`AMRNodeElliptic` User Guide: on irregular problem domains

Peter McCorquodale

April 15, 2003

Disclaimer

Contents

Chapter 1

Introduction

This document describes an extension to the Chombo package [CGL+00] for solving elliptic equations using adaptive mesh refinement on multiple levels with *node*-centered data on non-rectangular domains. See the `AMRNodeElliptic` design document [McC02] for a description of the algorithms used.

Chapter 2 describes the subroutines that the user must supply to describe the geometry and the operator. Chapter 3 describes the class `NodeFArrayBox` and functions that use this class to manipulate node-centered data. Interfaces for elliptic equation solvers are described in chapter 4. Some internal classes of the solvers are described in chapter 5. User interfaces for physical boundary conditions are described in chapter 6.

The `AMRNodeElliptic` package requires that Chombo library be installed. See chapter 1 of the Chombo design document [CGL+00] for requirements and installation of Chombo.

1.1 Constructing a makefile with `AMRNodeElliptic`

This section specifies how to construct a `GNUmakefile` compatible with the `AMRNodeElliptic` extension to Chombo. Most of this is also described in section 1.2.3 in the Chombo design document [CGL+00].

To make an external makefile, first say where Chombo is installed by specifying `CHOMBO_HOME` and the base name of the application by specifying `ebase`.

```
####define the location of Chombo
CHOMBO_HOME = ../../../../ChomboLib

####define the base name of the application
ebase := myProg
```

Then specify the names of the Chombo libraries that should be linked:

```
##
## names of Chombo libraries needed by this program, in order of search.
```

```
##
LibNames := AMRTools BoxTools
```

Now specify the path and source files external to Chombo and `AMRNodeElliptic`:

```
## path to look for source files not in Chombo
VPATH_LOCAL = .
INCLUDES_LOCAL = -I.

## C++ sources go here
CXX_SOURCES = NodeMaskPoissonBC.cpp testFuncs.cpp testRunPoint.cpp

## Chombo Fortran sources go here
FORT_SOURCES = testProb.ChF OutsideSphereMask.ChF

## standard C sources go here
C_SOURCES = myc1.c myc2.c

## standard Fortran sources go here
F_SOURCES = myfort1.f myfort2.f
```

So far, the description of the makefile is as for any makefile that uses the Chombo library. When using `AMRNodeElliptic`, end the makefile with the following lines:

```
####define location of AMRNodeElliptic source directory
SRC_DIR = ../../src
## these commands include the AMRNodeElliptic source
include $(SRC_DIR)/Make.package
INCLUDES_LOCAL += -I$(SRC_DIR)
VPATH_LOCAL += $(SRC_DIR)

## end of makefile for use with Chombo
include $(CHOMBO_HOME)/mk/Make.example
```

Chapter 2

Subroutines Needed for Non-Rectangular Domains

For non-rectangular domains, the user must supply two Fortran subroutines described in the `AMRNodeElliptic` design document [McC02]:

- `reachablenodes` specifies the geometry;
- `nodalcoefficients` specifies the coefficients of the operator.

2.1 The `reachablenodes` subroutine

This subroutine should return an array telling which nodes are in the domain and which are reachable from their neighbors along lines parallel to the coordinate axes, where "reachable" in this context means that the connecting segment lies entirely in the domain. We represent the domain by Ω.

```
subroutine reachablenodes(CHF_CONST_REAL[dx], CHF_FIA[mask])
```

Arguments:

- `dx` is the mesh spacing.
- `mask` is node-centered and has $2\mathbf{D}+1$ components.

At any node index $\boldsymbol{i}$ of the underlying `Box` of `mask`, the corresponding point in space is $\boldsymbol{x} = \boldsymbol{x}_0 + (\boldsymbol{i} - \frac{1}{2}\boldsymbol{u}) \cdot \mathtt{dx}$, and the components of `mask` should be set to 0 or 1 as follows (see Figure 2.1).

- component 0 is set to 1 if $\boldsymbol{x} \in \Omega$, and 0 otherwise.
- for each $d = 0, \ldots, \mathbf{D}-1$, component $2d+1$ is set to 1 if the entire segment $[\boldsymbol{x}, \boldsymbol{x} + \boldsymbol{e}^d \cdot \mathtt{dx}] \subset \Omega$, and 0 otherwise.

- for each $d = 0, \ldots, \mathbf{D} - 1$, component $2d + 2$ is set to 1 if the entire segment $[\boldsymbol{x}, \boldsymbol{x} - \boldsymbol{e}^d \cdot \mathtt{dx}] \subset \Omega$, and 0 otherwise.

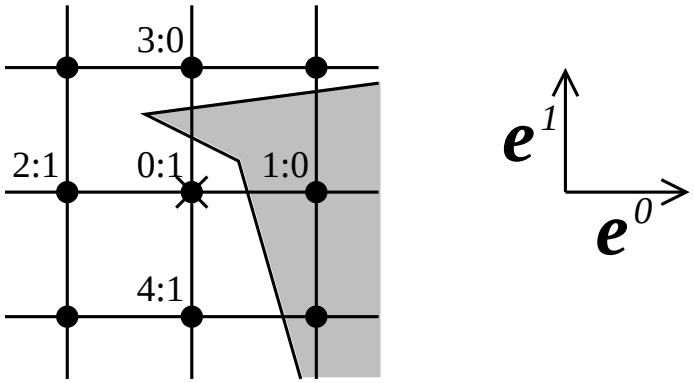

Figure 2.1: Components of the `mask` array at the center node indicated by $\times$, in two dimensions. The notation $i : v$ beside a node indicates that at node $\times$, component i has value v, as determined by the reachability of the labeled node from $\times$. The shaded area is outside the domain. We have $v = 1$ if the node is reachable from $\times$, and $v = 0$ if it is not. Note that the point at top center is considered to be *not* reachable from $\times$, even though it is in the domain, because the connecting segment does not lie entirely in the domain.

2.2 The `nodalcoefficients` subroutine

This subroutine should return an array containing the coefficients of the discrete operator at each point.

```
subroutine nodalcoefficients(CHF_CONST_REAL[dx], CHF_FRA[coeffs])
```

Arguments:

- `dx` is the mesh spacing.
- `coeffs` is node-centered and has $2\mathbf{D} + 2$ components.

At any node index $\boldsymbol{i}$ of the underlying `Box` of `coeffs`, the components of `coeffs` are set to the coefficients of the stencil for node $\boldsymbol{i}$, as follows:

- component 0 is the coefficient of the value with index $\boldsymbol{i}$.
- for each $d = 0, \ldots \mathbf{D} - 1$, component $2d + 1$ is the coefficient of the value with index $\boldsymbol{i} + \boldsymbol{e}^d$.

- for each $d = 0, \ldots \mathbf{D} - 1$, component $2d + 2$ is the coefficient of the value with index $\boldsymbol{i} - \boldsymbol{e}^d$.
- component $2\mathbf{D} + 1$ is the constant coefficient.

Summarizing, if $c_{\boldsymbol{i}}^0, \ldots c_{\boldsymbol{i}}^{2\mathbf{D}+1}$ are the coefficients for node $\boldsymbol{i}$ at a level with mesh spacing h, then the operator at point $\boldsymbol{i}$ is

$$c_{\boldsymbol{i}}^0 \cdot \varphi_{\boldsymbol{i}}^{N,l} + \sum_{d=0,\ldots,\mathbf{D}-1} (c_{\boldsymbol{i}}^{2d+1} \cdot \varphi_{\boldsymbol{i}+\boldsymbol{e}^d}^{N,l} + c_{\boldsymbol{i}}^{2d+2} \cdot \varphi_{\boldsymbol{i}-\boldsymbol{e}^d}^{N,l}) + c_{\boldsymbol{i}}^{2\mathbf{D}+1}.$$

Chapter 3

Node-Centered Data

3.1 The class NodeFArrayBox

The NodeFArrayBox class is a wrapper for a node-centered FArrayBox. This class is used in LevelData<NodeFArrayBox> with an underlying *cell*-centered DisjointBoxLayout.

The reason for introducing a new NodeFArrayBox class, instead of just using node-centered FArrayBoxes, is to allow the use of LevelData. A LevelData<FArrayBox> based on node-centered FArrayBoxes would have an underlying DisjointBoxLayout of node-centered boxes, and no two of the boxes could be adjacent, because if they were, they would share points, and the layout would not be disjoint. To get around this problem, we use a cell-centered DisjointBoxLayout and build a LevelData<NodeFArrayBox> on it, with the data located at the surrounding nodes of the cell-centered grids. Data at nodes shared by two or more grids is stored redundantly in all of the corresponding NodeFArrayBoxes.

The API for NodeFArrayBox is as follows.

- NodeFArrayBox()

  Default constructor. User must subsequently call define.

- Constructor.

  ```
  NodeFArrayBox(
      const Box& a_bx,
      int        a_nComp = 1)
  ```

  Creates a NodeFArrayBox object.
  Inputs:

  - a_bx specifies the cell-centered box. The data lie on the surrounding nodes of this box.
  - a_nComp specifies the number of components of the FArrayBox.

- `define(`
  `const Box& a_bx,`
  `int a_nComp = 1)`

  Defines a `NodeFArrayBox` object. Arguments are the same as those of the constructor above.

- `FArrayBox& getFab()`

  Returns a reference to the node-centered `FArrayBox` containing the data.

- `const Box& box()`

  Returns the cell-centered box of which the surrounding nodes contain the data.

- `Real norm(`
  `const Real a_dx,`
  `const int a_p = 2,`
  `const int a_startComp = 0,`
  `const int a_numComp = 1) const`

  For `a_p > 0`, returns the $L^{\texttt{a\_p}}$ norm of the data in `a_numComp` components, starting at component number `a_startComp`. The norm is computed by integration using the trapezoidal rule with mesh spacing `a_dx`, over those nodes that are in the domain as determined by calling `reachablenodes`. For `a_p = 0`, returns the L^∞ norm of the same data. An overloaded version of this function performs its operations over a subbox. Another overloaded version takes as an argument the `mask` array returned from `reachablenodes`, so that `reachablenodes` need not be called in this function.

- `Real maxnorm(`
  `const int a_startComp = 0,`
  `const int a_numComp = 1) const`

  Returns the maximum value of the data in `a_numComp` components, starting at component number `a_startComp`, in those nodes that are in the domain as determined by calling `reachablenodes`. An overloaded version of this function performs its operations over a subbox. Another overloaded version takes as an argument the `mask` array returned from `reachablenodes`, so that `reachablenodes` need not be called in this function.

Principal data members

- `Box m_box`. The cell-centered box of which the surrounding nodes contain the data.
- `FArrayBox m_fab`. The node-centered data.

