Chombo Software Package for AMR Applications
Design Document

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

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

September 12, 2003

Disclaimer

Contents

Chapter 1

Introduction

In many problems in partial differential equations, one is confronted with problems having multiple length scales and strong spatial localizations. Examples include nonlinear systems of hyperbolic partial differential equations containing complex combinations of discontinuities and smooth flow. Also included are combustion problems in which, at any given instant, burning is taking place in a small subset of the problem domain and problems with complex geometries in which localized geometric features can generate strong, localized solution gradients. Finite difference calculation using block-structured adaptive mesh refinement (AMR) is a powerful tool for computing solutions to partial differential equations involving such multiple scales. In this approach, the underlying problem domain is discretized using a rectangular grid and a solution is computed on that grid. Regions requiring additional resolution are identified by computing some local measure of the original error and covered by a disjoint union of rectangles in the domain, which are then refined by some integer factor. The solution is then computed on the composite grid. This process may be applied recursively, and for time-dependent problems, the error estimation and regridding can be integrated with the time evolution and refinement applied in time as well as in space. Such an approach was first introduced by Berger and Oliger [BO84] for computing time-dependent solutions to hyperbolic partial differential equations in multiple space dimensions. Since that time, the approach has been extended to a variety of problems in applied partial differential equations [BC89] [TF89] [BBSW94] [ABC94] [PBC+95] [HT97] [ABC+98] [JFH+98] [PHB+98] [HPC+99] [CDW99] .

One of the principal disadvantages of block-structured AMR is its relative difficulty to implement compared to single-grid algorithms. The algorithms are more complex and the data structures are unfamiliar to traditional FORTRAN programmers.

To ameliorate these difficulties, we have developed Chombo, a set of C++ classes designed to support block-structured AMR applications. Chombo is based in part on the BoxLib toolkit and related work done by our colleagues at the Center for Computational Sciences and Engineering (CCSE) at LBNL [CW93] [RBL+99]. The Chombo package at the present time consists of the following components:

- The BoxTools Library includes the BoxLib rectangular array library. BoxTools also contains a full set calculus on Z^n, and classes for defining data on unions of rectangles as well as mapping such data onto distributed memory systems.

- The AMRTools Library consists of classes which implement a number of operations that often appear in AMR algorithms: conservative interpolation, averaging between AMR levels, interpolation of boundary conditions at coarse-fine interfaces, and refluxing operations to maintain conservation at coarse-fine interfaces.
- The AMRTimeDependent library consists of classes which support the Berger-Oliger time stepping algorithm and examples of its use in solving systems of hyperbolic conservation laws..
- The AMRElliptic library consists of classes which support an AMR-multigrid algorithm for elliptic partial differential equations, and examples of its use in solving Poisson and Helmholtz equations.
- The ParticleTools library consists of classes which provide basic support for particle-in-cell methods.

In addition to these basic tools we have provided extensive documentation. There are some general comments regarding the use of the package, however, that are worth emphasizing here.

- As is the case with BoxLib, C++ rectangular array operations applied one point at a time in a for loop will not produce high performance on bulk rectangular operations. For this reason, BoxLib provides an interface between the array classes that allows them to be passed to FORTRAN routines. We have augmented this interface with a macro package, described in appendix A. The Chombo FORTRAN package additionally allows one to write dimension-independent FORTRAN. On the other hand, we have been willing to implement sparse irregular calculations in C++ directly using pointwise operations.
- We have attempted to leverage other related research activities. For example, HDF5 is an emerging standard for portable, self-describing, binary I/O. For this reason, we have based our I/O on HDF5. Similarly, we are working with the KeLP effort [FBK96] at UCSD/SDSC, and we expect ultimately that our parallel support will be built on top of KeLP.
- We are trying to facilitate others using parts of Chombo which they might find useful by pursuing a component-based design approach. For example, it is possible to use our implementation of the Berger-Rigoutsos [BR91] grid generation algorithm or the parallel data distribution support without using the rectangular array library.

Finally we want to emphasize that the developers of this package are themselves using Chombo to develop new algorithms and packages. This means that we will be actively adding capability that we expect to make available to other users of this package.

1.1 Requirements

Before we discuss the installation procedure, we must discuss what other software needs to be installed on a system in order to build Chombo.

- To build Chombo, the GNU version of make (GNUmake) must be installed. The Chombo makefile system requires GNU make version 3.77 or later. GNU software can be downloaded from many places, including:

  `ftp://ftp.gnu.org/gnu/make`

- HDF5 must be installed. This provides Chombo a mechanism for portable and parallel self-describing binary output. HDF5 can be downloaded from:

  `http://hdf.ncsa.uiuc.edu/HDF5`

  We recommend using HDF5 version 1.4.x. We suggest that it be configured with the option `"--enable-production"`. Chombo has not been tested with the latest version 1.6. If you try to use this version, please let us know how it goes.

- A functioning MPI-1.2 compliant C-binding is needed to build Chombo for parallel processing. This is only necessary if Chombo is compiled with the `MPI=TRUE` option. See section 1.2 for the various compilation options for Chombo. A parallel version of the HDF5 libraries must also be built. Configure HDF5 with `"--enable-parallel"` for a parallel version. Make sure to install it into a different directory from the serial version, since the libraries have the same names in both cases.

- A C++ compiler is required. Chombo makes heavy use of the ISO/IEC 14882 C++ Standard. Some compilers are not fully compliant with this specification, although most are. Chombo has been compiled and tested with[1]:

  - GNU g++ 2.95 or higher – stable releases only. Stay away from special releases, particularly version 2.96, which was released with some RedHat Linux distributions. Stable releases we have recently tested include 2.95.3, 3.0.x, 3.2.x.
  - Intel icc v6 or higher on Linux. Recently tested v6 and v7 on RedHat Linux v7.
  - IBM xlC version 5 or later. Recently tested on AIX v5.1.
  - MIPSpro Compilers, Version 7.3.1.2m. Recently tested in serial on a IRIX64 platform.
  - Sun Workshop 5.0.

[1]Support for the KCC compiler has been dropped. If this matters to you, contact us.

- A Fortran 77 compiler is required. Chombo has been compiled and tested with:
  - GNU g77
  - Intel ifc on Linux
  - IBM xlf, xlf90 on AIX
  - Portland Group pgf77, pgf90
  - HP/Compaq/DEC f77,f90,f95
  - Sun f77,f90
  - f90 on Cray T3E
- Perl version 5.0 or higher is required. The Chombo Fortran system uses perl to produce dimension-independent Fortran code. Perl 5 can be downloaded from:

  `http://www.perl.org/get.html`

1.2 Installation

The Chombo software package is distributed in a compressed tar file (file name ends in ".tar.gz"). You will need the GNU "gzip" program and a version of "tar" to unpack this file. First, run "gunzip" (or "gzip -d") on the "*.gz" file, then run "tar xf" on the resulting "*.tar" file. This will leave you with a subdirectory named "Chombo".

1.2.1 Configuring Chombo for a Particular System

Before Chombo can be built, the makefile system must be configured for the computer it is to be built on. The major configuration parameters relate to the locations of the other software on which Chombo depends and the compilers.

All Chombo configuration is done by setting variables that the makefiles use. The makefile system sets as many of these variables as possible, but some are hard to determine and others are purely a matter of user preference. All customizations of the makefile variables are done in a single file:

`Chombo/lib/mk/Make.defs.local`

This file does not exist in the Chombo distribution tar file. There are several ways to create it:

- copy the file `Chombo/lib/mk/Make.defs.local.template`
- run the commands: `cd Chombo/lib ; make setup`
- copy the system-specific customization file from `Chombo/lib/mk/local` if you are using a computer we already use (seaborg, halem, alvarez, cheetah, lomax)

The first two options produce a customization file with no variables defined, but with the important variables documented in comments. **The resulting** `Make.defs.local` **file must be editted to set the HDFINCFLAGS and HDFLIBFLAGS variables (see below) if either of these options is used.** The system-specific files in the third option set the variables for particular supercomputers that many people use and should work on those systems without modification.

The makefile variables that are most commonly customized include:

`DIM` the number of spatial dimensions in the calculations (=2 or 3). The default is 2.

`PRECISION` determines the size of floating point variables; the acceptable values are FLOAT and DOUBLE. The default is DOUBLE.

`DEBUG` determines whether to compile for debugging (=TRUE) or optimization (=FALSE). The default is TRUE.

`PROFILE` determines whether to compiler for performance profiling (=TRUE) or not (=FALSE). The default is FALSE.

`CXX` the command to run the C++ compiler (include path and options, if necessary). The default is g++, except on systems with a usable vendor-provided compiler.

`FC` the command to run the Fortran compiler (ditto). The default is g77, except on systems with a usable vendor-provided compiler.

`MPI` determines whether to compile for parallel (=TRUE) or serial (=FALSE) execution. The default is FALSE.

`MPICXX` when `$MPI` is TRUE, this specifies the command name of the parallel C++ compiler. The default is mpiCC, except on systems with a usable vendor-provided compiler.

`OBJMODEL` an optional flag value that specifies a special way to compile the Chombo code (e.g. for 64bits or dynamic libraries). The actual values are defined in the makefiles for the individual compilers. Most users will not need to set this. The default is blank.

`XTRACONFIG` an additional identification string to be added to filenames generated by the makefiles. This allows the user to build separate libraries based on parameters other than those specified by the makefile system. This string is empty by default.

`LD` the command to run the linker, if different from CXX

`HDFINCFLAGS` the C++ compiler options to compile with HDF (usually -I<hdf_dir>/include, where <hdf_dir> is the root directory of the HDF installation). The default is blank, but that usually will not work, so this variable must be set.

`HDFLIBFLAGS` the linker options to access the HDF libraries (usually -L<hdf_dir>/lib -lhdf5 -lz)

`HDFMPIINCFLAGS` same as HDFINCFLAGS, except for the parallel version of HDF. The default is blank. (this should be blank if parallel HDF is not installed)

`HDFMPILIBFLAGS` same as HDFLIBFLAGS, except for the parallel version of HDF (this should also be blank if parallel HDF is not installed)

The first 10 variables in this list (from DIM to XTRACONFIG) are called the "configuration" variables. The Chombo makefiles allow for different configurations to exist simultaneously by using the configuration in the names of the library and executable files. The normal procedure is to define a default configuration in the `Make.defs.local` file (or use the standard configuration defined in the `Chombo/lib/mk/Make.defs.defaults` file) and build alternate configurations by specifying the configuration variables explicitly on the make command line. For example:

make DIM=3 DEBUG=FALSE all

will build Chombo in 3 dimensions with compiler optimization enabled.

If the compilers on the system are not already known to the makefiles, it also may be necessary to set variables that determine what options to use in the compile commands. Variables for known compilers are set in files in the directory `Chombo/lib/mk/compiler`. The files in this directory have names of the form `Make.defs.`*compiler_name*. The *compiler_name* is taken from the CXX and FC configuration variables.

The recommended approach to setting compiler variables for unknown compilers is to first try to build the Chombo libraries and programs with the default compiler options variables, and if that doesn't work, to customize the variables in the `Make.defs.local` file.

The compiler variables are:

`cppdbgflags` options for the preprocessor step of C++ and Fortran compiles when DEBUG=TRUE (default is blank)

`cppoptflags` options for the preprocessor step of C++ and Fortran compiles when DEBUG=FALSE (default is blank)

`cxxdbgflags` options for C++ compiler and linker when DEBUG=TRUE (default is -g)

`cxxoptflags` options for C++ compiler and linker when DEBUG=FALSE (default is -O)

`fdbgflags` options for Fortran compiler when DEBUG=TRUE (default is -g)

`foptflags` options for Fortran compiler when DEBUG=FALSE (default is -O)

`lddbgflags` options for linker only when DEBUG=TRUE (default is blank)

`ldoptflags` options for linker only when DEBUG=FALSE (default is blank)

`cxxprofflags` options for C++ compiler and linker when PROFILE=TRUE (default is -pg) `fprofflags` options for Fortran compiler when PROFILE=TRUE (default is -pg)

These variables can be overridden on the make command line by setting the variables:

CPPFLAGS CXXFLAGS FFLAGS LDFLAGS

1.2.2 Compiling Chombo's Libraries

Once the makefile variables are properly customized in the `Make.defs.local` file, the libraries can be built. The commands to do this are:

```
cd Chombo/lib
make lib
```

Add to the "make" command any non-default definitions of configuration variables you wish to use.

This will produce a lot of output, most of which is compile commands and messages from make. Depending on which compilers are used, there may be some compiler warnings about unused variables and invalid offsets, but these can be safely ignored. None of the compiles should produce error messages. If this occurs, you have a problem. Usually the solution is to fix the compiler options. Problems with the files in `Chombo/lib/mk/compiler` should be reported to <chombo@anag.lbl.gov>.

1.2.3 Compiling and running Chombo's test programs

Once the libraries are successfully compiled, the test programs should be built and run. The commands to do this are:

```
cd Chombo/lib
make test
make run
```

Of course, any variables you defined on the command line when build the libraries also should be defined for these "make" commands.

The first "make" command compiles and links the test programs. Errors are usually in the link step, since the test code will usually compile if the library code compiles. Link errors are commonly due to bad or missing libraries, bad template instantiation by the compiler or problems with Fortran libraries. Make sure that the HDFLIBFLAGS variable has the correct value and that the HDF libraries were compiled with the same compiler that the Chombo build is using. Undefined references to routines named H5* is a symptom of problems with the HDF library. Other problems should be referred a local guru or, failing that, to <chombo@anag.lbl.gov>.

The "make run" command executes all the Chombo test programs, in a mode that produces minimal output. Successful execution of a test program is indicated by the message "... testFoo finished with status 0". If all you care about is whether the tests succeed or not, the command to use is:

make run | grep 'finished with'

If the status is anything other than 0, the test failed and you should rerun it by hand in verbose mode.

To do this, find the message that starts "make –no-print-directory –directory"' and occurs before the output of the test that failed. The word after "–directory" is the directory containing the failed test. Change directory into "test" then change into that directory. Run the command "make run VERBOSE=-v" and save the output. Then do "cd ../.." and run the command "make vars". Email the output from both commands to <chombo@anag.lbl.gov> and we'll try to suggest a solution.

1.2.4 Compiling and running Chombo's example applications

The test programs are all simple codes that exercise small pieces of the Chombo libraries, usually just single classes. They are not intended to show how the Chombo software should be used in a real application. For that, there are example applications.

Building and running the examples is the same as the test programs. The commands are:

```
cd Chombo/example
make all
make run
```

As before, any variables defined on the command line when you built the libraries should be added to this "make" command too.

This "make all" step usually succeeds if the libraries and tests built successfully. Errors should be reported.[2]

The "make run" step produces a lot of output, and can take a long time to run, depending on the computer and configuration. As with the tests, the command:

```
make run | grep 'finished with'
```

will print out a minimum of output and still indicate whether all the example programs ran successfully. It is recommended to run the examples once and look at the output to ensure that the programs actually ran correctly.

Some of the example programs use a lot of memory and take a long time to run in 3d: (e.g. AMRNodeElliptic/execPolytropic, AMRWaveEqn/exec). The command:

```
cd Chombo/example
make usage
```

will list all the example targets. An individual example can be run by running "make" with that target.

[2]Exception: the IBM xlC compiler complains about multiple definitions. It seems that it is safe to ignore these complaints as long as the executable files (*.ex) are created and run successfully.

1.2.5 Building an application using Chombo

There are two ways to build an application with the Chombo library: by using the Chombo makefiles and building the application as if it was a Chombo application, or by treating Chombo as just another library in an existing makefile.

1.2.5.1 Using the Chombo application makefiles

The Chombo makefiles support two different ways of building an application. One assumes all the source code for the application (aside from whatever libraries it uses) are stored in a single subdirectory. The other allows for source code in multiple directories. The former is simpler.

