MFChombo Software Package for Cartesian Grid, Multifluid Applications. Level 0 : Data Holders and Infrastructure

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

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

June 10, 2004

Contents

1 Introduction **2**

2 Overview of Multifluid Description **3**
2.1 Time Dependent Interface Geometries 5

3 Overview of API Design **9**

4 Data Structures for Graph Representation **9**
4.1 Overview 9
4.2 Class `MFIndexSpace` 10
4.3 Class `GeometryService` 11
4.4 Class `EBISBox` 14
4.5 Class `EBISLayout` 17
4.6 Class `VolIndex` 18

4.7 Class `FaceIndex` 18

5 Data Holders for Embedded Boundary Applications **19**
5.1 Class `BaseIVFAB<T>` 19
5.2 Class `BaseEBCellFAB<T>` 20
5.3 Class `EBCellFAB` 21
5.4 Class `MFCellFAB` 22
5.5 Class `MFCellFactory` 23
5.6 Class `BaseEBFaceFAB<T>` 23
5.7 Class `EBFaceFAB` 24

6 Data Structures for Pointwise Iteration **25**
6.1 Class `VoFIterator` 25
6.2 Class `FaceIterator` 26

7 Time-dependent functionality **28**
7.1 Class `MFRemapper` 28

1 Introduction

This document describes the MFTools component of the MFChombo distribution. This infrastructure is based upon the Chombo infrastructure developed by the Applied Numerical Algorithms Group at Lawrence Berkeley National Laboratory [1], and also draws heavily on the EBChombo software which extends Chombo for the case of embedded boundaries in a Cartesian mesh. MFTools is meant to be an infrastructure for Cartesian grid multifluid (MF) algorithms. This software aims to provide a relatively compact set of abstractions in which Cartesian grid multifluid algorithms can be expressed and implemented. The particular design we propose 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 MF 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 the others 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 AMR implementation strategies for block-structured refinement with the MF algorithms.

Because of the similarities between our multifluid approach and the embedded boundary approach for complex geometries used in the design of the EBChombo software, we will borrow heavily from the design and conceptual framework used for EBChombo. For more information regarding the EBChombo framework, see the EBChombo design documents.

2 Overview of Multifluid Description

Cartesian grids with embedded multifluid boundaries are useful to describe volume-of-fluid representations of irregular and non-static multifluid interfaces. In this description, geometry is represented by volumes and apertures. The areas / volumes, expressed in dimensionless terms are volume fractions $\kappa_{\boldsymbol{i}} = |V_{\boldsymbol{i}}|h^{-d}$, face apertures $\alpha_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}_s} = |A_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}_s}|h^{-(d-1)}$ and boundary apertures $\alpha^B_{\boldsymbol{i}} = |A^B_{\boldsymbol{i}}|h^{-(d-1)}$. We assume that we can compute estimates of these dimensionless quantities which are accurate to $O(h^2)$. See Figure 1 for an illustration. In the figure, the grey area represents one phase and the white region the second phase; the arrows represent fluxes for the "white" phase, including the flux across the multifluid interface. A conservative, "finite volume" discretization of a flux divergence $\nabla \cdot \vec{F}$ for the "white" region is of the form:

$$\begin{aligned}\nabla \cdot \vec{F} &\approx \frac{1}{\kappa h}\sum \vec{F}^{\alpha} \cdot \vec{A}^{\alpha} \\ &\approx \frac{1}{\kappa_{\boldsymbol{i}} h}\Big(\sum_{\pm=+,-}\sum_{s=1}^{d} \pm\alpha_{\boldsymbol{i}\pm\frac{1}{2}\boldsymbol{e}_s} F^s(\boldsymbol{x}_{\boldsymbol{i}\pm\frac{1}{2}\boldsymbol{e}_s}) + \alpha^B_{\boldsymbol{i}}\vec{n}^B_{\boldsymbol{i}} \cdot \vec{F}(\boldsymbol{x}^B_{\boldsymbol{i}})\Big) \qquad (1)\end{aligned}$$

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

$$\begin{aligned}\nabla \cdot \vec{F} = \Delta\phi = \rho \text{ on } \Omega, \qquad (2)\\ \frac{\partial \phi}{\partial n} = 0 \text{ on } \partial\Omega.\end{aligned}$$

