSolver<LevelData<EBCellFAB> > newBottomSolver;
  newBottomSolver.verbosity = 0;
  defineSolver(solver, grids, ebisl, newBottomSolver, a_params,1.e99);
  pout() << "solving " << endl;

  //solve the equation
  solver.solve(phi, rhs, a_params.maxLevel, 0);
}
```

4.11 Snippet to Project a Cell-Centered Velocity Field

```
void projectVel(
              const Vector< DisjointBoxLayout >&   a_grids,
              const Vector< EBISLayout >&          a_ebisl,
              const PoissonParameters&             a_params)
              const int&                           a_dofileout,
              const bool&                          a_isFine)
              Vector<LevelData<EBCellFAB>* >& velo,
              Vector<LevelData<EBCellFAB>* >& gphi)
{
  int nlevels = a_params.numLevels;

  RealVect dxLevCoarsest = RealVect::Unit;
  dxLevCoarsest *=a_params.coarsestDx;
  ProblemDomain domLevCoarsest(a_params.coarsestDomain);

  RealVect dxLev = dxLevCoarsest;

  Real domVal = 0.0;
  NeumannPoissonDomainBCFactory* domBCPhi = new NeumannPoissonDomainBCFactory();
  RefCountedPtr<BaseDomainBCFactory> baseDomainBCPhi = domBCPhi;
  domBCPhi->setValue(domVal);
  DirichletPoissonDomainBCFactory* domBCVel = new DirichletPoissonDomainBCFactory();
```

```
RefCountedPtr<BaseDomainBCFactory> baseDomainBCVel = domBCVel;
domBCVel->setValue(domVal);

NeumannPoissonEBBCFactory*      ebBCPhi = new NeumannPoissonEBBCFactory();
ebBCPhi->setValue(domVal);
RefCountedPtr<BaseEBBCFactory>     baseEBBCPhi     = ebBCPhi;


Vector<LevelData<EBCellFAB>*> rhoinv;
const int bottomSolverType = 1;

Vector<EBLevelGrid>                      eblg   (a_grids.size());
Vector<RefCountedPtr<EBQuadCFInterp> >   quadCFI(a_grids.size(), NULL);
domLev = domLevCoarsest;
for(int ilev = 0; ilev < a_grids.size(); ilev++)
  {
    int nvar = 1;
    int nref = a_params.refRatio[ilev];
    eblg[ilev] = EBLevelGrid(a_grids[ilev], a_ebisl[ilev], domLev);
    if(ilev > 0)
      {
        int nrefOld = a_params.refRatio[ilev-1];
        ProblemDomain domLevCoar = coarsen(domLev, nrefOld);
        quadCFI[ilev] = new EBQuadCFInterp(a_grids[ilev  ],
                                           a_grids[ilev-1],
                                           a_ebisl[ilev  ],
                                           a_ebisl[ilev-1],
                                           domLevCoar,
                                           nrefOld, nvar,
                                           *(eblg[ilev].getCFIVS()));

