MFChombo Software Package for Cartesian Grid, Multifluid Applications

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

July 8, 2003

Contents

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 [CGL+00], 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:

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

This is useful for many important partial differential equations. Consider Poisson's

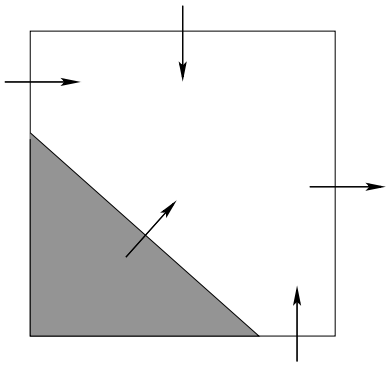

Figure 1: Multifluid cell. The grey area represents one phase, while the white region is a second phase in the same Cartesian cell. Arrows indicate fluxes for the white phase (including one across the white/grey multifluid interface)

equation with Neumann boundary conditions

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

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 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 [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, 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.

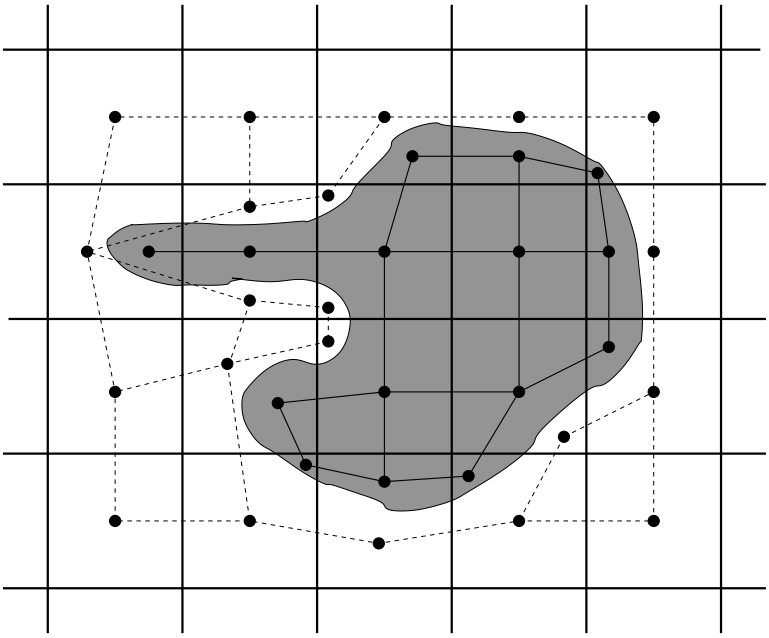

Figure 2: Example of a possible multifluid connectivity graph. The shaded region is one phase, while the unshaded region is a second phase. The dashed lines represent the graph connectivity of the unshaded phase, while the solid lines represent the connectivity of the shaded phase. For clarity, connectivity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

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

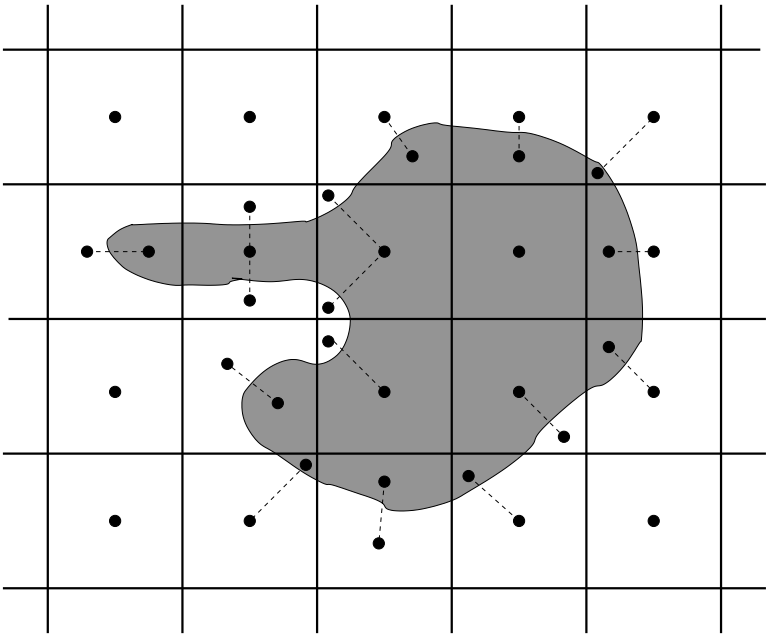

Figure 3: Multifluid example from Figure 2, illustrating connectivity between phases across the multifluid interface.