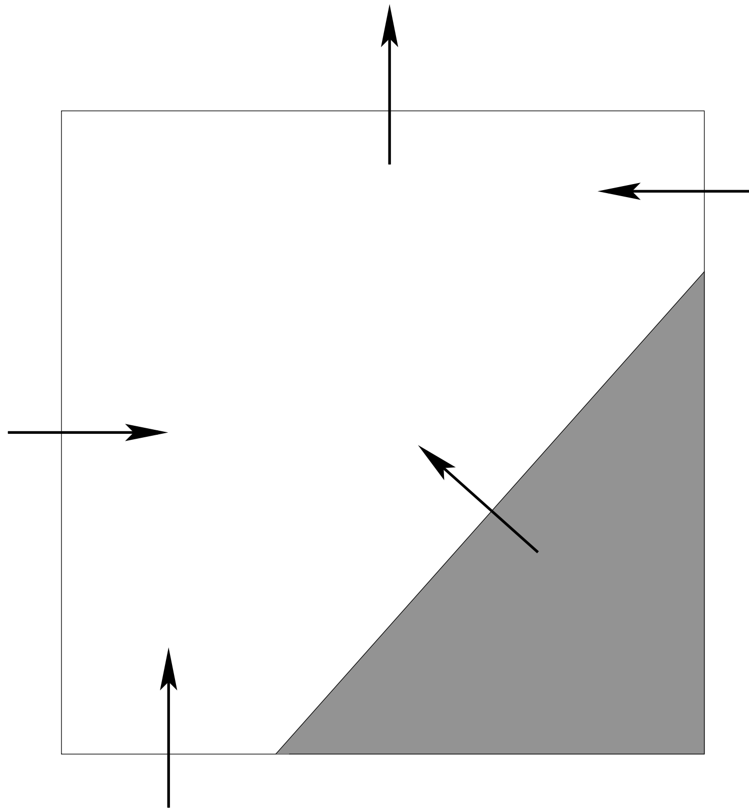


Figure 1: Multifluid cell. The grey area represents one phase, while the white region is a second phase in the same Cartesian cell. Arrows indicate fluxes for the white phase (including one across the white/grey multifluid interface)

The volume-of fluid description reduces the problem to finding sufficiently accurate gradients at the apertures. See Johansen and Colella [3] for a complete description of solving Poisson's equation with embedded boundaries; the approach we will take for multifluid interfaces will be similar. Hyperbolic conservation laws can be solved using similar divergence examples. See Modiano and Colella [4] for such an algorithm. Gueyffier, et al. [2] 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, as in Figure 2. The shaded region represents the area in one phase while the unshaded region represents a second phase. The solid lines represent the connectivity of the discrete domain for the shaded phase, while the dashed lines illustrate the connectivity for the unshaded phase. In addition, the two phases will generally be linked across the the multifluid interface through interface boundary conditions. Figure 3 illustrates the additional connectivity arising through a simple interface

boundary condition. The software infrastructure must support abstractions which can express this complexity.

Our solution to this abstraction problem is to define the multifluid grid as a graph. The irregular part of the index space for a phase *a* can be represented by a graph $G^a = \{N, E\}^a$, 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 Ω^a with the rectangular mesh, and the edges correspond to the parts of cell faces that abut a pair of irregular cell fragments. Interface connectivity between two phases a and b is defined by the set I^{ab} which determines the connectivity between N^a and N^b in much the same way that E^a determines the connectivity between the nodes in N^a.

For each phase, the remaining parts of space are indexed using elements of Z^d, or are contained in another phase 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. We recognize three types of grid cells or faces: a cell or face that the multifluid interface 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 for the given phase a is *covered*. In practice, we will not allow a multifluid interface to coincide with a cell face; instead the interface will be considered to be a small distance ϵ from the interface, and will be entirely on one side of the interface. This will help simplify connectivity issues.

An irregular volume of fluid (VoF) is formed from the intersection of a grid cell and the irregular phase domain Ω^a. We represent the segment of the multifluid interface as a single flat segment. Quantities located at the multifluid boundary are given the superscript B.

A VoF has a volume κh^{Dim}, where κ is its volume fraction. A face has an area $\ell h^{(Dim-1)}$, where ℓ is its area fraction. The polygonal representation is reconstructed from the volume and area fractions under the assumption that the VoF has one of the shapes above. Since the boundary segments are 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 multifluid face $\hat{n}$ and the area of that face $\ell^B h^{(Dim-1)}$.