      }
    domLev.refine(nref);
  }


EBCompositeCCProjector projectinator(eblg,  a_params.refRatio, quadCFI,
                                     a_params.coarsestDx,
                                     RealVect::Zero,
                                     baseDomainBCVel,
                                     baseDomainBCPhi,
                                     baseEBBCPhi,
                                     rhoinv, false, true, -1, 3 ,40,1.e99, 1,
projectinator.project(velo, gphi);
}
```

Chapter 5

Layer 5—EBAMRTimeDependent: Tools for Hyperbolic problems

5.1 Introduction

This document describes our numerical method for integrating systems of conservation laws (e.g., the Euler equations of gas dynamics) on an AMR grid hierarchy with embedded boundaries. We use an unsplit, second-order Godunov method, extending the algorithms developed by Colella [5] and Saltzman [13].

5.2 Notation

All these operations take place in a very similar context to that presented in [4]. For non-embedded boundary notation, refer to that document.

The standard (i, j, k) is not sufficient here to denote a computational cell as there can be multiple VoFs per cell. We define $\boldsymbol{v}$ to be the notation for a VoF and $\boldsymbol{f}$ to be a face. The function $ind(\boldsymbol{v})$ produces the cell which the VoF lives in. We define $\boldsymbol{v}^+(\boldsymbol{f})$ to be the VoF on the high side of face $\boldsymbol{f}$; $\boldsymbol{v}^-(\boldsymbol{f})$ is the VoF on the low side of face $\boldsymbol{f}$; $\boldsymbol{f}_d^+(\boldsymbol{v})$ is the set of faces on the high side of VoF $\boldsymbol{v}$; $\boldsymbol{f}_d^-(\boldsymbol{v})$ is the set of faces on the low side of VoF $\boldsymbol{v}$, where $d \in \{x, y, z\}$ is a coordinate direction (the number of directions is D). Also, we compose these operators to represent the set of VoFs directly connected to a given VoF: $\boldsymbol{v}_d^+(\boldsymbol{v}) = \boldsymbol{v}^+(\boldsymbol{f}_d^+(\boldsymbol{v}))$ and $\boldsymbol{v}_d^-(\boldsymbol{v}) = \boldsymbol{v}^-(\boldsymbol{f}_d^-(\boldsymbol{v}))$. The $\ll$ operator shifts data in the direction of the right hand argument. The shift operator can yield multiple VoFs. In this case, the shift operator includes averaging the values at the shifted-to VoFs.

We follow the same approach in the EB case in defining multilevel data and operators as we did for ordinary AMR. Given an AMR mesh hierarchy $\{\Omega^l\}_{l=0}^{lmax}$, we define the valid VoFs on level l to be

$$\mathcal{V}_{valid}^l = ind^{-1}(\Omega_{valid}^l) \tag{5.1}$$

and composite cell-centered data

$$\varphi^{comp} = \{\varphi^{l,valid}\}_{l=0}^{lmax}, \varphi^{l,valid} : \mathcal{V}_{valid}^{l} \rightarrow \mathbb{R}^m \tag{5.2}$$

For face-centered data,

$$\begin{aligned}\mathcal{F}_{valid}^{l,d} &= ind^{-1}(\Omega_{valid}^{l,\mathbf{e}^d})\\ \vec{F}^{l,valid} &= (F_0^{l,valid}, \ldots, F_{D-1}^{l,valid})\\ F_d^{l,valid} &: \mathcal{F}_{valid}^{l,d} \rightarrow \mathbb{R}^m\end{aligned} \tag{5.3}$$

For computations at cell centers the notation

$$CC = A \mid B \mid C$$

means that the 3-point formula A is used for CC if all cell centered values it uses are available, the 2-point formula B is used if current cell borders the high side of the physical domain (i.e., no high side value), and the 2-point formula C is used if current cell borders the low side of the physical domain (i.e., no low side value). A value is "available" if its VoF is not covered and is within the domain of computation. For computations at face centers the analogous notation

$$FC = A \mid B \mid C$$

means that the 2-point formula A is used for FC if all cell centered values it uses are available, the 1-point formula B is used if current face coincides with the high side of the physical domain (i.e., no high side value), and the 1-point formula C is used if current face coincided with the low side of the physical domain (i.e., no low side value).

5.3 Equations of Motion

We are solving a hyperbolic system of equations of the form

$$\frac{\partial U}{\partial t} + \sum_{d=0}^{\mathbf{D}-1} \frac{\partial F^d}{\partial x^d} = S \tag{5.4}$$

For 3D polytropic gas dynamics,

$$\begin{aligned}U &= (\rho, \rho u_x, \rho u_y, \rho u_z, \rho E)^T\\ F^x &= \left(\rho u_x, \rho u_x^2, \rho u_x u_y, \rho u_x u_z, \rho u_x E + u_x p\right)^T\\ F^y &= \left(\rho u_y, \rho u_x u_y, \rho u_y^2, \rho u_y u_z, \rho u_y E + u_y p\right)^T\\ F^z &= \left(\rho u_z, \rho u_x u_z, \rho u_z u_y, \rho u_z^2, \rho u_z E + u_z p\right)^T\\ E &= \frac{\gamma p}{(\gamma - 1)\rho} + \frac{|\vec{u}|^2}{2}\end{aligned} \tag{5.5}$$

We are given boundary conditions on faces at the boundary of the domain and on the embedded boundary. We also assume there may be a change of variables $W = W(U)$ ($W \equiv$ "primitive variables") that can be applied to simplify the calculation of the characteristic structure of the equations. This leads to a similar system of equations in W.

$$\begin{aligned} \frac{\partial W}{\partial t} + \sum_{d=0}^{\mathbf{D}-1} A^d(W)\, \frac{\partial W^d}{\partial x^d} = S' \\ A^d = \nabla_U W \cdot \nabla_U F^d \cdot \nabla_W U \\ S' = \nabla_U W \cdot S \end{aligned} \tag{5.6}$$

For 3D polytropic gas dynamics,

$$W = (\rho, u_x, u_y, u_z, p)^T$$

$$A^x = \begin{pmatrix} u_x & \rho & 0 & 0 & 0 \\ 0 & u_x & 0 & 0 & \frac{1}{\rho} \\ 0 & 0 & u_x & 0 & 0 \\ 0 & 0 & 0 & u_x & 0 \\ 0 & \rho c^2 & 0 & 0 & u_x \end{pmatrix}$$

$$A^y = \begin{pmatrix} u_y & 0 & \rho & 0 & 0 \\ 0 & u_y & 0 & 0 & 0 \\ 0 & 0 & u_y & 0 & \frac{1}{\rho} \\ 0 & 0 & 0 & u_y & 0 \\ 0 & 0 & \rho c^2 & 0 & u_y \end{pmatrix}$$

$$A^z = \begin{pmatrix} u_z & 0 & 0 & \rho & 0 \\ 0 & u_z & 0 & 0 & 0 \\ 0 & 0 & u_z & 0 & 0 \\ 0 & 0 & 0 & u_z & \frac{1}{\rho} \\ 0 & 0 & 0 & \rho c^2 & u_z \end{pmatrix}$$

5.4 Approximations to $\nabla \cdot F$.

To obtain a second-order approximation of the flux divergence in conservative form, first we must interpolate the flux to the face centroid. In two dimensions, this interpolation takes the form

$$\widetilde{F}_{\boldsymbol{f}}^{n+\frac{1}{2}} = F_{\boldsymbol{f}}^{n+\frac{1}{2}} + |\bar{x}|(F_{\boldsymbol{f} \ll sign(\bar{x})e^d}^{n+\frac{1}{2}} - F_{\boldsymbol{f}}^{n+\frac{1}{2}}) \tag{5.7}$$

where $\bar{x}$ is the centroid in the direction d perpendicular to the face normal. In three dimensions, define $(\bar{x}, \bar{y})$ to be the coordinates of the centroid in the plane (d^1, d^2) perpendicular to the face normal.

$$\widetilde{F}_{\boldsymbol{f}}^{n+\frac{1}{2}} = F_{\boldsymbol{f}}^{n+\frac{1}{2}}(1-\bar{x}\bar{y}+|\bar{x}\bar{y}|)+ \tag{5.8}$$

$$F_{\boldsymbol{f}\ll sign(\bar{x})\boldsymbol{e}^{d1}}^{n+\frac{1}{2}}(|\bar{x}|-|\bar{x}\bar{y}|)+ \tag{5.9}$$

$$F_{\boldsymbol{f}\ll sign(\bar{x})\boldsymbol{e}^{d2}}^{n+\frac{1}{2}}(|\bar{y}|-|\bar{x}\bar{y}|)+ \tag{5.10}$$

$$F_{\boldsymbol{f}<<sign(\bar{x})\boldsymbol{e}^{d1}<<sign(\bar{x})\boldsymbol{e}^{d2}}^{n+\frac{1}{2}}(|\bar{x}\bar{y}|) \tag{5.11}$$

Centroids in any dimension are normalized by Δx and centered at the cell center. This interpolation is only done if the shifts that are used in the interpolation are uniquely-defined and single-valued.

We then define the conservative divergence approximation.

$$\nabla\cdot\vec{F} \equiv (D\cdot\vec{F})^c = \frac{1}{k_{\boldsymbol{v}}h}((\sum_{d=0}^{\mathbf{D}-1}\sum_{\pm=+,-}\sum_{\boldsymbol{f}\in\mathcal{F}_{\boldsymbol{v}}^{d,\pm}} \pm\alpha_{\boldsymbol{f}}\widetilde{F}_{\boldsymbol{f}}^{n+\frac{1}{2}}) + \alpha_{\boldsymbol{v}}^B F_{\boldsymbol{v}}^{B,n+\frac{1}{2}}) \tag{5.12}$$

The non-conservative divergence approximation is defined below.

$$\nabla\cdot\vec{F} = (D\cdot\vec{F})^{NC} = \frac{1}{h}\sum_{\pm=+,-}\sum_{d=0}^{\mathbf{D}-1}\pm\bar{F}_{\boldsymbol{v},\pm,d}^{n+\frac{1}{2}} \tag{5.13}$$

$$\bar{F}_{\boldsymbol{v},\pm,d}^{n+\frac{1}{2}} = \begin{cases} \frac{1}{N(\mathcal{F}_{\boldsymbol{v}}^{d,\pm})}\sum_{\boldsymbol{f}\in\mathcal{F}_{\boldsymbol{v}}^{d,\pm}} F_{\boldsymbol{f}}^{n+\frac{1}{2}} & \text{if } N(\mathcal{F}_{\boldsymbol{v}}^{d,\pm})>0 \\ F_{\boldsymbol{v},\pm,d}^{\text{covered},n+\frac{1}{2}} & \text{otherwise} \end{cases} \tag{5.14}$$

The preliminary update update of the solution of the solution takes the form:

$$U_{\boldsymbol{v}}^{n+1} = U_{\boldsymbol{v}}^n - \Delta t((1-k_{\boldsymbol{v}})(D\cdot\vec{F})_{\boldsymbol{v}}^{NC} + k_{\boldsymbol{v}}(D\cdot\vec{F})_{\boldsymbol{v}}^c) \tag{5.15}$$

$$\delta M = -\Delta t k_{\boldsymbol{v}}(1-k_{\boldsymbol{v}})((D\cdot\vec{F})_{\boldsymbol{v}}^c - (D\cdot\vec{F})_{\boldsymbol{v}}^{NC}) \tag{5.16}$$

δM is the total mass increment that has been unaccounted for in the preliminary update. See the EBAMRTools document for how this mass gets redistributed in an AMR context. On a single level, the redistribution takes the following form:

$$U_{\boldsymbol{v}'}^{n+1} := U_{\boldsymbol{v}'}^{n+1} + w_{\boldsymbol{v},\boldsymbol{v}'},\delta M_{\boldsymbol{v}} \tag{5.17}$$

$$\boldsymbol{v}' \in \mathcal{N}(\boldsymbol{v}), \tag{5.18}$$

where $\mathcal{N}(\boldsymbol{v})$ is the set of VoFs that can be connected to $\boldsymbol{v}$ with a monotone path of length ≤ 1. The weights are nonnegative, and satisfy $\sum\limits_{\boldsymbol{v}'\in\mathcal{N}(\boldsymbol{v})}\kappa_{\boldsymbol{v}'}w_{\boldsymbol{v},\boldsymbol{v}'} = 1$.

5.5 Flux Estimation