Here is an example of a "GNUmakefile" that builds an application that has all its own code in a single directory. It is based on the file "Chombo/example/AMRPoisson/exec/GNUmakefile", with unnecessary lines removed for simplicity.

```
## path to the 'Chombo/lib' directory
CHOMBO_HOME := ../../../Chombo/lib

## defines the Chombo variables
include $(CHOMBO_HOME)/mk/Make.defs

## this is the name of the file with 'main()'
ebase := main    # change this !!

## Chombo libraries needed by this program
LibNames := AMRElliptic AMRTimeDependent AMRTools BoxTools

## Other libraries needed by this program (using -L and -l options)
XTRALIBFLAGS :=

## 'all' is the default target and 'all-test' is defined in Make.rules
all: all-test

## defines the rules to build everything
include $(CHOMBO_HOME)/mk/Make.rules
```

The first line defines the CHOMBO_HOME variable, which tells the rest of the makefiles where the Chombo home directory is. The directory containing the application code can be anywhere relative to the Chombo directory, but the CHOMBO_HOME variable must be defined appropriately. The next line uses that variable to define the variables the rest of the makefiles need. The next line will need to be changed for a different application. As it is shown here, there should be a file named "main.cpp" in the directory containing this makefile. The Chombo makefile system will compile this file and all the other source files in the directory. The next line specifies which Chombo libraries are needed. This might have to be changed. The next line shows the variable that should be defined to specify additional libraries. The value should contain the options to be given to the linker to access the libraries (e.g "-Lsomedir -lsomelib"). The next line defines the "all" target, which will be the default target. The Chombo makefile system handles everything when the target is invoked. Actually, the "all-test" target does the work. The final line includes the makefile that defines this target, and all the rules that it needs.

For more complex applications, a slightly more verbose approach allows the Chombo makefile system to build an application with source code in multiple directories. The files

Chombo/example/AMRWaveEqn/exec/GNUmakefile
Chombo/example/AMRWaveEqn/src/Make.package

show how to do this. In general, the GNUmakefile looks similar to the description above. The major difference is that each subdirectory that contains source code is mentioned explicitly in the GNUmakefile, and some variables are defined. Each source directory must contain a file called "Make.package" that specifies the files to be compiled. The final line includes another Chombo makefile ("Make.example" instead of "Make.rules" used in the first approach) that defines all the targets and rules. The "make all" command would be used to build the application, just as with the Chombo tests and examples.

1.2.5.2 Using an existing application makefile

To use an existing makefile, it is necessary to modify the rules for compiling C++ code that uses Chombo classes and to modify the link rule to use the Chombo libraries.[3]

The compile rules must be modified to add the "Chombo/lib/include" subdirectory to the search path for C++ header files (-I option for most compilers) and to define some C-preprocessor macro variables that the Chombo header files use. The compiler options for this will usually look something like:

-IChombo/lib/include -I<HDF_DIR>/include -DCH_SPACEDIM=<dim>
-DCH_USE_<precision> -DCH_<system> -DCH_LANG_CC -DHDF5 -DMPI

where <HDF_DIR> is the directory <dim> is the number of dimensions in the problem (2 or 3), <precision> is the type for floating point variables (FLOAT or DOUBLE) and <system> is the name of the operating system (see "Chombo/lib/mk/Make.defs" for the values of the "$system" makefile variable). Note that "-DMPI" should only be used when compiling for parallel execution.

The link rule must be modified to add the Chombo "lib" directory to the search path for libraries and to specify the Chombo libraries to be searched. The linker options for this will usually look something like:

-LChombo/lib -lamrelliptic<config> -lamrtimedependent<config>
-lamrtools<config> -lboxtools<config> -L<HDF_DIR>/lib -lhdf5 -lz

[3]This assumes that the Chombo libraries are compiled before trying to compile the application.

where <HDF_DIR> is as above, "-lhdf5 -lz" are the HDF5 libraries and <config> is the Chombo configuration string. See "Chombo/lib/mk/Make.defs.config" to see how to compute the configuration string.

Chapter 2

BoxTools

2.1 AMR Spatial Discretization

The underlying discretization of space is given as points $(i_0, ..., i_{\mathbf{D}-1}) = \boldsymbol{i} \in \mathbb{Z}^{\mathbf{D}}$. The problem domain is discretized using a grid $\Gamma \subset \mathbb{Z}^{\mathbf{D}}$ that is a bounded subset of the lattice. Γ is used to represent a cell-centered discretization of the continuous spatial domain into a collection of control volumes: $\boldsymbol{i} \in \Gamma$ represents a region of space $[\boldsymbol{x}_0 + (\boldsymbol{i} - \frac{1}{2}\boldsymbol{u})h, \boldsymbol{x}_0 + (\boldsymbol{i} + \frac{1}{2}\boldsymbol{u})h]$, where $\boldsymbol{x}_0 \in \mathbb{R}^{\mathbf{D}}$ is some fixed origin of coordinates, h is the mesh spacing, and $\boldsymbol{u} \in \mathbb{Z}^{\mathbf{D}}$ is the vector whose components are all equal to one. We can also define various face-centered and node-centered discretizations of space based on those control volumes. For example, we denote by $\Gamma^{\boldsymbol{v}}$ the set of points in physical space of the form $\boldsymbol{x}_0 + (\boldsymbol{i} \pm \frac{1}{2}\boldsymbol{v})h, \boldsymbol{i} \in \Gamma$. Here $\boldsymbol{v}$ can be any vector whose entries are equal to either zero or one.

We will find it useful to define a number of operators on points and subsets of $\mathbb{Z}^{\mathbf{D}}$. We denote by $|\boldsymbol{i}| = \underset{d=0...\mathbf{D}-1}{max}(|i_d|)$, $\Gamma + \boldsymbol{i}$ as the translation of a set by a point in $\mathbb{Z}^{\mathbf{D}}$, and $\mathcal{G}(\Gamma, r)$ to the set of all points within a $|\cdot|$-distance r of Γ

$$\mathcal{G}(\Gamma, r) = \underset{|\boldsymbol{i}|\leq r}{\cup} \Gamma + \boldsymbol{i}$$

We define a coarsening operator by $\mathcal{C}_r : \mathbb{Z}^{\mathbf{D}} \to \mathbb{Z}^{\mathbf{D}}$,

$$\mathcal{C}_r(\boldsymbol{i}) = (\lfloor \frac{i_0}{r} \rfloor, ..., \lfloor \frac{i_{d-1}}{r} \rfloor)$$

where r is a positive integer. The coarsening operator acting on subsets of $\mathbb{Z}^{\mathbf{D}}$ can be extended in a natural way to the other grid centerings: $\mathcal{C}_r(\Gamma^{\boldsymbol{v}}) \equiv (\mathcal{C}_r(\Gamma))^{\boldsymbol{v}}$.

We extend this discretization of space to represent a nested hierarchy of grids that discretize the same continuous spatial domain. We assume that our problem domain can be discretized by a nested hierarchy of grids $\Gamma^0 ... \Gamma^{lmax}$, with $\Gamma^{l+1} = \mathcal{C}^{-1}_{n^l_{ref}}(\Gamma^l)$, and $\boldsymbol{x}^l_0 - \frac{1}{2}\boldsymbol{u}h^l$ independent of l. The integer n^l_{ref} is the refinement ratio between level l and

$l+1$. We also assume that the mesh spacings h^l associated with Γ^l satisfy $\frac{h^{l+1}}{h^l} = n^l_{ref}$. These conditions imply that the underlying continuous spatial domains defined by the control volumes are all identical.

Adaptive mesh refinement calculations are performed on a hierarchy of meshes $\Omega^\ell \subset \Gamma^\ell$, with $\Omega^\ell \supset \mathcal{C}_{n^\ell_{ref}}(\Omega^{\ell+1})$. Typically, Ω^l is decomposed into a disjoint union of rectangles in order to perform calculations efficiently. We denote such a decomposition by $\mathcal{R}(\Omega^l) = \{\Omega^l_k\}_{k=1}^{N^l_{grids}}$, where the Ω^l_k's are rectangles and $\Omega^l_k \cap \Omega^l_{k'} = \emptyset$ if $k \neq k'$. We say that $\mathcal{R}(\Omega^l)$ is p-blocked, $p > 1$, if $\Omega^l_k = \mathcal{C}_p^{-1}(\mathcal{C}_p(\Omega^l_k))$ for all k. We will assume throughout that Ω^l admits a decomposition $\mathcal{R}(\Omega^l)$ that is n^{l-1}_{ref}-blocked for all $l > 0$. In particular, the control volume corresponding to a cell in Ω^{l-1} is either completely contained in the control volumes defined by Ω^l or its intersection has zero volume. We will also assume that there is at least one level l cell separating level $l+1$ cells from level $l-1$ cells: $\mathcal{G}(\mathcal{C}_{n^l_{nref}}(\Omega^{l+1}), 1) \cap \Gamma^l \subseteq \Omega^l$. We will refer to grid hierarchies that meet these two conditions as being *properly nested*. We emphasize that this form of proper nesting is a minimum requirement for the AMR algorithms discussed in this document. For some applications, it may be necessary to impose more stringent conditions on the grid hierarchy.

A discretized dependent variable in AMR is a *level array*

$$\varphi^l : \Omega^l \rightarrow \mathbb{R}^m$$

We denote by $\varphi_{\boldsymbol{i}} \in \mathbb{R}^m$ the value of φ at cell $\boldsymbol{i} \in \Omega^l$. We can also define level arrays on other grid centerings, i.e., $\psi : \Omega^{l,\boldsymbol{v}} \rightarrow \mathbb{R}^m$. In that case, we denote the indexing operation by $\psi_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{v}} \in \mathbb{R}^m$. In particular, we can define vector fields at a level

$$\vec{F}^l = (F^l_0, ..., F^l_{\mathbf{D}-1}) \ , \ F^l_d : \Omega^{l,\boldsymbol{e}^d} \rightarrow \mathbb{R}^m$$

We will be interested in operations on pairs of refined grids that are not necessarily contained in an AMR mesh hierarchy (e.g., during regridding). In those cases, we will denote by Γ^f, Γ^c the fine and coarse problem domains, n_{ref} the refinement ratio between the two levels, Ω^f, Ω^c the refined regions in the two domains, and φ^f, φ^c, etc., level arrays defined on Ω^f, Ω^c. We will always assume that the two levels are properly nested.

In the remainder of this section, we will describe *BoxTools*, a set of abstractions for defining points and regions in a multidimensional integer lattice index space, and representing aggregate data in such regions. The classes defined in the remainder of this chapter correspond to the mathematical objects described above in the following fashion.

- Points in the rectangular lattice $\boldsymbol{i} \in \mathbb{Z}^{\mathbf{D}} \Leftrightarrow$ the class `IntVect`.
- Rectangular subsets $\Gamma \subset \mathbb{Z}^{\mathbf{D}} \Leftrightarrow$ the class `Box`.
- Arbitrary subsets $\mathcal{I} \subset \mathbb{Z}^{\mathbf{D}} \Leftrightarrow$ the class `IntVectSet`.
- Rectangular arrays $\varphi : \Gamma \rightarrow \mathbb{R}^m \Leftrightarrow$ the class `FArrayBox`.
- Unions of rectangles at a fixed level of refinement $\Omega, \mathcal{R}(\Omega)$, and their distribution onto processors $\Leftrightarrow$ the class `DisJointBoxLayout`.
- Level arrays $\varphi : \Omega \rightarrow \mathbb{R}^m \Leftrightarrow$ the class `LevelData<FArrayBox>`.

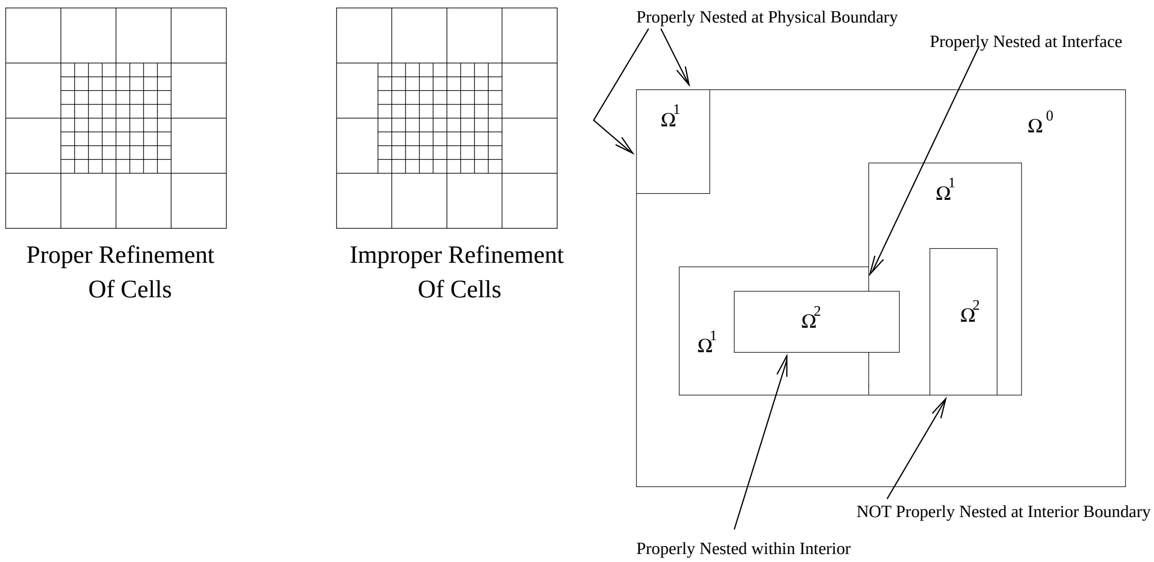

Figure 2.1: Examples illustrating proper nesting requirements for locally refined grids.

2.2 Points, Regions and Rectangular Arrays

BoxTools is a set of abstractions for defining points and regions in a multidimensional integer lattice index space, and representing aggregate data in such regions. The dimensionality $\mathbf{D}$ of the index space is a compilation-time constant. It is accessed as a macro `CH_SPACEDIM`, which is set in the Make process, and is propagated into Fortran .F or .ChF files in applying the C preprocessor. A second compile-time constant in BoxTools is that of the precision of floating-point variables. BoxTools provides a type `Real`, which is set using a `typedef` declaration to either `float` or `double` at compile time. The macro `REAL_T` serves the same function in Fortran. This macro is defined in the file `REAL.H`, which can be included as a header for both C++ and Fortran files.

2.2.1 The Class `IntVect`

`IntVect`s represent points in the rectangular lattice $\mathbb{Z}^{\mathbf{D}}$.

Operations on `IntVect`s. In the following definitions $\boldsymbol{i}$, $\boldsymbol{j}$ are `IntVect`s and s, d are integers, $0 \leq d < \mathbf{D}$.

• Constructors. `IntVect` has the usual default and copy constructors, as well as constructors that take tuples of integers as arguments, e.g., `IntVect`(i_0, i_1) (two dimensions), `IntVect`(i_0, i_1, i_2) (three dimensions).

• Arithmetic operators. $\boldsymbol{i} \oplus \boldsymbol{j}, \boldsymbol{i} \oplus s, \oplus \in \{$`+, -, *, /`$\}$ produce `IntVect`s by operating componentwise on the inputs. `+=, -=, *=, /=`, perform the same operations in place. e.g., $\boldsymbol{i}$`+=`$\boldsymbol{j}$ is the same as $\boldsymbol{i} = \boldsymbol{i} + \boldsymbol{j}$. `IntVect` also provides component-wise $\mathtt{min}(\boldsymbol{i}, \boldsymbol{j}), \mathtt{max}(\boldsymbol{i}, \boldsymbol{j})$ operators.