3.2 Notes on LevelData<NodeFArrayBox>::copyTo

The API for LevelData<NodeFArrayBox> is described in the Chombo design document [CGL+00], but one part of it deserves particular attention because it does not work in the same way.

```
void copyTo(
    const Interval&   a_srcComps,
    BoxLayoutData<T>& a_dest,
    const Interval&   a_destComps) const

void copyTo(
    const Interval& a_srcComps,
    LevelData<T>&   a_dest,
    const Interval& a_destComps) const
```

The LevelData<NodeFArrayBox>::copyTo function must be used with caution. If the source LevelData and the destination, a_dest, do not have the same underlying grid layout, then it is possible that some data will not be copied from the source to the destination. The reason is that, in determining points where data should be copied, copyTo finds intersections of the underlying *cell*-centered boxes and then copies at the surrounding nodes. See Figure 3.1 for an example in which data are not copied at some of the nodes shared by source and destination.

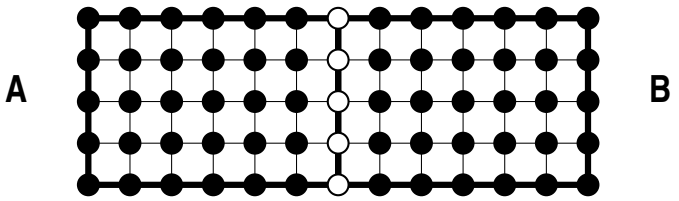

Figure 3.1: Suppose A and B are LevelData<NodeFArrayBox>es, each having an underlying grid layout of a single grid, and these two grids abut, as shown above where the nodes of intersection are indicated by hollow circles. Then the statement A.copyTo(A.interval(), B, B.interval()) results in *no* data being copied from A to B, even though they share nodes, because the two grids have no *cells* in common.

If all cells in the grid layout of the destination of the copyTo are also in the grid layout of the source, then this problem of uncopied data does not arise. For example, when copying all of the data on one AMR level to a coarsened version of the grids on a finer AMR level, copyTo works as expected, because of the nesting requirement for AMR levels.

But whenever we want to use LevelData<NodeFArrayBox>::copyTo from src where the underlying grid layout of the destination dest may contain cells that are not in the grid layout of this, we use a Copier object, as follows:

```
Copier myCopier(src.getBoxes(), dest.getBoxes(), IntVect::Unit);
src.copyTo(srcComps, dest, destComps, myCopier);
```

3.3 Node-centered norms

3.3.1 Norms of valid data on a single level

- ```
  Real norm(const LevelData<NodeFArrayBox>& a_phi,
            const ProblemDomain&            a_domain,
            const DisjointBoxLayout*        a_finerGridsPtr,
            const int                       a_nRefFine,
            const Real                      a_dx,
            const Interval                  a_comps,
            const int                       a_p,
            bool                            a_verbose = true)
  ```

- ```
  Real norm(const LevelData<NodeFArrayBox>& a_phi,
            const Box&                      a_domain,
            const DisjointBoxLayout*        a_finerGridsPtr,
            const int                       a_nRefFine,
            const Real                      a_dx,
            const Interval                  a_comps,
            const int                       a_p,
            bool                            a_verbose = true)
  ```

  Returns the norm of valid data on one level.
  Inputs:

  - a_phi is the data at this level.
  - a_domain is the entire cell-centered problem domain at this level.
  - a_finerGridsPtr is a pointer to the grid layout at the next finer level, or NULL if there is no finer level.
  - a_nRefFine is the refinement ratio between this level and the next finer level.
  - a_dx is the mesh spacing at this level.
  - a_comps specifies the components over which to take the norm.
  - a_p specifies which norm to take: if a_p = 0 then the max (L^∞) norm, and if a_p > 0 then the $L^{\text{a\_p}}$ norm.
  - a_verbose specifies whether to display the norm of each grid.

  This function calls reachablenodes. An overloaded version takes as an argument the mask array returned from reachablenodes, so that reachablenodes need not be called in this function.

Other overloaded versions of this function include arguments with the exterior boundary nodes of this level, and the interior boundary nodes of the coarsened finer-level grids.

- `Real maxnorm(const LevelData<NodeFArrayBox>& a_phi,`
  `const ProblemDomain& a_domain,`
  `const DisjointBoxLayout* a_finerGridsPtr,`
  `const int a_nRefFine,`
  `const Interval a_comps,`
  `bool a_verbose = true)`

- `Real maxnorm(const LevelData<NodeFArrayBox>& a_phi,`
  `const Box& a_domain,`
  `const DisjointBoxLayout* a_finerGridsPtr,`
  `const int a_nRefFine,`
  `const Interval a_comps,`
  `bool a_verbose = true)`

  Returns the max (L^∞) norm of valid data on one level. Inputs are a subset of those for the `norm` function above, and there are overloaded versions as with `norm`.

3.3.2 Norms of data on multiple levels

- `Real norm(`
  `const Vector<LevelData<NodeFArrayBox>* >& a_phi,`
  `const Vector<ProblemDomain>& a_domain,`
  `const Vector<int>& a_nRefFine,`
  `const Real a_dxCrse,`
  `const Interval a_comps,`
  `const int a_p,`
  `const int a_lBase,`
  `bool a_verbose = true)`

- `Real norm(`
  `const Vector<LevelData<NodeFArrayBox>* >& a_phi,`
  `const Vector<Box>& a_domain,`
  `const Vector<int>& a_nRefFine,`
  `const Real a_dxCrse,`
  `const Interval a_comps,`
  `const int a_p,`
  `const int a_lBase,`
  `bool a_verbose = true)`

  Returns the norm of valid data on multiple levels.
  Inputs:

- a_phi is a vector of pointers to data at multiple levels.
- a_domain specifies the entire cell-centered problem domain at all levels of resolution. Vector index corresponds to level number.
- a_nRefFine specifies the refinement ratios between adjacent levels. a_nRefFine[ilev] is the refinement ratio between levels ilev and ilev+1. a_nRefFine[ilev] must be a power of two, with a_nRefFine[ilev] ≥ 1.
- a_dxCrse is the mesh spacing at level a_lBase.
- a_comps specifies the components over which to take the norm.
- a_lBase is the index of the coarsest level on which the norm is to be computed.
- a_p specifies which norm to take: if a_p = 0 then the max (L^∞) norm, and if a_p > 0 then the $L^{\text{a\_p}}$ norm.
- a_verbose specifies whether to display the norm of each grid.

- ```
  Real maxnorm(
      const Vector<LevelData<NodeFArrayBox>* >& a_phi,
      const Vector<ProblemDomain>&              a_domain,
      const Vector<int>&                        a_nRefFine,
      const Interval                            a_comps,
      const int                                 a_lBase,
      bool                                      a_verbose = true)
  ```

- ```
  Real maxnorm(
      const Vector<LevelData<NodeFArrayBox>* >& a_phi,
      const Vector<Box>&                        a_domain,
      const Vector<int>&                        a_nRefFine,
      const Interval                            a_comps,
      const int                                 a_lBase,
      bool                                      a_verbose = true)
  ```

  Returns the max (L^∞) norm of valid data on multiple levels. Inputs are a subset of those for the multi-level norm function above.

3.4 Dot product for node-centered data

The DotProductNodes functions return the dot product of data distributed over two instances of LevelData<NodeFArrayBox> with the same layout. The dot product is the sum, over all valid nodes and over specified components, of the product of the data in the two data holders. That is, for data sets f and g on nodes Ω_N, the dot product is

$$f \cdot g = \sum_{\boldsymbol{i} \in \Omega_{N,valid}} f_{\boldsymbol{i}}\, g_{\boldsymbol{i}}.$$

- `Real DotProductNodes(`
  `const LevelData<NodeFArrayBox>& a_dataOne,`
  `const LevelData<NodeFArrayBox>& a_dataTwo,`
  `const BoxLayoutData< BaseFab<int> >& a_mask,`
  `const ProblemDomain& a_domain,`
  `const Interval& a_comps)`

- `Real DotProductNodes(`
  `const LevelData<NodeFArrayBox>& a_dataOne,`
  `const LevelData<NodeFArrayBox>& a_dataTwo,`
  `const BoxLayoutData< BaseFab<int> >& a_mask,`
  `const Box& a_domain,`
  `const Interval& a_comps)`

  Returns the dot product of data on one level.
  Inputs:

  - `a_dataOne` contains the first set of data on a level.
  - `a_dataTwo` contains the second set of data on a level.
  - `a_mask` is the mask array from `REACHABLENODES` on this level.
  - `a_domain` is the cell-centered problem domain on this level.
  - `a_comps` specifies the components to be used in computing the dot product.

  An overloaded version of this function includes the exterior boundary nodes of the level as an argument. It is more efficient to use this other version when the dot product is to be computed multiple times on data with the same layout.

3.5 Functions for interior and exterior boundary nodes

An *interior node* of a grid layout is any node in the layout with the property that all of the cells adjacent to the node are also contained in the grid layout. An *interior boundary node* of a grid layout is an interior node that lies on the boundary of one or more grids in the layout. See Figure 3.2 for a sample layout with interior boundary nodes marked.

An *exterior node* or *exterior boundary node* of a grid layout is any node of the layout that is not an interior node. See Figure 3.3 for the same layout as Figure 3.2, with exterior nodes marked.

3.5.1 `interiorBoundaryNodes` functions

The `interiorBoundaryNodes` functions return the interior boundary nodes of a `DisjointBoxLayout`. The interface would be simplest if the function returned a reference to an `IntVectSet`, but because of the way that `IntVectSet` is implemented,

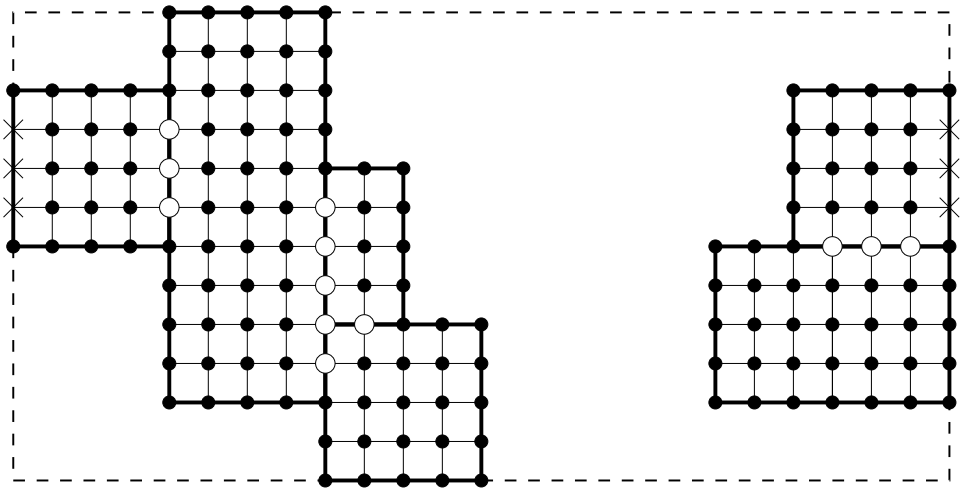

Figure 3.2: Hollow circles indicate the interior boundary nodes of a two-dimensional grid layout inside a problem domain delineated by the dashed lines. Nodes marked $\times$ are also interior boundary nodes if the problem domain is periodic in the horizontal direction.