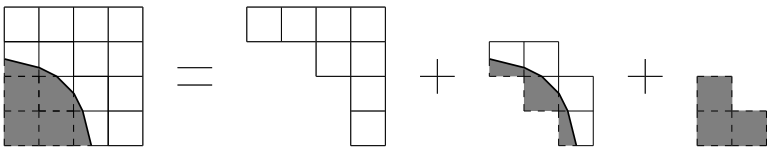

Figure 4: Decomposition of the grid for the unshaded phase into regular, irregular, and covered cells. The grey cells are outside the domain for the unshaded phase.

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 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 will change over time as well. For example, Figure 5(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 5(a) is shown in Figure 5(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.

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 `FaceIndex` class to index into *all* graph connections between VoFs, including those not located at a cell face. The class `MFIndexSpace` is a container for geometric information at all levels of refinement. The class `MFISLevel` contains the geometric information for a given level of refinement. `MFISLevel` is not part of the public API and is considered internal to `MFIndexSpace`. `MFISBox` represents the intersection between an `MFISLevel`

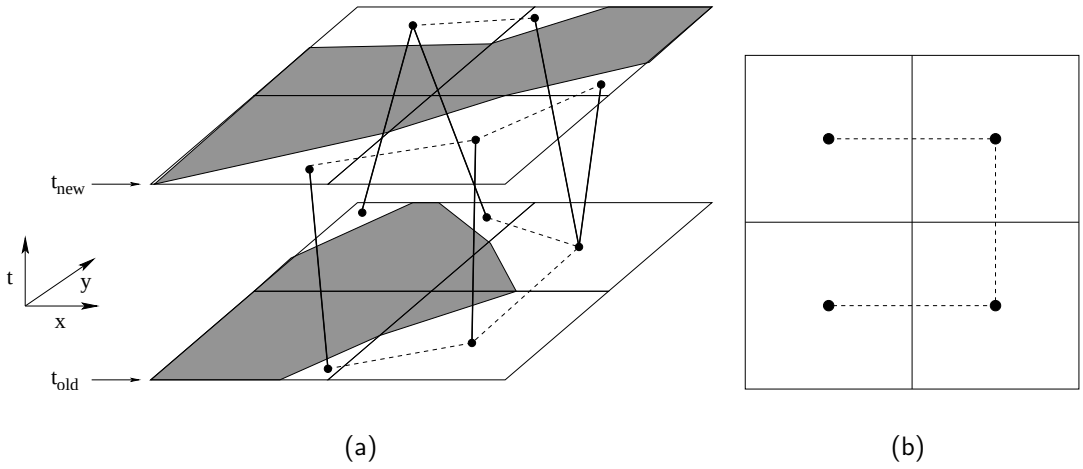

Figure 5: Evolution of a connectivity graph in time.In (a), dashed lines illustrate connnectivity of the unshaded phase at a single time level, while solid lines illustrate connectivity of VoFs between time levels. (b) Data graph for the time-dependent geometry shown in (a)

and a `Box` and is used for aggregate access of geometric information. `MFISLayout` is a set of `MFISBoxes` corresponding to the boxes in a `DisjointBoxLayout` grown by a specified number of ghost cells.

Along with the data graph, the `MFIndexSpace` and `MFISBox` classes contain two time levels for each resolution, one at an old time t_{old} and one at a new time $t_{new} = t_{old} + \Delta t$. It is assumed that the geometry at a time $t_{old} \leq t \leq t_{new}$ is a linear interpolation of the old- and new-time geometries. In particular, the `MFISBox` function `getDistance(RealVect loc, Real t)` returns the signed distance from the point `loc` to the nearest multifluid interface. The `MFISBox` computes this distance by keeping the distance to the nearest interface at each node at both old and new times, and then interpolating in time and space to compute the distance between `loc` and the multifluid interface at time t. 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, the `MFIndexSpace` class is a virtual base class, from which a user will derive a class which contains the specific method used to advance the interface.

In general, the user only accesses the data graph in the `MFIndexSpace` and `MFISBox` classes. Old and new time geometric information, such as normals, apertures, and centroids are accessed through connections to the data graph.

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

Concept	Chombo	MFChombo
Z^D	—-	MFIndexSpace
point	IntVect	VoF
region	Box	MFISBox
Union of Rectangles	BoxLayout	MFISLayout
Rectangular array	BaseFab	BaseMFCellFAB, BaseMFFaceFAB

Table 1: The concepts represented in Chombo and MFChombo.

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,) mfInterface}.`

`FaceIndex` types 0 through $(SpaceDim - 1)$ are face-centered connections between VoFs, and are identical to the `EBChombo` conception of `FaceIndexes`. A `FaceIndex` of type `mfInterface` is a multifluid interface within a cell, connecting two VoFs of different phases. A `FaceIndex` of this type can have an arbitrary location in a cell, except at the face at the edge of a cell, which is not allowed, as mentioned earlier.