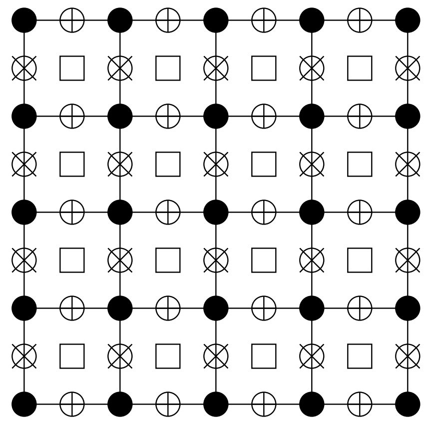

Figure 2.2: Vertex (•), cell (□) and face (⊕ and ⊗) sites on a grid.

• Logical operators. $\boldsymbol{i}_1$==$\boldsymbol{i}_2$, ($\boldsymbol{i}_1$!=$\boldsymbol{i}_2$) Test for mathematically equal (unequal) `IntVects`. Comparison operators are defined element-wise: >,>=,<,<=, $\boldsymbol{i} < \boldsymbol{j}$ iff $i_d < j_d$. Lexicographic ordering operators $\boldsymbol{i}$.`lexLT`($\boldsymbol{j}$), $\boldsymbol{i}$.`lexGT`($\boldsymbol{j}$) are also provided.

• Static members. `Unit` is the `IntVect` consisting of all ones. `Zero` is the vector consisting of all zeros. `BASISV`$(d), d = 0, ..., \mathbf{D} - 1$ returns the unit `IntVect` in the s direction.

• Indexing operations. $\boldsymbol{i}[d]$ returns the component of $\boldsymbol{i}$, and can be used to assign values to components: $\boldsymbol{i}[d] = q$.

2.2.2 The Class `Box`

A `Box` represents a rectangular region in $\mathbb{Z}^{\mathbf{D}}$, defined by specifying the `IntVects` defining its low and high corners. For each coordinate direction, a `Box` can be *cell-centered* or *node-centered*. This allows one to represent the various face-, edge-, and node-centered rectangular grids (figure 2.2).
Operations on `Box`es. In what follows, B, B_1, B_2 are `Boxes`, $\boldsymbol{i}, \boldsymbol{i}_1, \boldsymbol{i}_2, \boldsymbol{v}$ are `IntVects`, $\boldsymbol{v}$ having components equal to zero or one, and s, d are integers, $0 \leq d < \mathbf{D}$.

• Constructors. $B(\boldsymbol{i}_1, \boldsymbol{i}_2, \boldsymbol{v} =$ `Zero`$)$ Constructs a Box with low and high corners $\boldsymbol{i}_1, \boldsymbol{i}_2$, and centering defined by $\boldsymbol{v}$. If $v_d = 0$, then the `Box` is cell-centered in the d direction; if $v_d = 1$, then the `Box` is node-centered in the d direction. In particular, the default centering is cell-centered in all three directions. `Box` has a copy constructor and assignment operator. One can reset the low and high corners of the `Box` (`setSmall`($\boldsymbol{i}$), `setBig`($\boldsymbol{i}$)).

• Logical functions. B_1==B_2, B_1!=B_2 test whether B_1 and B_2 are equal or unequal, including having the same centering. B.`isEmpty`() tests whether B is empty. B.`contains`(B_1), B.`contains`($\boldsymbol{i}$), tests whether B contains $B_1, \boldsymbol{i}$. B_1.`intersects`(B_2) checks whether $B_1 \bigcap B_2 \neq \emptyset$. B_1.`sameType`(B_2), B_1.`sameSize`(B_2) check whether B_1 and B_2 have the same centering, or whether $B_1 = B_2 + \boldsymbol{i}$ for some $\boldsymbol{i}$. $B_1 < B_2$ if B_1.`smallEnd`().`LexLT`(B_2.`smallEnd`()).

• Shifting and Centering. B.convert($\boldsymbol{v}$) changes the centering of B to that specified by $\boldsymbol{v}$, as in the constructor. One can also change the centering in one direction. B.surroundingNodes() converts all the cell-centered directions to node-centered, and increments the high corner in those directions by 1. B.surroundingNodes(d) performs the same operation in the d coordinate direction. B.enclosedCells(), B.enclosedCells(d) perform the opposite operation, converting the centerings to be cell-centered, and decrementing by 1 the high corner of the box for those directions for which the centering changed. There are also corresponding friend functions, e.g., surroundingNodes(B) that return a new box with the appropriately modified centerings. The various grids depicted in figure 2.2 can be obtained from on another by application of the member functions surroundingNodes and enclosedCells. B.shift($\boldsymbol{i}$), B+=$\boldsymbol{i}$ perform the identical operation of replacing B with Box(B.smallEnd()$+\boldsymbol{i}$, B.bigEnd()$+\boldsymbol{i}$). B.shift(d, s) is the same as B+=$s *$ BASISV(d). B-=$\boldsymbol{i}$ is the same as B+=($-\boldsymbol{i}$).

• Size functions. B.smallEnd(), B.bigEnd(), B.size(), return IntVects containing the low corner, high corner, and size in each direction. The same functions called with an integer argument (B.size(s)) returns the s-th component of those IntVects. B.numPts() returns the discrete volume of B. B.loVect(), B.highVect(), return pointers to the $\mathbf{D}$-tuples of integers defining the low and high corners of B in order to pass them to Fortran.

• Set operations. Although it is not possible to define a complete set calculus on Boxes (the union of two rectangles is not always a rectangle), Box provides many of the set functions most commonly required. B_1&=B_2 sets $B_1 = B_1 \bigcap B_2$. B.minBox(B_2) sets B_1 to be the minimum sized Box containing B_1, B_2. B.grow(s) grows B in all directions by a size s (s can be negative corresponding to shrinking). B.grow($\boldsymbol{i}$) grows B by i_d in the dth direction, and B.grow(d, s) $=$ B.grow($s * BASISV(d)$). B.coarsen(s) $= \mathcal{C}_s(B)$, B.refine(s) $= {\mathcal{C}_s}^{-1}(B)$. B can also be coarsened and refined by different amounts in the various coordinate directions using B.coarsen($\boldsymbol{i}$), B.refine($\boldsymbol{i}$). Grow, minBox, coarsen, refine all have corresponding friend functions that return a new Box on which the operation has been performed, e.g., minBox(B_1, B_2). adjCellLo($B, d, s = 1$) returns the cell-centered box of width s direction adjacent to B on the low side in the dth coordinate direction. adjCellHi performs the same operation on the high side of B in the dth direction and adjBdryLo, adjBdryHi return the corresponding node-centered Boxes.

2.2.3 IntVectSet

IntVectSet represents an irregular region in an integer lattice $\mathbf{D}$-dimensional index space as an arbitrary collection of IntVects. A full set calculus is defined.
Operations on IntVectSets. In the following, $\mathcal{I}, \mathcal{I}_1, \mathcal{I}_2$ are IntVectSets, B is a Box, and s is an integer.

• Constructors. The default constructor constructs an empty IntVectSet. They can also be initialized at construction with an `IntVect`, a `Box` or another `IntVectSet`. An existing `IntVectSet` can also be re-initialized with any of those three objects using the member functions `define`. `IntVectSet` has an assignment operator.

• Set Operations. `IntVectSets` can be updated in place by taking unions ($\mathcal{I}_1$`|=`$\mathcal{I}_2$, $\mathcal{I}$`|=`B, $\mathcal{I}$`|=`$\boldsymbol{i}$) intersections ($\mathcal{I}_1$`&=`$\mathcal{I}_2$, $\mathcal{I}$`&=`B, $\mathcal{I}$`&=`$\boldsymbol{i}$) and set-theoretic differences ($\mathcal{I}_1$`-=`$\mathcal{I}_2$, $\mathcal{I}$`-=`B, $\mathcal{I}$`-=`$\boldsymbol{i}$) with another `IntVectSet`, a `Box` or an `IntVect`. $\mathcal{I}$.`coarsen`(s) sets $\mathcal{I}$ to $\mathcal{C}_s(\mathcal{I})$, $\mathcal{I}$.`refine`(s) changes $\mathcal{I}$ to $\mathcal{C}_s^{-1}(\mathcal{I})$. $\mathcal{I}$.`grow`(s) changes $\mathcal{I}$ to $\bigcup_{\boldsymbol{i}:|\boldsymbol{i}|\leq s}(\mathcal{I}+\boldsymbol{i})$. Union, intersection, difference, `coarsen`, and `refine` all have associated friend functions that return a new `IntVectSet` suitably modified. For example, $\mathcal{I}_1|\mathcal{I}_2$ returns $\mathcal{I}_1\bigcup\mathcal{I}_2$, leaving $\mathcal{I}_1$ and $\mathcal{I}_2$ unchanged. `shift`$(\mathcal{I},\boldsymbol{i})$ returns $\mathcal{I}+\boldsymbol{i}$.

• Other functions. $\mathcal{I}$.`isEmpty`() returns true if $\mathcal{I}=\emptyset$. $\mathcal{I}$.`minBox`() returns the minimum cell-centered `Box` containing $\mathcal{I}$. $\mathcal{I}$.`contains`(B), $\mathcal{I}$.`contains`($\boldsymbol{i}$) returns true if $B\subset\mathcal{I},\boldsymbol{i}\in\mathcal{I}$.

Performance Issues. `IntVectSet` uses two representations internally: a fast bitmap for small sets, and a slower tree representation for large sets. The heuristic employed between switching between the two representations is to use bitmaps for sets that are initialized to be rectangles, or that are obtained by applying intersection, set-theoretic difference, `coarsen`, `refine` or `grow` to `IntVectSet`(s) that are represented by bitmaps. However, the use of the union function causes the representation to be converted irreversibly to a tree representation, with a significant performance penalty. For that reason, the union operations should be used sparingly.

2.2.4 `Box` and `IntVectSet` Iterators

A `BoxIterator` or `IVSIterator` traverses a sequence of `IntVects` that comprise a given `Box` or `IntVectSet`. Each `IntVect` appears exactly once in the sequence. There is no guarantee that the `IntVects` will appear in any particular order.
Operations on `BoxIterator`, `IVSIterator`. In what follows, B is a `Box`, $\mathcal{I}$ is an `IntVectSet`, and `iter` is either a `BoxIterator` or an `IVSIterator`.

• Construction The iterators can be constructed with object to be iterated over (`iter`(B), `iter`($\mathcal{I}$)), or null-constructed and defined later (`iter.define`(B), `iter.define`($\mathcal{I}$)).

• Iteration. `iter.begin()` sets `iter` to the beginning of the iteration sequence, `++iter` advances `iter` to the next iterate, and `iter.ok()` checks to see if the current iterate is valid. A null-constructed iterator, or an iterator constructed with an empty `Box` or `IntVectSet` will always return false. `iter()` returns an `IntVect` containing the current value of the iterate.

2.2.5 Interval

An `Interval` consists of two ordered integers. An `Interval` can be created only by specifying its endpoints. The only operations that can be performed are to extract its endpoints or determine its size, which is the number of integers it contains. If the endpoints are equal, the size is one. It is permitted to define an `Interval` with zero or negative size. It is entirely the responsibility of the user to determine whether this is valid. `Interval` interacts only weakly with the other abstractions and is exclusively used to specify data component ranges in Chombo (see sections 2.2.6 and 2.4.3).

2.2.6 Rectangular arrays

A `BaseFab<T>` is a templated container class for multidimensional array data. It consists of three major elements: a `Box` to define the range of spatial indices over which the array is defined; an integer specifying the number of components; and a `T*` pointing to a contiguous block of array elements. The data is stored in Fortran order so that the pointer can be passed to a Fortran routine where it can be accessed as a multidimensional array.

A `BaseFab` is defined by specifying a domain in the form of a `Box`, which can have any centering, and the number of components, $ncomps$. This is intended to represent a $\mathbf{D}+1$ dimensional array in Fortran. A *component index* is an integer in the range 0 to $ncomps-1$ which is used to specify or select a component of a `BaseFab`. A range of component indices is often represented by an `Interval`. An already defined `BaseFab` can be redefined with a new domain and number of components. The behavior of existing data is undefined during redefinition. `BaseFabs` are large aggregate objects containing pointer data, so that shallow copy can lead to subtle bugs, and deep copying is expensive. For that reason, the assignment operator and copy constructor have been rendered inaccessible to the user by making them private. In particular, it is necessary to pass `BaseFabs` as reference parameters in procedure calls.

Operations on `BaseFabs`. In the following $\mathcal{A}, \mathcal{A}'$ are `BaseFabs`, B is a `Box`, $\boldsymbol{i}$ is an `IntVect`, and n_{comp}, d, s are integers, with $0 \leq d < \mathbf{D}, n_{comp} > 0$.

• Constructors. `BaseFab` has a default constructor, as well as a constructor `BaseFab`(B) that completely defines the BaseFab. $\mathcal{A}$.`resize`(B, n_{comp}) resets $\mathcal{A}$ to be defined over a `Box` B and with n components. Any data contained in $\mathcal{A}$ previously is discarded, and the data $\mathcal{A}$ is assumed to be uninitialized.

• Accessors. $\mathcal{A}(\boldsymbol{i}, s)$ is an indexing operator, returning a reference of type `T&` to the storage location for the value at point $\boldsymbol{i}$ and component s. For a `BaseFab` that is node-centered in one or more of the coordinate directions, the convention for indexing with an `IntVect` (which does not have centering) is that $\mathcal{A}(\boldsymbol{i}, s)$ returns the reference corresponding to $\boldsymbol{i} - \frac{1}{2}\boldsymbol{v}$, where $\boldsymbol{v}$ is the `IntVect` of zeros and ones defining the centering, i.e., the cell center and the node-centered points on the low side have the same index. $\mathcal{A}$.box() returns the `Box` over which $\mathcal{A}$ is defined, and $\mathcal{A}$.ncomp() the number of components. `BaseFab` provides an interface to the `Box` member functions `smallEnd()`, `bigEnd()`, `loVect()`, `hiVect()`: $\mathcal{A}$.`smallEnd()` == $\mathcal{A}$.box().`smallEnd()`, etc. $\mathcal{A}$.`dataPtr`(s) returns a pointer of type $T*$ to the data in $\mathcal{A}$ beginning at the nth component; n defaults to zero. $\mathcal{A}$.nCompPtr() returns a pointer to an integer containing the number of components. `loVect, hiVect, dataPtr, nCompPtr` are to be used in calling Fortran.

• <u>Data Modification Functions</u>. $\mathcal{A}$.setVal(t) sets all of the data values in $\mathcal{A}$ to the single value t. $\mathcal{A}$.copy($\mathcal{A}_2$) copies all of the values in $\mathcal{A}_2$ into the part of $\mathcal{A}_1$ defined on $\mathcal{A}$.box()&$\mathcal{A}_2$.box(). $\mathcal{A}_1$ and $\mathcal{A}_2$ must have the same number of components. Both `setVal` and `copy` have overloaded versions that permit the operations to be performed on a specified sub-rectangle and over subsets of the component ranges.

• <u>Domain Modification Functions</u> $\mathcal{A}$.shift($\boldsymbol{i}$) changes the `Box` over which $\mathcal{A}$ is defined to $\mathcal{A}.\mathtt{box}() + \boldsymbol{i}$, leaving the data unmodified. Mathematically, $\mathcal{A}$ becomes $\mathcal{A}'$, with $\mathcal{A}'(\boldsymbol{j}, s)$==$\mathcal{A}(\boldsymbol{j} - \boldsymbol{i}, s), \forall \boldsymbol{j} \in \mathcal{A}'$.box(). The `shift` function is overloaded to shift $\mathcal{A}$ by some distance in a single coordinate direction ($\mathcal{A}$.shift(d, s)). $\mathcal{A}$.shiftHalf($\boldsymbol{i}$) shifts the domain of $\mathcal{A}$ by $\boldsymbol{i}$ "halves" in each direction, where a half-shift changes the centering to the adjacent nodes/ cells centered `Box` in that direction.