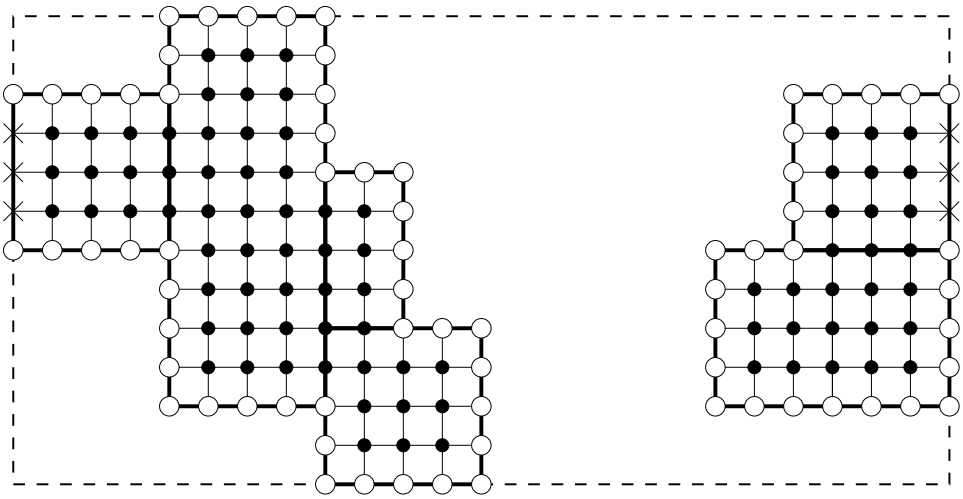

Figure 3.3: Hollow circles indicate the exterior boundary nodes of a two-dimensional grid layout inside a problem domain delineated by the dashed lines. Nodes marked $\times$ are also exterior boundary nodes if the problem domain is *not* periodic in the horizontal direction. Any node that is not an exterior boundary node is an interior node.

this would be very slow. Instead, the first interiorBoundaryNodes function returns a reference to a LayoutData< Vector<IntVectSet> > such that the union of all the IntVectSets in the Vectors contains exactly the indices of all interior boundary nodes of the input layout.

To make matters even more complicated, there is a second interiorBoundaryNodes function, which takes two DisjointBoxLayout arguments. This function returns a reference to a LayoutData that stores the indices of interior boundary nodes of one layout that are also nodes (interior or exterior) of a second layout. The output of this function is needed for calling copyInteriorNodes, described in section 3.5.3.

- ```
  void interiorBoundaryNodes(
      LayoutData< Vector<IntVectSet> >& a_IVSV,
      const DisjointBoxLayout&          a_boxes,
      const ProblemDomain&              a_domain)
  ```

- ```
  void interiorBoundaryNodes(
      LayoutData< Vector<IntVectSet> >& a_IVSV,
      const DisjointBoxLayout&          a_boxes,
      const Box&                        a_domain)
  ```

  Returns the interior boundary nodes of a layout.
  Inputs:

  - a_boxes specifies the cell-centered layout of boxes.
  - a_domain specifies the cell-centered problem domain that contains the layout.

  Output:

  - a_IVSV contains the interior boundary nodes of a_boxes in a LayoutData based on a_boxes. For each box in the layout of a_boxes, there is a Vector of IntVectSets that together contain the indices of the interior boundary nodes of the box.

- ```
  void interiorBoundaryNodes(
      LayoutData< Vector<IntVectSet> >& a_IVSV,
      const DisjointBoxLayout&          a_dest,
      const DisjointBoxLayout&          a_src,
      const ProblemDomain&              a_domain)
  ```

- ```
  void interiorBoundaryNodes(
      LayoutData< Vector<IntVectSet> >& a_IVSV,
      const DisjointBoxLayout&          a_dest,
      const DisjointBoxLayout&          a_src,
      const Box&                        a_domain)
  ```

Returns the interior boundary nodes of a source layout that are also nodes of a destination layout. This function is used when copying from data based on the source layout to data based on the destination layout.
Inputs:

- a_dest specifies the cell-centered layout of boxes on which to base the LayoutData to be returned.
- a_src specifies the cell-centered layout of boxes of which we wish to find the interior boundary nodes.
- a_domain specifies the cell-centered problem domain that contains both layouts.

Output:

- a_IVSV contains the interior boundary nodes of a_src in a LayoutData based on the layout in a_dest. For each box in the layout of a_dest, there is a Vector of IntVectSets that together contain the indices of all nodes of the box that are also interior boundary nodes of a_src.

If a_dest is the same as a_src, then the result is the same as calling the first interiorBoundaryNodes function with a_boxes set to a_dest.

3.5.2 exteriorBoundaryNodes function

- ```
  void exteriorBoundaryNodes(
      LayoutData< Vector<IntVectSet> >&       a_exterior,
      const LayoutData< Vector<IntVectSet> >& a_interior,
      const DisjointBoxLayout&                a_boxes)
  ```

  Returns the exterior boundary nodes (also called exterior nodes) of a_boxes.
  Inputs:

  - a_interior is the object returned by

    ```
    interiorBoundaryNodes(a_interior, a_boxes, a_domain)
    ```

    described in section 3.5.1.
  - a_boxes specifies the cell-centered layout of boxes.

  Output:

  - a_exterior contains the exterior boundary nodes of a_boxes in a LayoutData based on a_boxes. For each box in the layout of a_boxes, there is a Vector of IntVectSets that together contain the indices of the exterior boundary nodes of the box.
    ```

3.5.3 `copyInteriorNodes` function

The `copyInteriorNodes` function copies data from a source to a destination, but only from interior nodes of the source. Recall that the *interior* nodes are simply those that are not *exterior* nodes. This function is similar to `copyTo` but copies *valid* data only. See Figure 3.4 for an example.

- ```
  void copyInteriorNodes(
      LevelData<NodeFArrayBox>&                a_dest,
      const LevelData<NodeFArrayBox>&          a_src,
      const LayoutData< Vector<IntVectSet> >& a_IVSV)
  ```

  Copies data from the interior boundary nodes of `a_src` to `a_dest`.
  Inputs:

  - `a_src` contains the data to be copied at its interior nodes.
  - `a_IVSV` is the object returned by

    ```
    interiorBoundaryNodes(a_interior, a_dest.boxLayout(),
                          a_src.boxLayout(), a_domain)
    ```

    described in section 3.5.1.

  Output:

  - `a_dest`, at interior nodes of the layout of `a_src`, is replaced by data from `a_src`. Elsewhere, `a_dest` is unchanged.

3.5.4 `zeroBoundaryNodes` function

- ```
  void zeroBoundaryNodes(
      BoxLayoutData<NodeFArrayBox>&            a_dest,
      const LayoutData< Vector<IntVectSet> >& a_IVSV)
  ```

  Sets data to zero on specified nodes.
  Input:

  - `a_IVSV` is an object containing indices of nodes for each box, where data are to be set to zero. Usually it stores the indices of exterior boundary nodes of `a_dest`. These are obtained with the sequence of calls:

    ```
    interiorBoundaryNodes(a_IVSVint, a_dest.boxLayout(), a_domain);
    exteriorBoundaryNodes(a_IVSV, a_IVSVint, a_dest.boxLayout());
    ```

    where `a_domain` is the cell-centered problem domain.

  Output:

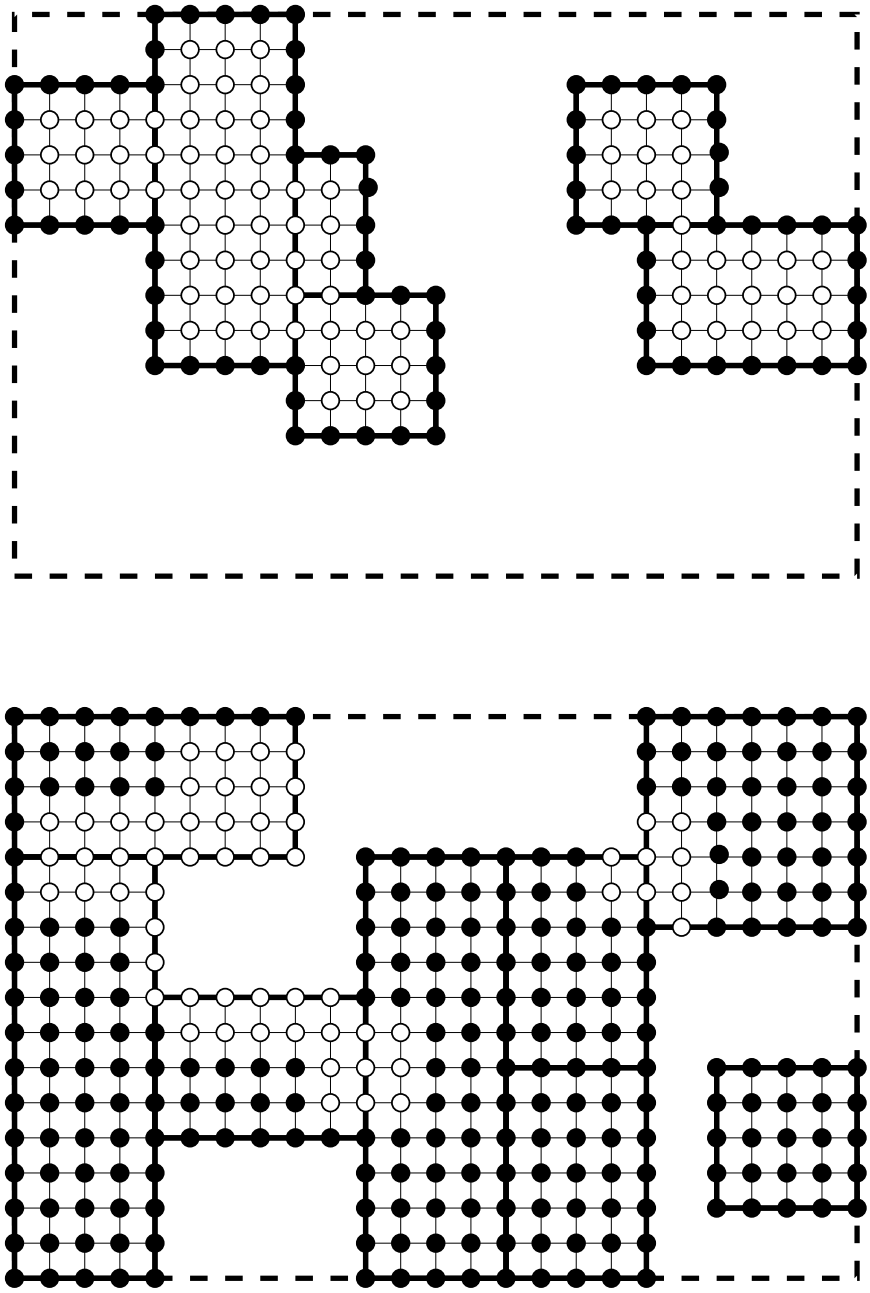

Figure 3.4: An example of a source and destination layout for `copyInteriorNodes`. The problem domain, delineated by dashed lines, is taken to be non-periodic.
Top: Hollow circles indicate the interior nodes of a source layout.
Bottom: Hollow circles indicate the nodes of a destination layout that are filled by `copyInteriorNodes` with data from the source layout above.