The entire time-dependent graph is represented in the virtual class `MFIndexSpace`, which stores all the graph connectivity and other geometric information (volume fractions, area fractions, etc). Because the geometry can change as a function of time, the `MFIndexSpace` class supports this by maintaining a data graph representation of the time-dependent geometry, as well as a graph with connectivity in time as well as space. `MFISBox` represents a subset of the `MFIndexSpace` at a particular refinement and over a particular box in the space, in a particular time interval ($t_{old} \leq t \leq t_{new}$). `MFISLayout` is a collection of `MFISBoxes` distributed over processors associated with an input `DisjointBoxLayout`.

Because there are multiple time-levels present in the `MFIndexSpace`, we also introduce a `TimeIndex` enumeration when it is necessary to distinguish between time levels:

`enum TimeIndex {oldTime = 0; newTime, average}`.

The `average` entry will refer to the average of the old and new time states.

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 ProblemDomain& domain,
              const RealVect& origin,
              const Real& dx,
              const Real& initialTime);
  ```

  Define data sizes. The `domain` argument defines the domain of the MFIndexSpace at its finest resolution. The arguments `origin` and `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. Coarser resolutions of the MFIndexSpace are also generated in the initialization process.

- ```
  void fillMFISLayout(MFISLayout& mfisLayout,
                      const DisjointBoxLayout& dbl,
                      const ProblemDomain& domain,
                      const int& nGhost);
  ```

  Define an `MFISLayout` for each box in the input layout `dbl` grown by the input ghost cells. The input `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 `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 `nghost` argument defines the number of ghost cells in each coordinate direction. The old- and new-time levels as well as the data graph for the `MFISLayout` will be taken from the given level in the `MFIndexSpace`.

- ```
  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`.

- `Real oldTime(int lev) const;`

  Returns the time at the old time level stored by the `MFISLayout` at the level `lev`.

- `Real newTime(int lev) const;`

  Returns the time at the new time level stored by the `MFISLayout` at the level `lev`.

- `virtual void advanceGeometry(int lev, Real dt)`

  This function will be provided by the derived class – it advances the geometry at level `lev` from t^{new} to $t^{new} + dt$. At the end of this function, the time levels have been updated, so that the new old-time is the old new-time and the new new-time is the old new-time $+ dt$. Also, the data graph is updated.

- `void createNewDataGraph(int lev)`

  Reconstitutes the data graph based on the current old- and new-time geometries; generally called by the `advanceGeometry` function.

`MFIndexSpace` can only be accessed through the the `Chombo_MFIS` singleton class. The usage pattern follows this model. At some point, one defines the singleton as follows:

```
MFIndexSpace* mfisPtr = Chombo_MFIS::instance();
mfisPtr->define(domain, origin, dx, time);
```

Since the `MFIndexSpace` is a virtual base class, the derived class will contain the functionality necessary to fully define the initial geometric configuration. Whenever one needs to define an `MFISLayout`, the usage is as follows:

```
void makeMFISL(MFISLayout& a_mfisl,
               const DisjointBoxLayout& a_grids,
               const ProblemDomain& a_domain,
               const int& a_nghost)
{
  const MFIndexSpace* const mfisPtr = Chombo_MFIS::instance();
  assert(mfisPtr->isDefined());
  mfisPtr->fillMFISLayout(a_mfisl, a_grids, a_domain, a_nghost);
}
```

4.3 Class `MFISBox`

`MFISBox` represents the geometric information of the domain at a given refinement and interval in time within the boundaries of a particular box. `MFISBox` can only be accessed by using the the `MFISLayout` interface. Like the `MFIndexSpace` and `MFISLayout` classes, `MFISBox` 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 `MFISBox`. Old and new time geometric information is then accessed through its connections with the data graph.

The important public member functions of `MFISBox` 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 `MFISBox` 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.

- `Vector<VolIndex> getVoFsNew(const VolIndex a_vof);`

  For the input `VolumeIndex` in the data graph, return the `VolumeIndexes` of all nodes in the new-time graph which are connected to `a_vof`. In most cases, this will return a `Vector` of length one (one-to-one correspondence between the data graph VoF and a new-time VoF). The only case where it will return a `Vector` with a length greater than one is when a single VoF in the old-time geometry splits into more than one VoF in the new-time geometry. A return `Vector` of length 0 indicates that an old-time VoF has disappeared from the new-time graph (normally because a multifluid interface has crossed a cell boundary and left the current cell).

- `Vector<VolIndex> getVoFsOld(const VolIndex a_vof);`

  For the input `VolumeIndex` in the data graph, return the `VolumeIndexes` of all nodes in the old-time graph which are connected to `a_vof`. In most cases, this will return a `Vector` of length one (one-to-one correspondence between the data graph VoF and a old-time VoF). The only case where it will return a `Vector` with a length greater than one is when multiple VoFs in the old-time geometry merge into one VoF in the new-time geometry (and so are represented as a single VoF in the data graph). A return `Vector` of length 0 indicates that an new-time VoF is created in the given cell during the interval from t_{old} to t_{new} (normally because a multifluid interface has crossed a cell boundary and left the current cell).