2.1 Time Dependent Interface Geometries

Because interfaces move as a function of time, `MFChombo` also has the concept of time as an integral part of the geometry. As the geometry changes, the graph and its connectivity

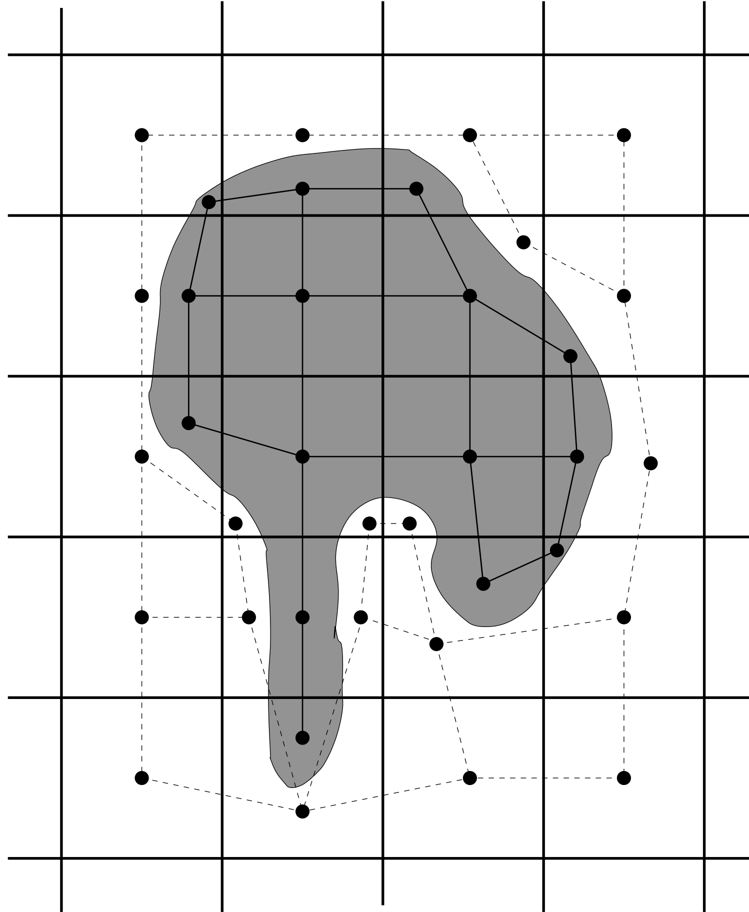


Figure 2: Example of a possible multifluid connectivity graph. The shaded region is one phase, while the unshaded region is a second phase. The dashed lines represent the graph connectivity of the unshaded phase, while the solid lines represent the connectivity of the shaded phase. For clarity, connectivity across the multifluid interface is ignored in this depiction.

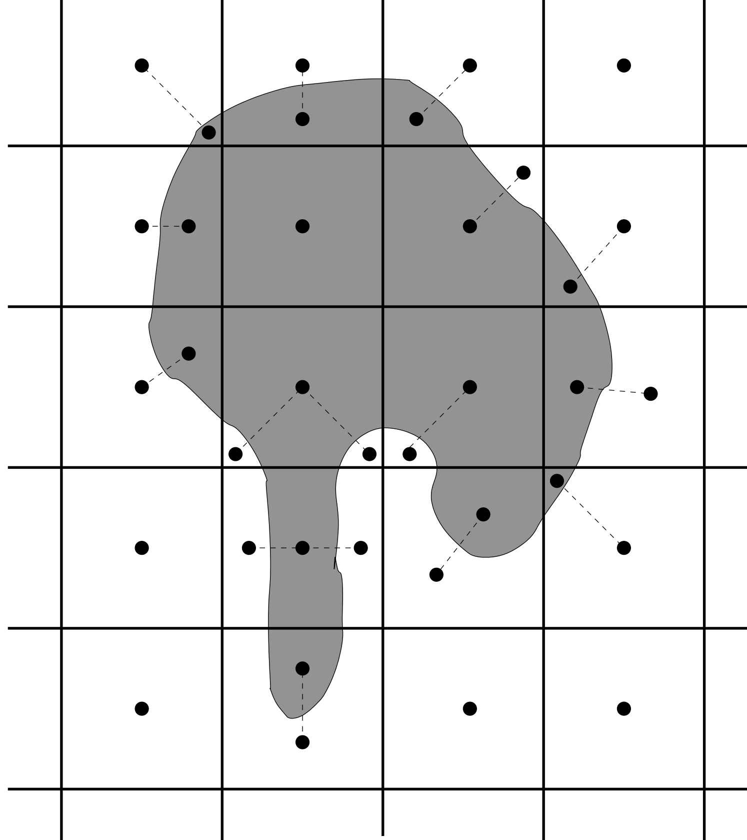
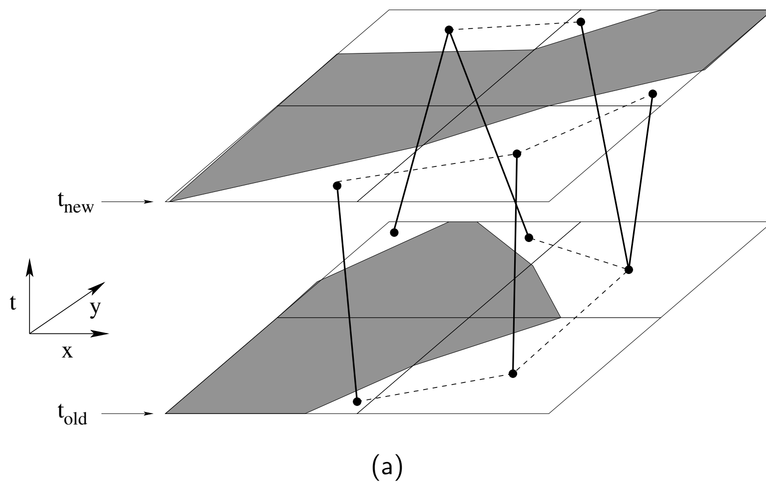
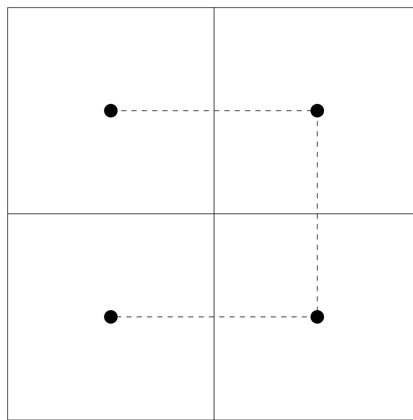