FArrayBox

An `FArrayBox` is-a `BaseFab<Real>` which contains floating-point data. A number of aggregate floating-point arithmetic operations are provided. `FArrayBox` is implemented as a derived class from `BaseFab<Real>`. In addition to `BaseFab` operations, `FArrayBox` has a collection of operations that are specialized to real-valued arrays.

• <u>Pointwise Arithmetic Operators</u> $\mathcal{A}_1 \oplus = \mathcal{A}_2$, $\oplus \in \{$`+, -, *, /`$\}$, updates in place the values of $\mathcal{A}_1(\boldsymbol{i}, s)$ with $\mathcal{A}_1(\boldsymbol{i}, s) \oplus \mathcal{A}_2(\boldsymbol{i}, s)$, for $0 \leq s < \mathcal{A}_1.\mathtt{nComps}() = \mathcal{A}_2.\mathtt{nComps}()$, and $\boldsymbol{i} \in \mathcal{A}_1.\mathtt{box}() \bigcap \mathcal{A}_2.\mathtt{box}()$. There are also a collection of member functions `plus, minus, mult, divide`, that perform these operations over subboxes and subranges of the components. $\mathcal{A}$.abs() updates in place the values of $\mathcal{A}$ with their absolute values. `abs` is overloaded with versions specifying subbox and a single component. The unary operators `negate` and `invert` behave in a similar fashion to `abs`.

• <u>Reduction Operators</u> $\mathcal{A}$.sum(s), $\mathcal{A}$.min(s), $\mathcal{A}$.max(s), return real values containing the sum, minimum, and maximum of the values of the s-th component of $\mathcal{A}$. $\mathcal{A}$.minIndex(s), $\mathcal{A}$.maxIndex(s) return `IntVects` corresponding to one of the locations $\boldsymbol{i}$ such that the minimum or maximum is attained. $\mathcal{A}.\mathtt{norm}(p, s), p \geq 1$ returns the discrete p norm of the s-th component of $\mathcal{A}$. $\mathcal{A}.\mathtt{norm}(p, s) = (\sum_{\boldsymbol{i} \in \mathcal{A}.\mathtt{box}()} |\mathcal{A}(\boldsymbol{i}, s)|^p)^{1/p}$. There are also overloaded versions of these functions that perform their operations over a subbox, or for a range of components.

• <u>Mask Functions</u> $\mathcal{A}$.maskLT(M, a, s) sets the values of the input `BaseFab<int>` M to one or zero, depending on whether $\mathcal{A}(\boldsymbol{i}, s) < a$ or not. M is resized by the function so that $M.\mathtt{box}() = \mathcal{A}.\mathtt{box}()$. `maskLT` also returns the integer number of non-zero entries in M. `maskLE, maskEQ, maskGT, maskGE` are defined similarly.

The Fortran Interface. The collection of values taken on by a `BaseFab` $\mathcal{A}$ is stored in a contiguous block of storage beginning at $\mathcal{A}$.dataPtr(). The data is stored in Fortran ordering corresponding with the spatial indices first, followed by the component index. Specifically, if a Fortran routine is called from C++

```
extern "C" {foo_(real*, int*, int* , int* );}

  FArrayBox A(B,nc);
  foo_(A.dataPtr(),A.loVect(),A.hiVect(),A.nCompPtr())
```

The indexing of $\mathcal{A}$ in the Fortran routine is given by

```
      subroutine foo(a,lo,hi,nc)

      integer lo(0:CHF_SPACEDIM-1)
      integer hi(0:CHF_SPACEDIM-1)
      real_t a(lo(0):hi(0),lo(1):hi(1),0:nc-1)

      do ic =0,nc-1
      do j = lo(1),hi(1)
      do i = lo(0),hi(0)

      a(i,j,ic) = ...
```

For further details on the Fortran interface, see Chapter 8.

2.2.6.1 Aliasing

`BaseFab<T>`, and by inheritance `FArrayBox`, can also be built as an *alias* of another `BaseFab<T>` (where T is the same for the two objects). For example:

```
  BaseFab<int> original(b, 4);
  Interval     subcomps(2, 3);
  BaseFab<int> alias(subcomps, original);
  // component 0 of alias is equivalent to component 2 of original
```

This BaseFab does not allocate any memory, but sets its data pointer into the memory pointed to by the argument BaseFab. It is the users responsibility to ensure this aliased BaseFab is not used after the original BaseFab has deleted its data member (resize, define(..) called, or destruction, etc.).

This is similar to using an offset pointer into an array. The offset pointer is only valid as long as the original array is valid.

This aliased BaseFab will also generate side effects (modifying the values of data in one will modify the other's data). Deleting the alias will not affect the original.

This aliased BaseFab will have `subcomps.size()` components, starting at zero. The alilased BaseFab can only have the same `Box` domain as the original.

2.3 The Class ProblemDomain

`ProblemDomain` is a class to handle interaction with boundary conditions at the edge of the computational domain, either physical boundary conditions or periodic ones. This class contains much of the functionality of the Box class, since logically the computational domain is generally a Box.

Intersection with a ProblemDomain object will result in only removing regions which are outside the physical domain in non-periodic directions. Regions outside the logical computational domain in periodic directions will be treated as ghost cells which can be filled with an exchange() function or through suitable interpolation from a coarser domain.

Since ProblemDomain will contain a Box, it is a dimension-dependent class, so SpaceDim must be defined as either 1, 2, or 3 when compiling.

Note that this implementation of ProblemDomain is inherently cell-centered.

The user interface for `ProblemDomain` is as follows:

- `ProblemDomain()`

  Default constructor – the object is defined in an unusable state until the user calls the `define` function.

- `ProblemDomain(const Box& domBox, const bool* isPeriodic)`

  Full constructor. Places the `BRMeshRefine` object in a usable state.

  Arguments:

  - `domBox` Computational domain.
  - `isPeriodic` SpaceDim array of bools which defines whether BC's are physical or periodic in each coordinate direction.

- `ProblemDomain(const Box& domBox)`

  Partial constructor, creates non-periodic (in any coordinate direction) ProblemDomain.

  Arguments:

  - `domBox` Computational domain.

- `ProblemDomain(const IntVect& small, const IntVect& big, const bool* isPeriodic)`

  Full constructor, creates ProblemDomain.

  Arguments:

  - `small` Location of lower-left corner of domain box

- `big` Location of upper-right corner of domain box
- `isPeriodic` SpaceDim array of bools which defines whether BC's are physical or periodic in each coordinate direction.

- `ProblemDomain(const IntVect& small, const IntVect& big)`

  Partial constructor, creates non-periodic (in any coordinate direction) ProblemDomain.

  Arguments:

  - `small` Location of lower-left corner of domain box
  - `big` Location of upper-right corner of domain box

- ```
  ProblemDomain(const IntVect& small, const int* vec_len,
                const bool* isPeriodic)
  ```

  Full constructor, creates ProblemDomain.

  Arguments:

  - `small` Location of lower-left corner of domain box
  - `vec_len` Size of domain in each direction.
  - `isPeriodic` SpaceDim array of bools which defines whether BC's are physical or periodic in each coordinate direction.

- `ProblemDomain(const IntVect& small, const int* vec_len)`

  Partial constructor, creates ProblemDomain with non-periodic boundary conditions by default.

  Arguments:

  - `small` Location of lower-left corner of domain box
  - `vec_len` Size of domain in each direction.

- `ProblemDomain(const ProblemDomain& src)`

  Copy constructor

- `const Box& domainBox() const`

  Returns logical computational domain.

- `bool isPeriodic(int dir) const`

  Returns `true` if boundary condition is periodic in direction `dir`.
  ```

- `bool isPeriodic() const`

  Returns `true` if boundary condition is periodic in any direction.

- `ShiftIterator shiftIterator() const`

  Returns ShiftIterator for this problem domain. ShiftIterator is a utility class to aid with periodic boundary condtions whose use is mostly internal to BoxLayout, Copier, etc which allows looping over IntVects which used to shift the domain for enforcing periodic boundary conditions.

- `bool isEmpty() const`

  Returns `true` if this ProblemDomain has an empty domainBox.

- `bool contains(const IntVect& p) const`

  Returns true if argument is contained within this ProblemDomain. An empty ProblemDomain does not contain and is not contained by any ProblemDomain. In a periodic direction, all locations are contained, since a periodic domain is an infinite domain. If periodic in all directions, this will always return `true`.

- `bool contains(const Box& b) const`

  Returns true if argument is contained within this ProblemDomain. An empty ProblemDomain does not contain and is not contained by any ProblemDomain. In a periodic direction, all locations are contained, since a periodic domain is an infinite domain. If periodic in all directions, this will always return `true`.

- `bool intersects( const Box& a_box) const`

  Returns true if this ProblemDomain and the argument have non-null intersections. It is an error if a_box is not cell-centered. An empty ProblemDomain does not intersect any Box. Boxes always intersect in periodic dimensions, since a periodic domain is an infinite domain. If periodic in all directions, this will always return `true`.

- `bool intersectsNotEmpty (const Box& a_box) const`

  Returns true if this ProblemDomain and the argument have non-null intersections. It is an error if a_box is not cell-centered. This routine does not perform the check to see if *this or b are empty boxes. It is the callers responsibility to ensure that this never happens. If you are unsure, the use the .intersects(..) routine. In periodic directions, will always return true.

- `bool intersects(const Box& box1, const Box& box2) const`

  Returns true of box1 and box2 and any of their periodic images intersect. (This is useful for checking disjointness).

- `ProblemDomain& operator= (const ProblemDomain& b)`

  Assignment operator.

- `void setPeriodic(int a_dir, bool a_isPeriodic)`

  Sets whether boundary condition is periodic in direction a_dir (`true` is periodic).

- `friend`
  `Box bdryLo(const ProblemDomain& a_pd, int a_dir, int a_len=1)`

  Returns the edge-centered box (in direction a_dir) defining the low side of the domainBox in the argument ProblemDomain. The neighbor of an empty ProblemDomain is an empty Box of the appropriate type. If dir is a periodic direction, will return an empty box.

  Arguments:

  - `a_pd` input ProblemDomain
  - `a_dir` normal direction of edge to return. Directions are zero-based and must be $0 \leq$`a_dir` $<$`SpaceDim`.
  - `a_len` Width of returned box in normal direction `a_dir`.

- `friend`
  `Box bdryHi(const ProblemDomain& a_pd, int a_dir, int a_len=1)`

  Returns the edge-centered box (in direction a_dir) defining the high side of the domainBox in the argument ProblemDomain. The neighbor of an empty ProblemDomain is an empty Box of the appropriate type. If dir is a periodic direction, will return an empty box.

  Arguments:

  - `a_pd` input ProblemDomain
  - `a_dir` normal direction of edge to return. Directions are zero-based and must be $0 \leq$`a_dir` $<$`SpaceDim`.
  - `a_len` Width of returned box in normal direction `a_dir`.

- `friend`
  `Box adjCellLo(const ProblemDomain& a_pd, int a_dir, int a_len=1)`

  Returns the cell-centered box (in direction a_dir) adjacent to the low side of the domainBox in the argument ProblemDomain. The neighbor of an empty ProblemDomain is an empty Box of cell-centered type. If dir is a periodic direction, will return an empty box.

  Arguments:

– `a_pd` input ProblemDomain

– `a_dir` normal direction of side to return. Directions are zero-based and must be $0 \leq$`a_dir` $<$`SpaceDim`.

– `a_len` Width of returned box in normal direction `a_dir`.

- ```
  friend
  Box adjCellLo(const ProblemDomain& a_pd, int a_dir, int a_len=1)
  ```

  Returns the cell-centered box (in direction a_dir) adjacent to the low side of the domainBox in the argument ProblemDomain. The neighbor of an empty ProblemDomain is an empty Box of cell-centered type. If dir is a periodic direction, will return an empty box.

  Arguments:

  – `a_pd` input ProblemDomain

  – `a_dir` normal direction of side to return. Directions are zero-based and must be $0 \leq$`a_dir` $<$`SpaceDim`.

  – `a_len` Width of returned box in normal direction `a_dir`.

- `Box operator& (const Box& b) const`

  Returns the Box that is the intersection of the input box `b` and the ProblemDomain. The Box `b` must be cell-centered. The intersection of an empty ProblemDomain and any box is the Empty Box. This operator does nothing in periodic directions (since a periodic domain is an infinite domain).

- `ProblemDomain& refine(int a_refinement_ratio)`

  Modifies this ProblemDomain by refining it by (the positive) `a_refinement_ratio`. The empty ProblemDomain is not modified by this function.

- ```
  friend
  ProblemDomain refine(const ProblemDomain& a_probdomain,
                       int a_refinement_ratio)
  ```

  Returns a ProblemDomain that is the argument ProblemDomain refined by (the positive) `a_refinement_ratio`. If `a_probdomain` is an Empty ProblemDomain, then an empty problem domain is produced.

  `ProblemDomain& refine(const IntVect& a_refinement_ratio)`

  Modifies this ProblemDomain by refining it by the given refinement ratio in each direction. The empty ProblemDomain is not modified by this function.

```
friend
ProblemDomain refine (const ProblemDomain&     a_probdomain,
                      const IntVect& a_refinement_ratio)
```

Returns a ProblemDomain that is the argument ProblemDomain refined by (the positive) a_refinement_ratio. If a_probdomain is an Empty ProblemDomain, then an empty problem domain is produced.

- `ProblemDomain& coarsen(int a_refinement_ratio)`

  Modifies this ProblemDomain by coarsening it by (the positive) a_refinement_ratio. The empty ProblemDomain is not modified by this function.

- ```
  friend
  ProblemDomain coarsen(const ProblemDomain& a_probdomain,
                        int a_refinement_ratio)
  ```

  Returns a ProblemDomain that is the argument ProblemDomain coarsened by (the positive) a_refinement_ratio. If a_probdomain is an Empty ProblemDomain, then an empty problem domain is produced.

```
ProblemDomain& coarsen(const IntVect& a_refinement_ratio)
```

Modifies this ProblemDomain by coarsening it by the given refinement ratio in each direction. The empty ProblemDomain is not modified by this function.

```
friend
ProblemDomain refine (const ProblemDomain&     a_probdomain,
                      const IntVect& a_refinement_ratio)
```

Returns a ProblemDomain that is the argument ProblemDomain coarsened by (the positive) a_refinement_ratio. If a_probdomain is an Empty ProblemDomain, then an empty problem domain is produced.

```
friend
std::ostream& operator<< (std::ostream& os, const ProblemDomain& b)
```

Writes and ASCII representation to the ostream.

```
friend
std::istream& operator<< (std::istream& is, ProblemDomain& b)
```

read from istream.
```

2.4 Data on Unions of Rectangles

This section describes our tools for doing calculations over unions of rectangles. These tools may be used either in serial or in parallel though the design reflects our parallel programming model. In this section, we explain the tools we use to describe unions of rectangles and the data which lives over these regions.

2.4.1 Introduction

We wish to represent data defined on unions of rectangles. Such data can be mapped naturally onto distributed memory by assigning boxes to processors, with data defined on those boxes stored on the processor to which the box is assigned. This approach has been used quite successfully. Berger and Bokhari [BB86], Kohn and Baden [KB96], Rendleman, et. al. [RBL$^+$99], and others have used this technique. Our API is derived from joint work with Baden to develop an abstract version of KeLP [FBK96]. It is implemented using the following three sets of classes:

• `BoxLayout, DisjointBoxLayout`—classes that represent unions of rectangles and the mapping of those rectangles to processors.

• `LayoutData, BoxLayoutData, LevelData`— templated classes for distributing data over processors.