- `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,`
  `int phase);`

  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. For a multifluid interface, we determine high or low by the phases themselves. A "high" face is one that connects the given VoF with a higher phase index, while a "low" face connects the given VoF with a phase of a lower index number. In a two-phase computation, only one `sd` will return a non-empty set of multifluid interface faces.

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

  Returns true if the input cell is a regular VoF in the data graph for the given phase. A cell is considered regular for the data graph if it is regular in both the old-time and new-time graphs.

- `bool isRegular(const Box& box, int phase) const;`

  Returns true if every cell in the input Box is a regular VoF in the data graph for the given phase.

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

  Returns true if the input cell is a covered cell in the data graph for the given phase. A cell is considered to be "covered" for the purposes of the data graph if it is covered in both the old-time and new-time graphs.

- `bool isCovered(const Box& box, int phase) const;`

  Returns true if every cell in the input box is a covered cell in the data graph for the given 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. Note that since the VoF indexed by the `VolIndex vofin` has a distinguished phase, phase arguments are not used for this function, or any function using a `VolIndex` as an argument.

- `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 isInterfaceConnected(const VolIndex& vof1,`
  `const VolIndex& vof2) const;`

  Return true if the two input VoFs are connected at a multifluid interface (through a `FaceIndex` of type `mfInterface`). `vof1` and `vof2` should be at the same time.

- `bool isAllCovered(int phase);`

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

- `bool isAllRegular(int phase);`

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

- `RealVect normal(const FaceIndex& faceIn, TimeIndex timeIn) const;`

  Returns the normal to the interface pointed to by `faceIn` at the time level referred to by `timeIn`. Return the zero vector if the answer is undefined (for example, if the interface referenced by `intIn` does not exist at the time referenced by `timeIn`) Note that no time need be specified, since the time level is given implicitly by `faceIn`.

- `RealVect centroid(const VolIndex& vofin, TimeIndex timeIn) const;`

  Returns the centroid of the VoF at the time level pointed to by `timeIn`. 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, TimeIndex timeIn) const;`

  Return centroid of input face at the time level indicated by `timeIn` as a RealVect. Return the zero vector if the face is covered or regular, or if the face is a `timeFace`. 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).

- `Real areaFrac(const FaceIndex& a_vof1, TimeIndex timeIn);`

  Return the area fraction of the face at the time level indicated by `timeIn`.

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

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

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

  Returns the corresponding VoF from the next coarser `MFISLevel` (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 MFISBox& a_source, const Interval& Cs);
  ```

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

- `Real getDistance(RealVect& loc, Real time) const`

  Returns the signed distance from the point `loc` to the nearest multifluid interface at the time given. If there is no interface, returns a value larger than the size of the domain.

4.4 Class `MFISLayout`

`MFISLayout` is a collection of `MFISBoxes` distributed across processors and associated with a `DisjointBoxLayout` and a number of ghost cells. In a parallel context, `MFISLayout` is the way the user can create parallel, distributed data. `MFISLayouts` are null-constructed and are defined by sending them to the `fillMFISLayout(...)` function of `MFIndexSpace`. `MFISLayout` is constructed around a reference-counted pointer of an `MFISLayoutImplem` object so copying `MFISLayouts` 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 `MFISBox` class directly. Because `MFISBox` archives the information to do this, it is an inexpensive operation. Coarsening and refinement using larger factors of refinement must be done through `MFISLayout` and it can be expensive, especially in terms of memory usage. When one sets the maximum levels of refinement and coarsening, `MFISLayout` creates mirrors of itself at all intermediate levels of refinement and holds those new `MFISLayouts` as member data. Refinement and coarsening is done by threading through these intermediate levels. The important functions of `MFISLayout` follow.

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

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

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

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

- `setMaxCoarseningRatio(const int& a_maxCoarsen);`

  Sets the maximum level of coarsening that this `MFISLayout` will have to perform. Creates and holds new `MFISLayouts` 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 `MFISLayout`

4.5 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.

4.6 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 multifluid interface can have an area fraction which is greater than 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 (`MFISBox, 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. If this `FaceIndex` is of type `mfInterface`, the "low" and "high" sides will be the Vof which is lower or higher, respectively, in its lexicographic ordering, as discussed in the `Chombo` design document.

- `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 MFISBox. A BaseMFIVFAB<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 MFISBox. Multiple data components per VolIndex may be specified in the BaseMFIVFAB definition.

A BaseMFIFFAB<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 MFISBox. All the faces in a BaseMFIFFAB will have the same FaceIndexType, which is specified in the BaseMFIFFAB definition. Multiple data components per face may be specified in the definition. BaseMFCellFAB 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 MFISBox. BaseMFFaceFAB is a templated class which holds face-centered data over a similar region.

5.1 Class BaseMFIFFAB<T>

A BaseMFIFFAB<T> is a templated array of data defined over an irregular region of space. Storage is allocated for all phases in a single BaseMFIFFAB. The irregular region is specified by the faces of an IntVectSet, intersected with the data graph of an MFISBox. All the faces in a BaseMFIFFAB have the same FaceIndexType, which is specified in the BaseMFIFFAB definition. Multiple data components per face may be specified in the definition. The important functions of BaseMFIFFAB follow.

