EBChombo Software Package for Cartesian Grid, Embedded Boundary Applications

P. Colella
D. T. Graves
T. J. Ligocki
D. Modiano
B. Van Straalen

Applied Numerical Algorithms Group
NERSC Division
Lawrence Berkeley National Laboratory
Berkeley, CA

April 16, 2003

Contents

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 [CGL+00]. 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 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.

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 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" discretizations of a flux divergence $\nabla \cdot \vec{F}$ is of the form:

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

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

$$\nabla \cdot \vec{F} = \Delta \phi = \rho \text{ on } \Omega \ . \tag{2}$$

$$\frac{\partial \phi}{\partial n} = 0 \text{ on } \partial\Omega \tag{3}$$

The volume-of fluid description reduces the problem to finding sufficiently accurate gradients at the apertures. See Johansen and Colella [JC98] for a complete description of solving Poisson's equation on embedded boundaries. Hyperbolic conservation laws can be

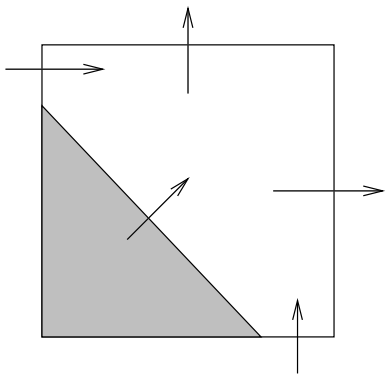

Figure 1: Embedded boundary cell. The grey area represents the region excluded from the solution domain and the arrows represent fluxes.

solved using similar divergence examples. See Modiano and Colella [MC00] for such an algorithm. Gueyffier, et. al. [GLN$^+$99] 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 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 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 require grid coarsening (multigrid, for example) also produce grids where such complex connectivity occurs. The connectivity can become arbitrarily complex (see figure 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

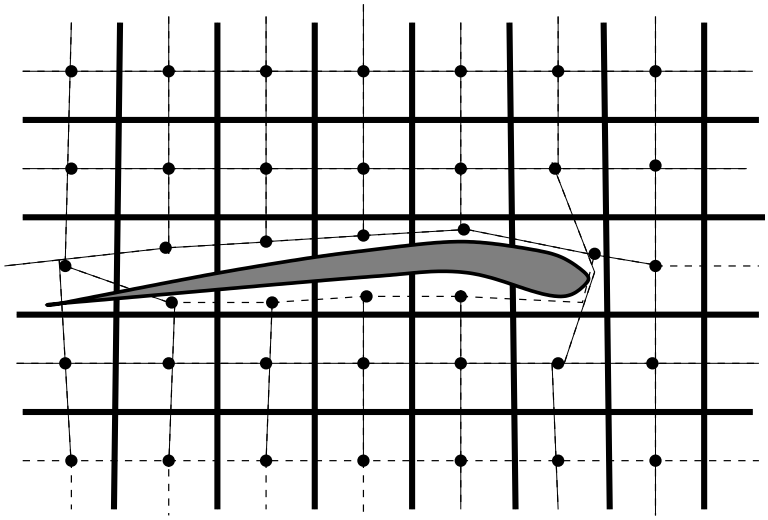

Figure 2: Example of embedded boundary description of an interface. The interior of the cartoon airfoil represents the area excluded from the domain and the lines connecting the cell centers represent the graph connectivity of the discrete domain.

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 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 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 which grid faces the embedded boundary intersects, the irregular cell can be a pentagon, a trapezoid, or a triangle, as shown in figure 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 $\hat{n}$

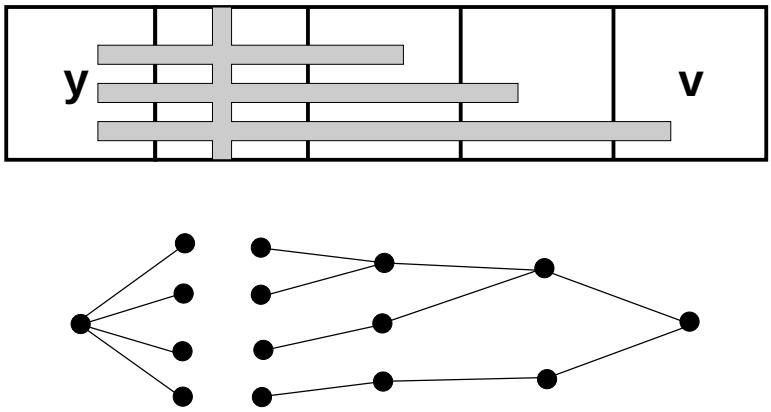

Figure 3: Example of embedded boundary description of an interface and its corresponding graph description. The graph can be almost arbitrarily complex when the grid is underresolved.