– a_dest is set to zero at nodes in a_IVSV. Elsewhere, a_dest is unchanged.

The underlying layouts of a_dest and a_IVSV must be the same.

Chapter 4

Interface for AMRNodeElliptic Solver

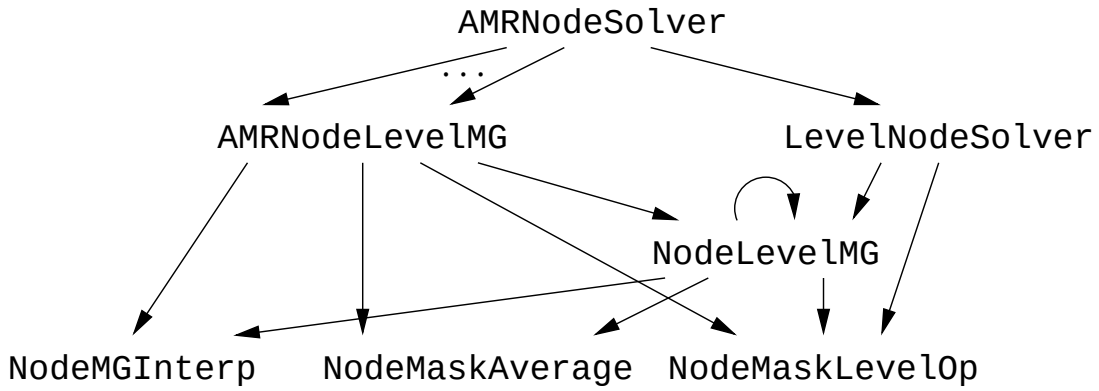

Figure 4.1: This chart shows the major classes of `AMRNodeElliptic` used by `AMRNodeSolver`. An arrow from one class to another indicates that an instance of the first class defines an instance of the second class. In each case, only one instance of the second class is defined, except that the `AMRNodeSolver` defines an instance of `AMRNodeLevelMG` for each level in the AMR hierarchy.

4.1 The class `AMRNodeSolver`

The `AMRNodeSolver` class solves an elliptic equation on an adaptive mesh refinement hierarchy of node-centered grids. The boundary conditions are obtained from a combination of interpolation from the next coarser level of grids, plus some set of physical boundary conditions on a rectangular domain.
The API is as follows.

- `void define(`
  `const Vector<DisjointBoxLayout>& a_gridsLevel,`
  `const Vector<ProblemDomain>& a_domainLevel,`

```
    const Vector<Real>&              a_dxLevel,
    const Vector<int>&               a_refRatio,
    int                              a_numLevels,
    int                              a_lBase,
    const NodeMaskLevelOp* const     a_opin,
    int                              a_minLength = 1)
```

- `void define(`
```
    const Vector<DisjointBoxLayout>& a_gridsLevel,
    const Vector<Box>&               a_domainLevel,
    const Vector<Real>&              a_dxLevel,
    const Vector<int>&               a_refRatio,
    int                              a_numLevels,
    int                              a_lBase,
    const NodeMaskLevelOp* const     a_opin,
    int                              a_minLength = 1)
```

  Sets up the internal state of an `AMRNodeSolver` object with information about the grid hierarchy and the operator.
  Inputs:

  - `a_gridsLevel` specifies the layout of the cell-centered grids on all levels. Vector index corresponds to level number.
  - `a_domainLevel` specifies the entire cell-centered problem domain at all levels of resolution. Vector index corresponds to level number.
  - `a_dxLevel` specifies the mesh spacing at all levels. Vector index corresponds to level number.
  - `a_refRatio` specifies the refinement ratios between adjacent levels. `a_refRatio[ilev]` is the refinement ratio between levels `ilev` and `ilev+1`. `a_refRatio[ilev]` must be a power of two, with `a_refRatio`$[$`ilev`$] \geq 1$.
  - `a_numLevels` is the number of AMR levels in the calculation. The length of the `Vectors` `a_gridsLevel`, `a_domainLevel` and `a_dxLevel` must be at least `a_numLevels`, and the length of the `Vector` `a_refRatio` must be at least `a_numLevels-1`.
  - `a_lBase` is the index of the coarsest level on which the solution is to be computed.
  - `a_opin` is a pointer to the `NodeMaskLevelOp` to use in solving.
  - `a_minLength` is the minimum length of the maximally coarsened box in `LevelNodeSolver`, or 0 if there is to be no coarsening. If `a_minLength` ≥ 1 then the multigrid solver will coarsen boxes only so far as to maintain the lengths of all boxes as multiples of `a_minLength`.

- `void solveAMR(`
  `Vector<LevelData<NodeFArrayBox>*>& a_phiLevel,`
  `const Vector<LevelData<NodeFArrayBox>*>& a_rhsLevel)`

  Solves the elliptic equation over the hierarchy of levels `m_lBase ... m_finestLevel` where `m_finestLevel = m_numLevels-1`. If `m_lBase > 0`, then the data at level `m_lBase-1` is used to interpolate boundary conditions at boundary cells that are not adjacent to the domain boundary. Solves to tolerance `m_tolerance`.
  Inputs:

  - `a_phiLevel` contains pointers to the current guess at the solution values for levels `(lMin = max(m_lBase-1, 0)) .. m_finestLevel`. Index in the `Vector` corresponds to level number. These values are updated in place.
  - `a_rhsLevel` contains pointers to values of the right-hand side for levels `m_lBase .. m_finestLevel`.

  Outputs:

  - `a_phiLevel` contains pointers to values of the updated solution at the same levels.

- `void AMRVCycleMG(`
  `Vector<LevelData<NodeFArrayBox>*>& a_phiLevel,`
  `const Vector<LevelData<NodeFArrayBox>*>& a_rhsLevel)`

  Does one relaxation V-cycle using an AMR multigrid solver.
  Inputs:

  - `a_phiLevel` contains pointers to current guess at values of the solution at levels `m_lBase .. m_finestLevel`. Vector index corresponds to level number.
  - `a_rhsLevel` contains pointers to values of the right-hand side on the same levels as `a_phiLevel`.

  Outputs:

  - `a_phiLevel` contains pointers to values of the updated solution.

- `Real computeResidualNorm(int a_normNtype)`

  Returns the norm of the multilevel residual on levels `m_lBase` to `m_finestLevel`, where the residual at level `ilev` has been stored in `m_amrmgLevel[ilev]->m_resid`. The residual must have been computed already before this function is called.
  Inputs:

  - `a_normType` is the type of norm: `0` for max norm, or `p > 0` for L_p norm.

- `void applyAMROperator(`
  `LevelData<NodeFArrayBox>& a_lofPhi,`
  `Vector<LevelData<NodeFArrayBox>*>& a_phiLev,`
  `int a_ilev)`

  Applies the multilevel AMR operator to the data on a level.
  Inputs:

  - `a_phiLev` contains pointers to current guess at values of the solution at levels `m_lBase .. m_finestLevel`. Vector index corresponds to level number.
  - `a_ilev` is level on which operator is to be applied. The result depends on `a_phiLev[a_ilev]`, as well as `a_phiLev[a_ilev-1]` and `a_phiLev[a_ilev+1]` if they are defined.

  Outputs:

  - `a_lofPhi` contains value of operator applied on valid region of level `a_ilev`.

- `void computeAMRResidual(`
  `LevelData<NodeFArrayBox>& a_res,`
  `Vector<LevelData<NodeFArrayBox>*>& a_phiLevel,`
  `const Vector<LevelData<NodeFArrayBox>*>& a_rhsLevel,`
  `int a_ilev)`

  Computes the residual on a level using the multilevel operator.
  Inputs:

  - `a_phiLevel` contains pointers to current guess at values of the solution at levels `m_lBase .. m_finestLevel`. Vector index corresponds to level number.
  - `a_rhsLevel` contains pointers to the right-hand side at the same levels as `a_phiLevel`.
  - `a_ilev` is level on which the residual is to be computed. The result depends on `a_rhsLevel[a_ilev]` and `a_phiLevel[a_ilev]`, as well as `a_phiLevel[a_ilev-1]` and `a_phiLevel[a_ilev+1]` if they are defined.

  Outputs:

  - `a_res` contains residual on valid region of level `a_ilev`. This is also stored in `m_amrmgLevel[ilev]->m_resid`.

- `void setTolerance(Real a_tolerance)`

  Sets the tolerance in the AMR solver: iterations end when the norm of the residual is less than `a_tolerance` times the initial residual. Default is `1.0e-10`.

- `void setOperatorTolerance(Real a_operatorTolerance)`

  Sets the "operator tolerance" of the AMR solver: iterations end if the ratio of the new residual to the old residual exceeds $1 -$ `a_operatorTolerance` and at least `m_minIter` iterations have been performed. Default is `1.0e-5`.

- `void setBottomTolerance(Real a_tolerance)`

  Sets the tolerance in the bottom solver, `LevelNodeSolver m_levelSolver`.

- `void setMaxIter(int a_maxIter)`

  Sets the maximum number of iterations in the AMR solver. Default is 42.

- `void setMinIter(int a_minIter)`

  Sets the minimum number of iterations in the AMR solver before it will terminate due to lack of convergence. Default is 5.

- `void setBottomMaxIter(Real a_maxIter)`

  Sets the maximum number of iterations in the bottom solver, `LevelNodeSolver::levelSolve` or `LevelNodeSolver::levelSolveH`. Default is 33.

- `void setNumSmoothDown(int a_numSmoothDown)`

  Sets the number of iterations of smoothing for the downward part of the V-cycle in `NodeLevelMG::mgRelax`. Default is 4.

- `void setNumSmoothUp(int a_numSmoothUp)`

  Sets the number of iterations of smoothing for the upward part of the V-cycle in `NodeLevelMG::mgRelax`. Default is 4.

- `void setNumBottomGSRB(int a_numBottomGSRB)`

  Sets the number of iterations of smoothing at the bottom of the V-cycle in `NodeLevelMG::mgRelax`. Default is 16.

- `void setBottomSmoothing(bool a_doBottomSmooth)`

  Sets whether the smoother is applied at the bottom of the V-cycle in `NodeLevelMG::mgRelax`. Default is `true`.

Principal data members

- `Vector<AMRNodeLevelMG*> m_amrmgLevel` is a vector of length `m_numLevels` containing pointers to `AMRNodeLevelMG` objects, each of which manages the data and operations on one level.

- LevelNodeSolver m_levelSolver solves elliptic equations on level m_lBase using multigrid.
- Vector<DisjointBoxLayout> m_gridsLevel, Vector<Box> m_domainLevel, Vector<Real> m_dxLevel, Vector<int> m_refRatio, int m_numLevels, int m_lBase are set to the arguments a_gridsLevel, a_domainLevel, a_dxLevel, a_refRatio, a_numLevels, a_lBase in the constructor.