- ```
  BaseMFIFFAB(const IntVectSet& iggeom_in,
              const MFISBox& a_mfisBox,
              FaceIndexType a_faceType,
              int nvarin,
              bool interiorOnly=false);
  ```

  Defining constructor. The arguments specify the valid domain in the form of an IntVectSet, the FaceIndexType 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 (only relevant if a_faceType is xFace, yFace, or zFace).

- `void setVal(T value);`

  Set a value everywhere. Every data location in this BaseMFIFFAB is set to value.

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

  Set a value everywhere, for the phase a_phase. Every data location in the given phase in this BaseMFIFFAB is set.
  ```

- `void copy(const Box& a_intBox, const Interval& Cd,`
  `const Box& a_toBox`
  `const BaseMFIFFAB<T>& a_source, const Interval& Cs);`

  Copy the contents of another `BaseMFIFFAB` into this `BaseMFIFFAB` over the specified regions and intervals. Both `BaseMFIFFAB`s must be defined for the same `FaceIndexType`.

- `int nComp() const;`

  Return the number of data components of this `BaseMFIFFAB`.

- `FaceIndexType type() const;`

  Return the `FaceIndexType` of the faces of this `BaseMFIFFAB`.

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

  Indexing operator. Return a reference to the contents of this `BaseMFIFFAB`, at the specified face and data component, where `varlocin` may range from zero, to *nvar-1*. The returned object is a modifiable lvalue.

5.2 Class `BaseMFIVFAB<T>`

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

- `BaseMFIVFAB(const IntVectSet& iggeom_in,`
  `const MFISBox& 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 `BaseMFIVFAB` is set to `value`.

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

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

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

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

- `int nComp() const;`

  Return the number of data components of this `BaseMFIVFAB`.

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

  Indexing operator. Return a reference to the contents of this `BaseMFIVFAB`, at the specified VoF and data component, where `varlocin` may range from zero to *nvar-1*. The returned object is a modifiable lvalue.

5.3 Class `BaseMFCellFAB<T>`

A `BaseMFCellFAB<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 `BaseMFCellFAB` 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 `BaseMFIVFAB`. `BaseMFCellFAB` 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 `BaseMFCellFAB` is defined as follows.

- ```
  void define(const MFISBox a_mfis,const Box& a_region,
              int a_nVar);
  ```

  Full define function. Defines the domain of the `BaseMFCellFAB` to be the intersection of the input `Box` and the domain of the input `MFISBox`. 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 BaseMFCellFAB<T>& a_srcFab,
            const Interval& srcInt);
  ```

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

- `int nComp() const;`

  Return the number of data components of this `BaseMFCellFAB`.

- `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.

5.4 Class `MFCellFAB`

An `MFCellFAB` 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 `MFISBox`. It is an extension of a `BaseMFCellFAB<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 `BaseMFIVFAB<Real>`. `MFCellFAB` 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. `MFCellFAB` has all the functions of `BaseMFCellFAB<Real>` and the following extra functions:

- `void reMap()`

  Updates the data in this MFCellFab to the current data graph. Interpolates existing data as necessary to fit the current data graph. This function is generally called for each `MFCellFab` after the geometry in the underlying `MFIndexSpace` has been advanced.

- `FArrayBox& getRegFAB();`

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

- ```
  MFCellFAB& operator+=(const Real& a_valin);
  MFCellFAB& operator-=(const Real& a_valin);
  MFCellFAB& operator*=(const Real& a_valin);
  MFCellFAB& 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.

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

5.5 Class BaseMFFaceFAB<T>

A BaseMFFaceFAB<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 MFISBox. At every uncovered face in this intersection, the BaseMFFaceFAB 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 BaseMFIFFAB. BaseMFFaceFAB 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 BaseMFFaceFAB are defined as follows.

- ```
  void define(const MFISBox& a_mfis,
              const Box& a_region,
              FaceIndexType a_faceType,
              int a_nVar,
              bool interiorOnly = false);
  ```

  Full define function. Defines the domain of the BaseMFFaceFAB to be the intersection of the input Box and the faces of the input MFISBox 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 `BaseMFFaceFAB`.