Figure 3: Multifluid example from Figure 2, illustrating connectivity between phases across the multifluid interface.

will change over time as well. For example, Figure 4(a) illustrates a possible evolution of a multiphase interface and the changing connectivity graph of the unshaded phase. Note that the two VoFs at the old time in the rear left cell merge, while the VoF in the rear right cell splits as the geometry of the interface evolves. The changing geometry means that there is a connectivity graph in time (illustrated by the dark solid lines) as well as space (illustrated by the dashed lines at each time level). The additional connectivity in time and space leads to the concept of a *data graph*, which is the simplest possible connectivity graph based on the aggregate old- and new-time connectivity. For example, the data graph for the time-dependent geometry shown in Figure 4(a) is shown in Figure 4(b). The key feature of the data connectivity graph is that VoFs in a single cell which are linked by time connections in the connection graph (i.e. cases where a VoF either splits into multiple VoFs or merges with another VoF during a timestep) are represented by one node on the data graph. Topologically, the data graph is identical to the new-time graph except in locations where a single old-time VoF has been split into more than one node in the new-time graph; in this case, the multiple child nodes in the new time graph are represented as a single node in the data graph. In general, a user will access graph connectivity information through the data graph; old- and new-time geometric information is then accessed through connections with the data graph. Solution updates are computed using connectivities based on the data graph, and when data storage is allocated, it is based on the data graph, rather than the old- or new-time graphs. This will simplify time-dependent computations.



(a)



(b)

Figure 4: Evolution of a connectivity graph in time.In (a), dashed lines illustrate connnectivity of the unshaded phase at a single time level, while solid lines illustrate connectivity of VoFs between time levels. (b) Data graph for the time-dependent geometry shown in (a)

Concept	Chombo	MFChombo
Z^D	—-	MFIndexSpace
point	IntVect	VoF
region	Box	EBISBox
Union of Rectangles	BoxLayout	EBISLayout
Rectangular array	BaseFab	MFCellFAB
iterator over points	BoxIterator	VoFIterator, FaceIterator

Table 1: The concepts represented in Chombo and MFChombo.

3 Overview of API Design

The pieces of the graph of the discrete space are represented by the classes `VolIndex` and `FaceIndex`, which are elements of the `EBChombo` software. `VolIndex` is an abstract index into cell-centered locations corresponding to the nodes of the graph (VoFs). In `EBChombo`, the `FaceIndex` class is an abstract index into edge-centered locations (connections between VoFs). To handle the additional needs of multifluid computations, we extend the `VolIndex` class to index into interphase graph connections between VoFs. `FaceIndex` controls access to faces within the same phase.

The class `MFIndexSpace` is a container for geometric information at all levels of refinement. Each fluid phase is represented by data on an `EBIndexSpace`. `EBIndexSpace` is the EBChombo class for representing embedded boundary applications (cut-cell technology). Each fluid phase has it's own consistent `EBIndexSpace`.