• `LayoutIterator/LayoutIndex, DataIterator/DataIndex`— classes for iterating over and indexing into the classes above.

2.4.2 Layouts

The classes `BoxLayout` and `DisjointBoxLayout` represent unions of rectangles and the mapping of the rectangles onto processors. `BoxLayout` represents an arbitrary union of valid boxes. `DisjointBoxLayout` is a `BoxLayout` and has the additional property that none of the boxes intersect. Both types of layout have two states: open and closed. During construction, a layout is open. In its open state, a user can add boxes and modify the mapping of boxes to processor. When a user is finished changing a `BoxLayout` to her satisfaction, she invokes the `close()` function. After closing, the `BoxLayout` cannot be accessed in a non-const manner. There is no way to reopen a closed `BoxLayout`. The closed property propagates through assignment and copy construction. Only closed layouts may be used to build the distributed data classes.

2.4.2.1 BoxLayout

A `BoxLayout` is a collection of boxes. On parallel platforms, `BoxLayout` includes a mapping to processors. In both cases, the data holders `LayoutData`, `BoxLayoutData`, and `LevelData` define mappings from the `Boxes` in the `BoxLayout` to objects of the template type `T`. The important functions of `BoxLayout` are as follows:

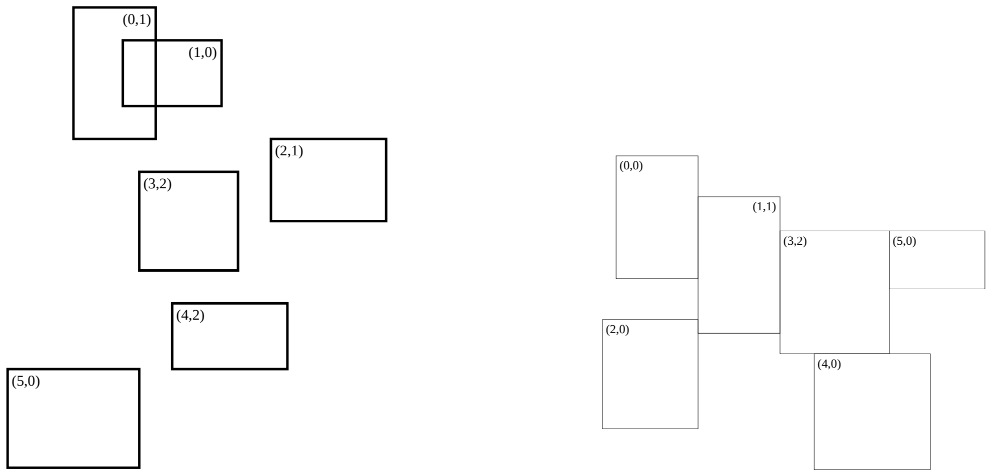

Figure 2.3: Left: Example of a `BoxLayout`. The first integer in the pair identifies the `Box`, and the second integer the processor ID. In this case we have the following assignments. Processor 0: B_1, B_5. Processor 1: B_0, B_2. Processor 2: B_3, B_4. Note that B_0 and B_1 have a non-empty intersection. Right: Example of a `disjoint BoxLayout`. The first integer in the pair identifies the `Box`, and the second integer the processor ID. In this case we have the following assignments. Processor 0: B_0, B_2, B_4, B_5. Processor 1: B_1. Processor 2: B_3. Note that a disjoint `BoxLayout` has empty intersections.

- <u>Construction</u>

```
BoxLayout(const Vector<Box>& boxes, const Vector<int>& procIDs)
void define(const Vector<Box>& boxes, const Vector<int>& procIDs)
virtual void deepCopy(const BoxLayout& source)
DataIndex addBox(const Box& box, int iProc)
virtual void close()
```

The constructor and define functions construct a `BoxLayout` from a vector of Boxes and a vector of processor assignments. The input `procIDs` must all be in the range `[0...numProcs()-1]` where the function `numProcs()`, located in `SPMD.H`, returns the number of processors being used in the calculation. `procIDs[i]` is the processor number of the processor on which the data that maps to the box `boxes[i]` is stored. The processor assignment Vector must be the same length as the `Vector<Box>` argument. On exit, the `BoxLayout` will be closed. One can either null construct the `BoxLayout` and call the define function or construct and define at once. If the user is not using MPI, the `procIDs` argument is ignored. The new object created with `deepCopy` disassociates itself with original implementation safely. This object now is considered 'open' and can be non-const modified. There is no assurance that the order in which this `BoxLayout` is indexed corresponds to the indexing of `source`. `addBox` incrementally adds a box and its processor assignment to an open layout (if the layout has been closed, calling this function generates a run-time error) and returns a `DataIndex` object. The `DataIndex` object is valid both before and after `close` is called. It can be used later to access this box again, or access the data object (`T`) in a `BoxLayoutData` that is built from this `BoxLayout` object. `close` marks this `BoxLayout` as complete and unchangeable. It is here that the layout gets sorted. This must be called before any data containers are constructed using the layout.

- Boolean functions.

```
bool operator==(const BoxLayout& rhs) const
bool check(const DataIndex& index) const
bool isClosed()
```

Equality for `BoxLayout` is a reference-counted pointer check. This returns true if these two objects share the same implementation. Important Warning: Two layouts can have the same boxes and same processor mapping and still return false if they were built separately. To force equality of two layouts, use the copy constructor. `check` returns true if the input `DataIndex` matches the layout. `isClosed` returns true if close() has been called.

- Accessors.

```
Box& operator[](const DataIndex& it)
DataIterator dataIterator() const
LayoutIterator layoutIterator() const
```

This allows access to an individual box through the iterator. One must be iterating through the correct layout (`check` must return true) in order for the accessor operator to work correctly. The member functions `dataIterator`, `layoutIterator` return the iterators associated with this layout.

- Coarsening and Refinement Operations.

```
friend void coarsen(BoxLayout& output, const BoxLayout& input,
                    int refinement)
friend void refine(BoxLayout& output, const BoxLayout& input,
                   int refinement)
```

The functions `coarsen`, `refine` coarsens or refines each box in the layout by the input refinement ratio. Iterator objects that worked for the input will work for the output.

2.4.2.2 DisjointBoxLayout

`DisjointBoxLayout` is-a `BoxLayout`. The difference between them is that, for `DisjointBoxLayout`, closed also implies that the boxes in a `DisjointBoxLayout` have no non-trivial intersection with one another in index space. Any attempt to close a `DisjointBoxLayout` object with boxes which have non-trivial intersection will result in a run-time error. Coarsening may not preserve disjointedness, and applying the `coarsen` operator to a DisjointBoxLayout will generate also a run-time error if the new coarsened boxes aren't disjoint. Otherwise, all of the functions of BoxLayout carry over to DisjointBoxLayout.

If the problem domain is periodic, disjointness is tied to the periodicity of the domain – a box in the BoxLayout may intersect the periodic image of another box. To account for this, DisjointBoxLayout can also be defined with a ProblemDomain. In the default case, the domain is defined to be non-periodic in all directions. If the domain is periodic, then the periodicity of the domain is taken into account when checking for disjointness of the boxes in the BoxLayout.

The important extra functions of DisjointBoxLayout are as follows:

- Constructors

```
DisjointBoxLayout(const Vector<Box>& boxes,
                  const Vector<int>& procIDs,
                  const ProblemDomain& probDomain)
void define(const Vector<Box>& boxes, const Vector<int>& procIDs,
            const ProblemDomain& probDomain)
void define(BoxLayout& a_layout, const ProblemDomain& a_physDomain);
virtual void deepCopy(const BoxLayout& a_source,
                      const ProblemDomain& a_physDomain)
```

These functions are the same as the corresponding functions in BoxLayout, but with the addition of a ProblemDomain argument.

- Checking functions

```
bool checkPeriodic(const ProblemDomain& probDomain)
```

The checkPeriodic function returns true if the argument ProblemDomain is consistent with the ProblemDomain used to define the DisjointBoxLayout. Two ProblemDomains are consistent if they are periodic in the same directions, and they have same domain size in any periodic directions. In non-periodic directions, no consistency is required.

2.4.3 Templated Data Holders

LayoutData<T>, BoxLayoutData<T>, and LevelData<T> are templated data holders over a BoxLayout that hold one T at each box in the layout. Each class represents a different level of functionality. LayoutData<T> is a holder for creating local data corresponding to the part of the BoxLayout assigned to that processor. In particular, there is no support in Chombo for communicating LayoutData<T> information between processors. LevelData<T> implements an abstract form of a cell-centered level array, represented as a collection of rectangular "arrays" (i.e., objects of type T), each of which is defined on an element of $\mathcal{R}(\Omega)$. These arrays are distributed over processors using the rule encoded in the DisjointBoxLayout used to construct them. Finally, a BoxLayoutData<T> is a generalization of a LevelData<T>, in that the underlying BoxLayout is allowed to have overlapping Boxes. Thus one can copy from a LevelData<T> to a LevelData<T> or BoxLayoutData<T>, but not from a BoxLayoutData<T>, since the latter is not guaranteed to be single-valued on each cell.

2.4.3.1 LayoutData

LayoutData is a templated data holder for a collection of Box-oriented objects. A LayoutData can be built upon either a BoxLayout or a DisjointBoxLayout. The arrangement of the T objects is given by the underlying BoxLayout object. Each box in the BoxLayout will have a corresponding T object in the LayoutData object. The T objects contained within a LayoutData object should be accessed through a DataIterator. Non-local access to a LayoutData (access to a T that lives on another processor) is an error. Data in a LayoutData *cannot* be communicated to other processors. The class T must provide null construction.
The important parts of the LayoutData<T> API are as follows:

- Construction.

```
LayoutData(const BoxLayout& dp);
void define(const BoxLayout& dp);
```

The constructor allocates a T object for every box in the BoxLayout dp using the T() (null) constructor. The function define performs the same task for a null-constructed LayoutData. The dp must be closed or a runtime error will occur.

- Accessors.

```
DataIterator iterator() const;
T& operator[](const DataIndex& index);
```

The input DataIndex for the indexing operator [] must match the BoxLayout which was used in construction of the LayoutData. It must also correspond to an element in the BoxLayout on myProc(). iterator returns an iterator which provides the DataIndex(es) which can be used to access the objects T which live at each box.

2.4.3.2 BoxLayoutData

Requirements on the template class T: BoxLayoutData<T> requires that T provides the following member functions, in addition to a null constructor for T:

- Constructors.

```
T(const Box& box, int comps)
define(const Box& box, int comps)
```

Allocate all the memory for data given a region and the number of components. The data does not necessarily need to be initialized.

• Copiers.

```
copy(const Box& regionFrom, const Interval& destInterval,
     const Box& regionTo, const T& source,
     const Interval& sourceInterval)
void linearOut(void* const buf, const Box& R, const Interval& comps)
const void linearIn(const void* const buf, const Box& R, const Interval& comps)
int size(const Box& R, const Interval& comps)
const static int preAllocatable()
```

`copy` copies the data from `source` over the `regionFrom` to the `regionTo` in the destination. The two regions must be the same size, and must be valid in the source and destination, respectively. The component range specified by `sourceInterval` is copied to the component range specified by `destInterval` and the size of these two `Intervals` must be the same. `linearIn/linearOut` copy the object from/to the stream of bytes `buf`. This stream is assumed to be allocated by the calling program. `size` returns the size of the linearized object in bytes. `preAllocateable` returns:

1. if the `size` function is strictly a function of Box and Interval, and does not depend on the current state of the T object.

2. if `size` is symmetric, in that sender and receiver T object can size their message buffers, but a static object cannot.

3. if the object is truly dynamic. the message `size` is subject to unique object data.

A `BoxLayoutData` can be built upon either a `BoxLayout` or a `DisjointBoxLayout`. `BoxLayoutData<T>` is-a `LayoutData<T>` which means that it has all of the member functions of `LayoutData<T>`. The important extra functions of `BoxLayoutData<T>` are:

• Constructors.

```
BoxLayoutData(const BoxLayout& boxes, int comps);
virtual void define(const BoxLayout& boxes, int comps);
virtual void define(const BoxLayoutData<T>& da);
virtual void define(const BoxLayoutData<T>& da,
                    const Interval& comps);
```

Defines the object from a layout and a number of components. Because of the semantics of inheritance, any `DisjointBoxLayout` can be used as an argument here instead of `BoxLayout`. The second `define` explicitly defines this `BoxLayoutData` from input. This includes copying the data values. The third `define` defines this `BoxLayoutData` to be on the same `BoxLayout` as the input `da` but only for the components defined by the `Interval comps`.

• Accessors.

```
int nComp() const;
Interval interval() const;
```

nComp returns the number of components in the data holder. interval returns the component range of the data holder (0:nComp()-1).

2.4.3.3 LevelData

A LevelData can be built only upon DisjointBoxLayouts. LevelData<T> has the same requirements on its T that BoxLayoutData<T> has. It also contains the important extra concepts of ghost values and data communication. Each box in the input layout is grown in each direction by the number of ghost cells in that direction. The data that lives on the input region (the part inside of the ghost cells) is considered "valid" data and the data on the ghost cells is considered "ghost" data. There are two data communication paradigms. One is the exchange function which copies data from the valid regions to the ghost regions where they intersect. The other function is copyTo which allows data communication between data holders. The source of a copyTo must be a LevelData<T>. The destination of copyTo may be either a LevelData<T> or a BoxLayoutData<T>. LevelData<T> is-a BoxLayoutData<T> (which is-a LayoutData<T>) which means that it has all of the member functions of BoxLayoutData<T> (and, by transitivity, all the member functions of LayoutData<T>). The important extra functions of LevelData<T> are as follows:

• Constructors.

```
LevelData(const DisjointBoxLayout& dp, int comps,
          const IntVect& ghost = IntVect::TheZeroVector());
virtual void define(const DisjointBoxLayout& dp, int comps,
                    const IntVect& ghost = IntVect::TheZeroVector());
virtual void define(const LevelData<T>& da);
virtual void define(const LevelData<T>& da, const Interval& comps);
```

The construction functions work in the same way as the construction functions for BoxLayoutData. The main difference is that for each Box B in the BoxLayout, the object of type T associated using the Box grown by ghost, i.e., T(grow(B,ghost),comps).

• Copiers.

```
virtual void copyTo(const Interval& srcComps,
                    BoxLayoutData<T>& dest,
                    const Interval& destComps) const;
virtual void copyTo(const Interval& srcComps,
                    LevelData<T>& dest,
```

```
                    const Interval& destComps) const;
virtual void exchange(const Interval& comps);
const IntVect& ghostVect();
```

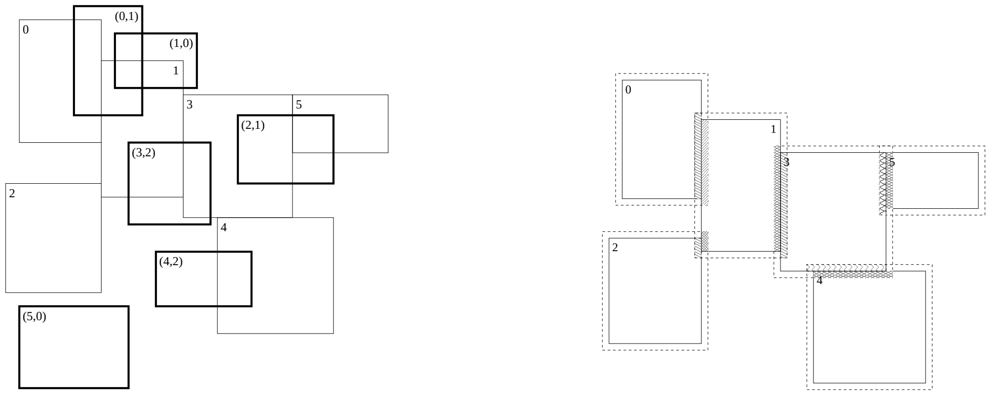

Figure 2.4: Left: CopyTo example. This figure illustrates copying from a LevelData built on the DisjointBoxLayout in figure 2.3 to a BoxLayoutData built on top of the BoxLayout in figure 2.3. A single call to Copy would perform the following data movements: Data from B_0 copied to B'_0. Data from B_1 copied to B'_0,B'_1,B'_3. Data from B_3 copied to B'_2,B'_3. Data from B_4 copied to B'_4. Data from B_5 copied to B'_2. No data is copied from B_2 or to B'_5. Right: exchange example. This figure illustrates copying data from the valid regions of a LevelData built on top of the DisjointBoxLayout in figure 2.3 to ghost cell regions of the same LevelData. The dashed Boxes indicate which ghost cell regions will be filled by a single call to exchange.