- `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,
            const Box& a_RegionTo,
            const MFFaceFAB<T>& a_source,
            const Interval& a_srcInt);
  ```

  Copy the data from `a_source` into the current `BaseMFFaceFAB` over regions and intervals specified. The two `MFFaceFAB`s 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.

5.6 Class `MFFaceFAB`

An `MFFaceFAB` 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 `BaseMFFaceFAB<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 `BaseMFIFFAB<Real>`. `MFFaceFAB` has all the functions of `BaseMFFaceFAB<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.

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

- ```
  MFFaceFAB& operator+=(const Real& a_valin);
  MFFaceFAB& operator-=(const Real& a_valin);
  MFFaceFAB& operator*=(const Real& a_valin);
  MFFaceFAB& 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 MFISBox. Its important functions are as follows

- ```
  VoFIterator(const IntVectSet& a_ivs,
              const MFISBox& a_mfisBox,
              const int phase);
  void define(const IntVectSet& a_ivs,
              const MFISBox& a_mfisBox,
              const int phase);
  ```

  Define the VoFIterator with the input IntVectSet and the MFISBox. The IntVectSet defines the points that will be iterated over and should be contained within the region of MFISBox. 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(MFCellFAB& a_phi,
                   const IntVectSet& a_ivs,
                   const MFISBox& 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, 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.

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 MFISBox& a_mfisBox,
               const int a_phase,
               const FaceIndexType& a_faceType,
               const FaceStop::WhichFaces& a_location);

  void define(const IntVectSet& a_ivs,
              const MFISBox& 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(MFFaceFAB& a_phi,
                       const IntVectSet& a_ivs,
                       const MFISBox& 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, 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 AMR Tools for Multifluid computations

Our strategy for computing adaptive mesh refinement (AMR) solutions for multifluid problems will be to refine the multifluid interfaces to the finest level possible. Extending the `AMRTools` infrastructure to the multifluid case is straightforward because multifluid interfaces do not cross coarse-fine interfaces.

As in the "AMRTools" section in [CGL+00], we describe the algorithmic and software support for implementing AMR algorithms in a multifluid setting. For more detail about the basic design philosophy for the software in this section, see [CGL+00].

7.1 C++ Classes for Two-Level Operators

7.1.1 The Class `MFCoarseAverage`

This class sets data on a level equal to an average of the data from a finer level.

- ```
  void
  define(const MFISLayout& a_fine_domain,
         int a_numcomps,
         int a_ref_ratio);
  ```

  Arguments:

  - `a_fine_domain` (not modified): the fine level domain.
  ```

- `a_numcomps` (not modified): the number of components of coarse and fine data sets.
- `a_ref_ratio` (not modified): the refinement ratio n_{ref}.

- ```
  void
  averageToCoarse(LevelData<MFCellFAB>& a_coarse_data,
                  const LevelData<MFCellFAB>& a_fine_data);
  ```
  Replaces coarse data with the average of fine data, in the valid fine domain.
  Arguments:
  - `a_coarse_data` (modified): coarse data set, destination of averaging.
  - `a_fine_data` (not modified): fine data set, source of averaging.

7.2 The Class `MFFineInterp`

This class fills the valid region of a level of data by piecewise linear interpolation from data on a coarser level of refinement, using the piecewise linear interpolation operator described in section.

- ```
  void
  define(const MFISLayout& a_fine_domain,
         int a_numcomps,
         int a_ref_ratio,
         const ProblemDomain& a_problem_domain);

  void
  define(const MFISLayout& a_fine_domain,
         int a_numcomps,
         int a_ref_ratio,
         const Box& a_problemDomain)
  ```
  Arguments:
  - `a_fine_domain` (not modified): domain of the fine level.
  - `a_numcomps` (not modified): number of components of the coarse and fine data.
  - `a_ref_ratio` (not modified): the refinement ratio $N_r = \Delta x^c / \Delta x^f$.
  - `a_problem_domain` (not modified): the problem domain in the fine level index space.

- ```
  void
  interpToFine(LevelData<MFCellFAB>& a_fine_data,
  ```
  ```

```
const LevelData<MFCellFAB>& a_coarse_data);
```

Replaces fine data by interpolation from coarse data.
Arguments:

- `a_fine_data` (modified): the fine data set, destination of interpolation.
- `a_coarse_data` (not modified): the coarse data set, source of interpolation.