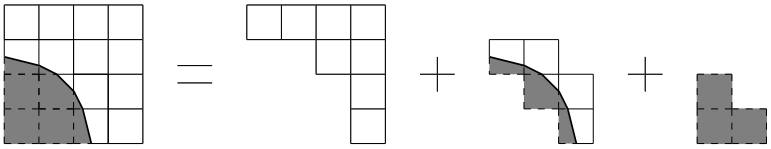

Figure 4: Decomposition of the grid into regular, irregular, and covered cells. The gray regions are outside the solution domain.

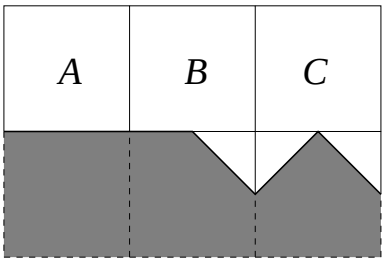

Figure 5: Cells with unit volume that are irregular.

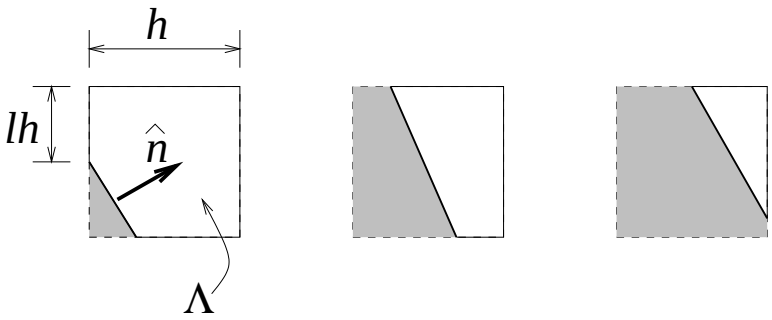

Figure 6: Representable irregular cell geometry. The gray regions are outside the solution domain.

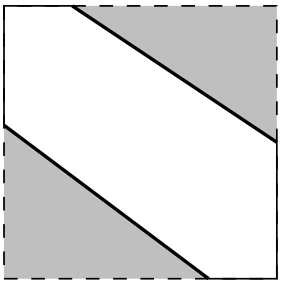

Figure 7: Unrepresentable irregular cell geometry. The gray regions are outside the solution domain.

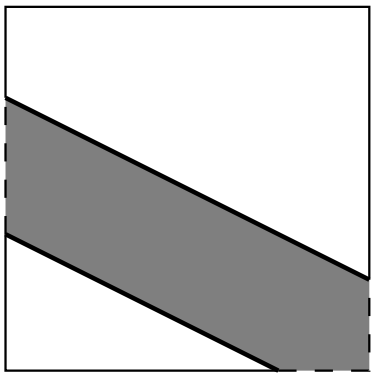

Figure 8: Multiple irregular VoFs sharing a grid cell. The left face of the grid cell is also multi-valued. The gray region is outside the solution domain.

and the area of that face $\ell^B h$.

We do not represent irregular cells such as shown in figure 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 8.

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):

$$\int_\Omega \nabla \cdot \vec{B} dV = \int_{\partial\Omega} \vec{B} \cdot \hat{n} dA \tag{4}$$

where $\hat{n}$ is the unit normal vector to the boundary of the domain. We discretize this equation so that a given 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 4 as follows:

$$V_v \vec{B} = \sum_f \vec{B} \cdot \hat{n} A_f. \tag{5}$$

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.

3.1 Interface Normal and Area

Suppose $\vec{B} = \hat{e}_x$, the unit vector in the x direction. Equation 4 becomes

$$\int_{\partial\Omega} n_x = 0, \tag{6}$$

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 5 becomes

$$A_{x0,h} - A_{x0,l} = n^I_{x0} A_I \tag{7}$$

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

$$A_I = (\sum_{i=0}^{D-1} (A_{xi,h} - A_{xi,l})^2)^{\frac{1}{2}} \tag{8}$$

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

$$n^I_{x0} = \frac{A_{x0,h} - A_{x0,l}}{A_I}. \tag{9}$$

For VoFs with multiple faces in a particular direction, we use the sum of the face areas in equations 8 and 9.

3.2 Centroid

Centroids are calculated by dividing the uncovered region of the VoF into simple shapes ("simplices"). The centroid of the VoF is computed using the centroids of the simplices. In both dimensions, compute the centroid assuming the normals are positive. We then adjust for negative normals.