The first copyTo copies all of the data in the valid regions of this object to dest where the two BoxLayouts intersect. The length of the input and output intervals must be the same. The second version of copyTo copies to the LevelData dest filling the ghost cells of 'dest' with data from 'this' also (figure 2.4). The exchange function copies data from the valid regions to the ghost regions where they intersect (figure 2.4). If the DisjointBoxLayout used to define this LevelData is periodic in any direction, both copyTo and exchange will also fill cells from valid regions of the appropriate periodic images as necessary. ghostvect returns the IntVect defining the size of the ghost region.

2.4.3.4 Aliasing

For template classes that support an *aliasing* constructor, eg:

```
BaseFab<int> original(b, 4);
Interval     subcomps(2, 3);
BaseFab<int> alias;
alias.define(subcomps, original);
```

then a user can alias an entire LevelData at once using the function

```
template <class T>
void aliasLevelData(LevelData<T>& a_alias,
                    LevelData<T>* a_original,
const Interval& a_interval);
```

See section 2.2.6.1 for semantics of aliasing.

2.4.4 Iterators

There are two iterators over multiple-box objects in Chombo, `LayoutIterator` and `DataIterator`. They each return objects (`LayoutIndex`, `DataIndex`) that can be used to index into layouts and data holders. A layout may be indexed into by either `LayoutIndex` or a `DataIndex`, while a data holder may only be indexed into using a `DataIndex`. In serial the iteration patterns of the two types of iterator are exactly the same. The iterators iterate through every box in the layout. In parallel, the `LayoutIterator` still iterates through every box in the layout but the `DataIterator` iterates through only boxes whose data resides upon the current processor.
Principal Operations on `DataIterator`, `LayoutIterator`. In the following, BL is a `BoxLayout`, DBL is a `DisjointBoxLayout`, and `iter` is either a `DataIterator` or a `LayoutIterator`.

- <u>Construction.</u> The iterators can be constructed with the object to be iterated over (`iter`(BL), `iter`(DBL)), or null-constructed and defined later (`iter.define`(BL), `iter.define`(DBL)).
- <u>Iteration</u>. `iter.begin`() sets `iter` to the beginning of the iteration sequence, `++iter` advances `iter` to the next iterate, and `iter.ok`() checks to see if the current iterate is valid. `iter()` returns the current value of the iterate which is a `DataIndex` if `iter()` is a `DataIterator` or a `LayoutIndex` if `iter()` is a `LayoutIterator`.

We give examples of the use of `LayoutIterator` and `DataIterator`. In the first example, we iterate over all the `Boxes` in the layout to determine whether they cover the `Box` B.

```
Box B;
IntVectSet ivs(B);
BoxLayout bl;
...
LayoutIterator liter(bl);

for (liter.begin();liter.ok();++liter)
{
```

```
    ivs -= bl[liter];
}

if (ivs.isEmpty())
{
...
}
```

In the second example, we set the values of all the components in all the FArrayBoxes in a BoxLayoutData<FArrayBox> to zero.

```
BoxLayout bl;
...
BoxLayoutData<FArrayBox> bld(bl,1);
DataIterator diter(bl);
for (diter.begin();diter.ok();++diter)
{
    bld[diter].setVal(0.0);
}
```

Chapter 3

AMRTools

3.1 Multilevel Operators.

In this section, we describe algorithmic and library support suitable for implementing extensions of second-order accurate discretizations of quasi-linear elliptic, parabolic, and hyperbolic PDE's in conservation form to AMR. Our approach will be to express the AMR discretizations in terms of the corresponding uniform grid discretizations at each level, using appropriate interpolation operators to provide ghost cell values for points in the stencil extending outside of the grids at that level. We will also define a conservative discretization of the divergence operator on multilevel data.

From a formal numerical analysis standpoint, a solution on an adaptive mesh hierarchy $\{\Omega^l\}_{l=0}^{l_{max}}$ approximates the exact solution to the PDE only on those cells that are not covered by a grid at a finer level. We define the valid region of Ω^l

$$\Omega^l_{valid} = \Omega^l - \mathcal{C}_{n^l_{ref}}(\Omega^{l+1})$$

A composite array φ^{comp} is a collection of discrete values defined on the valid regions at each of the levels of refinement.

$$\varphi^{comp} = \{\varphi^{l,valid}\}_{l=0}^{l_{max}} \ , \ \varphi^{l,valid} : \Omega^l_{valid} \rightarrow \mathbb{R}^m$$

We can also define valid regions and composite arrays for other centerings. $\Omega^{l,\boldsymbol{e}^d}_{valid} = \Omega^{l,\boldsymbol{e}^d} - \mathcal{C}_{n^l_{ref}}(\Omega^{l+1,\boldsymbol{e}^d})$. Thus $\Omega^{l,\boldsymbol{e}^d}_{valid}$ consists of d-faces that are not covered by the d-faces at the next finer level. A composite vector field $\vec{F}^{comp} = \{\vec{F}^{l,valid}\}_{l=0}^{l_{max}}$ is defined as follows.

$$\vec{F}^{l,valid} = (F_0^{l,valid} \dots F_{\mathbf{D}-1}^{l,valid})$$
$$F_d^{l,valid} : \Omega^{l,\boldsymbol{e}^d}_{valid} \rightarrow \mathbb{R}^m$$

Thus a composite vector field has values at level l on all of the faces that are not covered by faces at the next finer level.

We want to compute finite difference approximations to differential operators. For example, let L be a finite difference approximation to a linear differential operator $\mathcal{L}$. On a uniform grid, L typically takes the form

$$(L\varphi)_{\boldsymbol{i}} = \sum_{|\boldsymbol{s}|\leq S} c_{\boldsymbol{i},\boldsymbol{s}} \varphi_{\boldsymbol{i}+\boldsymbol{s}} \tag{3.1}$$

Starting from this operator, we can extend L to be defined on an AMR grid hierarchy in the following fashion. For each $\Omega^l_k \in \mathcal{R}(\Omega^l)$

$$\begin{aligned} \varphi^{l,ext}_{\boldsymbol{i}} &= \varphi^l_{\boldsymbol{i}} \text{ on } \Omega^l_{valid} \\ &= I(\varphi^{comp}, \boldsymbol{x}^l_0 + \boldsymbol{i}h^l) \text{ on } \mathcal{G}(\Omega^l_k, S) \cap \Gamma^l - \Omega^l_{valid} \end{aligned}$$

$$(L\varphi)_{\boldsymbol{i}} = \sum_{|\boldsymbol{s}|\leq S} c_{\boldsymbol{i},\boldsymbol{s}} \varphi^{l,ext}_{\boldsymbol{i}+\boldsymbol{s}} \text{ on } \Omega^l_k$$

Here $I = I(\varphi^{comp}, \boldsymbol{x})$ is an interpolation operator that takes some combination of the valid composite data and constructs an interpolant at the point $\boldsymbol{x} \in \mathbb{R}^{\mathbf{D}}$.

Let ψ be a smooth function on $\mathbb{R}^{\mathbf{D}}$, and define the level array

$$\psi^l = \psi(\boldsymbol{x}^l_0 + \boldsymbol{i}h^l) \text{ on } \Gamma^l$$

and composite array

$$\begin{aligned} \psi^{comp} &= \{\psi^{l,valid}\}^{l_{max}}_{l=0} \\ \psi^{l,valid} &= \psi^l \text{ on } \Omega^l_{valid} \end{aligned}$$

Then the truncation error of the operator L can be computed as follows. For $\boldsymbol{i} \in \Omega^l_{valid}$

$$\begin{aligned} \tau_{\boldsymbol{i}} &\equiv L^{comp}(\psi^{comp})_{\boldsymbol{i}} - \mathcal{L}(\psi)(\boldsymbol{x}^l_0 + \boldsymbol{i}h^l) \\ &= \sum_{|\boldsymbol{s}|\leq \mathcal{S}} c_{\boldsymbol{i},\boldsymbol{s}} \psi_{\boldsymbol{i}+\boldsymbol{s}} - \mathcal{L}(\psi)(\boldsymbol{x}^l_0 + \boldsymbol{i}h^l) \\ &- \sum_{\substack{|\boldsymbol{s}|\leq \mathcal{S} \\ \boldsymbol{i}+\boldsymbol{s}\notin\Omega^l_{valid}}} c_{\boldsymbol{i},\boldsymbol{s}} (\psi_{\boldsymbol{i}+\boldsymbol{s}} - I(\psi^{comp}, \boldsymbol{x}^l_0 + (\boldsymbol{i}+\boldsymbol{s})h^l)) \end{aligned}$$

The first sum is the truncation error on a uniform grid, while the second sum gives the effect of replacing the uniform grid values of the smooth function ψ by those obtained by interpolation.

Unfortunately, this process, when used by itself, becomes unwieldy for any but the simplest finite difference approximations. Typically, in order to obtain $\tau = O(h^q)$ it is necessary to compute $I(\varphi^{comp}, \boldsymbol{x}^l_0 + \boldsymbol{i}h)$ to an accuracy of $O(h^{p+q})$, where p is order of the highest derivative of the operator, due to the contributions of the second summand

$(\max_{|\boldsymbol{s}|\leq\mathcal{S}}(|c_{\boldsymbol{i},\boldsymbol{s}}|) = O(h^{-p}))$. To obtain interpolants of such accuracy, we must either use general polynomial interpolants using data located on multiple levels of refinement, or impose minimum distance requirements between grids at different levels of refinement. The alternative is to accept a larger truncation error near the boundary between levels of refinement. In the AMR algorithms that motivate the design of Chombo, we use a combination of all three techniques. This approach is motivated by the following mathematical and algorithmic considerations.

• Our target applications involve solving first - and second - order quasi-linear systems of PDE's of classical type, i.e., elliptic, parabolic, and hyperbolic.

• Our underlying uniform-grid discretizations are based on second-order accurate methods, mainly in discrete conservation form.

The latter property is one that we would like to preserve in the AMR versions of these algorithms. However, the requirement for discrete conservation form leads to a loss of accuracy at coarse-fine boundaries. Finite difference methods rely on the cancellation of truncation error terms in the differenced quantities in order to obtain a given accuracy on a uniform grid. This is the mechanism, for example, by which the second divided difference approximates the second derivative to $O(h^2)$, even though it is a divided difference of quantities that are themselves accurate only to $O(h^2)$. This mechanism fails at the interface between different levels of refinement. If one is to approximate the divergence operator with a divided difference of single-valued fluxes, it is not possible to compute the flux so that the truncation error cancels that of the fluxes on both the adjacent coarse and fine faces.

Fortunately, our choice of target applications makes this local loss of accuracy acceptable. For elliptic and parabolic problems, a truncation error of $O(h^{p-1})$ on a set of codimension one induces a solution error of $O(h^p)$, due to a discrete form of elliptic regularity. In hyperbolic problems, a truncation error of $O(h^{p-1})$ on a set of codimension one induces a total error of $O(h^p)$ in L^1 (and in L^∞ as well if the boundary is non-characteristic).

In the following, we give the details of the algorithms for interpolating between levels that arise in this approach. They include averaging and interpolation methods for transferring information between levels; specialized operators for interpolating boundary conditions at boundaries between levels; and a conservative multilevel discretization of the divergence operator. For all of these cases, we will describe the algorithms for the case of data defined on two successive levels $\Omega^f, \Omega^c_{valid}$. The resulting operators can all be extended to the full AMR hierarchy by applying them to a pair of levels at a time, provided that appropriate nesting conditions are met. For the most part, only proper nesting is required. When that is not the case, we will explicitly state the nesting conditions required on grids at successive levels.

3.1.1 Interlevel Transfer Operators

3.1.1.1 Conservative Averaging.

This operator is used to average from finer levels on to coarser levels, or for constructing averaged residuals in multigrid iterations.

$$Average(\varphi, n_{ref})_{\boldsymbol{i}^c} = \frac{1}{(n_{ref})^d} \sum_{\boldsymbol{i} \in \mathcal{C}^{-1}_{n_{ref}}(\boldsymbol{i}^c)} \varphi_{\boldsymbol{i}}$$

This process produces values on the coarse grid that are an $O(h^2)$ estimate of the solution on the fine grid.

3.1.1.2 Piecewise Constant Interpolation.

This operator is primarily used in multigrid iteration to interpolate the correction from the coarser level to the next finer level.

$$I_{pwc}(\varphi)_{\boldsymbol{i}^f} = \varphi_{\boldsymbol{i}}$$

where $\boldsymbol{i} = \mathcal{C}_{n_{ref}}(\boldsymbol{i}^f)$. This method has an interpolation error of $O(h)$.

3.1.1.3 Piecewise Linear Interpolation.

This method is primarily used to initialize fine grid data after regridding. Given a level array φ^c on Ω^c, we want to compute $I_{pwl}(\varphi)$ defined on an Ω^f properly nested in Ω^c.

$$I_{pwl}(\varphi)_{\boldsymbol{i}^f} = \varphi_{\boldsymbol{i}} + \sum_{d=0}^{\mathbf{D}-1} (\frac{(i^f_d + \frac{1}{2})}{n_{ref}} - (i_d + \frac{1}{2}))(\delta^d \varphi)_{\boldsymbol{i}}$$

$$(\delta^d \varphi)_{\boldsymbol{i}} = \begin{cases} \eta_{\boldsymbol{i}} (\delta^d_c \varphi)_{\boldsymbol{i}} & \text{if both } \boldsymbol{i} \pm \boldsymbol{e}^d \in \Gamma^c \\ \varphi_{\boldsymbol{i}+\boldsymbol{e}^d} - \varphi_{\boldsymbol{i}} & \text{if } \boldsymbol{i} - \boldsymbol{e}^d \notin \Gamma^c \\ \varphi_{\boldsymbol{i}} - \varphi_{\boldsymbol{i}-\boldsymbol{e}^d} & \text{if } \boldsymbol{i} + \boldsymbol{e}^d \notin \Gamma^c \end{cases}$$

$$\eta_{\boldsymbol{i}} = \chi(min(\varphi^{max}_{\boldsymbol{i}} - \varphi_{\boldsymbol{i}}, \varphi_{\boldsymbol{i}} - \varphi^{min}_{\boldsymbol{i}}), \sum_{d=0}^{\mathbf{D}-1} |\delta^d_c \varphi|_{\boldsymbol{i}})$$

$$(\delta^d_c \varphi)_{\boldsymbol{i}} = \begin{cases} \frac{1}{2}(\varphi_{\boldsymbol{i}+\boldsymbol{e}^d} - \varphi_{\boldsymbol{i}-\boldsymbol{e}^d}) & \text{if both } \boldsymbol{i} \pm \boldsymbol{e}^d \in \Gamma^c \\ \varphi_{\boldsymbol{i}+\boldsymbol{e}^d} - \varphi_{\boldsymbol{i}} & \text{if } \boldsymbol{i} - \boldsymbol{e}^d \notin \Gamma^c \\ \varphi_{\boldsymbol{i}} - \varphi_{\boldsymbol{i}-\boldsymbol{e}^d} & \text{if } \boldsymbol{i} + \boldsymbol{e}^d \notin \Gamma^c \end{cases}$$

$$\chi(a,b) = \begin{cases} \frac{a}{b} & \text{if } a < b \\ 1 & \text{oththerwise} \end{cases}$$

$$\varphi_{\boldsymbol{i}}^{max} = \underset{|\boldsymbol{s}|\leq 1}{max}(\varphi_{\boldsymbol{i}+\boldsymbol{s}}) \ , \ \varphi_{\boldsymbol{i}}^{min} = \underset{|\boldsymbol{s}|\leq 1}{min}(\varphi_{\boldsymbol{i}+\boldsymbol{s}})$$

At cells adjacent to the boundary of the computational domain Γ^c, the maximum and minimum are taken over the points $\boldsymbol{i}+\boldsymbol{s}$ that are contained in the computational domain. Also note, the arguments to χ are always non-negative.