4.1.1 Internal class AMRNodeLevelMG

AMRNodeLevelMG manages the data and operations for AMRNodeSolver on one level. AMRNodeSolver defines a vector of AMRNodeLevelMG objects, one for each level. This class should be considered internal to AMRNodeSolver and should not be considered part of the Chombo API.

These functions are called from AMRNodeSolver. In a multigrid V-cycle, AMRNodeSolver::AMRVCycleMG calls downSweep at each level from the finest down to one above the base level, solves at the base level with class LevelNodeSolver, and then calls upSweep at each level from one above the base up to the finest level.

- ```
  void define(
      const AMRNodeSolver* const   a_parent,
      int                          a_level,
      const NodeMaskLevelOp* const a_opin)
  ```

  Sets up the internal state of an AMRNodeLevelMG object.
  Inputs:
  - a_parent is a pointer to the AMRNodeSolver object that called this constructor.
  - a_level is the level for which this object manages data and operations.
  - a_opin is a pointer to the NodeMaskLevelOp to use in solving.

- ```
  void applyAMROperator(
      LevelData<NodeFArrayBox>&           a_Lofphi,
      Vector<LevelData<NodeFArrayBox>*>& a_phiLevel)
  ```

  Calculates the multilevel AMR operator to the data on this level. This operator uses inhomogeneous coarse-fine boundary conditions and inhomogeneous physical domain boundary conditions.
  Inputs:
  - a_phiLevel contains pointers to current guess at values of the solution. Vector index corresponds to level number. The result depends on a_phiLev[m_level], as well as a_phiLev[m_level-1] and a_phiLev[m_level+1] if they are defined.

Outputs:

- a_lofPhi contains value of operator applied on valid region of level m_level.

- void computeAMRResidual(
  Vector<LevelData<NodeFArrayBox>*>& a_phiLevel,
  const Vector<LevelData<NodeFArrayBox>*>& a_rhsLevel)

  Computes the residual on this level using the multilevel operator. The result is stored in local data m_resid.
  Inputs:

  - a_phiLevel contains pointers to current guess at values of the solution. Vector index corresponds to level number. The result depends on *a_phiLevel[m_level], as well as *a_phiLevel[m_level-1] and *a_phiLevel[m_level+1] if they are defined.
  - a_rhsLevel contains pointers to the right-hand side at the same levels as a_phiLevel. The result depends on *a_rhsLevel[m_level].

- void downSweep(
  Vector<LevelData<NodeFArrayBox>*>& a_phiLevel,
  const Vector<LevelData<NodeFArrayBox>*>& a_rhsLevel)

  Sweeps down a multigrid V-cycle from this level to next coarser level. This function calculates the correction m_corr and updates the solution at this level with it. It also overwrites the residual at the next coarser level, if any, with the averaged residual at this level.
  Inputs:

  - a_phiLevel contains pointers to current guess at values of the solution. Vector index corresponds to level number.
  - a_rhsLevel contains pointers to the right-hand side at the same levels as a_phiLevel.

  Outputs:

  - a_phiLevel contains pointers to updated solution.

- void upSweep(
  Vector<LevelData<NodeFArrayBox>*>& a_phiLevel,
  const Vector<LevelData<NodeFArrayBox>*>& a_rhsLevel)

  Sweeps up a multigrid V-cycle from the next coarser level to this level. This function interpolates the correction m_corr from the next coarser level, uses it to finish the calculation of residual m_resid, and updates the solution at this level.
  Inputs:

- a_phiLevel contains pointers to current guess at values of the solution. Vector index corresponds to level number.
- a_rhsLevel contains pointers to the right-hand side at the same levels as a_phiLevel.

Outputs:

- a_phiLevel contains pointers to updated solution.

- `Real computeResidualNorm(int a_normNtype)`

  Returns the norm of internal data m_resid.
  Inputs:

  - a_normType is the type of norm: 0 for max norm, or p > 0 for L_p norm.

Principal data members

- LevelData<NodeFArrayBox> m_resid is the residual calculated in computeAMRResidual.
- LevelData<NodeFArrayBox> m_corr is the correction at this level.
- int m_level is the level for which this object manages data and operations, set to a_level argument in constructor.
- NodeMaskLevelOp* m_levelOpPtr is the pointer to the level operator, set to a_opin argument in constructor.
- NodeMGInterp m_mginterp is used for interpolating the correction from the next coarser level to this level in upSweep.
- NodeMaskAverage m_averageOp is used for averaging the residual down to the next coarser level in downSweep.

4.1.2 Internal class NodeLevelMG

The NodeLevelMG class is a multigrid solver on a level. It is used by the classes LevelNodeSolver and AMRNodeLevelMG (internal to AMRNodeSolver). This class should be considered internal to AMRNodeSolver and LevelNodeSolver, and should not be considered part of the Chombo API.

- ```
  void define(
      const DisjointBoxLayout&     a_grids,
      const DisjointBoxLayout*     a_gridsCoarsePtr,
      const ProblemDomain&         a_domain,
  ```
  ```

```
    Real                           a_dx,
    int                            a_refToCoarse,
    const NodeMaskLevelOp* const a_opin,
    int                            a_nCoarserLevels)
```

- `void define(`
```
    const DisjointBoxLayout&     a_grids,
    const DisjointBoxLayout*     a_gridsCoarsePtr,
    const Box&                   a_domain,
    Real                         a_dx,
    int                          a_refToCoarse,
    const NodeMaskLevelOp* const a_opin,
    int                          a_nCoarserLevels)
```

  Defines the internal state of the `NodeLevelMG` object and allocates space for the residual. The arguments are the same as for the `LevelNodeSolver` constructor, but instead of `a_minLength` we have `a_nCoarserLevels`. This argument represents the number of coarser-level `NodeLevelMG` objects to be defined recursively. Namely,

  - when defined from `LevelNodeSolver`,
    - if `a_minLength` is `0` then `a_nCoarserLevels` is also `0`;
    - if `a_minLength` ≥ 1 then `a_nCoarserLevels` is the exponent of the largest power of 2 that divides the lengths of every grid at this level divided by `a_minLength`. Thus after `a_nCoarserLevels` coarsenings by 2, the lengths of all grids will still be divisible by `a_minLength`.
  - when defined from `AMRNodeLevelMG`,
    - if `a_minLength` is `0` or `a_level` is `0` then `a_nCoarserLevels` is also `0`;
    - if `a_minLength` ≥ 1 and `a_level` ≥ 1 then `a_nCoarserLevels` $= \log_2(r) - 1$ where r is the refinement ratio to the next coarser level.

- `void mgRelax(`
```
    LevelData<NodeFArrayBox>&       a_phi,
    const LevelData<NodeFArrayBox>& a_rhs,
    bool                            a_bottomsolveflag)
```

  Invokes a relaxation step, updating `a_phi` with right-hand side `a_rhs`. It is assumed that the problem is in residual-correction form. In particular, only the homogeneous form of the physical and coarse-fine boundary conditions need be invoked. `mgRelax` calls itself recursively on the coarsened grids.

Principal data members

- `LevelData<NodeFArrayBox> m_resid` is the residual at this level.

- LevelData<NodeFArrayBox> m_crseResid is the residual on the grids of this level coarsened by 2.
- LevelData<NodeFArrayBox> m_crseCorr is the correction on the grids of this level coarsened by 2.
- NodeMaskLevelOp* m_levelOpPtr is the pointer to the level operator, set to a_opin argument in constructor.
- NodeMaskAverage m_averageOp is used for averaging the residual down to the residual on the coarsened grids of this level, in mgRelax.
- NodeMGInterp m_mginterp is used for interpolating the correction in mgRelax.

4.2 The class LevelNodeSolver

The LevelNodeSolver class solves elliptic equations on a level using the multigrid method. It is used by AMRNodeSolver to solve at the base level.
The API is as follows.

- ```
  void define(
      const DisjointBoxLayout&     a_grids,
      const DisjointBoxLayout*     a_gridsCoarsePtr,
      const ProblemDomain&         a_domain,
      Real                         a_dx,
      int                          a_refToCoarse,
      const NodeMaskLevelOp* const a_opin,
      int                          a_minLength = 1)
  ```
- ```
  void define(
      const DisjointBoxLayout&     a_grids,
      const DisjointBoxLayout*     a_gridsCoarsePtr,
      const Box&                   a_domain,
      Real                         a_dx,
      int                          a_refToCoarse,
      const NodeMaskLevelOp* const a_opin,
      int                          a_minLength = 1)
  ```

  Sets up the internal state of a LevelNodeSolver object.
  Inputs:
  - a_grids specifies the layout of the cell-centered grids at this level.
  - a_gridsCoarsePtr is a pointer to the layout of the cell-centered grids at the next coarser level, or NULL if there is no coarser level.

- a_domain specifies the cell-centered problem domain on this level.
- a_dx is the mesh spacing at this level.
- a_refToCoarse is the refinement ratio between this level and the next coarser level. Ignored if there is no coarser level.
- a_opin is a pointer to the NodeMaskLevelOp to use in solving.
- a_minLength is the minimum length of the maximally coarsened box, or 0 if there is to be no coarsening. If a_minLength ≥ 1 then the multigrid solver will coarsen boxes only so far as to maintain the lengths of all boxes as multiples of a_minLength.

- `void levelSolve(`
  ```
  LevelData<NodeFArrayBox>&       a_phi,
  const LevelData<NodeFArrayBox>* a_phiCoarse,
  const LevelData<NodeFArrayBox>& a_rhs,
  bool                            a_initializePhiToZero = true)
  ```
  Does a level solve on this level using multigrid and inhomogeneous boundary conditions at coarse/fine interfaces.
  Inputs:
  - a_phi is initial guess at solution at this level.
  - a_phiCoarse is a pointer to solution at next coarser level, or NULL if there is no coarser level.
  - a_rhs contains the right-hand side at this level.
  - a_initializePhiToZero tells whether to initialize the solution to zero.

  Output:
  - a_phi is updated solution at this level.

- `void levelSolveH(`
  ```
  LevelData<NodeFArrayBox>&       a_phi,
  const LevelData<NodeFArrayBox>& a_rhs,
  bool                            a_initializePhiToZero = true)
  ```
  Does a level solve on this level using multigrid and homogeneous boundary conditions at coarse/fine interfaces.
  Inputs:
  - a_phi is initial guess at solution at this level.
  - a_rhs contains the right-hand side at this level.
  - a_initializePhiToZero tells whether to initialize the solution to zero.

Output:

- a_phi is updated solution at this level.

- `void setTolerance(Real a_tolerance)`

  Sets the tolerance in the level solver: iterations end when the norm of the residual is less than a_tolerance times the initial residual. Default is `1.0e-10`.

- `void setOperatorTolerance(Real a_operatorTolerance)`

  Sets the "operator tolerance" of the solver: iterations end if the ratio of the new residual to the old residual exceeds $1 -$ a_operatorTolerance and at least m_minIter iterations have been performed. Default is `1.0e-4`.

- `void setMaxIter(int a_maxIter)`

  Sets the maximum number of relaxation iterations. Default is 33.

- `void setMinIter(int a_minIter)`

  Sets the minimum number of iterations in the solver before it will terminate due to lack of convergence. Default is 4.

- `void setnumSmoothDown(int a_numSmoothDown)`

  Sets the number of iterations of smoothing for the downward part of the V-cycle in m_levelMG.mgRelax. Default is 4.

- `void setnumSmoothUp(int a_numSmoothUp)`

  Sets the number of iterations of smoothing for the upward part of the V-cycle in m_levelMG.mgRelax. Default is 4.

- `void setnumBottomGSRB(int a_numBottomGSRB)`

  Sets the number of iterations of smoothing at the bottom of the V-cycle in m_levelMG.mgRelax. Default is 16.

- `void setBottomSmoothing(bool a_bottomSolveFlag)`

  Sets whether the smoother is applied at the bottom of the V-cycle in m_levelMG.mgRelax. Default is true.

Principal data members

- NodeLevelMG m_levelMG is a multigrid level solver object to relax on this level.

- NodeMaskLevelOp* m_levelOpPtr is a pointer to the NodeMaskLevelOp to use in solving.

- LevelData<NodeFArrayBox> m_resid is the residual in the level solve.

- LevelData<NodeFArrayBox> m_corr is the correction in the level solve.

4.3 The NodeMaskLevelOp interface

NodeMaskLevelOp is a pure base class to encapsulate level operations for node-centered elliptic solvers.

- `virtual NodeMaskLevelOp* new_levelop() = 0`

  Returns a pointer to a new instance of NodeMaskLevelOp of the same type. This gets around the "no virtual constructor" rule.