Given $U^n_{\boldsymbol{i}}$ and $S^n_{\boldsymbol{i}}$, we want to compute a second-order accurate estimate of the fluxes: $F^{n+\frac{1}{2}}_{\boldsymbol{f}} \approx F^d(\boldsymbol{x}_0 + (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, t^n + \frac{1}{2}\Delta t)$. Specifically, we want to compute the fluxes at the center of the Cartesian grid faces corresponding to the faces of the embedded boundary geometry. In addition, we want to compute fluxes at the centers of Cartesian grid faces corresponding to faces adjacent to VoFs, but that are completely covered. Pointwise operations are conceptually the same for both regular and irregular VoFs. In other operations we specify both the regular and irregular VoF calculation. The transformations $\nabla_U W$ and $\nabla_W U$ are functions of both space and time. We shall leave the precise centering of these transformations vague as this will be application-dependent. In outline, the method is given as follows.

5.5.1 Flux Estimation in Two Dimensions

1. Transform to primitive variables.

$$W^n_{\boldsymbol{v}} = W(U^n_{\boldsymbol{v}}) \tag{5.19}$$

2. Compute slopes $\Delta^d W_{\boldsymbol{v}}$. This is described separately in section 5.6.

3. Compute the effect of the normal derivative terms and the source term on the extrapolation in space and time from cell centers to faces. For $0 \leq d < \mathbf{D}$,

$$\begin{aligned} W_{\boldsymbol{v},\pm,d} &= W^n_{\boldsymbol{v}} + \frac{1}{2}(\pm I - \frac{\Delta t}{h} A^d_{\boldsymbol{v}}) P_\pm(\Delta^d W_{\boldsymbol{v}}) \\ A^d_{\boldsymbol{v}} &= A^d(W_{\boldsymbol{v}}) \\ P_\pm(W) &= \sum_{\pm\lambda_k > 0} (l_k \cdot W) r_k \\ W_{\boldsymbol{v},\pm,d} &= W_{\boldsymbol{v},\pm,d} + \frac{\Delta t}{2} \nabla_U W \cdot S^n_{\boldsymbol{v}} \end{aligned} \tag{5.20}$$

where λ_k are eigenvalues of $A^d_{\boldsymbol{i}}$, and l_k and r_k are the corresponding left and right eigenvectors. We then extrapolate to the covered faces. First we define the VoFs involved.

$$\begin{aligned} d' &= 1 - d \\ s^d &= sign(n^d) \\ \boldsymbol{v}^{up} &= ind^{-1}(ind(\boldsymbol{v}) + s^{d'}\boldsymbol{e}^{d'} - s^d\boldsymbol{e}^d) \\ \boldsymbol{v}^{side} &= ind^{-1}(ind(\boldsymbol{v}) + s^d\boldsymbol{e}^d) \\ \boldsymbol{v}^{corner} &= ind^{-1}(ind(\boldsymbol{v}) + s^{d'}\boldsymbol{e}^{d'}) \end{aligned} \tag{5.21}$$

Define $W^{\text{up,side,corner}}$, extrapolations to the edges near the VoFs near $\boldsymbol{v}$.

$$\begin{aligned} W^{\text{up}} &= W_{\boldsymbol{v}^{\text{up}},\mp,d} \\ W^{\text{side}} &= W_{\boldsymbol{v}^{\text{side}},\mp,d} - s^d \Delta^d W \\ W^{\text{corner}} &= W_{\boldsymbol{v}^{\text{corner}},\mp,d} \\ \Delta^d W &= \begin{cases} \Delta^d W^n_{\boldsymbol{v}^{\text{side}}} & \text{if } n^d > n^{d'} \\ \Delta^d W^n_{\boldsymbol{v}^{\text{corner}}} & \text{otherwise} \end{cases} \\ \Delta^{d'} W &= \begin{cases} \Delta^{d'} W^n_{\boldsymbol{v}^{\text{corner}}} & \text{if } n^d > n^{d'} \\ \Delta^{d'} W^n_{\boldsymbol{v}^{\text{up}}} & \text{otherwise} \end{cases} \end{aligned} \tag{5.22}$$

where the slopes are defined in section 5.6 If any of these vofs does not have a monotone path to the original VoF $\boldsymbol{v}$, we drop order the order of interpolation.

If $|n_d| < |n_{d'}|$:

$$W^{\text{full}} = \frac{|n_d|}{|n_{d'}|} W^{\text{corner}} + (1 - \frac{|n_d|}{|n_{d'}|}) W^{\text{up}} - (\frac{|n^d|}{|n^{d'}|} s^d \Delta^d W + s^{d'} \Delta^{d'} W) \tag{5.23}$$

$$W^{\text{covered}}_{\boldsymbol{v},\pm,d} = \begin{cases} W^{\text{full}} & \text{if both exist} \\ W^{\text{up}} & \text{if only } \boldsymbol{v}^{\text{up}} \text{ exists} \\ W^{\text{corner}} & \text{if only } \boldsymbol{v}^{\text{corner}} \text{ exists} \\ W^n_{\boldsymbol{v}} & \text{if neither exists} \end{cases} \tag{5.24}$$

If $|n_d| \geq |n_{d'}|$:

$$W^{\text{full}} = \frac{|n_{d'}|}{|n_d|} W^{\text{corner}} + (1 - \frac{|n_{d'}|}{|n_d|}) W^{\text{side}} - (\frac{|n^{d'}|}{|n^d|} s^{d'} \Delta^{d'} W + s^d \Delta^d W) \tag{5.25}$$

$$W^{\text{covered}}_{\boldsymbol{v},\pm,d} = \begin{cases} W^{\text{full}} & \text{if both exist} \\ W^{\text{side}} & \text{if only } \boldsymbol{v}^{\text{side}} \text{ exists} \\ W^{\text{corner}} & \text{if only } \boldsymbol{v}^{\text{corner}} \text{ exists} \\ W^n_{\boldsymbol{v}} & \text{if neither exists} \end{cases} \tag{5.26}$$

4. Compute estimates of F^d suitable for computing 1D flux derivatives $\frac{\partial F^d}{\partial x^d}$ using a Riemann solver for the interior, R, and for the boundary, R_B.

$$\begin{aligned} F^{\text{1D}}_{\boldsymbol{f}} &= R(W_{\boldsymbol{v}_-(\boldsymbol{f}),+,d}, W_{\boldsymbol{v}_+(\boldsymbol{f}),-,d}, d) \\ &\quad \mid \; R_B(W_{\boldsymbol{v}_-(\boldsymbol{f}),+,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \\ &\quad \mid \; R_B(W_{\boldsymbol{v}_+(\boldsymbol{f}),-,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \\ d &= dir(\boldsymbol{f}) \end{aligned} \tag{5.27}$$

5. Compute the covered fluxes $F^{\text{1D,covered}}$

$$\begin{aligned} F^{\text{1D, covered}}_{\boldsymbol{v},+,d} &= R(W_{\boldsymbol{v},+,d}, W^{\text{covered}}_{\boldsymbol{v},+,d}, d) \\ F^{\text{1D, covered}}_{\boldsymbol{v},-,d} &= R(W^{\text{covered}}_{\boldsymbol{v},-,d}, W_{\boldsymbol{v},-,d}, d) \end{aligned} \tag{5.28}$$

6. Compute corrections to $W_{\boldsymbol{i},\pm,d}$ due to the transverse derivatives. For regular cells, this takes the following form.

$$W^{n+\frac{1}{2}}_{\boldsymbol{i},\pm,d} = nW_{\boldsymbol{i},\pm,d} \; - \; \frac{\Delta t}{2h}\nabla_U W \cdot (F^{\text{1D}}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^{d_1}} - F^{\text{1D}}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^{d_1}}) \tag{5.29}$$

$$d \neq d_1, \; 0 \leq d, d_1 < \mathbf{D} \tag{5.30}$$

For irregular cells, we compute the transverse derivatives and use them to correct the extrapolated values of U and obtain time-centered fluxes at centers of Cartesian faces. In two dimensions, this takes the form

$$\begin{aligned} D^{d,\perp}F_{\boldsymbol{v}} &= \frac{1}{h}(\bar{F}_{\boldsymbol{v},+,d_1} - \bar{F}_{\boldsymbol{v},-,d_1}) \\ \bar{F}_{\boldsymbol{v},\pm,d'} &= \begin{cases} \frac{1}{N_{\boldsymbol{v},\pm,d'}} \sum_{\boldsymbol{f}\in\mathcal{F}_{\boldsymbol{v},\pm,d'}} F^{1D}_{\boldsymbol{f},\pm,d'} & \text{if } N_{\boldsymbol{v},\pm,d'} > 0 \\ F^{\text{1D, covered}}_{\boldsymbol{v},\pm,d'} & \text{otherwise} \end{cases} \\ d \neq d_1, &\quad 0 \leq d, d_1 < \mathbf{D} \\ W^{n+\frac{1}{2}}_{\boldsymbol{v},\pm,d} &= W_{\boldsymbol{v},\pm,d} - \frac{\Delta t}{2}\nabla_U W(D^{d,\perp}F_{\boldsymbol{v}}) \end{aligned} \tag{5.31}$$

Extrapolate to covered faces with the procedure described above using $W^{n+\frac{1}{2}}_{\cdot,\mp,d}$ to form $W^{n+\frac{1}{2},\text{covered}}_{\cdot,\pm,d}$.

7. Compute the flux estimate.

$$\begin{aligned}
F_{\boldsymbol{f}}^{n+\frac{1}{2}} &= R(W^{n+\frac{1}{2}}_{\boldsymbol{v}^-(\boldsymbol{f}),+,d}, W^{n+\frac{1}{2}}_{\boldsymbol{v}^+(\boldsymbol{f}),-,d}, d) \\
&\mid\ R_B(W^{n+\frac{1}{2}}_{\boldsymbol{v}^-(\boldsymbol{f}),+,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \\
&\mid\ R_B(W^{n+\frac{1}{2}}_{\boldsymbol{v}^+(\boldsymbol{f}),-,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \\
F^{n+\frac{1}{2},\text{covered}}_{\boldsymbol{v},-,d} &= R(W^{n+\frac{1}{2},\text{covered}}_{\boldsymbol{v},+,d}, W^{n+\frac{1}{2}}_{\boldsymbol{v},-,d}, d) \\
F^{n+\frac{1}{2},\text{covered}}_{\boldsymbol{v},+,d} &= R(W^{n+\frac{1}{2}}_{\boldsymbol{v},+,d}, W^{n+\frac{1}{2},\text{covered}}_{\boldsymbol{v},+,d}, d)
\end{aligned} \tag{5.32}$$

8. Modify the flux with artificial viscosity where the flow is compressive.

5.5.2 Flux Estimation in Three Dimensions

1. Transform to primitive variables.

$$W^n_{\boldsymbol{v}} = W(U^n_{\boldsymbol{v}}) \tag{5.33}$$

2. Compute slopes $\Delta^d W_{\boldsymbol{v}}$. This is described separately in section 5.6.

3. Compute the effect of the normal derivative terms and the source term on the extrapolation in space and time from cell centers to faces. For $0 \leq d < \mathbf{D}$,

$$\begin{aligned}
W_{\boldsymbol{v},\pm,d} &= W^n_{\boldsymbol{v}} + \frac{1}{2}(\pm I - \frac{\Delta t}{h} A^d_{\boldsymbol{v}}) P_\pm(\Delta^d W_{\boldsymbol{v}}) \\
A^d_{\boldsymbol{v}} &= A^d(W_{\boldsymbol{v}}) \\
P_\pm(W) &= \sum_{\pm\lambda_k > 0} (l_k \cdot W) r_k \\
W_{\boldsymbol{v},\pm,d} &= W_{\boldsymbol{v},\pm,d} + \frac{\Delta t}{2} \nabla_U W \cdot S^n_{\boldsymbol{v}}
\end{aligned} \tag{5.34}$$

   where λ_k are eigenvalues of $A^d_{\boldsymbol{i}}$, and l_k and r_k are the corresponding left and right eigenvectors.

   We then extrapolate to the covered faces. Define the direction of the face normal to be d_f and d_1, d_2 to be the directions tangential to the face. The procedure develops as follows

   - We define the associated vofs.
   - We form a 2x2 grid of values along a plane h away from the covered face and bilinearly interpolate to the point where the normal intersects the plane.
   - We use the slopes of the solution to extrapolate along the normal to get a second-order approximation of the solution at the covered face.

Which plane is selected is determined by the direction of the normal. If any of these VoFs does not have a monotone path to the original VoF $\boldsymbol{v}$, we drop order the order of interpolation.

If $|n_f| \geq |n_{d_1}|$ and $|n_{d_f}| \geq |n_{d_2}|$:

$$\begin{aligned}
\boldsymbol{v}^{00} =& ind^{-1}(ind(\boldsymbol{v}) + s^{d_f}\boldsymbol{e}^{d_f}) \\
\boldsymbol{v}^{10} =& ind^{-1}(ind(\boldsymbol{v}) + s^{d_1}\boldsymbol{e}^{d_1}) \\
\boldsymbol{v}^{01} =& ind^{-1}(ind(\boldsymbol{v}) + s^{d_2}\boldsymbol{e}^{d_2}) \\
\boldsymbol{v}^{11} =& ind^{-1}(ind(\boldsymbol{v}) + s^{d_1}\boldsymbol{e}^{d_1} + s^{d_2}\boldsymbol{e}^{d_2}) \\
W^{00} =& W_{\boldsymbol{v}^{00},\mp,d_f} - s^{d_f}\Delta^{d_f}W_{\boldsymbol{v}^{00}} \\
W^{10} =& W_{\boldsymbol{v}^{10},\mp,d_f} \\
W^{01} =& W_{\boldsymbol{v}^{01},\mp,d_f} \\
W^{11} =& W_{\boldsymbol{v}^{11},\mp,d_f}
\end{aligned} \tag{5.35}$$