The `MFIndexSpace` class also contains the interface to the functionality needed to advance the interface in time. Because it is expected that the method of advancing the multifluid interface in time will be dependent on the physics of the actual problem being solved.

In general, the user only accesses the data graph through each phase's `EBISBox` descriptor.

In an AMR computation with refinement in time, different refinement levels will be have different old and new times as the hierarchy of levels is advanced in time. Therefore, except at the initial time when the `MFIndexSpace` is defined, time variables will depend on the individual `MFISLevels`.

4 Data Structures for Graph Representation

4.1 Overview

The class `VolIndex` is an abstract index into cell-centered locations (VoFs) corresponding to the nodes of the graph. The class `FaceIndex` is an abstract index into connections 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. There are multiple types of FaceIndexes, depending on the relationship between the VoFs connected by the FaceIndex. The FaceIndex type is determined by an enumeration; there are $(SpaceDim + 1)$ types. The enumeration types are named:

```
enum FaceIndexType {xFace=0, yFace, (zFace,)}
```

FaceIndex types 0 through $(SpaceDim - 1)$ are face-centered connections between VoFs, and are identical to the EBChombo conception of FaceIndexes.

EBISBox represents a subset of the MFIndexSpace at a particular refinement and over a particular box in the space at a particular time, for a single fluid phase.

4.2 Class MFIndexSpace

The entire time-dependent graph description of the geometry is represented in the class MFIndexSpace, which stores the data graph, along with the graph connectivity and other geometric information (volume fractions, area fractions, etc) at two time levels (t_{old} and t_{new}) and the connectivity of the graph between these two times. The important member functions of MFIndexSpace are as follows.

- ```
  void
  define(const Box& a_domain,
         const RealVect& a_origin,
         const Real& a_dx,
         const Vector<GeometryService*>& a_geoservers,
         int a_nCellMax = -1,
  int maxCoarsenings = -1);
  ```

  Define data sizes. The a_domain argument defines the domain of the MFIndexSpace at its finest resolution. The arguments a_origin and a_dx specify the location of the zero vector in the index space and the grid spacing in each coordinate direction at the finest resolution. The initialTime argument is the solution time at which the initial geometry is defined. The a_geoservers argument is the service class which tells the MFIndexSpace how to build itself. It contains a GeometryService description of each fluid pahse. See section 4.3 for a description of the GeometryService interface class. Coarser resolutions of the MFIndexSpace are also generated in the initialization process. The degree of coarsening is controled by the optional argument maxCellCoarsenings

- ```
  void
  fillEBISLayout(EBISLayout& a_ebis,  int phase,
                 const DisjointBoxLayout& a_grids,
                 const Box& a_domain,
  ```

```
                    const int & nghost) const;
```

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

- `int numPhases() const`
- `int numLevels() const;`

  Return the number of levels of refinement represented in the `MFIndexSpace`

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

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

A major modification to the Embedded Boundary Chombo code (**EBCHOMBO**) was the ability to have multiple `EBIndexSpace` instances. We use this ability to model each fluid phase as it's own cut-cell AMR data representaion. The `EBISBox` interface privides the needed extra functionality for users to access the additional inter-phase surface data.

Another critical extension of the `EBIndexSpace` was the ability to have multiple *boundary* faces per volume-of-fluid. This can be finessed in a single phase embedded boundary algorithm, but is essential for correct multi-phase dynamics.

4.3 Class `GeometryService`

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

In the multifluid context "covered" means the volume has no volume of the phase in question. "regular" means the cell is entirely this fluid phase.

The interface of `GeometryService` is

- ```
  virtual bool isRegular(const Box& region, const ProblemDomain& domain,
                         const RealVect& origin, const Real& dx)=0;
  virtual bool isCovered(const Box& region, const ProblemDomain& 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 ProblemDomain& 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. `MFIndexSpace` 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 4.4. 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

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 4.4.
  ```

`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_mfisBox) 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_mfisBox) 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_mfisBox) 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_mfisBox) 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_mfisBox) 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_mfisBox) 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.

4.4 Class EBISBox

EBISBox

represents the geometric information of the domain at a given refinement and time instance within the boundaries of a particular box. EBISBox can only be accessed by using the the EBISLayout interface. Like the MFIndexSpace and EBISLayout classes, EBISBox has two different time levels, t_{old} and t_{new}, and contains the geometries at each time level, along with the data graph based on these geometries. Geometric information is also organized by phase. In general, the old- and new-time graphs are hidden from the user, who instead accesses the data graph for a given EBISBox. Old and new time geometric information is then accessed through its connections with the data graph.