- ```
  virtual void define(
      const DisjointBoxLayout& a_grids,
      const DisjointBoxLayout* a_gridsCoarsePtr,
      Real                     a_dx,
      int                      a_refToCoarse,
      const ProblemDomain&     a_domain,
      bool                     a_homogeneousOnly = false,
      int                      a_ncomp = 1) = 0
  ```

- ```
  virtual void define(
      const DisjointBoxLayout& a_grids,
      const DisjointBoxLayout* a_gridsCoarsePtr,
      Real                     a_dx,
      int                      a_refToCoarse,
      const Box&               a_domain,
      bool                     a_homogeneousOnly = false,
      int                      a_ncomp = 1) = 0
  ```

  Full define function. Makes all coarse/fine information and sets internal variables.
  Inputs:

  - a_grids specifies the layout of the cell-centered grids at this level.
  - a_gridsCoarsePtr is a pointer to the layout of the cell-centered grids at the next coarser level, or NULL if there is no coarser level.
  - a_domain specifies the cell-centered problem domain on this level.
  - a_dx is the mesh spacing at this level.
  - a_refToCoarse is the refinement ratio between this level and the next coarser level. Ignored if there is no coarser level.
  - a_homogeneousOnly specifies whether all coarse/fine interpolation is homogeneous.
  - a_ncomp is the number of components in the operator argument.

- `virtual void define(`
  `const NodeMaskLevelOp* a_opfine,`
  `int a_refToFine) = 0`

  Full define function. Makes all coarse/fine information and sets internal variables from finer `NodeMaskLevelOp`. Any `NodeMaskLevelOp` defined with this function is not able to execute inhomogeneous boundary conditions at the coarse/fine interface.
  Inputs:

  - `a_opfine` is the finer-level operator.
  - `a_refToFine` is the refinement ratio between this level and the next finer level.

- `virtual void CFInterp(`
  `LevelData<NodeFArrayBox>& a_phi,`
  `const LevelData<NodeFArrayBox>& a_phiCoarse,`
  `bool a_inhomogeneous) = 0`

  Fills the nodes on the coarse/fine interface with interpolated data from the coarser level.
  Inputs:

  - `a_phi` is the solution on the current level.
  - `a_phiCoarse` is the solution at the next coarser level.
  - `a_inhomogeneous` specifies whether the physical boundary condition is inhomogeneous.

- `virtual void homogeneousCFInterp(`
  `LevelData<NodeFArrayBox>& a_phi) = 0`

  Zeroes out the nodes that lie on the interface with the coarser level.
  Input:

  - `a_phi` is the solution on this level.

  Output:

  - `a_phi` is the solution on this level with interface nodes set to zero.

- `virtual void smooth(`
  `LevelData<NodeFArrayBox>& a_phi,`
  `const LevelData<NodeFArrayBox>& a_rhs) = 0`

  Smoother. This smooths the solution (in the multigrid sense) on a level. It assumes that the problem has already been put into residual-correction form, so that coarse/fine boundary conditions are homogeneous.
  Inputs:

- a_phi is the solution on this level.
- a_rhs is the right-hand side on this level.

Output:

- a_phi is the smoothed solution on this level.

- `virtual void applyOpH(`
  `LevelData<NodeFArrayBox>& a_LofPhi,`
  `LevelData<NodeFArrayBox>& a_phi) = 0`

- `virtual void applyOpI(`
  `LevelData<NodeFArrayBox>& a_LofPhi,`
  `LevelData<NodeFArrayBox>& a_phi,`
  `const LevelData<NodeFArrayBox>* a_phiCoarsePtr) = 0`

- `virtual void applyOpHcfIphys(`
  `LevelData<NodeFArrayBox>& a_LofPhi,`
  `LevelData<NodeFArrayBox>& a_phi) = 0`

- `virtual void applyOpIcfHphys(`
  `LevelData<NodeFArrayBox>& a_LofPhi,`
  `LevelData<NodeFArrayBox>& a_phi,`
  `const LevelData<NodeFArrayBox>* a_phiCoarsePtr) = 0`

  Evaluate the operator. The four functions differ in whether the boundary conditions are homogeneous or inhomogeneous on the coarse/fine interface and on the physical boundary:

function	coarse/fine interface	physical boundary
applyOpH	homogeneous	homogeneous
applyOpI	inhomogeneous	inhomogeneous
applyOpHcfIphys	homogeneous	inhomogeneous
applyOpIcfHphys	inhomogeneous	homogeneous

Inputs:

- a_phi is the solution on this level.
- a_phiCoarsePtr is a pointer to the solution at the next coarser level, required for inhomogeneous coarse/fine boundary conditions.

Output:

- a_LofPhi is the result of the operator on this level.

- ```
  virtual void residualH(
      LevelData<NodeFArrayBox>&       a_resid,
      LevelData<NodeFArrayBox>&       a_phi,
      const LevelData<NodeFArrayBox>& a_rhs) = 0
  ```

- ```
  virtual void residualI(
      LevelData<NodeFArrayBox>&       a_resid,
      LevelData<NodeFArrayBox>&       a_phi,
      const LevelData<NodeFArrayBox>* a_phiCoarsePtr,
      const LevelData<NodeFArrayBox>& a_rhs) = 0
  ```

- ```
  virtual void residualHcfIphys(
      LevelData<NodeFArrayBox>&       a_resid,
      LevelData<NodeFArrayBox>&       a_phi,
      const LevelData<NodeFArrayBox>& a_rhs) = 0
  ```

- ```
  virtual void residualIcfHphys(
      LevelData<NodeFArrayBox>&       a_resid,
      LevelData<NodeFArrayBox>&       a_phi,
      const LevelData<NodeFArrayBox>* a_phiCoarsePtr,
      const LevelData<NodeFArrayBox>& a_rhs) = 0
  ```

  Calculate the residual of the operator. If L is the operator, φ is the solution to which the operator is applied, and ρ is the right-hand side, then the residual is simply $\rho - L(\varphi)$. The four functions differ in whether the boundary conditions are homogeneous or inhomogeneous on the coarse/fine interface and on the physical boundary:

function	coarse/fine interface	physical boundary
`residualH`	homogeneous	homogeneous
`residualI`	inhomogeneous	inhomogeneous
`residualHcfIphys`	homogeneous	inhomogeneous
`residualIcfHphys`	inhomogeneous	homogeneous

  Inputs:

  - `a_phi` is the solution on this level.
  - `a_phiCoarsePtr` is a pointer to the solution at the next coarser level, required for inhomogeneous coarse/fine boundary conditions.
  - `a_rhs` is the right-hand side on this level.

  Output:

  - `a_resid` is the residual on this level.

- `virtual void bottomSmoother(`
  `LevelData<NodeFArrayBox>& a_phi,`
  `const LevelData<NodeFArrayBox>& a_rhs) = 0`

  Performs smoothing at the bottom level. This is used when it is not possible to coarsen the grid. Typically, the function is either a point relaxation or a conjugate-gradient type of method.
  Inputs:

  - `a_phi` is the solution on this level.
  - `a_rhs` is the right-hand side on this level.

  Output:

  - `a_phi` is the smoothed solution on this level.

- `virtual void levelPreconditioner(`
  `LevelData<NodeFArrayBox>& a_phihat,`
  `const LevelData<NodeFArrayBox>& a_rhshat) = 0`

  Applies preconditioner. In the notation of [BBC+94], if the preconditioner is M, which is an approximation to the operator, then this solves the related equation $M\hat{\phi} = \widehat{\rho}$. In the AMR multigrid implementation, this function is generally most often used by the bottom solver.
  Input:

  - `a_rhshat` is the right-hand side of the preconditioning.

  Output:

  - `a_phihat` is the result of the preconditioning.

4.3.1 The `NodeMaskBaseBottomSmoother` interface

`NodeMaskBaseBottomSmoother` is a pure base class to encapsulate operations for smoothers used by node-centered elliptic solvers.

- `virtual NodeMaskBaseBottomSmoother* new_bottomSmoother() const = 0`

  Returns a pointer to a new instance of `NodeMaskBaseBottomSmoother` of the same type.

- `virtual void doBottomSmooth(`
  `LevelData<NodeFArrayBox>& a_phi,`
  `const LevelData<NodeFArrayBox>& a_rhs,`
  `const BoxLayoutData< BaseFab<int> >& a_mask,`
  `NodeMaskLevelOp* a_levelop_ptr) = 0`

Performs smoothing of $L(\varphi) = \rho$.
Inputs:

- a_phi is the solution on a level.
- a_rhs is the right-hand side on a level.
- a_mask is the mask array from REACHABLENODES on this level.
- a_levelop_ptr is a pointer to an instance of the operator.

Output:

- a_phi is the smoothed solution on the level.

Examples of smoothers are conjugate gradient and BiCGStab, described in [BBC+94]. These are implemented as the NodeMaskCGSmoother and NodeMaskBiCGStabSmoother classes, respectively, described in detail in [McC02].

4.3.2 The class NodeMaskPoissonOp

NodeMaskPoissonOp is a subclass of NodeMaskLevelOp that includes operations for the discrete Laplacian operator. Here we describe only the functions and members of NodeMaskPoissonOp that are not described in the section on NodeMaskLevelOp.

- `void setBottomSmoother(const NodeMaskBaseBottomSmoother& a_bottomSmoother)`

  Sets the bottom smoother to be used in the smooth function. The default bottom smoother for NodeMaskPoissonOp is an object of class NodeMaskCGSmoother.

- `void setInterpolationDegree(int a_interpolationDegree)`

  Sets the degree of interpolation to be used in NodeQuadCFInterp: 1 for linear in two dimensions or bilinear in three dimensions; 2 for quadratic in two dimensions or biquadratic in three dimensions. Default is 2. This function must be called before define. The interpolation degree cannot be changed later without calling define again.

- `void setDomainNodeBC(const DomainNodeBC& a_dombcIn)`

  Sets the boundary conditions of the physical domain.

4.4 The class AMRNodeSolverAlt

The AMRNodeSolverAlt class solves an elliptic equation on an adaptive mesh refinement hierarchy using an alternative algorithm to the AMR multigrid solver implemented in AMRNodeSolver. AMRNodeSolverAlt solves one level at a time, from coarsest to finest, interpolating from coarser to finer solutions as it proceeds. See Figure 4.2 for a chart of major classes used in AMRNodeSolverAlt.

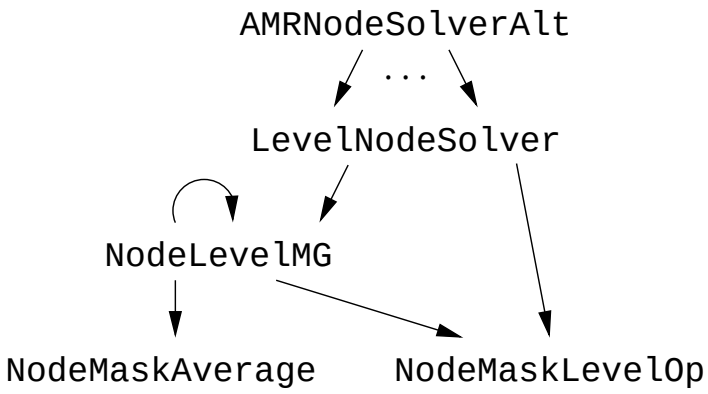

Figure 4.2: This chart shows the major classes of `AMRNodeElliptic` used by `AMRNodeSolverAlt`. An arrow from one class to another indicates that an instance of the first class defines an instance of the second class. In each case, only one instance of the second class is defined, except that the `AMRNodeSolverAlt` defines an instance of `LevelNodeSolver` for each level in the AMR hierarchy.