We form a bilinear function $W(x_{d_1}, x_{d_2})$ in the plane formed by the four faces at which the values live:

$$\begin{aligned}
W(x_{d_1}, x_{d_2}) = & Ax_{d_1} + Bx_{d_2} + Cx_{d_1}x_{d_2} + D \\
A =& s^{d_1}(W^{10} - W^{00}) \\
B =& s^{d_2}(W^{01} - W^{00}) \\
C =& s^{d_1}s^{d_2}(W^{11} - W^{00}) - (W^{10} - W^{00}) - (W^{01} - W^{00}) \\
D =& W^{00}
\end{aligned} \tag{5.36}$$

We then extrapolate to the covered face from the point on the plane where the normal intersects

$$W^{\mathsf{full}} = W(s^{d_1}\frac{|n_{d_1}|}{|n_{d_f}|}, s^{d_2}\frac{|n_{d_2}|}{|n_{d_f}|}) - \Delta^{d_f}W_{\boldsymbol{v}^{00}} - s^{d_1}\frac{|n_{d_1}|}{|n_{d_f}|}\Delta^{d_1}W_{\boldsymbol{v}^{10}} - s^{d_2}\frac{|n_{d_2}|}{|n_{d_f}|}\Delta^{d_2}W_{\boldsymbol{v}^{01}} \tag{5.37}$$

Otherwise (assume $|n_{d_1}| \geq |n_{d_f}|$ and $|n_{d_1}| \geq |n_{d_2}|$):

$$\begin{aligned}
\boldsymbol{v}^{00} =& ind^{-1}(ind(\boldsymbol{v}) + s^{d_1}\boldsymbol{e}^{d_1}) \\
\boldsymbol{v}^{10} =& ind^{-1}(ind(\boldsymbol{v}) + s^{d_1}\boldsymbol{e}^{d_1}) - s^{d_f}\boldsymbol{e}^{d_f} \\
\boldsymbol{v}^{01} =& ind^{-1}(ind(\boldsymbol{v}) + s^{d_1}\boldsymbol{e}^{d_1}) + s^{d_2}\boldsymbol{e}^{d_2} \\
\boldsymbol{v}^{11} =& ind^{-1}(ind(\boldsymbol{v}) + s^{d_1}\boldsymbol{e}^{d_1} - s^{d_f}\boldsymbol{e}^{d_f} + s^{d_2}\boldsymbol{e}^{d_2} \\
W^{00} =& W_{\boldsymbol{v}^{00},\mp,d_f} \\
W^{10} =& W_{\boldsymbol{v}^{10},\mp,d_f} \\
W^{01} =& W_{\boldsymbol{v}^{01},\mp,d_f} \\
W^{11} =& W_{\boldsymbol{v}^{11},\mp,d_f}
\end{aligned} \tag{5.38}$$

We form a bilinear function $W(x_{d_1}, x_{d_2})$ in the plane formed by the four faces at which the values live. This is shown in equation 5.36. We then extrapolate to the covered face from the point on the plane where the normal intersects

$$W^{\text{full}} = W(s^{d_f}\frac{|n_{d_f}|}{|n_{d_1}|}, s^{d_2}\frac{|n_{d_2}|}{|n_{d_1}|}) - \Delta^{d_1}W_{\boldsymbol{v}^{00}} - s^{d_f}\frac{|n_{d_f}|}{|n_{d_1}|}\Delta^{d_f}W_{\boldsymbol{v}^{10}} - s^{d_2}\frac{|n_{d_2}|}{|n_{d_1}|}\Delta^{d_2}W_{\boldsymbol{v}^{01}} \tag{5.39}$$

In either case,

$$W^{\text{covered}}_{\boldsymbol{v},\pm,d} = \begin{cases} W^{\text{full}} & \text{if all four VoFs exist} \\ W^n_{\boldsymbol{v}} & \text{otherwise} \end{cases} \tag{5.40}$$

4. Compute estimates of F^d suitable for computing 1D flux derivatives $\frac{\partial F^d}{\partial x^d}$ using a Riemann solver for the interior, R, and for the boundary, R_B.

$$\begin{aligned} F^{\text{1D}}_{\boldsymbol{f}} &= R(W_{\boldsymbol{v}_-(\boldsymbol{f}),+,d}, W_{\boldsymbol{v}_+(\boldsymbol{f}),-,d}, d) \\ &\quad |\ R_B(W_{\boldsymbol{v}_-(\boldsymbol{f}),+,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \\ &\quad |\ R_B(W_{\boldsymbol{v}_+(\boldsymbol{f}),-,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \\ d &= dir(\boldsymbol{f}) \end{aligned} \tag{5.41}$$

5. Compute the covered fluxes $F^{\text{1D,covered}}$

$$\begin{aligned} F^{\text{1D, covered}}_{\boldsymbol{v},+,d} &= R(W_{\boldsymbol{v},+,d}, W^{\text{covered}}_{\boldsymbol{v},+,d}, d) \\ F^{\text{1D, covered}}_{\boldsymbol{v},-,d} &= R(W^{\text{covered}}_{\boldsymbol{v},-,d}, W_{\boldsymbol{v},-,d}, d) \end{aligned} \tag{5.42}$$

6. Compute corrections to $U_{\boldsymbol{i},\pm,d}$ corresponding to one set of transverse derivatives appropriate to obtain $(1, 1, 1)$ diagonal coupling. This step is only meaningful in three dimensions. We compute 1D flux differences, and use them to compute $U_{\boldsymbol{v},\pm,d_1,d_2}$, the d_1-edge-centered state partially updated by the effect of derivatives in the d_1, d_2 directions.

$$\begin{aligned} D^{\text{1D}}_d F^{\text{1D}}_{\boldsymbol{v}} &= \frac{1}{h}(\bar{F}^{\text{1D}}_{\boldsymbol{v},+,d} - \bar{F}^{\text{1D}}_{\boldsymbol{v},-,d}) \\ \bar{F}_{\boldsymbol{v},\pm,d} &= \begin{cases} \frac{1}{N_{\pm,d}}(\sum\limits_{\boldsymbol{f}\in\mathcal{F}_{\boldsymbol{v},\pm,d}} F^{\text{1D}}_{\boldsymbol{f}}) & \text{if } N_{\boldsymbol{v},\pm,d} > 0 \\ F^{\text{1D, covered}}_{\boldsymbol{v},\pm,d} & \text{otherwise} \end{cases} \end{aligned} \tag{5.43}$$

$$W_{\boldsymbol{v},\pm,d_1,d_2} = W_{\boldsymbol{v},\pm,d_1} - \frac{\Delta t}{3}\nabla_U W(D^{\text{1D}}_{d_2}F^{\text{1D}})_{\boldsymbol{v}} \tag{5.44}$$

We then extrapolate to covered faces with the procedure described above using $W_{\cdot,\pm,d_1,d_2}$ to form $W^{\text{covered},d}_{\cdot,\pm,d_1,d_2}$ and compute an estimate to the fluxes:

$$\begin{aligned}
F_{\boldsymbol{f},d_1,d_2} &= R(W_{\boldsymbol{v}_-(\boldsymbol{f}),+,d_1,d_2}, W_{\boldsymbol{v}_+(\boldsymbol{f}),-,d_1,d_2}, d_1) \\
&\quad \mid\ R_B(W_{\boldsymbol{v}_-(\boldsymbol{f}),+,d_1,d_2}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d_1) \\
&\quad \mid\ R_B(W_{\boldsymbol{v}_+(\boldsymbol{f}),-,d_1,d_2}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d_1) \\
d &= dir(\boldsymbol{f}) \\
F^{\text{covered}}_{\boldsymbol{v},-,d_1,d_2} &= R(W^{\text{covered}}_{\boldsymbol{v},-,d1,d2}, W_{\boldsymbol{v},-,d_1,d_2}, d_1) \\
F^{\text{covered}}_{\boldsymbol{v},+,d_1,d_2} &= R(W_{\boldsymbol{v},+,d1,d2}, W^{\text{covered}}_{\boldsymbol{v},+,d_1,d_2}, d_1)
\end{aligned} \tag{5.45}$$

7. Compute final corrections to $W_{\boldsymbol{i},\pm,d}$ due to the final transverse derivatives. We compute the $2\mathbf{D}$ transverse derivatives and use them to correct the extrapolated values of U and obtain time-centered fluxes at centers of Cartesian faces. In three dimensions, this takes the form:

$$\begin{aligned}
D^{d,\perp}F_{\boldsymbol{v}} &= \frac{1}{h}(\bar{F}_{\boldsymbol{v},+,d_1,d_2} - \bar{F}_{\boldsymbol{v},-,d_1,d_2} + \bar{F}_{\boldsymbol{v},+,d_2,d_1} - \bar{F}_{\boldsymbol{v},-,d_2,d_1}) \\
\bar{F}_{\boldsymbol{v},\pm,d',d''} &= \begin{cases} \frac{1}{N_{\boldsymbol{v},\pm,d'}} \sum_{\boldsymbol{f}\in\mathcal{F}_{\boldsymbol{v},\pm,d'}} F_{\boldsymbol{f},\pm,d',d''} & \text{if } N_{\boldsymbol{v},\pm,d'} > 0 \\ F^{\text{covered}}_{\boldsymbol{v},\pm,d',d''} & \text{otherwise} \end{cases} \\
d \neq d_1 \neq d_2 &\quad 0 \leq d, d_1, d_2 < \mathbf{D} \\
W^{n+\frac{1}{2}}_{\boldsymbol{v},\pm,d} &= W_{\boldsymbol{v},\pm,d} - \frac{\Delta t}{2}\nabla_U W(D^{d,\perp}F_{\boldsymbol{v}})
\end{aligned} \tag{5.46}$$

We then extrapolate to covered faces with the procedure described above using $W^{n+\frac{1}{2}}_{\cdot,\pm,d}$ to form $W^{n+\frac{1}{2},\text{covered},d}_{\cdot,\pm,d}$.

8. Compute the flux estimate.

$$\begin{aligned}
F^{n+\frac{1}{2}}_{\boldsymbol{f}} &= R(W^{n+\frac{1}{2}}_{\boldsymbol{v}^-(\boldsymbol{f}),+,d}, W^{n+\frac{1}{2}}_{\boldsymbol{v}^+(\boldsymbol{f}),-,d}, d) \\
&\quad \mid\ R_B(W^{n+\frac{1}{2}}_{\boldsymbol{v}^-(\boldsymbol{f}),+,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \\
&\quad \mid\ R_B(W^{n+\frac{1}{2}}_{\boldsymbol{v}^+(\boldsymbol{f}),-,d}, (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h, d) \\
F^{n+\frac{1}{2},\text{covered}}_{\boldsymbol{v},-,d} &= R(W^{n+\frac{1}{2},\text{covered}}_{\boldsymbol{v},+,d}, W^{n+\frac{1}{2}}_{\boldsymbol{v},-,d}, d) \\
F^{n+\frac{1}{2},\text{covered}}_{\boldsymbol{v},+,d} &= R(W^{n+\frac{1}{2}}_{\boldsymbol{v},+,d}, W^{n+\frac{1}{2},\text{covered}}_{\boldsymbol{v},+,d}, d)
\end{aligned} \tag{5.47}$$

9. Modify the flux with artificial viscosity where the flow is compressive.

5.5.3 Modifications for R-Z Computations

For R-Z calculations, we make some adjustments to the algorithm. Specifically, we separate the radial pressure force as a separate flux. This makes free-stream preservation in the radial direction easier to achieve. For this section, we will confine ourselves to the compressible Euler equations.

5.5.3.1 Equations of Motion

The compressible Euler equations in R-Z coordinates are given by

$$\frac{\partial U}{\partial t} + \frac{1}{r}\frac{\partial (rF^r)}{\partial r} + \frac{1}{r}\frac{\partial (rF^z)}{\partial z} + \frac{\partial H}{\partial r} + \frac{\partial H}{\partial z} = 0 \tag{5.48}$$

where

$$\begin{aligned} U &= (\rho, \rho u_r, \rho u_z, \rho E)^T \\ F^r &= (\rho u_r, \rho u_r^2, \rho u_r u_z, \rho u_r(E+p))^T \\ F^z &= (\rho u_z, \rho u_r u_z, \rho u_z^2, \rho u_z(E+p))^T \\ H &= (0, p, p, 0)^T \end{aligned} \tag{5.49}$$

5.5.3.2 Flux Divergence Approximations

In section 5.4, we describe our solution update strategy and this remains largely unchanged. Our update still takes the form of equation 5.16 and redistribution still takes the form of equation 5.18. The definitions of the divergence approximations do change, however. The volume of a full cell ΔV_j is given by

$$\Delta V_j = (j + \frac{1}{2})h^3 \tag{5.50}$$

where $(i,j) = ind^{-1}(\boldsymbol{v})$. Define $\kappa_{\boldsymbol{v}}^{vol}$ to be the real volume of the cell that the VoF occupies.

$$\kappa_{\boldsymbol{v}}^{vol} = \frac{1}{\Delta V}\int_{\Delta_{\boldsymbol{v}}} r dr dz = \frac{1}{\Delta V}\int_{\partial\Delta_{\boldsymbol{v}}} \frac{r^2}{2} n_r dl \tag{5.51}$$

$$\kappa_{\boldsymbol{v}}^{vol} = \frac{h}{2\Delta V}((\alpha r^2)_{\boldsymbol{f}(\boldsymbol{v},+,r)} - (\alpha r^2)_{\boldsymbol{f}(\boldsymbol{v},-,r)} - \alpha_B \bar{r}_{\delta\boldsymbol{v}}^2 n^r) \tag{5.52}$$

The conservative divergence of the flux in RZ is given by

$$\begin{aligned}(D \cdot \vec{F})_{\boldsymbol{v}}^c = \frac{h}{\Delta V \kappa_{\boldsymbol{v}}^{vol}}&((r\bar{F}^r\alpha)_{\boldsymbol{f}(\boldsymbol{v},+,r)} - (r\bar{F}^r\alpha)_{\boldsymbol{f}(\boldsymbol{v},-,r)} \\ &+ (\bar{r}\bar{F}^z\alpha)_{\boldsymbol{f}(\boldsymbol{v},+,z)} - (\bar{r}\bar{F}^z\alpha)_{\boldsymbol{f}(\boldsymbol{v},-,z)})\end{aligned}$$