We use the limiter η to keep the interpolated values from exceeding local estimates of the maximum and minimum values of the solution on the coarser grid. As long as $\eta = 1$, i.e., the limiter does not reduce the values of the slopes, the error in the interpolated values is $O(h^2)$.

3.1.2 Coarse-Fine Boundary Interpolation

3.1.2.1 Piecewise Linear Interpolation

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

$$\varphi^f : \Omega^f \rightarrow \mathbb{R}$$
$$\varphi^{c,valid} : \Omega^c_{valid} \rightarrow \mathbb{R}$$

We want to compute an extension $\tilde{\varphi}^f$ of φ^f on $\tilde{\Omega}^f = \mathcal{G}(\Omega^f, p) \cap \Gamma^f, p > 0$, which is accurate to $O(h^2)$, assuming only that $\mathcal{C}_r(\tilde{\Omega}^f) \cap \Gamma^c \subset \Omega^c$. There must be enough points on the coarse level to interpolate out to a distance of p fine cells from Ω^f. One way to achieve this goal is by choosing an appropriate blocking factor, i.e., we assume that Ω^f is $n_{ref}(\lfloor \frac{p}{n_{ref}} \rfloor + min(1, p \ mod \ n_{ref}))$ - blocked. Combined with proper nesting, this ensures that there are sufficient cells in Ω^c to perform the interpolation.
We perform this calculation in these steps

(i) Extend $\varphi^{c,valid}$ to φ^c, defined on all of $\Omega^c : \varphi^c = Av(\varphi^f, n_{ref})$ on $\mathcal{C}_{n_{ref}}(\Omega^f)$

(ii) For each $\boldsymbol{i}^f \in \tilde{\Omega}^f - \Omega^f$, compute a piecewise linear interpolant. For $\boldsymbol{i} = \mathcal{C}_{n_{ref}}(\boldsymbol{i}^f)$,

$$\tilde{\varphi}^f_{\boldsymbol{i}^f} = \varphi_{\boldsymbol{i}} + \sum_{d=0}^{\mathbf{D}-1} (\frac{(i^f_d + \frac{1}{2})}{n_{ref}} - (i_d + \frac{1}{2}))(\delta^d \varphi)_{\boldsymbol{i}} \equiv I^B_{pwl}(\varphi^f, \varphi^{c,valid})_{\boldsymbol{i}^f}$$

Unlike in the interlevel transfer operator I_{pwl}, we use a minimal stencil for $(\delta^d\varphi)_{\boldsymbol{i}}$ (Figure 3.1).

$$(\delta^d \varphi)_{\boldsymbol{i}} = \begin{cases} \delta_{vL}(\varphi_{\boldsymbol{i}-\boldsymbol{e}^d}, \varphi_{\boldsymbol{i}}, \varphi_{\boldsymbol{i}+\boldsymbol{e}^d}) & \text{if both } \boldsymbol{i} \pm \boldsymbol{e}^d \in \Omega^c \\ \varphi_{\boldsymbol{i}+\boldsymbol{e}^d} - \varphi_{\boldsymbol{i}} & \text{if } \boldsymbol{i} - \boldsymbol{e}^d \notin \Omega^c \\ \varphi_{\boldsymbol{i}} - \varphi_{\boldsymbol{i}-\boldsymbol{e}^d} & \text{if } \boldsymbol{i} + \boldsymbol{e}^d \notin \Omega^c \end{cases}$$

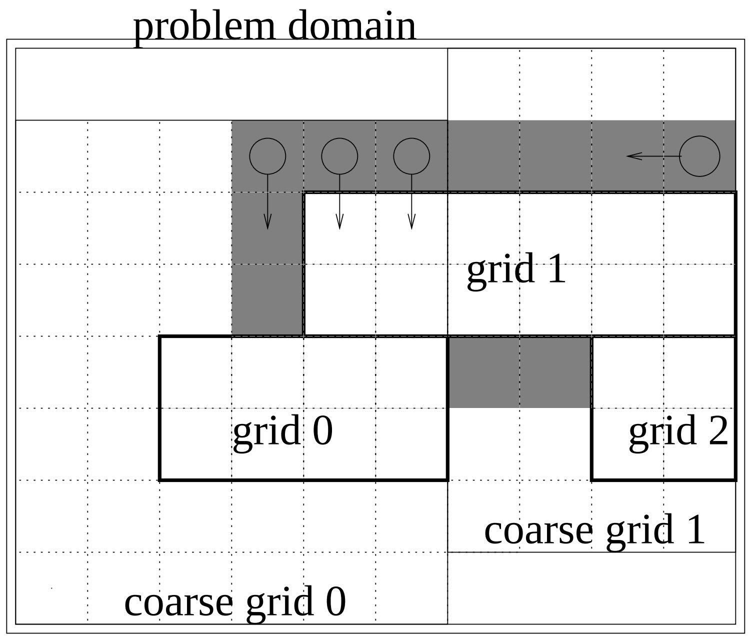

Figure 3.1: Interpolation on the coarse grid. One-sided differences are used at cells marked with circles.

$$\delta_{vL}(a_l, a_c, a_r) = \begin{cases} min(\frac{1}{2}|a_l - a_r|, |a_l - a_c|, |a_r - a_c|)sign(a_r - a_l) & \text{if } (a_l - a_c)(a_c - a_r) > 0 \\ 0 & \text{otherwise} \end{cases}$$

(iii) $\tilde{\varphi}^f = \varphi^f$ on Ω^f

The truncation error of this interpolation operator is $O(h^2)$, i.e., if $\psi = \psi(\boldsymbol{x})$ is a smooth function, and

$$\psi_{\boldsymbol{i}}^f = \psi(\boldsymbol{x}_0^f + \boldsymbol{i}h^f) \text{ on } \Omega^f$$
$$\psi_{\boldsymbol{i}}^{c,valid} = \psi(\boldsymbol{x}_0^c + \boldsymbol{i}h^c) \text{ on } \Omega^{c,valid}$$

then

$$\tilde{\psi}_{\boldsymbol{i}^f}^f = \psi(\boldsymbol{x}_0^f + \boldsymbol{i}h^f) + O(h^2) \text{ for } \boldsymbol{i} \in \tilde{\Omega}^f - \Omega^f$$

where $\tilde{\psi}^f$ is the extension of (ψ^f, ψ^c) computed using the process outlined above. The key point is that, as long as the extension of ψ^c to $\mathcal{C}_{n_{ref}}(\Omega^f)$ is accurate to $O(h^2)$, the undivided difference formula approximates $h^c \frac{\partial \psi}{\partial x^d}$ to $O(h)$, and differs from the Taylor expansion of ψ around $(\boldsymbol{x}^c + \boldsymbol{i}h^c)$ by $O(h^2)$.

3.1.2.2 Quadratic Coarse-Fine Boundary Interpolation

This interpolation scheme is motivated by the requirements of constructing consistent discretizations of second-order operators. Given φ^f, $\varphi^{c,valid}$, we want to compute a level vector field $\vec{G}^f = (G_0^f, \ldots G_{\mathbf{D}-1}^f)$ that approximates the gradient to sufficient accuracy so that, when we take its divergence, we obtain at least an $O(h)$ approximation to the Laplacian. For each $\Omega^{f,k} \in \mathcal{R}(\Omega^f)$, we construct an extension $\tilde{\varphi}$ of φ^f.

$$\tilde{\varphi} : \tilde{\Omega}_k^f \rightarrow \mathbb{R}^m$$
$$\tilde{\Omega}_k^f = (\bigcup_{\pm=+,-} \bigcup_{d=0}^{\mathbf{D}-1} \Omega_k^f \pm \boldsymbol{e}^d) \cap \Gamma^f$$

Then, for each $\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d$ such that both $\boldsymbol{i}, \boldsymbol{i}+\boldsymbol{e}^d \in \tilde{\Omega}_k^f$ we can compute a centered difference approximation to the gradient on a staggered grid

$$G^f_{d,\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = \frac{1}{h^f}(\tilde{\varphi}_{\boldsymbol{i}+\boldsymbol{e}^d} - \tilde{\varphi}_{\boldsymbol{i}})$$

For this estimate of the gradient to be accurate to $O(h^2)$, it is necessary to compute an $O(h^3)$ extension of φ^f. On $\tilde{\Omega}_k^f \cap \Omega^f$, the values for $\tilde{\varphi}$ will be given by $\tilde{\varphi}_{\boldsymbol{i}} = \varphi^f_{\boldsymbol{i}}$. The values for the remaining points in $\tilde{\Omega}_k^f - \Omega^f$ will be obtained by interpolating data from φ^f and φ^c.

To perform this interpolation, we first observe that, given $\boldsymbol{i} \in \tilde{\Omega}_k^f - \Omega^f$, there is a unique choice of $\pm$ and d, such that $\boldsymbol{i} \mp \boldsymbol{e}^d \in \Omega_k^f$. Having specified that choice, the interpolant is constructed in two steps (figure 3.2).

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

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

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

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

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

(ii) Interpolation in the normal direction.

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

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

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

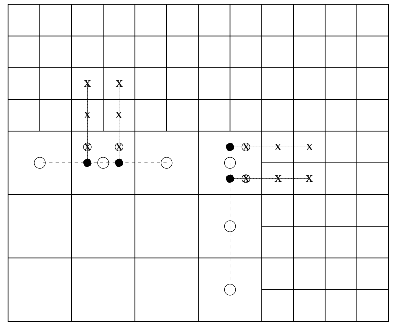

Figure 3.2: Interpolation at a coarse-fine interface. Left stencil is the usual stencil. Right stencil is the modified interpolation stencil; since the upper coarse cell is covered by a fine grid, use shifted coarse grid stencil (open circles) to get intermediate values (solid circles), then perform final interpolation as before to get "ghost cell" values (circled X's). Note that to perform interpolation for the horizontal coarse-fine interface, we need to shift the coarse stencil left.

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

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

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

In the case of $D^{d'd''}\varphi^c$, we use an average of all of the four-point difference approximations $\frac{\partial^2}{\partial x_{d'}\partial x_{d''}}$ centered at d', d'' corners adjacent to $\boldsymbol{i}$ such that all four points in the stencil are in Ω^c_{valid} (Figure 3.3)

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

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

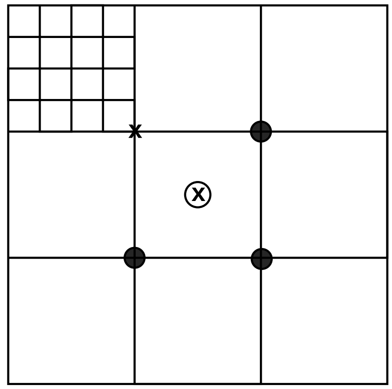

Figure 3.3: Mixed-derivative approximation illustration. The upper-left corner is covered by a finer level so the mixed derivative in the upper left (the uncircled x) has a stencil which extends into the finer level. We therefore average the mixed derivatives centered on the other corners (the filled circles) to approximate the mixed derivatives for coarse-fine interpolation in three dimensions.

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

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

3.1.2.3 Level Divergence, Composite Divergence, and Refluxing

Let $\vec{F}$ be a level vector field on Ω. We define a discretized divergence operator as follows.

$$(D\vec{F})_{\boldsymbol{i}} = \frac{1}{h}\sum_{d=0}^{\mathbf{D}-1}(F_{d,\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} - F_{d,\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^d}), \boldsymbol{i} \in \Omega \tag{3.2}$$

Let $\vec{F}^{comp} = \{\vec{F}^f, \vec{F}^{c,valid}\}$ be a two-level composite vector field. We want to define a composite divergence $D^{comp}(\vec{F}^f, \vec{F}^{c,valid})_{\boldsymbol{i}}$ for $\boldsymbol{i} \in \Omega^c_{valid}$. To do this, we construct an extension of $\vec{F}^{c,valid}$ to the edges adjacent to Ω^c_{valid} that are covered by fine level faces. On the valid coarse-level d-faces, $\hat{F}_{d,\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = F^{c,valid}_{d,\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d}$. On the faces adjacent to cells in Ω^c_{valid}, but not in $\Omega^{l,\boldsymbol{e}^d}_{valid}$, we set $\hat{F}_d$ to be $< F^f_d >$, the average of F^f_d onto the next coarser level.

$$< F^f_d >_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = \frac{1}{(n_{ref})^{\mathbf{D}-1}} \sum_{\boldsymbol{i}^f+\frac{1}{2}\boldsymbol{e}^d \in \mathcal{F}^d} F^f_{d,\boldsymbol{i}^f+\frac{1}{2}\boldsymbol{e}^d}$$

$$\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d \in \zeta^f_{d,+} \cup \zeta^f_{d,-}$$

Here the sum is over the set of all fine level d-faces that are covered by $[\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d]$, which is given as a rectangle in $\Gamma^{f,\boldsymbol{e}^d}$.

$$\mathcal{F}^d = [\boldsymbol{i}\ n_{ref} + \frac{1}{2}\boldsymbol{e}^d, (\boldsymbol{i} + (\boldsymbol{u} - \boldsymbol{e}^d))n_{ref} + \frac{1}{2}\boldsymbol{e}^d]$$

$\zeta^f_{d,\pm}$ consists of all the d-faces in Ω^c on the boundary of Ω^{l+1}, with valid cells on the low $(\pm = -)$ or high $(\pm = +)$ side.

$$\zeta^f_{d,\pm} = \{\boldsymbol{i} \pm \frac{1}{2}\boldsymbol{e}^d : \boldsymbol{i} \pm \boldsymbol{e}^d \in \Omega^c_{valid}, \boldsymbol{i} \in \mathcal{C}_{n_{ref}}(\Omega^f)\}$$

For both performance reasons and algorithmic reasons, it is useful to express D^{comp} as a succession of applications of the level divergence operator D applied to extensions of $\vec{F}^{l,valid}$ to the entire level, followed by a step that corrects the cells in Ω^c_{valid} that are adjacent to Ω^f. We define a *flux register* $\delta\vec{F}^f$ associated with the fine level

$$\delta\vec{F}^f = (\delta F^f_0, ..., \delta F^f_{\mathbf{D}-1})$$
$$\delta F^f_d : \zeta^f_{d,+} \cup \zeta^f_{d,-} \to \mathbb{R}^m$$

Let $\vec{F}^c$ be *any* coarse level vector field that extends $\vec{F}^{c,valid}$, i.e.

$$F^c_d = F^{c,valid}_d \text{ on } \Omega^{c,\boldsymbol{e}^d}_{valid}$$

Then for $\boldsymbol{i} \in \Omega^c_{valid}$,

$$D^{comp}(\vec{F}^f, \vec{F}^{c,valid})_{\boldsymbol{i}} = (D\vec{F}^c)_{\boldsymbol{i}} + D_R(\delta\vec{F}^c)_{\boldsymbol{i}} \tag{3.3}$$

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

$$\delta F^f_d =< F^f_d > - F^c_d \text{ on } \zeta^c_{d,+} \cup \zeta^c_{d,-}$$

D_R is the reflux divergence operator, given by the following for valid coarse level cells adjacent to Ω^f.

$$D_R(\delta\vec{F}^f)_{\boldsymbol{i}} = \frac{1}{h^c} \sum_{d=0}^{\mathbf{D}-1} \sum_{\substack{\pm=+,-: \\ \boldsymbol{i}\pm\frac{1}{2}\boldsymbol{e}^d \in \zeta^f_{d,\mp}}} \pm\delta F^f_{d,\boldsymbol{i}\pm\frac{1}{2}\boldsymbol{e}^d}$$

For the remaining cells in Ω^c_{valid}, $D_R(\delta\vec{F}^f)$ is defined to be identically zero.