- ```
  void define(
      const Vector<DisjointBoxLayout>& a_gridsLevel,
      const Vector<ProblemDomain>&     a_domainLevel,
      const Vector<Real>&              a_dxLevel,
      const Vector<int>&               a_refRatio,
      int                              a_numLevels,
      int                              a_lBase,
      const NodeMaskLevelOp* const     a_opin,
      int                              a_minLength = 1)
  ```

- ```
  void define(
      const Vector<DisjointBoxLayout>& a_gridsLevel,
      const Vector<Box>&               a_domainLevel,
      const Vector<Real>&              a_dxLevel,
      const Vector<int>&               a_refRatio,
      int                              a_numLevels,
      int                              a_lBase,
      const NodeMaskLevelOp* const     a_opin,
      int                              a_minLength = 1)
  ```

  Sets up the internal state of an `AMRNodeSolverAlt` object with information about the grid hierarchy and the operator. The arguments are the same as for the `define` function of `AMRNodeSolver`.

- ```
  void solveAMR(
      Vector<LevelData<NodeFArrayBox>*>&       a_phiLevel,
  ```
  ```

const Vector<LevelData<NodeFArrayBox>*>& a_rhsLevel)

Solves the elliptic equation over a hierarchy of levels. The arguments are the same as for the solveAMR function of AMRNodeSolver.

- void setTolerance(Real a_tolerance)

  Sets the tolerance in the one-level solvers: iterations at a level end when the norm of the residual is less than a_tolerance times the initial residual. Default is 1.0e-10.

- void setMaxIter(int a_maxIter)

  sets the maximum number of iterations in the solvers at all levels. Default is set in LevelNodeSolver.

- void setBottomSmoothing(bool a_doBottomSmooth)

  sets whether the smoother is applied at the bottom of the V-cycle in the function NodeLevelMG::mgRelax. Default is true.

Principal data members.

- Vector<LevelNodeSolver*> m_levelSolver is a vector of length m_numLevels containing pointers to LevelNodeSolver objects, each of which manages the data and operations on one level.

- Vector<DisjointBoxLayout> m_gridsLevel, Vector<Box> m_domainLevel, Vector<Real> m_dxLevel, Vector<int> m_refRatio, int m_numLevels, int m_lBase are set to the arguments a_gridsLevel, a_domainLevel, a_dxLevel, a_refRatio, a_numLevels, a_lBase in the constructor.

Chapter 5

C++ Classes for Two-Level Operators

5.1 The class NodeMaskAverage

The NodeMaskAverage class is used for replacing coarse-level data with an average of fine-level data in the classes NodeLevelMG and AMRNodeLevelMG.

- ```
  void define(
      const DisjointBoxLayout& a_gridsFine,
      const DisjointBoxLayout& a_gridsCoarse,
      int                      a_numcomps,
      int                      a_refRatio,
      const ProblemDomain&     a_domainFine,
      Real                     a_dx)
  ```

- ```
  void define(
      const DisjointBoxLayout& a_gridsFine,
      const DisjointBoxLayout& a_gridsCoarse,
      int                      a_numcomps,
      int                      a_refRatio,
      const Box&               a_domainFine,
      Real                     a_dx)
  ```

  Defines the internal state of the NodeMaskAverage object.
  Inputs:

  - a_gridsFine specifies the layout of the cell-centered grids at the finer level.
  - a_gridsCoarse specifies the layout of the cell-centered grids at the coarser level.
  - a_numcomps is the number of components in the data.
  - a_refRatio is the refinement ratio between the two levels.
  - a_domainFine specifies the cell-centered problem domain

- a_dx is the mesh spacing at the finer level.

- ```
  void define(
      const DisjointBoxLayout& a_gridsCoarse,
      int                      a_numcomps,
      int                      a_refRatio,
      const ProblemDomain&     a_domainFine,
      Real                     a_dx)
  ```

- ```
  void define(
      const DisjointBoxLayout& a_gridsCoarse,
      int                      a_numcomps,
      int                      a_refRatio,
      const Box&               a_domainFine,
      Real                     a_dx)
  ```

  Defines the internal state of the NodeMaskAverage object. The arguments are the same as that of the longer define function above, except that a_gridsFine is not present; the grids at the finer level are taken to be refinements of the grids at the coarser level.

- ```
  void averageToCoarse(
      LevelData<NodeFArrayBox>& a_coarse,
      LevelData<NodeFArrayBox>& a_fine)
  ```

  Replaces coarse-level data with an average of nearby finer-level cells' data, on the interior of the coarsened finer-level domain. Elsewhere, the coarse-level data is unchanged.
  Input:

  - a_fine is finer-level data.

  Output:

  - a_coarse is coarser-level data with interior nodes replaced by average of finer-level data.

5.2 The class NodeMGInterp

The NodeMGInterp class is used for interpolating data from a coarser to a finer AMR level in the class AMRNodeLevelMG.

- ```
  void define(
      const DisjointBoxLayout& a_grids,
      int                      a_numcomps,
  ```

```
    int                         a_refRatio,
    const ProblemDomain&        a_domain,
    Real                        a_dx)
```

- ```
  void define(
      const DisjointBoxLayout& a_grids,
      int                      a_numcomps,
      int                      a_refRatio,
      const Box&               a_domain,
      Real                     a_dx)
  ```

  Defines the internal state of the `NodeMGInterp` object.
  Inputs:

  - `a_grids` specifies the layout of the cell-centered grids at the finer level.
  - `a_numcomps` is the number of components in the data.
  - `a_refRatio` is the refinement ratio between the two levels.
  - `a_domain` specifies the cell-centered problem domain on the finer level.
  - `a_dx` is the mesh spacing at the finer level.

- ```
  void interpToFine(
      LevelData<NodeFArrayBox>&       a_fine,
      const LevelData<NodeFArrayBox>& a_coarse)
      bool                            a_sameGrids = false)
  ```

  Modifies the finer-level data by adding interpolated values of the coarser-level data.
  Input:

  - `a_fine` is original finer-level data.
  - `a_coarse` is coarser-level data.
  - `a_sameGrids` specifies whether the finer-level grids are refinements of the coarser-level grids.

  Output:

  - `a_fine` is updated finer-level data.

5.3 The class `NodeQuadCFInterp`

The `NodeQuadCFInterp` class interpolates data from a coarse level to a fine level on coarse/fine interfaces.

- ```
  void define(
      const DisjointBoxLayout&           a_grids,
      Real                               a_dx,
      const ProblemDomain&               a_domain,
      const LayoutData<NodeCFIVS>* const a_loCFIVS,
      const LayoutData<NodeCFIVS>* const a_hiCFIVS,
      int                                a_refToCoarse,
      int                                a_interpolationDegree = 2,
      int                                a_ncomp = 1)
  ```

- ```
  void define(
      const DisjointBoxLayout&           a_grids,
      Real                               a_dx,
      const Box&                         a_domain,
      const LayoutData<NodeCFIVS>* const a_loCFIVS,
      const LayoutData<NodeCFIVS>* const a_hiCFIVS,
      int                                a_refToCoarse,
      int                                a_interpolationDegree = 2,
      int                                a_ncomp = 1)
  ```

  Defines the internal state of the `NodeQuadCFInterp` object.
  Inputs:

  - `a_grids` specifies the layout of the cell-centered grids at the finer level.
  - `a_dx` is the mesh spacing at the finer level.
  - `a_domain` specifies the cell-centered problem domain on the finer level.
  - `a_loCFIVS` and `a_hiCFIVS` are pointers to arrays of length `SpaceDim` of `LayoutData<NodeCFIVS>`, each of which holds information about nodes on the coarse/fine interface for each finer-level box. The underlying layout in every case is `a_grids`. The arrays of `LayoutData<NodeCFIVS>` are indexed by dimension, with `a_loCFIVS` for faces on the low sides and `a_hiCFIVS` for faces on the high sides.
  - `a_refToCoarse` is the refinement ratio between the two levels. It must be a power of 2.
  - `a_interpolationDegree` is the degree of interpolation. It is 1 if interpolation is linear in two dimensions or bilinear in three dimensions. It is 2 if interpolation is quadratic in two dimensions or biquadratic in three dimensions.
  - `a_ncomp` is the number of components in the data.

- ```
  void coarseFineInterp(
      LevelData<NodeFArrayBox>&       a_phiFine,
      const LevelData<NodeFArrayBox>& a_phiCoarse,
  ```
  ```

```
    bool                                a_inhomogeneous)
```

Fills the nodes of the fine level of the coarse/fine interfaces with interpolated data from the coarser level.
Inputs:

- a_phiCoarse is coarser-level data.
- a_inhomogeneous specifies whether the physical boundary condition is inhomogeneous.

Output:

- a_phiFine is the finer-level data, modified at nodes on interfaces with the coarser level.

- `void setDomainNodeBC(const DomainNodeBC& a_dombcIn)`

  Sets the boundary conditions of the physical domain. This is required if the refinement ratio is more than 2, because then the boundary conditions need to be set on the intermediate levels between successive interpolations by refinement ratio of 2.

5.4 The class NodeCFIVS

The NodeCFIVS class determines the set of nodes that lie on the coarse/fine interface, for a particular face of a particular box at the finer level.