$$\begin{aligned}\left(\frac{\partial H}{\partial r}\right)^c &= \frac{1}{\kappa_{\boldsymbol{v}} h^2}\int \frac{\partial H}{\partial r} drdz = \frac{1}{\kappa_{\boldsymbol{v}} h^2}\int H n_r dl \\ \left(\frac{\partial H}{\partial z}\right)^c &= \frac{1}{\kappa_{\boldsymbol{v}} h^2}\int \frac{\partial H}{\partial z} drdz = \frac{1}{\kappa_{\boldsymbol{v}} h^2}\int H n_z dl\end{aligned}$$

We always deal with these divergences in a form multiplied by the volume fraction κ.

$$\begin{aligned}\kappa_{\boldsymbol{v}}(D \cdot \vec{F})^c_{\boldsymbol{v}} = \frac{h\kappa_{\boldsymbol{v}}}{\Delta V \kappa^{vol}_{\boldsymbol{v}}}&((r\bar{F}^r\alpha)_{\boldsymbol{f}(\boldsymbol{v},+,r)} - (r\bar{F}^r\alpha)_{\boldsymbol{f}(\boldsymbol{v},-,r)} \\ &+(\bar{r}\bar{F}^z\alpha)_{\boldsymbol{f}(\boldsymbol{v},+,z)} - (\bar{r}\bar{F}^z\alpha)_{\boldsymbol{f}(\boldsymbol{v},-,z)})\end{aligned}$$

$$\begin{aligned}\kappa_{\boldsymbol{v}}\left(\frac{\partial H}{\partial r}\right)^c &= \frac{1}{h^2}\int H n_r dl = \frac{1}{h}((H\alpha)_{\boldsymbol{f}(\boldsymbol{v},+,r)} - (H\alpha)_{\boldsymbol{f}(\boldsymbol{v},-,r)}) \\ \kappa_{\boldsymbol{v}}\left(\frac{\partial H}{\partial z}\right)^c &= \frac{1}{h^2}\int H n_z dl = \frac{1}{h}((H\alpha)_{\boldsymbol{f}(\boldsymbol{v},+,z)} - (H\alpha)_{\boldsymbol{f}(\boldsymbol{v},-,z)})\end{aligned}$$

where $\bar{F}$ has been interpolated to face centroids where α denotes the ordinary area fraction. The nonconservative divergence of the flux in RZ is given by

$$\begin{aligned}(D \cdot \vec{F})^{nc}_{\boldsymbol{v}} = \frac{1}{hr_{\boldsymbol{v}}}&((rF^r)_{\boldsymbol{f}(\boldsymbol{v},+,r)} - (rF^r)_{\boldsymbol{f}(\boldsymbol{v},-,r)}) \\ &+\frac{1}{h}(F^z_{\boldsymbol{f}(\boldsymbol{v},+,z)} - F^z_{\boldsymbol{f}(\boldsymbol{v},-,z)})\end{aligned}$$

$$\begin{aligned}\left(\frac{\partial H}{\partial r}\right)^{nc} &= \frac{1}{h}(H_{\boldsymbol{f}(\boldsymbol{v},+,r)} - H_{\boldsymbol{f}(\boldsymbol{v},-,r)}) \\ \left(\frac{\partial H}{\partial z}\right)^{nc} &= \frac{1}{h}(H_{\boldsymbol{f}(\boldsymbol{v},+,z)} - H_{\boldsymbol{f}(\boldsymbol{v},-,z)})\end{aligned}$$

5.5.3.3 Primitive Variable Form of the Equations

In the predictor step, we use the nonconservative form of the equations of motion. See Courant and Friedrichs [6] for derivations.

$$\frac{\partial W}{\partial t} + A^r\frac{\partial W}{\partial r} + A^z\frac{\partial W}{\partial z} = S \qquad (5.53)$$

where

$$\begin{aligned}W &= (\rho, u_r, u_z, p)^T \\ S &= \left(-\rho\frac{u_r}{r}, 0, 0, -\rho c^2\frac{u_r}{r}\right)^T\end{aligned}$$

$$A^r = \left(\begin{array}{cccc} u_r & \rho & 0 & 0 \\ 0 & u_r & 0 & \frac{1}{\rho} \\ 0 & 0 & u_r & 0 \\ 0 & \rho c^2 & 0 & u_r \end{array} \right)$$

$$A^r = \left(\begin{array}{cccc} u_z & \rho & 0 & 0 \\ 0 & u_z & 0 & 0 \\ 0 & 0 & u_z & \frac{1}{\rho} \\ 0 & 0 & \rho c^2 & u_z \end{array} \right)$$

5.5.3.4 Flux Registers

Refluxing is the balancing the fluxes at coarse-fine interfaces so the coarse side of the interface is using the same flux as the integral of the fine fluxes over the same area. In this way, we maintain strong mass conservation at coarse-fine interfaces. As shown in equation, 5.5.3.2, the conservative divergence in cylindrical coordinates is has a difference form than in Cartesian coordinates. It is therefore necessary to describe the refluxing operation specifically for cylindrical coordinates.

Let $\vec{F}^{comp} = \{\vec{F}^f, \vec{F}^{c,valid}\}$ be a two-level composite vector field. We want to define a composite divergence $D^{comp}(\vec{F}^f, \vec{F}^{c,valid})_{\boldsymbol{v}}$, for $\boldsymbol{v} \in V^c_{valid}$. We do this by extending $F^{c,valid}$ to the faces adjacent to $\boldsymbol{v} \in V^c_{valid}$, but are covered by $\mathcal{F}^f_{valid}$.

$$< F^f_z >_{\boldsymbol{f}_c} = \left(\frac{\kappa_{\boldsymbol{v}_c}}{\kappa^{vol}_{\boldsymbol{v}_c} \Delta V_{\boldsymbol{v}_c}} \right) \left(\frac{h^2}{(n_{ref})^{(\mathbf{D}-1)}} \right) \sum_{\boldsymbol{f} \in \mathcal{C}^{-1}_{n_{ref}(\boldsymbol{f}_c)}} (\bar{r}\alpha)_{\boldsymbol{f}} (\bar{F}^z + \bar{H})_{\boldsymbol{f}}$$

$$< F^f_r >_{\boldsymbol{f}_c} = \left(\frac{\kappa_{\boldsymbol{v}_c}}{\kappa^{vol}_{\boldsymbol{v}_c} \Delta V_{\boldsymbol{v}_c}} \right) \left(\frac{h^2}{(n_{ref})^{(\mathbf{D}-1)}} \right) \sum_{\boldsymbol{f} \in \mathcal{C}^{-1}_{n_{ref}(\boldsymbol{f}_c)}} (r\alpha)_{\boldsymbol{f}} (F^r + H)_{\boldsymbol{f}}$$

$$F^c_{r,\boldsymbol{f}_c} = \left(\frac{\kappa_{\boldsymbol{v}_c}}{\kappa^{vol}_{\boldsymbol{v}_c} \Delta V_{\boldsymbol{v}_c}} \right) (h^2 (r\alpha)_{\boldsymbol{f}_c}) (F^r + H)_{\boldsymbol{f}_c}$$

$$F^c_{z,\boldsymbol{f}_c} = \left(\frac{\kappa_{\boldsymbol{v}_c}}{\kappa^{vol}_{\boldsymbol{v}_c} \Delta V_{\boldsymbol{v}_c}} \right) (h^2 (\bar{r}\alpha)_{\boldsymbol{f}_c}) (\bar{F}^z + \bar{H})_{\boldsymbol{f}_c}$$

$$\boldsymbol{f}_c \in ind^{-1}(\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d), \boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d \in \zeta^f_{d,+} \cup \zeta^f_{d,-}$$

$$\zeta^f_{d,\pm} = \{\boldsymbol{i} \pm \frac{1}{2}\boldsymbol{e}^d : \boldsymbol{i} \pm \boldsymbol{e}^d \in \Omega^c_{valid}, \boldsymbol{i} \in \mathcal{C}_{n_{ref}}(\Omega^f)\}$$

The VoF $\boldsymbol{v}_c$ is the coarse volume that is adjacent to the coarse-fine interface and $r_{\boldsymbol{v}_c}$ is the radius of its cell center. Then we can define $(D \cdot \vec{F})_{\boldsymbol{v}}, \boldsymbol{v} \in \mathcal{V}^c_{valid}$, using the expression above, with $\tilde{F}_{\boldsymbol{f}} =< F^f_d >$ on faces covered by $\mathcal{F}^f$. We can express the composite divergence in terms of a level divergence, plus a correction. We define a flux register $\delta\vec{F}^f$, associated with the fine level

$$\delta\vec{F}^f = (\delta F^f_{0,\dots}\delta F^f_{D-1})$$
$$\delta F^f_d : ind^{-1}(\zeta^f_{d,+} \cup \zeta^f_{d,-}) \rightarrow \mathbb{R}^m$$

If $\vec{F}^c$ is any coarse level vector field that extends $\vec{F}^{c,valid}$, i.e. $F^c_d = F^{c,valid}_d$ on $\mathcal{F}^{c,d}_{valid}$ then for $\boldsymbol{v} \in \mathcal{V}^c_{valid}$

$$D^{comp}(\vec{F}^f, \vec{F}^{c,valid})_{\boldsymbol{v}} = (D\vec{F}^c)_{\boldsymbol{v}} + D_R(\delta\vec{F}^c)_{\boldsymbol{v}} \tag{5.54}$$

Here $\delta\vec{F}^f$ is a flux register, set to be

$$\delta F^f_d = < F^f_d > - F^c_d \text{ on } ind^{-1}(\zeta^c_{d,+} \cup \zeta^c_{d,-}) \tag{5.55}$$

D_R is the reflux divergence operator. For valid coarse vofs adjacent to Ω^f it is given by

$$\kappa_{\boldsymbol{v}}(D_R\delta\vec{F}^f)_{\boldsymbol{v}} = \sum_{d=0}^{D-1}\Big(\sum_{\boldsymbol{f}:\boldsymbol{v}=\boldsymbol{v}^+(\boldsymbol{f})} \delta F^f_{d,\boldsymbol{f}} - \sum_{\boldsymbol{f}:\boldsymbol{v}=\boldsymbol{v}^-(\boldsymbol{f})} \delta F^f_{d,\boldsymbol{f}}\Big) \tag{5.56}$$

For the remaining vofs in $\mathcal{V}^f_{valid}$,

$$(D_R\delta\vec{F}^f) \equiv 0 \tag{5.57}$$

We then add the reflux divergence to adjust the coarse solution U^c to preserve conservation.

$$U^c_{\boldsymbol{v}} += \kappa_{\boldsymbol{v}}(D_R(\delta F))_{\boldsymbol{v}} \tag{5.58}$$

5.5.4 Artificial Viscosity

The artificial viscosity coefficient is K_0, the velocity is $\vec{u}$ and $d = dir(\boldsymbol{f})$.

$$\begin{aligned}
(D\vec{u})_{\boldsymbol{f}} &= (u^d_{\boldsymbol{v}^+(\boldsymbol{f})} - u^d_{\boldsymbol{v}^-(\boldsymbol{f})}) + \sum_{d' \neq d} \frac{1}{2}(\Delta^{d'} u^{d'}_{\boldsymbol{v}^+(\boldsymbol{f})} + \Delta^{d'} u^{d'}_{\boldsymbol{v}^-(\boldsymbol{f})}) \\
K_{\boldsymbol{f}} &= K_0 \text{ max}(-(D\vec{u})_{\boldsymbol{f}}, 0) \\
F^{n+\frac{1}{2}}_{\boldsymbol{f}} &= F^{n+\frac{1}{2}}_{\boldsymbol{f}} - K_{\boldsymbol{f}}(U^n_{\boldsymbol{v}^+(\boldsymbol{f})} - U^n_{\boldsymbol{v}^-(\boldsymbol{f})}) \\
F^{\text{covered}}_{\boldsymbol{v},\pm,d} &= F^{\text{covered}}_{\boldsymbol{v},\pm,d} - K_{\boldsymbol{f}}(U^n_{\boldsymbol{v}^+(\boldsymbol{f})} - U^n_{\boldsymbol{v}^-(\boldsymbol{f})})
\end{aligned}$$

We modify the covered face with the same divergence used in the adjacent uncovered face.

$$\begin{aligned}
F^{\text{covered}}_{\boldsymbol{v},\pm,d} =& F^{\text{covered}}_{\boldsymbol{v},\pm,d} - K_{\boldsymbol{f}}(U^n_{\boldsymbol{v}^+(\boldsymbol{f})} - U^n_{\boldsymbol{v}^-(\boldsymbol{f})}) \\
\boldsymbol{f} =& \boldsymbol{f}(\boldsymbol{v}, \mp, d)
\end{aligned}$$

This has the effect of negating the effect of artificial viscosity on the non-conservative divergence of the flux at irregular cells. We describe later that the solid wall boundary condition at the embedded boundary is also modified with artificial viscosity.

5.6 Slope Calculation

We will use the 4th order slope calculation in Colella and Glaz [3] combined with characteristic limiting.

$$
\begin{aligned}
\Delta^d W_{\boldsymbol{v}} &= \zeta_{\boldsymbol{v}}\ \widetilde{\Delta}^d W_{\boldsymbol{v}} \\
\widetilde{\Delta}^d W_{\boldsymbol{v}} &= \Delta^{vL}(\Delta^B W_{\boldsymbol{v}}, \Delta^L W_{\boldsymbol{v}}, \Delta^R W_{\boldsymbol{v}}) \mid \Delta_2^d W_{\boldsymbol{v}} \mid \Delta_2^d W_{\boldsymbol{v}} \\
\Delta_2^d W_{\boldsymbol{v}} &= \Delta^{vL}(\Delta^C W_{\boldsymbol{v}}, \Delta^L W_{\boldsymbol{v}}, \Delta^R W_{\boldsymbol{v}}) \mid \Delta^{VLL} W_{\boldsymbol{v}} \mid \Delta^{VLR} W_{\boldsymbol{v}} \\
\Delta^B W_{\boldsymbol{v}} &= \frac{2}{3}((W - \frac{1}{4}\Delta_2^d W) \ll \boldsymbol{e}^d)_{\boldsymbol{v}} - ((W + \frac{1}{4}\Delta_2^d W) \ll -\boldsymbol{e}^d)_{\boldsymbol{v}}) \\
\Delta^C W_{\boldsymbol{v}} &= \frac{1}{2}((W^n \ll \boldsymbol{e}^d)_{\boldsymbol{v}} - (W^n \ll -\boldsymbol{e}^d)_{\boldsymbol{v}}) \\
\Delta^L W_{\boldsymbol{v}} &= W^n_{\boldsymbol{v}} - (W^n \ll -\boldsymbol{e}^d)_{\boldsymbol{v}} \\
\Delta^R W_{\boldsymbol{v}} &= (W^n \ll \boldsymbol{e}^d)_{\boldsymbol{v}} - W^n_{\boldsymbol{v}} \\