The important public member functions of EBISBox are as follows:

- IntVectSet getMultiCells(const Box& subbox, int phase) const;

  Returns a list all multi-valued cells at the given level of refinement within the input Box subbox in the *data graph* for the phase denoted by phase.

- IntVectSet getIrregIVS(const Box& boxin, int phase) const;

  Returns the irregular cells of the EBISBox data graph that are within the input subbox for the given phase. For the purposes of the data graph, any cell which is irregular in the old- or new-time graphs is considered irregular.

- Vector<VolIndex> getVoFs(const IntVect& iv, int phase);

  Gets all the VoFs in the data graph in a particular cell for the given phase.

- int numVoFs(const IntVect& iv, int phase) const;

  Returns the number of VoFs in the data graph in a particular cell for the given phase.

- Vector<FaceIndex> getFaces(const VolIndex& vof,
FaceIndexType faceType,
Side::LoHiSide sd);

  Gets all faces in the data graph of the type denoted by faceType for the given VoF and phase. If the faceType is either xFace, yFace, or zFace, the Side sd denotes whether it is a high or low side face in the direction given.

- bool isRegular(const IntVect& iv) const;

  Returns true if the input cell is a regular VoF in the data graph

- 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 in the data graph for this phase. A cell is considered to be "covered" for the purposes of the data graph if there is no fluid of this phase in this cell.

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

  Returns true if every cell in the input box is a covered cell in the data graph in this phase.

- `bool isIrregular(const IntVect& iv, int phase) const;`

  Returns true if the input cell is an irregular cell in the data graph for the given phase.

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

  Returns the number of faces the input VoF has in the given type and side (if `faceType` is `xFace`, `yFace`, or `zFace`). Returns zero if the VoF has no faces of the given type.

- `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 (connected by a `FaceIndex` of type `xFace`, `yFace`, or `zFace`).

- `bool isAllCovered();`

  Return true if every cell in the `EBISBox` is covered (not in the given phase) in the data graph.

- `bool isAllRegular();`

  Return true if every cell in the `EBISBox` is regular (in the domain of the given phase) in the data graph.

- `RealVect normal(const VolIndex& vofin, int phase) const;`

  Returns the normal to the fluid interface with respect to the opposing indicated phase 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).

- `RealVect bndryCentroid(const VolIndex& a_vof, int phase) const;`

  Returns the centroid of the area of the fluid interface with respect to the opposing indicated phase at the input VoF. Return the zero vector if the answer is undefined (for example, if the VoF is regular or covered).

- `Real bndryArea(const VolIndex& a_vof, int phase) const;`

  Returns the surface area of the fluid interface with respect to the opposing indicated phase at the input VoF. Returns zero if the answer is undefined (for example, if the VoF is regular or covered, or this phase is not adjacent).

- `int numFacePhase(const VolIndex& a_vof) const ;`

  returns the number of distinct inter-phase faces for this volume of fluid.

- `int facePhase(const VolIndex& a_vof, int face) const ;`

  returns the ID of the fluid phase for a specified VoF inter-phase face.