Consider figure 9. In the rightmost figure (labeled "C") the uncovered region forms a triangle. In the leftmost figure ("A"), the covered region forms a triangle. In the middle figure, the uncovered region forms a trapezoid. We find the geometric configuration of a given VoF by comparing its volume fraction with the largest possible triangular

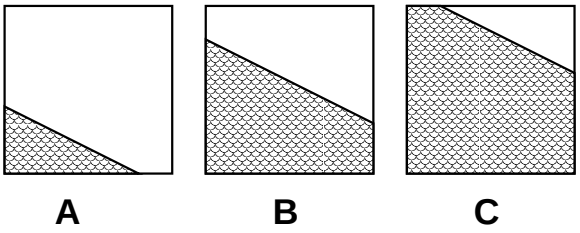

Figure 9: Geometric configurations that a two-dimensional VoF can assume. The shaded area is covered by the body.

configuration and the largest possible trapezoidal configuration. Once we know the shape of the uncovered region, the calculation is straightforward.

We compute face centroids in three dimensions exactly as we compute volume centroids in two dimensions.

In three dimensions, we substantially reduce the number of cases to consider by ordering the normals by size and creating a mapping such that $nx0 < nx1 < nx2$. After the centroid is calculated we simply invert the mapping. We also only consider the centroid of the part of the cell that has a volume fraction less than one half. If the VoF has a greater volume fraction, we compute the centroid of the complement of the VoF in the cell first and use that to compute the VoF centroid. Figure 10 shows the three configurations that the region formed by a plane cutting a cell can assume with these constraints in place. Suppose the fluid is below the plane. Define $\bar{x}$ to be the centroid of the fluid and $\bar{x}_A$ to be the centroid of section A and so on. Define $\bar{x}_T$ to be the centroid of the tetrahedron formed by by the intersection of the plane with the lower coordinate axes of the cell. In case 1,

$$\bar{x} = \bar{x}_A. \tag{10}$$

In case 2

$$\bar{x} = \bar{x}_B = \frac{1}{V_B}(V_T\bar{x}_T - V_C\bar{x}_C). \tag{11}$$

In case 3

$$\bar{x} = \bar{x}_F = \frac{1}{V_F}(V_T\bar{x}_T - V_D\bar{x}_C - V_E\bar{x}_E). \tag{12}$$

Regions A, C, D, E, and T are tetrahedra. The equation for a plane is given by

$$\alpha = n_0x_0 + n_1x_1 + n_2x_2 \tag{13}$$

The centroid of the tetrahedron formed by this plane and the coordinate axes is given by

$$\bar{x}_i = \frac{1}{V_t}\frac{\alpha^4}{24n_i^2n_jn_k} = \frac{\alpha}{4n_i} \tag{14}$$

where volume where the tetrahedron is given by

$$V_t = \frac{\alpha^3}{6n_in_jn_k} \tag{15}$$

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: The concepts represented in Chombo and EBChombo.

This assumes that all normals are positive.

4 Overview of API Design

The pieces of the graph of the discrete space is 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 `EBISBoxes` corresponding to the boxes in a `DisjointBoxLayout` grown by a specified number of ghost cells.

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`.