\Delta^{3L} W_{\boldsymbol{v}} &= \frac{1}{2}(3W^n_{\boldsymbol{v}} - 4(W^n \ll -\boldsymbol{e}^d)_{\boldsymbol{v}} + (W^n \ll -2\boldsymbol{e}^d)_{\boldsymbol{v}}) \\
\Delta^{3R} W_{\boldsymbol{v}} &= \frac{1}{2}(-3W^n_{\boldsymbol{v}} + 4(W^n \ll \boldsymbol{e}^d)_{\boldsymbol{v}} - (W^n \ll 2\boldsymbol{e}^d)_{\boldsymbol{v}}) \\
\Delta^{VLL} W_{\boldsymbol{v}} &= \begin{cases} \text{min}(\Delta^{3L} W_{\boldsymbol{v}}, \Delta^L_{\boldsymbol{v}}) & \text{if } \Delta^{3L} W_{\boldsymbol{v}} \cdot \Delta^L W_{\boldsymbol{v}} > 0 \\ 0 & \text{otherwise} \end{cases} \\
\Delta^{VLR} W_{\boldsymbol{v}} &= \begin{cases} \text{min}(\Delta^{3R} W_{\boldsymbol{v}}, \Delta^R_{\boldsymbol{v}}) & \text{if } \Delta^{3R} W_{\boldsymbol{v}} \cdot \Delta^R W_{\boldsymbol{v}} > 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

At domain boundaries, $\Delta^L W_{\boldsymbol{v}}$ and $\Delta^R W_{\boldsymbol{v}}$ may be overwritten by the application. There are two versions of the van Leer limiter $\Delta^{vL}(\delta W_C, \delta W_L, \delta W_R)$ that are commonly used. One is to apply a limiter to the differences in characteristic variables.

1. Compute expansion of one-sided and centered differences in characteristic variables.

$$\alpha^k_L = l^k \cdot \delta W_L \tag{5.59}$$

$$\alpha^k_R = l^k \cdot \delta W_R \tag{5.60}$$

$$\alpha^k_C = l^k \cdot \delta W \tag{5.61}$$

2. Apply van Leer limiter

$$\alpha^k = \begin{cases} \text{min}(2\,|\,\alpha^k_L\,|,\ 2\,|\,\alpha^k_R\,|,\ |\,\alpha^k_C\,|) & \text{if } \alpha^k_L \cdot \alpha^k_R > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.62}$$

3. $\Delta^{vL} = \sum_k \alpha^k r^k$

Here, $l^k = l^k(W^n_{\boldsymbol{i}})$ and $r^k = r^k(W^n_{\boldsymbol{i}})$.

For a variety of problems, it suffices to apply the van Leer limiter componentwise to the differences. Formally, this can be obtain from the more general case above by taking the matrices of left and right eigenvectors to be the identity.

5.6.1 Flattening

Finally, we give the algorithm for computing the flattening coefficient $\zeta_{\boldsymbol{i}}$. We assume that there is a quantity corresponding to the pressure in gas dynamics (denoted here as p) which can act as a steepness indicator, and a quantity corresponding to the bulk modulus (denoted here as K, given as γp in a gas), that can be used to non-dimensionalize differences in p.

$$\zeta_{\boldsymbol{v}} = \begin{cases} \underset{0 \le d < \mathbf{D}}{min} \zeta^d_{\boldsymbol{v}} & \text{if } \sum_{d=0}^{\mathbf{D}-1} \Delta^d_1 u^d_{\boldsymbol{v}} < 0 \\ 1 & \text{otherwise} \end{cases} \tag{5.63}$$
$$\zeta^d_{\boldsymbol{v}} = min_3(\widetilde{\zeta}^d, d)_{\boldsymbol{v}}$$
$$\widetilde{\zeta}^d_{\boldsymbol{v}} = \eta(\Delta^d_1 p_{\boldsymbol{v}},\, \Delta^d_2 p_{\boldsymbol{v}},\, min_3(K, d)_{\boldsymbol{v}})$$
$$\Delta^d_1 p_{\boldsymbol{v}} = \Delta^C p_{\boldsymbol{v}} \mid \Delta^L p_{\boldsymbol{v}} \mid \Delta^R p_{\boldsymbol{v}}$$
$$\Delta^d_2 p_{\boldsymbol{v}} = (\Delta^d_1 p \ll \boldsymbol{e}^d)_{\boldsymbol{v}} + (\Delta^d_1 p \ll -\boldsymbol{e}^d)_{\boldsymbol{v}} \mid 2\Delta^d_1 p_{\boldsymbol{v}} \mid 2\Delta^d_1 p_{\boldsymbol{v}}$$

The functions min_3 and ζ are given below.

$$min_3(q, d)_{\boldsymbol{v}} = \text{min}((q \ll \boldsymbol{e}^d)_{\boldsymbol{v}}, q_{\boldsymbol{v}}, (q \ll -\boldsymbol{e}^d)_{\boldsymbol{v}}) \mid \text{min} q_{\boldsymbol{v}}, (q \ll -\boldsymbol{e}^d)_{\boldsymbol{v}}) \mid \text{min}(q \ll \boldsymbol{e}^d)_{\boldsymbol{v}}, q_{\boldsymbol{v}})$$
$$\zeta(\delta p_1, \delta p_2, p_0) = \begin{cases} 0 & \text{if } \frac{|\delta p_1|}{p_0} > d \text{ and } \frac{|\delta p_1|}{|\delta p_2|} > r_1 \\ 1 - \frac{\frac{|\delta p_1|}{|\delta p_2|} - r_0}{r_1 - r_0} & \text{if } \frac{|\delta p_1|}{p_0} > d \text{ and } r_1 \ge \frac{|\delta p_1|}{|\delta p_2|} > r_0 \\ 1 & \text{otherwise} \end{cases}$$
$$r_0 = 0.75,\ r_1 = 0.85,\ d = 0.33 \tag{5.64}$$

Note that min_3 is not the minimum over all available VoFs but involves the minimum of shifted VoFs which includes an averaging operation.

5.7 Computing fluxes at the irregular boundary

The flux at the embedded boundary is centered at the centroid of the boundary $\bar{\mathbf{x}}$. We extrapolate the primitive solution in space from the cell center. We then transform to the conservative solution and extrapolate in time using the stable, non-conservative estimate of the flux divergence described in equation 5.14.

$$W_{\boldsymbol{v},B} = W^n_{\boldsymbol{v}} + \sum_{d=0}^{D-1}(\bar{x}_d \Delta^d W^n_{\boldsymbol{v}} - \frac{\Delta t}{2\Delta x} A^d \Delta^d W^n_{\boldsymbol{v}}) \tag{5.65}$$

$$F^{n+\frac{1}{2}}_{\boldsymbol{v},B} = R_B(U^{n+\frac{1}{2}}_{\boldsymbol{v},B}, \boldsymbol{n}^B_{\boldsymbol{v}}) \tag{5.66}$$

For polytropic gas dynamics, this becomes

$$\begin{aligned}
\rho_{\boldsymbol{v},B} =& \rho^n_{\boldsymbol{v}} + \sum_{d=0}^{D-1}(\bar{x}_d \Delta^d \rho^n_{\boldsymbol{v}} - \frac{\Delta t}{2\Delta x}(u^d \Delta^d \rho + \rho \Delta^d u^d)^n_{\boldsymbol{v}}) \\
p_{\boldsymbol{v},B} =& p^n_{\boldsymbol{v}} + \sum_{d=0}^{D-1}(\bar{x}_d \Delta^d p^n_{\boldsymbol{v}} - \frac{\Delta t}{2\Delta x}(u^d \Delta^d p + \rho c^2 \Delta^d u^d)^n_{\boldsymbol{v}}) \\
u^{d1}_{\boldsymbol{v},B} =& (u^{d1})^n_{\boldsymbol{v}} + \sum_{d=0}^{D-1}(\bar{x}_d \Delta^d (u^{d1})^n_{\boldsymbol{v}}) \\
&- \frac{\Delta t}{2\Delta x}(u^{d1} \Delta^{d1} u^{d1} + \frac{1}{\rho}\Delta^{d1} p)^n_{\boldsymbol{v}} \\
&- \frac{\Delta t}{2\Delta x}(\sum_{d2 \neq d1}(u^{d2} \Delta^{d2} u^{d1}))
\end{aligned} \tag{5.67}$$

If we are using solid-wall boundary conditions at the irregular boundary, we calculate an approximation of the divergence of the velocity at the irregular cell $D(\vec{u})_{\boldsymbol{v}}$ and use it to modify the flux to be consistent with artificial viscosity. The d-direction momentum flux at the irregular boundary is given by $-p^r n^d$ where p^r is the pressure to emerge from the Riemann solution in equation 5.67. For artificial viscosity, we modify this flux as follows.

$$\begin{aligned}
(D\vec{u})_{\boldsymbol{v}} &= \sum_{d'=0}^{D-1} \Delta^{d'} u^{d'}_{\boldsymbol{v}} \\
p^r &= p^r - 2K_0 \,\mathsf{max}(-(D\vec{u})_{\boldsymbol{v}}, 0)\vec{u} \cdot \hat{n}
\end{aligned}$$

5.8 Class Hierarchy

The principal EBAMRGodunov classes follow.

- `EBAMRGodunov`, the `AMRLevel`-derived class which is driven by the `AMR` class.
- `EBLevelGodunov`, a class owned by `AMRGodunov`. `EBLevelGodunov` advances the solution on a level and can exist outside the context of an AMR hierarchy. This class makes possible Richardson extrapolation for error estimation.

- EBPatchGodunov, is a base class which encapsulates the operations required to advance a solution on a single patch.
- `EBPhysIBC` is a base class which encapsulates initial conditions and flux-based boundary conditions.

5.8.1 Class `EBAMRGodunov`

`EBAMRGodunov` is the `AMRLevel`-derived class with which the `AMR` class will directly interact. Its user interface is therefore constrained by the `AMRLevel` interface. The important data members of the `EBAMRGodunov` class are as follows.

- `LevelData<EBCellFAB> m_state_old, m_state_new;`

  The state data at old and new times. Both need to be kept because subcycling in time requires temporal interpolation.

- `Real m_cfl, m_dx;`

  CFL number and grid spacing for this level.

- `EBPWLFineInterp m_fine_interp;`

  Interpolation operator for refining data during regridding that were previously only covered by coarser data.

- `EBCoarseAverage m_coarse_average;`

  This is the averaging operator which replaces data on coarser levels with the average of the data on this level where they coincide in space.

- `RefCountedPtr<EBPhysIBC> m_phys_ibc_ptr;`

  This boundary condition operator provides flux-based boundary data at domain boundaries and also provides initial conditions.

The `EBAMRGodunov` implementation of the `AMRLevel` currently does the following for each of the important interface functions.

- `Real EBAMRGodunov::advance()`

  This function advances the conservative state by one time step. It calls the `EBLevelGodunov::step` function. The timestep returned by that function is stored in member data.

- `void  EBAMRGodunov::postTimeStep()`

  This function calls refluxing from the next finer level and averages its solution to the next finer level.

- `void regrid(const Vector<Box>& a_new_grids)`

  This function changes the union of rectangles over which the data is defined. At places where the two sets of rectangles intersect, the data is copied from the previous set of rectangles. At places where there was only data from the next coarser level, piecewise linear interpolation is used to fill the data.

- `void initialData()`

  In this function the initial state is filled by calling `m_phys_ibc_ptr->initialize`.

- `void computeDt()`

  This function returns the timestep stored during the `advance()` call.

- `void computeInitialDt()`

  This function calculates the time step using the maximum wavespeed returned by a `EBLevelGodunov::getMaxWaveSpeed` call. Define the maximum wavespeed to be w and the initial timestep multiplier to be K and the grid spacing at this level to be h,

  $$\Delta t = \frac{Kh}{w}. \tag{5.68}$$

- `DisjointBoxLayout loadBalance(const Vector<Box>& a_grids)`

  Calls the Chombo load balancer to create the returned layout.

5.8.2 Class `EBLevelGodunov`

`EBLevelGodunov` is a class owned by `AMRGodunov`. `EBLevelGodunov` advances the solution on a level and can exist outside the context of an AMR hierarchy. This class makes possible Richardson extrapolation for error estimation. The important functions of the public interface of `EBLevelGodunov` follow.