- `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.
  ```

4.5 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 MFIndexSpace. 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`

4.6 Class `VolIndex`

The class `VolIndex` is an abstract index into cell-centered locations which corresponds to the nodes of the data graph. 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 multifluid interface 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.

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.

4.7 Class `FaceIndex`

The class `FaceIndex` is an abstract index into connections between VoFs in the graph. A `FaceIndex` exists between two VoFs and is defined by those VoFs. Each `FaceIndex` has an associated type, given by the `FaceIndexType` enumeration in Section 4.1. Every face referred to by a `FaceIndex` has an associated area fraction. Note that while a face-centered `FaceIndex` can have an area fraction between zero and one. A FaceIndex with zero area fraction has no flow area between the VoFs connected by the face. A face with unity area fraction has an uncovered area equal to an uncovered cell face. Only friend classes (`EBISBox, MFIndexSpace...`) may call the defining constructors. Only the null constructor of `FaceIndex` should be used by users.

The important member functions of this class are:

- `const FaceIndexType& faceType() const`

  Returns the `FaceIndexType` of this `FaceIndex`.

- `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. If the `FaceIndexType` is of `mfInterface`, "low" and "high" are defined in the same way as described previously. 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.

- `bool isBoundary() const`

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

5 Data Holders for Embedded Boundary Applications

All multifluid data holders are defined on the data graph of an `EBISBox`. A `BaseIVFAB<T>` is an array of data defined in an irregular region of space. The irregular region is specified by the `VolIndexes` of an `IntVectSet` and the data graph of a `EBISBox`. Multiple data components per `VolIndex` may be specified in the `BaseEBIVFAB` definition.

A `BaseEBIFFAB<T>` is an array of data defined over an irregular region of space. The irregular region is specified by the faces of an `IntVectSet` within the data graph of an `EBISBox`. All the faces in a `BaseEBIFFAB` will have the same `FaceIndexType`, which is specified in the `BaseEBIFFAB` 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 a multifluid interface in the data graph of an `EBISBox`. `BaseEBFaceFAB` is a templated class which holds face-centered data over a similar region.

5.1 Class `BaseIVFAB<T>`

A `BaseIVFAB<T>` is a templated array of data defined over an irregular region of space. The irregular region is specified by the `VolIndexs` of an `IntVectSet` intersected with the data graph of an `EBISBox`. 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_mfisBox,
              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 `value`.

- `void setVal(T value, int phase);`

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

- `int nComp() const;`

  Return the number of data components of 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, where `varlocin` may range from zero to *nvar-1*. The returned object is a modifiable lvalue.

5.2 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 the data graph in an `MFIndexSpace`. 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_mfis,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 setVal(T a_value, int phase);
  ```
  Set the value of all data for the given phase 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.

- ```
  int nComp() const;
  ```
  Return the number of data components of this BaseEBCellFAB.

- ```
  T& operator()(const VolIndex& a_vof, int a_nVarLoc);
  ```
  Returns the data at VoF a_vof for variable number a_nVarLoc. Returns a modifiable lvalue.

- ```
  BaseFab<T>& getRegFAB();
  ```
  Returns the regular data holder. This is useful so that the data can be passed to Fortran using the BaseFab interface.

- ```
  getCellDataType(BaseFab<int>& cellTypes)
  ```
  Fills a basefab¡int¿ with values indicating which type of cell is in each cell of the BaseFab¡T¿ returned by the getRegFab function. In a regular cell, the value is the index number of the phase occupying the cell. If the cell is an irregular cell, the value is -1.

- ```
  const IntVectSet& getMultiCells() const;
  ```
  Returns the IntVectSet of all the multiply-valued cells.

5.3 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 the data graph for a single phase in an EBISBox. It is an extension of a BaseEBCellFAB<Real> which includes arithmetic functions. At singly valued cells, 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);
  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 or the same data graph.

5.4 Class `MFCellFAB`

Class `MFCellFAB` represents a collection of `EBCellFAB`s for each fluid phase. It provides limited functionality itself but does organize code design and interfaces.

- ```
  MFCellFAB(const Vector<EBISBox>& a_phaseGraphs,
            const Box& a_region, const Vector<int>& a_nVar)
  ```

  Constructor. `a_phaseGraphs` has a length equal to the number of fluid phases. Each fluid phase is defined by it's own `EBISBox`

- `EBCellFAB& getPhase(int a_phase)`

- `int nComp(int a_phase) const`

- ```
  void copy(const Box& RegionFrom,
            const Interval& destInt,
            const Box& RegionTo,
            const MFCellFAB& source,
            const Interval& srcInt);
  static int preAllocatable()
  int size(const Box& R, const Interval& comps) const
  void linearOut(void* buf, const Box& R, const Interval& comps) const
  void linearIn(void* buf, const Box& R, const Interval& comps);
  ```

Linearization routines required by the LevelData<T> parallel data holder template class.

5.5 Class MFCellFactory

Used to imnplement the Factory Design Pattern for our parallel data holders. This is the technique used in Chombo to allow objects with different construction requirements to re-use our parallel data holder and communication system.

The only critical user interface function of this class is:

- ```
  MFCellFactory(const MFIndexSpace& a_mf, const DisjointBoxLayout& a_dbl,
                const Box& a_domain, const Vector<int>& a_ncomps,
                int ghost);
  ```

This class is used as follows:

```
LevelData<MFCellFAB> state;
MFCellFactory factory(mfIndexSpace, dBoxLayout,
                      domain, numComps, nghost);
state.define(dBoxLayout, maxComps, nghost, factory);
```

5.6 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 the faces of a given FaceIndexType in the data graph of an EBISBox. 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_mfis,
              const Box& a_region,
              FaceIndexType a_faceType,
              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 for the given FaceIndexType. 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 (only relevant if a_faceType is xFace, yFace, or zFace).

- void setVal(T a_value);

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

- void setVal(T a_value, int phase);

  Set the value of all data in the container which is of the given phase to a_value.

- FaceIndexType type() const;

  Return the FaceIndexType of the faces of this BaseEBFaceFAB.