7.3 The Class `MFPiecewiseLinearFillPatch`

This class fills some of the ghost cells of a level of data by piecewise linear interpolation from data on a coarser level of refinement. It is intended to be used in the context of a multilevel time-dependent adaptive mesh refinement (AMR) calculation. The algorithm used is that described in [CGL+00]. The interface described here is slightly more general, as it allows for the coarse grid data to be a linear combination of the form

$$\varphi^{c,valid} = \alpha\varphi^{c,old} + (1-\alpha)\varphi^{c,new}$$

Note that cells outside the problem domain are never filled; it is the application developer's responsibility to fill them elsewhere according to the application-specific boundary conditions. Cells outside the computational domain in periodic direction, however, are considered to be inside the problem domain and are filled.

- ```
  void
  define(const MFISLayout& a_fine_domain,
         const MFISLayout& a_coarse_domain,
         int a_num_comps,
         const ProblemDomain& a_coarse_problem_domain,
         int a_ref_ratio,
         int a_interp_radius);

  void
  define(const MFISLayout& a_fine_domain,
         const MFISLayout& a_coarse_domain,
         int a_num_comps,
         const Box& a_coarse_problem_domain,
         int a_ref_ratio,
         int a_interp_radius);
  ```

  Defines domains of the levels and other persistent data.
  Arguments:

  - `a_fine_domain` (not modified): domain of fine level.
  - `a_coarse_domain` (not modified): domain of coarse level.
  ```

- a_num_comps (not modified): number of components of state vector.
- a_coarse_problem_domain (not modified): problem domain on the coarse level.
- a_ref_ratio (not modified): refinement ratio.
- a_interp_radius (not modified): number of layers of fine ghost cells to fill by interpolation.

- ```
  void
  fillInterp(LevelData<MFCellFAB>& a_fine_data,
             const LevelData<MFCellFAB>& a_old_coarse_data,
             const LevelData<MFCellFAB>& a_new_coarse_data,
             Real a_time_interp_coef,
             int a_src_comp,
             int a_dest_comp,
             int a_num_comp);
  ```

  Fills the ghost cells of the fine level data by interpolation.
  Arguments:
  - a_fine_data (modified): fine data whose ghost cells are to be filled.
  - a_old_coarse_data (not modified): coarse level data at the old time.
  - a_new_coarse_data (not modified): coarse level data at the new time.
  - a_time_interp_coef (not modified): time interpolation coefficient, α. It is required that $0 \leq \alpha \leq 1$.
  - a_src_comp (not modified): starting coarse data component.
  - a_dest_comp (not modified): starting fine data component.
  - a_num_comp (not modified): number of data components to be interpolated.

7.4 The Class MFQuadCFInterp

The class MFQuadCFInterp interpolates data onto the ghost cells on the faces of a LevelData<MFCellFAB>, using the algorithm described in [CGL+00]. It uses one-sided differencing in places where the stencil to do full centered differencing is partially covered by finer grids. The user interface of MFQuadCFInterp is given as follows.

- ```
  void define(const MFISLayout& a_fineBoxes,
              const MFISLayout* a_coarBoxes,
              Real a_dx,
              int a_refRatio,
              int a_nComp,
              const ProblemDomain& a_domf);
  ```