- ```
  void define(
      const ProblemDomain&     a_domain,
      const Box&               a_box,
      const DisjointBoxLayout& a_levelBoxes,
      int                      a_idir,
      Side::LoHiSide           a_hiorlo)
  ```

- ```
  void define(
      const Box&               a_domain,
      const Box&               a_box,
      const DisjointBoxLayout& a_levelBoxes,
      int                      a_idir,
      Side::LoHiSide           a_hiorlo)
  ```

  Defines the internal state of the NodeCFIVS object.
  Inputs:

  - a_domain specifies the cell-centered problem domain on the finer level.
  - a_box specifies the finer-level box.

- a_levelBoxes specifies the layout of all cell-centered boxes at the finer level.
- a_idir and a_hilorlo specify the particular face of a_box.

- `const IntVectSet& getFineIVS() const`

  Returns the indices of the finer-level nodes on this face that also lie on the interface with the coarser level. The `IntVectSet` excludes indices of nodes that lie on the boundary of the problem domain, and nodes that are on faces of any other boxes in a_levelBoxes on the side of a_box specified by a_idir and a_hilorlo. See Figure 5.1.

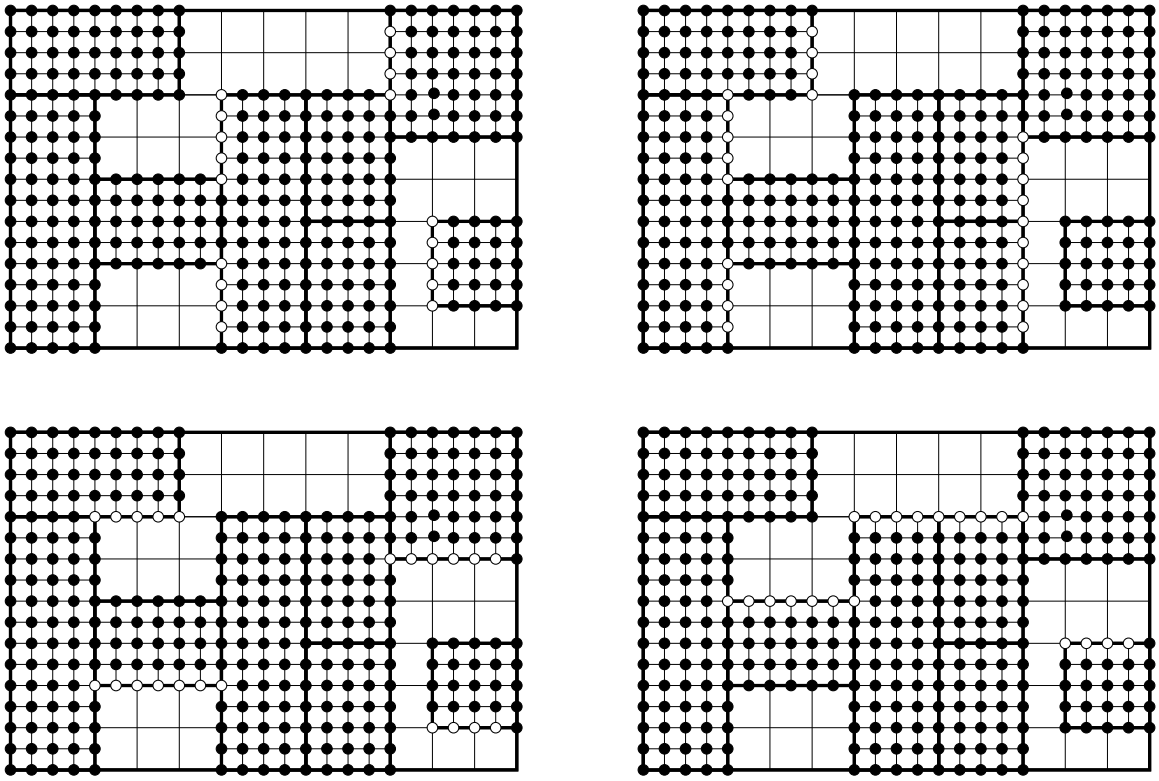

Figure 5.1: This figure indicates, with hollow circles, all nodes with indices returned by getFineIVS over all settings of a_box in a two-dimensional layout over a non-periodic domain.
Top: a_idir = 0, and a_hiorlo = (left) Side::Lo or (right) Side::Hi.
Bottom: a_idir = 1, and a_hiorlo = (left) Side::Lo or (right) Side::Hi.

Chapter 6

External Boundary Conditions

Boundary conditions on the faces of the rectangular problem domain are enforced with the class `DomainNodeBC`, which contains a pointer to an instance of the class `FaceNodeBC` for each boundary face. The user adds a `FaceNodeBC`-derived class for each face to enforce particular boundary conditions.

The `AMRNodeElliptic` package allows boundary conditions that are expressed as

$$A\frac{\partial \varphi}{\partial n} + B\varphi = C \tag{6.1}$$

for each node lying on the boundary, where A, B, and C may vary over the boundary. The class derived from `FaceNodeBC` provides a function to fill A, B, and C on a boundary face. `DomainNodeBC` calls this function and fills the boundary nodes appropriately. Examples:

- Dirichlet: $A = 0$, $B = 1$. For homogeneous, $C = 0$.
- Neumann: $A = 1$, $B = 0$. For homogeneous, $C = 0$.

The user needs to define an instance of `DomainNodeBC` only once, regardless of the number of levels of refinement. The problem domain and the mesh spacing for the level are inputs to the functions that set and apply the boundary conditions.

6.1 The `DomainNodeBC` class

There is one instance of `DomainNodeBC` for the whole problem domain.

- `DomainNodeBC()`

  This constructor defines the object in an unusable state until the user calls `setFaceNodeBC` for each face.

- ```
  void setFaceNodeBC(
       const FaceNodeBC& a_bc)
  ```
  ```

Sets boundary conditions at a face.
Input:

- a_bc is boundary condition at a face.

- ```
  void applyHomogeneousBCs(
      NodeFArrayBox&       a_state,
      const ProblemDomain& a_domain,
      Real                 a_dx) const
  ```

- ```
  void applyHomogeneousBCs(
      NodeFArrayBox& a_state,
      const Box&     a_domain,
      Real           a_dx) const
  ```

- ```
  void applyInhomogeneousBCs(
      NodeFArrayBox&       a_state,
      const ProblemDomain& a_domain,
      Real                 a_dx) const
  ```

- ```
  void applyInhomogeneousBCs(
      NodeFArrayBox& a_state,
      const Box&     a_domain,
      Real           a_dx) const
  ```

  Apply boundary conditions. The difference between applyHomogeneousBCs and applyInhomogeneousBCs is that applyHomogeneousBCs sets $C = 0$ in (6.1).
  Inputs:

  - a_domain is the cell-centered problem domain on the level.
  - a_dx is the mesh spacing on the level.

  Output:

  - a_state contains the data. Data at nodes on the boundary of the problem domain are modified according to the boundary conditions. Data at other nodes are unchanged.

6.2 The FaceNodeBC interface

There is a FaceNodeBC object for each of the 2*SpaceDim faces of the problem domain.

- ```
  FaceNodeBC(
      int            a_dir,
      Side::LoHiSide a_sd)
  ```
  ```

- FaceNodeBC(
  ```
  int              a_dir,
  Side::LoHiSide   a_sd,
  const Interval&  a_comps)
  ```
  Constructor that defines boundary conditions on a face.
  Inputs:
  - a_dir is dimension of face, 0 up to SpaceDim-1.
  - a_sd is either Side::Lo or Side::Hi in specifying the face.
  - a_comps, if present, specifies the interval over which components of the data will have boundary values set. If absent, then the interval is (0:0).

- virtual FaceNodeBC* new_boxBC() const = 0

  Returns a pointer to a new instance of FaceNodeBC of the same type.

- void define(
  ```
  int             a_dir,
  Side::LoHiSide  a_sd)
  ```

- void define(
  ```
  int              a_dir,
  Side::LoHiSide   a_sd,
  const Interval&  a_comps)
  ```
  Define the FaceNodeBC object with the same arguments as the constructors.

- virtual void applyHomogeneousBCs(
  ```
  FArrayBox&           a_state,
  const ProblemDomain& a_domain,
  Real                 a_dx) const
  ```

- virtual void applyHomogeneousBCs(
  ```
  FArrayBox& a_state,
  const Box& a_domain,
  Real       a_dx) const
  ```

- virtual void applyInhomogeneousBCs(
  ```
  FArrayBox&           a_state,
  const ProblemDomain& a_domain,
  Real                 a_dx) const
  ```

- virtual void applyInhomogeneousBCs(
  ```
  FArrayBox& a_state,
  const Box& a_domain,
  ```

```
    Real       a_dx) const
```

The difference between applyHomogeneousBCs and applyInhomogeneousBCs is that applyHomogeneousBCs sets $C = 0$ in (6.1).
Inputs:

- a_domain is the cell-centered problem domain on the level.
- a_dx is the mesh spacing on the level.

Output:

- a_state contains the data. Data on nodes of a_state that lie on the boundary of the cell-centered problem domain a_domain are modified according to the boundary conditions. Data on other nodes of a_state are unchanged.

- ```
  virtual void fillBCValues(
      FArrayBox&           a_neumfac,
      FArrayBox&           a_dircfac,
      FArrayBox&           a_inhmval,
      Real                 a_dx,
      const ProblemDomain& a_domain) const = 0
  ```

- ```
  virtual void fillBCValues(
      FArrayBox& a_neumfac,
      FArrayBox& a_dircfac,
      FArrayBox& a_inhmval,
      Real       a_dx,
      const Box& a_domain) const = 0
  ```

  Sets the coefficients in (6.1). This function must be provided in the derived class.
  Inputs:

  - a_dx is the mesh spacing on the level.
  - a_domain is the cell-centered problem domain on the level.

  Outputs: Data in these node-centered FArrayBoxes are set on the nodes that lie on this face of the boundary of a_domain. When called by applyEitherBCs, all three of them have the same underlying node-centered box, which will be a subbox of this boundary face. In the boundary condition equation $A\frac{\partial \varphi}{\partial n} + B\varphi = C$ (6.1):

  - a_neumfac contains A.
  - a_dircfac contains B.
  - a_inhmval contains C.

Bibliography

[BBC+94] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1994.

[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.

[McC02] P. McCorquodale. AMRNodeElliptic Software Package: Node-Centered AMR for Elliptic Problems. unpublished, 2002.