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

  Returns the data at face a_face for variable number a_nVarLoc. Returns a modifiable lvalue.

- void copy(const Box& a_RegionFrom, const Interval& a_destInt,

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

  Copy the data from a_source into the current BaseEBFaceFAB over regions and intervals specified. The two MFFaceFABs must have the same FaceIndexType and be based on the same data graph.

- 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.

5.7 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 MFIndexSpace. It is an extension of a BaseEBFaceFAB<Real> which includes arithmetic functions. At single-valued faces, 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 (note that unlike the corresponding cell-centered class, there is no remap functionality provided. It is assumed that face-centered data is transient and cannot be remapped.):

- 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);
  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.

6 Data Structures for Pointwise Iteration

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

6.1 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_mfisBox,
              const int phase);
  void define(const IntVectSet& a_ivs,
              const EBISBox& a_mfisBox,
              const int phase);
  ```

  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_mfisBox,
                   const Real& a_value)
{
   int thisPhase = a_phi.phase();
   VoFIterator vofit(a_ivs, a_mfisBox, thisPhase);
   for(vofit.reset(); vofit.ok(); ++vofit)
     {
        const VolIndex& vof = vofit();
        a_phi(vof) = 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.

6.2 Class FaceIterator

The `FaceIterator` class is used to iterate over faces of a particular `FaceIndexType` and phase in an `IntVectSet`. First we must define `FaceStop`, the enumeration class which distinguishes which faces at which a given `FaceIterator` will stop if it is of the `xFace`, `yFace`, or `zFace` `FaceType`. If the `FaceIndexType` is `mfInterface`, then the `FaceStp` case has no meaning. 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 interface of `FaceIterator`:

- ```
  FaceIterator(const IntVectSet& a_ivs,
               const EBISBox& a_mfisBox,
               const int a_phase,
               const FaceIndexType& a_faceType,
               const FaceStop::WhichFaces& a_location);

  void define(const IntVectSet& a_ivs,
              const EBISBox& a_mfisBox,
              const int a_phase,
              const FaceIndexType& a_faceType,
              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_mfisBox,
                       const Real& a_value)
{
   int type = a_phi.type();
   int phase = a_phi.phase();
   FaceIterator faceit(a_ivs, a_mfisBox, phase, type,
                       FaceStop::SurroundingWithBoundary);
   for(faceit.reset(); faceit.ok(); ++faceit)
     {
        const FaceIndex& face = faceit();
        a_phi(face) = 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 Time-dependent functionality

As the fluid interface moves there is the need to be remapping the static data structures.

7.1 Class `MFRemapper`

Class `MFRemapper` handles two forms of remapping operation required for time-dependent computations.

- ```
  void remap(const MFIndexSpace& a_sourceMF,
             const LevelData<MFCellFAB>& a_source,
             const MFIndexSpace& a_destMF,
             LevelData<MFCellFAB>& a_dest)
  ```

  Remap operation that takes an existing state data the source `MFIndexSpace` and maps it too the same `DisjointBoxLayout` but with a new `MFIndexSpace`.

  The new `MFIndexSpace` is usually the result an interface movement operation, followed by some form of LevelSet computation. At the end of this, a new `MFIndexSpace` is created. The current state data needs to be mapped into a new `LevelData<MFCellFAB>` based on this new `MFIndexSpace`. The
  ```

DisjointBoxLayouts from a_source and a_dest are required to be identical. Thus, no coarse-fine AMR operations are performed.

An important algorithmic consideration for this routine involves determining the correct data to fill into a cell that was covered in the source MFIndexSpace but becomes uncovered in the destination MFIndexSpace. The operation must not transfer state data from another fluid phase. It should also be done in a manner that preserves the order of the numerical scheme at the fluid interface. Different schemes are currently being investigated.

-
```
void remap(const MFIndexSpace& a_MF,
           const ProblemDomain& a_domainCoar,
           const ProblemDomain& a_domainFine,
           const LevelData<MFCellFAB>& a_source,
           const LevelData<MFCellFAB>& a_coarse,
           const int& nref,
           const int& nghost,
           LevelData<MFCellFAB>&  a_dest)
```

  Regridding version of remapping operation. Same MFIndexSpace, same state data, different grid configurations. This routine is usually called after a MeshRefine operation. This routine does have to perform coarse-fine interpolation to initialize new fine grid locations. nref is the integer refinement factor between this level of refinment and the next coarser AMR level. nghost is the number of ghost cells in the a_dest that the user needs to be filled in.

References

[1] 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.

[2] 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.

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

[4] 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.