-
  ```
  void define(const DisjointBoxLayout& a_thisDBL,
              const DisjointBoxLayout& a_coarDBL,
              const EBISLayout&        a_thisEBISL,
              const EBISLayout&        a_coarEBISL,
              const RedistSTencil&     a_redStencil,
              const Box&               a_DProblem,
              const int&               a_numGhost,
              const int&               a_nRefine,
              const Real&              a_dx,
              const EBPatchGodunov* const a_integrator,
              const bool& a_hasCoarser,
              const bool& a_hasFiner);
  ```

  Define the internal data structures. For the coarsest level, an empty DisjointBoxLayout is passed in for coarserDisjointBoxLayout.

  - `a_thisDBL, a_coarDBL` The layouts at this level and the next coarser level. For the coarsest level, an empty `DisjointBoxLayout` is passed in for `coarDBL`.
  - `a_DProblem, a_dx` The problem domain and grid spacing at this level.
  - `a_nRefine` The refinement ratio between this level and the next coarser level.
  - `a_numGhost` The number of ghost cells (assumed to be isotropic) required to advance the solution.
  - `a_bc` Boundary conditions and initial conditions are encapsulated in this object.

-
  ```
  Real step(LevelData<EBCellFAB>&          a_U,
            LevelData<BaseIVFAB<Real> >&  a_massDiff,
            EBFluxRegister&               a_coarFluxRegister,
            EBFluxRegister&               a_fineFluxRegister
            const LevelData<EBCellFAB>&   a_UCoarseOld,
            const LevelData<EBCellFAB>&   a_UCoarseNew,
            const Real&                   a_time,
            const Real&                   a_TCold,
            const Real&                   a_TCNew,
            const Real&                   a_dt);
  ```

  Advance the solution at this timeStep for one time step.

  - `a_UCoarseOld, a_UCoarseNew` The solution at the next coarser level at the old and new coarse times.
  - `a_time, a_TCold, a_TCNew` The time of this solution (before the advance) and the old and new coarse solution times.
  - `a_dt` The time step at this level.
  - `a_U` The solution at this level.
  - `a_massDiff` Redistribution mass.
  - `a_coarFluxRegister, a_fineFluxRegisters` The flux registers between this level and the adjacent levels.

- `Real getMaxWaveSpeed(const LevelData<EBCellFAB>& a_state);`

  Return the maximum wave speed of input `a_state` for purposes of limiting the time step.

5.8.3 Class `EBPatchGodunov`

The base class `EBPatchGodunov` provides a skeleton for the application-dependent pieces of a second-order unsplit Godunov method. The virtual functions are called by `EBLevelGodunov`, which manages the overall assembly of the second-order unsplit fluxes. As part of `EBPatchGodunov`, we provide some member functions (slope, flattening), that we expect to be useful across applications, but require either virtual functions or parameter information by the user.

There are three types of grid variables that appear in the unsplit Godunov method in section (**??**): conserved quantities, primitive variables, and fluxes, denoted below by U, q, F, respectively. It is often convenient to have the number of components for primitive variables and for fluxes exceed that for conserved quantities. In the case of primitive variables, redundant quantities are carried that parameterize the equation of state in order to avoid multiple calls to that function. In the case of fluxes, it is often convenient to split the flux for some variables into multiple components, e.g., dividing the momentum flux into advective and pressure terms. The API given here provides the flexibility to support these various options.
Construction Methods:

- `void setPhysIBC(RefCountedPtr<EBPhysIBC> a_bc)`

  Set the boundary condition pointer of the integrator.