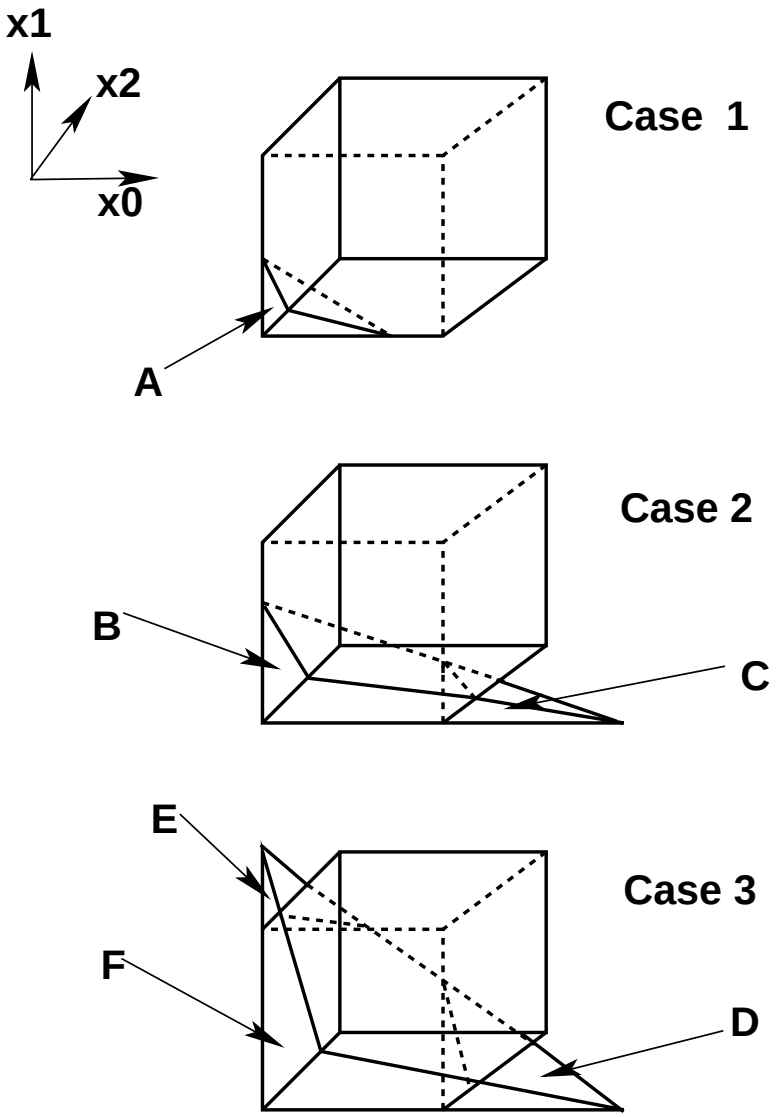

Figure 10: All the geometric configurations that the region formed by a plane cutting a cell when the following conditions are met: all the normals are positive; the normals are ordered such that $nx0 < nx1 < nx2$; and at and the volume fraction is less than one half.

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);
  ```

  Define data sizes. BoundingBox is 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. The geometry argument is the service class which tells the EBIndexSpace how to build itself. See section 5.2 for a description of the GeometryService interface class.

- ```
  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 the problem. 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);
}
```

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 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 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 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. 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 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 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 8.1.

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 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 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 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. 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_region, const Interval& Cd,`
  `const EBISBox& a_source, const Interval& Cs);`

  Copy the information from `a_source` to the over the intersection of the box `a_region` the box of the current `EBISBox` and the box of `a_source`. 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.

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 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`

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.

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;`

  Returning 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.

6 Data Holders for Embedded Boundary Applications

A BaseIVFAB 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 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.

6.1 Class BaseIFFAB

A BaseIFFAB 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. A BaseIFFAB can hold multiple components. 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_region, const Interval& Cd,
            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.

6.2 Class `BaseIVFAB`

A `BaseIVFAB` is a templated 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. 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_region, const Interval& destInterval,
            const BaseIVFAB<T>& src, const Interval& srcInterval);
  ```

  Copy the contents of another `BaseIVFAB` into this `BaseIVFAB`.

- `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.

6.3 Class `BaseEBCellFAB`

A `BaseEBCellFAB` 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 EBCellFAB& a_srcFab,const Box& a_intBox,`
  `int a_srcComp,int a_destComp,int a_numComp);`

  Copy the data from a_srcFab into the current BaseEBCellFAB over the intersection of the current domain, the domain of the input fab, and the input Box a_intBox.

- `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>& getRegFAB();`

  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.

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& getRegFAB();`

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

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

  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.

6.5 Class `BaseEBFaceFAB`

A `BaseEBFaceFAB` 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 EBFaceFAB& a_srcFab,`
  `const Box& a_intBox,int a_srcComp,int a_destComp,int a_numComp);`

  Copy the data from `a_srcFab` into the current `BaseEBFaceFAB` over the intersection of the current domain, the domain of the input fab, and the input `Box a_intBox`.

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

  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.

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. 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& getRegFAB();`

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

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

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

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.

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 EBISBox& a_ebisBox);
  void define(const IntVectSet& a_ivs,
              const EBISBox& 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);
   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.

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.

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.

- LoNoBoundary means stop at all faces on the high side of `IntVectSet` cells, excluding faces on the domain boundary.

Now we may define the important classes of `FaceIterator`:

- ```
  FaceIterator(const IntVectSet& a_ivs,
               const EBISBox& a_ebisBox,
               const int& a_direction,
               const FaceStop::WhichFaces& a_location);
  void define(const IntVectSet& a_ivs,
              const EBISBox& 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, 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.

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.

8 Usage Patterns

Here we present the usage patterns of the concepts presented in section 4. We present an initialization pattern and a calculation pattern along with an example of a `GeometryService` class.

8.1 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)
}
```

8.2 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 is 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]);
}
```

8.3 Finite Difference Calculations using EBChombo

Here we present our calculation usage pattern of 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);
    }
}
```

9 Landmines

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

9.1 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.

9.1.1 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 irregulr cells properly.

9.1.2 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++.

9.1.3 Stencil Size and Data Holders

By caveat 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.

9.2 Sending Irregular Data to Fortran

If one indends 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 indicies 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 indicies 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.

References

[CGL+00] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, D. Modiano, D. B. Serafini, and B. Van Straalen. Chombo Software Package for AMR Applications - Design Document. unpublished, 2000.

[GLN+99] 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.

[JC98] Hans Johansen and Phillip Colella. A cartesian grid embedded boundary method for Poisson's eqaution on irregular domains. *J. Comput. Phys.*, 1998.

[MC00] 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.