```
void define(const MFISLayout& a_fineBoxes,
            const MFISLayout* a_coarBoxes,
            Real a_dx,
            int a_refRatio,
            int a_nComp,
            const Box& a_domf);
```

Full define function. This makes all coarse-fine information and sets internal variables.
Arguments:

- `a_fineBoxes` (not modified): The grids at the current level.
- `a_coarBoxes` (not modified): The grids at the next coarser level in the AMR hierarchy.
- `a_dx` (not modified): The grid spacing at the current level.
- `a_refRatio` (not modified): The refinement ratio between this level and the next coarser level in the AMR hierarchy.
- `a_nComp` (not modified): The number of components in the data to be interpolated.
- `a_domf` (not modified): The domain at the current level.

- `void coarseFineInterp(LevelData<MFCellFAB>& a_phif,`
  `const LevelData<MFCellFAB>& a_phic) const;`

  Coarse-fine interpolation operator. Fills all the ghost cells on all the faces of the `LevelData<MFCellFAB> a_phif` with values interpolated with `a_phic`.
  Arguments:

  - `a_phif` (modified): The solution at the current level.
  - `a_phic` (not modified): The solution at the next coarser level in the AMR hierarchy.

7.5 The Class `MFLevelFluxRegister`

`MFLevelFluxRegister` manages the manipulations at coarse-fine boundaries associated with maintaining conservation form of cell-centered discretizations of the divergence operator, using the algorithm described in [CGL+00]. Unlike the previous operators, `MFLevelFluxRegister` holds data, corresponding to the flux register $\delta \vec{F}^f$ defined in [CGL+00] The class also manages the manipulation of that data.

The user interface for `MFLevelFluxRegister` is as follows.

- `void define(const MFISLayout& a_dbl,`
  `const MFISLayout& a_dblCoarse,`
  `const ProblemDomain& a_dProblem,`
  `int a_nRefine,`
  `int a_nComp);`

  `void define(const MFISLayout& a_dbl,`
  `const MFISLayout& a_dblCoarse,`
  `const Box& a_dProblem,`
  `int a_nRefine,`
  `int a_nComp);`

  Defines the internal state of the flux register, allocating space for the register itself, as well as the indexing information required to perform the other operations.
  Arguments:

  - `a_dbl` (not modified): The grids at the current level.
  - `a_dblCoarse` (not modified): The grids at the next coarser level in the AMR hierarchy.
  - `a_dProblem` (not modified): The domain at the current level.
  - `a_nRefine` (not modified): The refinement ratio between this level and the next coarser level.
  - `a_nComp` (not modified): The number of variables used in the computation.

- `void setToZero()` Initializes the register to all zeros.

- `void incrementCoarse(MFFaceFAB& a_coarseFlux,`
  `Real a_scale,`
  `const DataIndex& a_coarsePatchIndex,`
  `const Interval& a_srcInterval,`
  `const Interval& a_dstInterval,`
  `int a_dir);`

  Increments the register with data from `a_coarseFlux`, multiplied by `a_scale` (α): $\delta F_d^f := \delta F_d^f + \alpha F_d^c$, for all of the d-faces where the input flux (defined on a single rectangle) coincide with the d-faces on which the flux register is defined. `a_coarseFlux` contains fluxes in the `a_dir` direction for the grid `a_dblCoarse[a_coarsePatchIndex]`. Only the registers corresponding to the low faces of `a_dblCoarse[a_coarsePatchIndex]` in the `a_dir` direction are incremented (this avoids double-counting at coarse-coarse interfaces. `a_srcInterval` gives the `Interval` of components of `a_coarseFlux` that correspond to `a_dstInterval` of components of the flux register.
  Arguments:

- a_coarseFlux (not modified): Flux to put into the flux register. This is not const because its box is shifted back and forth - no net change occurs.
- a_scale (not modified): Factor by which to multiply a_coarseFlux in flux register.
- a_coarsePatchIndex (not modified): Index which corresponds to which box in the LevelData<MFCellFAB> solution from which a_coarseFlux was calculated.
- a_srcInterval (not modified): The Interval of components to put into the flux register.
- a_dstInterval (not modified): The Interval of components of the flux register into which the flux data is put.
- a_dir (not modified): Direction of faces upon which fluxes live.

- ```
  void incrementFine(MFFaceFAB& a_fineFlux,
                     Real a_scale,
                     const DataIndex& a_finePatchIndex,
                     const Interval& a_srcInterval,
                     const Interval& a_dstInterval,
                     int a_dir,
                     Side::LoHiSide a_sd);
  ```

  Increments the register with the average over each face of data from a_fineFlux, scaled by a_scale (α): $\delta F_d^f = \delta F_d^f + \alpha < F_d^f >$, for all of the d-faces where the input flux (defined on a single rectangle) cover the d-faces on which the flux register is defined. a_fineFlux contains fluxes in the a_dir direction for the grid a_dbl[a_finePatchIndex]. Only the register corresponding to the direction a_dir and the side a_sd is initialized. a_srcInterval and a_dstInterval are as above.

  Arguments:

  - a_fineFlux (not modified): Flux to put into the flux register. This is not const because its box is shifted back and forth - no net change occurs.
  - a_scale (not modified): Factor by which to multiply a_fineFlux in flux register.
  - a_finePatchIndex (not modified): Index which corresponds to which box in the LevelData<MFCellFAB> solution from which a_fineFlux was calculated.
  - a_srcInterval (not modified): The Interval of components to put into the flux register.
  - a_dstInterval (not modified): The Interval of components of the flux register into which the flux data is put.
  ```

- `a_dir` (not modified): Direction of faces upon which fluxes live.
- `a_sd` (not modified): Side of the fine face where coarse-fine interface lies.

- ```
  void reflux(LevelData<MFCellFAB>& a_uCoarse,
              const Interval& a_coarse_interval,
              const Interval& a_flux_interval,
              Real a_scale);
  ```

  Increments `a_uCoarse` with the reflux divergence of the contents of the flux register, scaled by `a_scale` (α): $U^c := U^c + \alpha D_R(\delta \vec{F})$. `a_flux_interval` gives the `Interval` of components of the flux register that correspond to `a_coarse_interval` of components of `a_uCoarse`.
  Arguments:
  - `a_uCoarse` (modified): `LevelData<MFCellFAB>` that gets modified by refluxing.
  - `a_coarse_interval` (not modified): The `Interval` of components to put into `a_uCoarse`.
  - `a_flux_interval` (not modified): The `Interval` of components to use from the flux register.
  - `a_scale` (not modified): Factor by which to scale the flux register.

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.
  ```