-
  ```
  virtual void define(
               const Box& a_domain,
               const Real& a_dx);
  ```

  Set the domain variables for this level.

- `virtual EBPatchGodunov* new_patchGodunov = 0;`

  Factory method. Return pointer to new `PatchGodunov` object with its boundary conditions defined.

`EBLevelGodunov` API: (Translation: these are the only things that actually get called by `EBLevelGodunov`.

-
  ```
  virtual void
  regularUpdate(EBCellFAB&       a_consState,
                EBFluxFAB&       a_flux,
                BaseIVFAB<Real>& a_nonConservativeDivergence,
                const EBCellFAB& a_source,
                const Box&       a_box);
  ```

  Update the state using flux difference that ignores EB. Store fluxes used in this update Store non-conservative divergence. Flux coming out of this this should exist at cell face centers.

-
  ```
  interpolateFluxToCentroids(BaseIFFAB<Real>              a_centroidFlux[SpaceDim],
                             const BaseIFFAB<Real>* const a_fluxInterpolant[SpaceDim],
                             const IntVectSet&            a_irregIVS);
  ```

  Interpolates cell-face centered fluxes to centroids over irregular cells. Flux going into this should exist at cell face centers.

-
  ```
  virtual void
  irregularUpdate(EBCellFAB&             a_consState,
                  Real&                  a_maxWaveSpeed,
                  BaseIVFAB<Real>&       a_massDiff,
                  const BaseIFFAB<Real>  a_centroidFlux[SpaceDim],
                  const BaseIVFAB<Real>& a_nonConservativeDivergence,
                  const Box&             a_box,
                  const IntVectSet&      a_ivs);
  ```
  Update the state at irregular VoFs and compute mass difference and the maximum wave speed over the entire box. Flux going into this should exist at VoF centroids.

-
  ```
  virtual Real getMaxWaveSpeed(
               const EBCellFAB& a_U,
               const Box& a_box)= 0;
  ```
  Return the maximum wave speed on over this patch.

-
  ```
  void setValidBox(const Box& a_validBox,
                   const EBISBox& a_ebisbox,
                   const Real& a_time,
                   const Real& a_dt);
  ```
  Set the valid box of the patch.

Virtual interface:

-
  ```
  virtual void consToPrim(EBCellFAB&       a_primState,
                          const EBCellFAB& a_conState) = 0;
  ```
  Compute the primitive state given the conserved state. $W_i = W(U_i)$.

-
  ```
  virtual void incrementWithSource(
                               EBCellFAB&       a_primState,
                               const EBCellFAB& a_source,
                               const Real& a_scale,
                               const Box&  a_box) = 0;
  ```
  Increment the primitive variables by the source term, as in (5.34). `a_scale = 0.5*dt`.

-
  ```
  virtual void normalPred(EBCellFAB&       a_qlo,
                          EBCellFAB&       a_qhi,
                          const EBCellFAB& a_q,
                          const EBCellFAB& a_dq,
                          const Real&      a_scale,
                          const int&       a_dir,
                          const Box&       a_box) = 0;
  ```
  Extrapolate in the low and high direction from q, as in (5.34). A default implementation is provided which assumes the existence of the virtual functions `limit`.

-
  ```
  virtual void riemann(EBFaceFAB&       a_flux,
                       const EBCellFAB& a_qleft,
                       const EBCellFAB& a_qright,
                       const int&        a_dir,
                       const Box&       a_box) = 0;
  virtual void riemann(BaseIVFAB<Real>&       a_coveredFlux,
                       const BaseIVFAB<Real>& a_extendedState,
                       const EBCellFAB&       a_primState,
                       const IntVecSet&       a_coveredFace,
                       const int&             a_dir,
                       const Side::LoHiSide&  a_sd) = 0;
  ```

  Given input left and right states, compute a suitably-upwinded flux (e.g. by solving a Riemann problem), as in equation 5.41.

-
  ```
  virtual void updateCons(EBCellFAB&             a_conState,
                          const EBFaceFAB&       a_flux,
                          const BaseIVFAB<Real>& a_coveredFluxMinu,
                          const BaseIVFAB<Real>& a_coveredFluxPlus,
                          const IntVecSet&       a_coveredFaceMinu,
                          const IntVecSet&       a_coveredFacePlus,
                          const int&             a_dir,
                          const Box&             a_box,
                          const Real&            a_scale) = 0;
  ```

  Given the value of the flux, update the conserved quantities and modify in place the flux for the purpose of passing it to a `EBFluxRegister`.

  ```
  consstate_i +=a_scale*(flux_i-1/2 - flux_i+1/2)
  ```

  .

-
  ```
  virtual void updatePrim(EBCellFAB&             a_qminus,
                          EBCellFAB&             a_qplus,
                          const EBFaceFAB&       a_flux,
                          const BaseIVFAB<Real>& a_coveredFluxMinu,
                          const BaseIVFAB<Real>& a_coveredFluxPlus,
                          const IntVecSet&       a_coveredFaceMinu,
                          const IntVecSet&       a_coveredFacePlus,
                          const int&             a_dir,
                          const Box&             a_box,
                          const Real&            a_scale) = 0;
  ```

  Given `a_flux`, the value of the flux in the direction `a_dir`, update `q_plus`, `q_minus`, the extrapolated primitive quantities, as in (**??**,5.29,5.30).

  ```
  primstate_i += a_scale*Grad_W U(flux_i-1/2 - flux_i+1/2)
  ```

-
  ```
  virtual void applyLimiter(EBCellFAB&       a_dq,
                            const EBCellFAB& a_dql,
                            const EBCellFAB& a_dqr,
                            const int&       a_dir,
                            const Box&       a_box) = 0;
  ```

  Given left and right one-sided undivided differences `a_dql`,`a_dqr`, apply van Leer limiter vL defined in section (5.6) to `a_dq`. Called by the default implementation of `EBPatchGodunov::slope`.

- `virtual int numPrimitives() const = 0;`

  Returns number of components for primitive variables.

- `virtual int numFluxes() const = 0;`

  Returns number of components for flux variables.

- `virtual int numConserved() const = 0;`

  Returns number of components for conserved variables.

- `virtual Interval velocityInterval() const = 0;`

  Returns the interval of component indices in the primitive variable `EBCellFAB` for the velocities.

- `virtual int pressureIndex() const = 0;`

  Returns the component index for the pressure. Called only if flattening is used.

- `virtual int bulkModulusIndex() const = 0;`

  Returns the component index for the bulk modulus, used as a normalization to measure shock strength in flattening. Called only if flattening is used.

- `virtual Real artificialViscosityCoefficient() const = 0;`

  Returns value of artificial viscosity. Called only if artificial viscosity is being used.

Useful member functions:

-
  ```
  void slope(EBCellFAB&       a_dq,
             const EBCellFAB& a_q,
             const EBCellFAB& a_flattening,
             int              a_dir,
             const Box&       a_box) const;
  ```

  Compute the limited slope a_dq of the primitive variables a_q for the components in the interval `a_interval`, using the algorithm described in (5.6). Calls user-supplied `EBPatchGodunov::applyLimiter`.

- `void getFlattening(const EBCellFAB& a_q);`

  Computes the flattening coefficient (5.63) and stores it in the member data `m_flatcoef`. Called from `EBPatchGodunov::slope`, if required.

5.8.4 Class `EBPhysIBC`

`EBPhysIBC` is an interface class owned and used by PatchGodunov through which a user specifies the initial and boundary of conditions of her particular problem. These boundary conditions are flux-based. `EBPhysIBC` contains as member data the mesh spacing (`Real a_dx`) and the domain of computation (`ProblemDomain a_domain`). The important user functions of `EBPhysIBC` are as follows.

-
  ```
  virtual void define(const Box&  a_domain
                      const Real& a_dx) = 0;
  ```

  Define the internals of the class.

- `virtual EBPhysIBC* new_ebphysIBC() = 0;`

  Factory method. Return a new `EBPhysIBC` object.

-
  ```
  virtual void fluxBC(EBFaceFAB& a_flux,
                      const EBCellFAB& a_Wextrap,
                      const EBCellFAB& a_Wcenter,
                      const int& a_dir,
                      const Side::LoHiSide& a_side,
                      const Real& a_time) = 0;
  ```

  Enforce the flux boundary condition on the boundary of the domain and place the result in `a_flux`. The arguments to this function are as follows

  – `a_flux` is the array of the fluxes over the box. This values in the array that correspond to the boundary faces of the domain are to be replaced with boundary values.

  – `a_Wextrap` is the extrapolated value of the state's primitive variables. This data is cell-centered.

  – `a_Wcenter` is the cell-centered value of the primitive variables at the start of the time step. This data is cell-centered.

  – `a_dir, a_side` is the direction normal and the side of the domain where the function will be enforcing boundary conditions.

  – `a_time` is the time at which boundary conditions will be imposed.

- `virtual void initialize(LevelData<FArrayBox>& a_conState);`

  Fill the input with the initial conserved variable data of the problem.

-
  ```
  void
  setBndrySlopes(EBCellFAB&       a_deltaPrim,
                 const EBCellFAB& a_primState,
                 const int&       a_dir)
  ```
  Set the slopes at domain boundaries as described in section 5.6.

Bibliography

[1] J. B. Bell, P. Colella, and H. M. Glaz. A second-order projection method for the incompressible Navier-Stokes equations. *J. Comput. Phys.*, 85:257–283, 1989.

[2] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53:484–512, March 1984.

[3] P. Colella and H. M. Glaz. Efficient solution algorithms for the Riemann problem for real gases. *J. Comput. Phys.*, 59:264, 1985.

[4] P. Colella, D. T. Graves, N.D. Keen, T. J. Ligocki, D. F. Martin, P.W. McCorquodale, D. Modiano, P.O. Schwartz, T.D. Sternberg, and B. Van Straalen. Chombo Software Package for AMR Applications - Design Document. unpublished, 2000.

[5] Phillip Colella. Multidimensional upwind methods for hyperbolic conservation laws. *J. Comput. Phys.*, 87:171–200, 1990.

[6] R. Courant and K. O. Friedrichs. *Supersonic Flow and Shock Waves*. NYU, New York, NY, 1948.

[7] Denis Gueyffier, Jie Li, Ali Nadim, Ruben Scardovelli, and Stephane Zaleski. Volume-of-fluid interface tracking with smooth surface stress methods for three dimensional flows. *J. Comput. Phys.*, 152:423–456, 1999.

[8] H. Johansen and P. Colella. A Cartesian grid embedded boundary method for Poisson's equation on irregular domains. *J. Comput. Phys.*, 147(2):60–85, December 1998.

[9] Hans Johansen and Phillip Colella. A cartesian grid embedded boundary method for Poisson's equation on irregular domains. *J. Comput. Phys.*, 1998.

[10] Hans Svend Johansen. *Cartesian Grid embedded Boundary Finite Difference Methods for Elliptic and Parabolic Partial Differential Equations on Irregular Domains*. PhD thesis, University of California, Berkeley, 1997.

[11] D. F. Martin and K. L. Cartwright. Solving Poisson's equation using adaptive mesh refinement. *Technical Report UCB/ERI M96/66 UC Berkeley*, 1996.

[12] D. Modiano and P. Colella. A higher-order embedded boundary method for time-dependent simulation of hyperbolic conservation laws. In *ASME 2000 Fluids Engineering Division Summer Meeting*, 2000.

[13] Jeff Saltzman. An unsplit 3d upwind method for hyperbolic conservation laws. *J. Comput. Phys.*, 115:153–168, 1994.

[14] P. Schwartz, M. Barad, P. Colella, and T. Ligocki. A Cartesian grid embedded boundary method for the heat equation and Poisson's equation in three dimensions. *J. Comput. Phys.*, 211(2):531–550, January 2006.

[15] E.H. Twizell, A.B. Gumel, and M.A. Arigu. Second-order, l_0-stable methods for the heat equation with time-dependent boundary conditions. *Advances in Computational Mathematics*, 6:333–352, 1996.