Let $\vec{H} = (H_0 \dots H_{\mathbf{D}-1})$, $H_d : \mathbb{R}^{\mathbf{D}} \to \mathbb{R}^m$ be a smooth vector field, and define discrete level and composite vector fields by evaluating $\vec{H}$ on the grid.

$$\vec{H}^f = (H_0^f \dots H_{\mathbf{D}-1}^f) \ , \ H^f_{d,\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = H_d(\boldsymbol{x}_0^f + (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h^f)$$
$$\vec{H}^c = (H_0^c \dots H_{\mathbf{D}-1}^c) \ , \ H^c_{d,\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = H_d(\boldsymbol{x}_0^c + (\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d)h^c)$$

We can then compute the truncation error of the composite divergence using (3.3).

$$\begin{aligned} D^{comp}(\vec{H}^f, \vec{H}^{c,valid}) &= D(\vec{H}^c) + D_R(\delta\vec{H}) \\ &= \nabla \cdot \vec{H} + O(h^2) + D_R(\delta\vec{H}) \end{aligned}$$

Here $H_d^{c,valid}$ is given by restricting H_d^c to $\Omega^{c,\boldsymbol{e}^d}_{valid}$, and we make use of the observation that the centered difference approximation to the divergence given by (3.2) is second order accurate. Away from the cells adjacent to Ω^f, the contribution from the reflux divergence is zero, and the truncation error is $O(h^2)$. To estimate the truncation error at cells adjacent to the coarse - fine interface, we note that

$$\delta H_d^f = < H_d^f > - H_d^c = O((h^c)^2)$$

So that $D_R(\delta\vec{H}) = O(h^c)$, i.e., we lose one order of accuracy due to the correction to the divergence that maintains conservation form.

Laplacian

Using the operators described above, we can now define a discretization of the Laplacian on an adaptive mesh hierarchy. Let φ^{comp} a composite array defined on an AMR grid hierarchy satisfying proper nesting. The Laplacian is defined as the divergence of the gradient:

$$(L^{comp}\varphi^{comp})_{\boldsymbol{i}} \equiv D^{comp}(\vec{G}^{l+1,valid}, \vec{G}^{l,valid})_{\boldsymbol{i}} \ , \ \boldsymbol{i} \in \Omega^l_{valid} \tag{3.4}$$

where $\vec{G}^{l,valid} = \vec{G}(\varphi^{l,valid}, \varphi^{l-1,valid})$ is computed using the algorithm in section 3.1.2.2. It is assumed here that the discrete gradients can be computed using the boundary conditions for the faces that lay on the boundary of the domain. It is not difficult to check that, if the grids are properly nested, that the stencil of $(L^{comp}\varphi^{comp})_{\boldsymbol{i}}$ is contained in the valid regions of the meshes at levels l, $l \pm 1$. Away from boundaries between levels, this discretization reduces to the standard $2\mathbf{D}+1$ point discretization of the Laplacian. On grid interiors, L^{comp} has a truncation error of $O(h^2)$ due to cancellation of error terms in the centered-difference stencil. At coarse-fine interfaces, this drops to $O(h)$ due to division of the $O(h^3)$ interpolant by h^2 and the loss of centered-difference cancellations. However, if the discrete equation $L^{comp}\varphi = \rho$ is solved using these operators, the resulting solution φ is second-order accurate, because this loss of accuracy occurs on a set of codimension one [JC98]. The dependencies of the Laplacian operators may again be expressed explicitly: if $L^{comp,l}(\varphi^{comp})$ is $L^{comp}(\varphi^{comp})$ restricted to Ω^l_{valid}, then $L^{comp,l}(\varphi^{comp}) = L^{comp,l}(\varphi^{l,valid}, \varphi^{l+1,valid}, \varphi^{l-1,valid})$.

3.2 C++ Classes for Two-Level Operators

In this section, we document the user interfaces for a set of C++ classes that implement the operators described above. Typically, the interface has two parts. The constructor and define function constructs the persistent data, such as interpolation coefficients and the `IntVectSets` defining the irregular regions where the operator must be applied. This can either be done by calling a defining constructor, or by calling a member function `define` with the same arguments on an object that has been constructed with a default constructor. Note that for classes where problem domain information is required for construction, there are generally two sets of constructors and define functions – one with a `Box` to represent the domain, the second with a `ProblemDomain`; if the functions with a `Box` are used, a non-periodic domain is assumed. The second part of the interface consists of the functions that actually apply the operator to the data. Once the operator has been defined, the user can apply it multiple times to different data sets. The operator must be redefined only when the grids change.

3.2.1 The Class `CoarseAverage`

This class sets data on a level equal to an average of the data on a finer level of refinement, using the averaging operator in section 3.1.1.

- ```
  void
  define(const DisjointBoxLayout& 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<FArrayBox>& a_coarse_data,
                  const LevelData<FArrayBox>& 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.

3.2.2 The Class FineInterp

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 3.1.1.

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

  void
  define(const DisjointBoxLayout& 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<FArrayBox>& a_fine_data,
               const LevelData<FArrayBox>& 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.

3.2.3 The Class PiecewiseLinearFillPatch

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 section 3.1.2.1. 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 DisjointBoxLayout& a_fine_domain,
         const DisjointBoxLayout& a_coarse_domain,
         int a_num_comps,
         const ProblemDomain& a_coarse_problem_domain,
         int a_ref_ratio,
         int a_interp_radius);

  void
  define(const DisjointBoxLayout& a_fine_domain,
         const DisjointBoxLayout& 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<FArrayBox>& a_fine_data,
             const LevelData<FArrayBox>& a_old_coarse_data,
             const LevelData<FArrayBox>& 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.

3.2.4 The Class QuadCFInterp

The class QuadCFInterp interpolates data onto the ghost cells on the faces of a LevelData<FArrayBox>, using the algorithm described in section 3.1.2.2. 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 QuadCFInterp is given as follows.

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

  void define(const DisjointBoxLayout& a_fineBoxes,
              const DisjointBoxLayout* 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<FArrayBox>& a_phif,
                        const LevelData<FArrayBox>& a_phic) const;
  ```

  Coarse-fine interpolation operator. Fills all the ghost cells on all the faces of the LevelData<FArrayBox> 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.

3.2.5 The Class LevelFluxRegister

LevelFluxRegister 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 section 3.1.2.3. Unlike the previous operators, LevelFluxRegister holds data, corresponding to the flux register $\delta\vec{F}^f$ defined in section 3.1.2.3. The class also manages the manipulation of that data.

The user interface for LevelFluxRegister is as follows.

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

  void define(const DisjointBoxLayout& a_dbl,
              const DisjointBoxLayout& 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(FArrayBox& 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<FArrayBox> 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(FArrayBox& 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<FArrayBox> 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<FArrayBox>& 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<FArrayBox> 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.

3.3 The Class BRMeshRefine

BRMeshRefine is an object which produces a hierarchy of block-structured grids which obey proper-nesting requirements. See Berger and Colella [BC89] for an explanation of proper nesting. BRMeshRefine follows the algorithm of Berger and Rigoutsos [BR91] to generate the grids from tagged points in discrete index space. There are two interfaces for BRMeshRefine grid generation: one takes tags at all levels in the hierarchy and one takes tags only at the coarsest level. If the BRMeshRefine object is defined with a ProblemDomain which is periodic in one or more directions, grids generated will be properly nested in the periodic directions.

The user interface for BRMeshRefine is as follows:

- BRMeshRefine();

  Default constructor – the object is defined in an unusable state until the user calls the define function.

- ```
  BRMeshRefine(
               const ProblemDomain&   a_baseDomain,
               const Vector<int>&     a_refRatios,
               const Real             a_fillRatio,
               const int              a_blockFactor,
               const int              a_bufferSize,
               const int              a_maxSize );

  BRMeshRefine(
               const Box&             a_baseDomain,
               const Vector<int>&     a_refRatios,
               const Real             a_fillRatio,
               const int              a_blockFactor,
               const int              a_bufferSize,
               const int              a_maxSize );
  ```

  Full constructor. Places the BRMeshRefine object in a usable state.

  Arguments:

  – a_baseDomain Problem domain at the coarsest (level 0) level. Output grids will be constrained to be within the computational domain on each level.
  ```

– a_refRatios Refinement ratios between the levels. a_refRatio[i] represents the refinement ratio between levels i and i+1. The vector indices must correspond to level number.

– a_fillRatio Overall grid efficiency to be generated. If this number is set low, the grids will tend to be larger and less filled with tags. If this number is set high, the grids will tend to be smaller and more filled with tags. This controls the aggressiveness of agglomeration by box merging.

– a_blockFactor Blocking factor. For each box B in the grids, this is the number $Nref$ for which it is guaranteed to be true that $refine(coarsen(B, Nref), Nref) == B$. Default = 1. Note that this will also be the minimum possible box size.

– a_bufferSize Proper nesting buffer size. This will be the minimum number of level ℓ cells between any level $\ell + 1$ cell and a level $\ell - 1$ cell. Default = 1.

– a_maxSize Maximum length of a grid in any dimension. An input value of 0 means the maximum value will be ∞ (no limit).

- ```
  void
  define(
         const ProblemDomain&    a_baseDomain,
         const Vector<int>&      a_refRatios,
         const Real              a_fillRatio,
         const int               a_blockFactor,
         const int               a_bufferSize,
         const int               a_maxSize );

  void
  define(
         const Box&              a_baseDomain,
         const Vector<int>&      a_refRatios,
         const Real              a_fillRatio,
         const int               a_blockFactor,
         const int               a_bufferSize,
         const int               a_maxSize );
  ```

  Defines (or redefines) a BRMeshRefine object and places it in a usable state.

  Arguments:

  – a_baseDomain Problem domain at the coarsest (level 0) level. Output grids will be constrained to be within the computational domain on each level.

  – a_refRatios Refinement ratios between the levels. RefRatio[i] represents the refinement ratio between levels i and i+1. The vector indices must correspond to level number.
  ```

- a_fillRatio Overall grid efficiency to be generated. If this number is set low, the grids will tend to be larger and less filled with tags. If this number is set high, the grids will tend to be smaller and more filled with tags. This controls the aggressiveness of agglomeration by box merging.
- a_blockFactor Blocking factor. For each box B in the grids, this is the number $Nref$ for which it is guaranteed to be true that $refine(coarsen(B, Nref), Nref) == B$. Default = 1. Note that this will also be the minimum possible box size.
- a_bufferSize Proper nesting buffer size. This will be the minimum number of level ℓ cells between any level $\ell + 1$ cell and a level $\ell - 1$ cell. Default = 1.
- a_maxSize Maximum length of a grid in any dimension. An input value of 0 means the maximum value will be ∞ (no limit).

•
```
int
regrid(
       Vector<Vector<Box> >&   a_newmeshes,
       IntVectSet&             a_tags,
       const int               a_baseLevel,
       const int               a_topLevel,
       const Vector<Vector<Box> >& a_oldMeshes) const;
```

The interface for BRMeshRefine which takes only a single level of tags and generates a multilevel hierarchy of grids which covers those tags while satisfying the proper nesting requirements. Note that the proper nesting requirement is an overriding constraint – if a tagged cell cannot be refined while satisfying proper nesting, it is not refined. (This is only an issue if BaseLevel > 0.). The grids produced by this function will also satisfy the constraints placed by the BlockFactor, FillRatio, and MaxSize. Returns the finest level on which grids are defined (for this function, this will normally be TopLevel+1)

Arguments:

- a_newmeshes The new set of grids at every level. This is resized and filled in the function.
- a_tags Tagged cells on BaseLevel.
- a_baseLevel Index of base mesh level. This is the finest level which does *not* change. For example, if all grids except level 0 are going to be changed by BRMeshRefine, a_baseLevel = 0.
- a_topLevel Index of top level of relevant tags which is the same as one level *below* the highest level of grids that will change. So if the AMR hierarchy has 9 levels and one wants all of them to change except level 0, she would set a_baseLevel = 0 and a_topLevel = 7 (highest level number is 8).

– a_oldMeshes Grids before BRMeshRefine is called. If there are no previous grids, set a_oldMeshes to be the domains. See the example shown in figure 3.4 The vector indices must correspond to level number.

– Returns the finest level on which grids are defined in newmeshes.

- ```
  int
  regrid(
         Vector<Vector<Box> >&   a_newmeshes,
         Vector<IntVectSet>&     a_tags,
         const int               a_baseLevel,
         const int               a_topLevel,
         const Vector<Vector<Box> >& a_oldMeshes) const;
  ```

  The interface for BRMeshRefine which takes tags at all levels and generates a new multilevel hierarchy of grids which covers the tags at each level while satisfying the proper nesting requirements. Note that the proper nesting requirement is an overriding constraint – if a tagged cell cannot be refined while satisfying proper nesting, it is not refined. (This is only an issue if a_baseLevel $>$ 0.). The grids produced by this function will also satisfy the constraints placed by the BlockFactor, FillRatio, and MaxSize. Returns the finest level on which grids are defined.

  Arguments:

  – a_newmeshes The set of grids at every level. This is resized and filled in this function.

  – a_tags Tagged cells on every level from a_baseLevel to a_topLevel-1. The vector indices must correspond to level number.

  – a_baseLevel Index of base mesh level. This is the finest level which does *not* change. For example, if all grids except level 0 are going to be changed by BRMeshRefine, a_baseLevel = 0.

  – a_topLevel Index of top level of relevant tags which is the same as one level *below* the highest level of grids that will change. So if the AMR hierarchy has 9 levels and one wants all of them to change except level 0, she would set a_baseLevel = 0 and a_topLevel = 7 (highest level number is 8).

  – a_oldMeshes Grids before BRMeshRefine is called. If there are no previous grids, set a_oldMeshes to be the domains. See the example shown in figure 3.4 The vector indices must correspond to level number.

  – Returns the finest level on which grids are defined in a_newmeshes.

Figure 3.4 is a sample code to show the use of BRMeshRefine to create lists of grids from tags. For an explanation of how to use LoadBalance to transform these into DisjointBoxLayouts see 7.4.
  ```

```
int setGrids(
             Vector<Vector<Box> >&        a_vectGrids,
             const Vector<ProblemDomain>& a_vectDomain,
             Vector<int>&                 a_vectRefRatio,
             int&                         a_numlevels,
             Real                         a_fillRat,
             int                          a_maxboxsize)
{
  Box btag = a_vectDomain[0].domainBox();
  int ishrink = btag.size(0)/4;
  btag.grow(-ishrink);
  IntVectSet tags(btag);

  Vector<Vector<Box> > VVBoxNew(a_numlevels);
  Vector<Vector<Box> > VVBoxOld(a_numlevels);
  for(int ilev = 0; ilev <a_numlevels; ilev++)
    {
      VVBoxOld[ilev].push_back(a_vectDomain[ilev].domainBox());
    }
  int baseLevel = 0;
  int topLevel  = a_numlevels - 2;
  int blockFactor = 2;
  int buffersize = 1;
  if(topLevel >= 0)
    BRMeshRefine meshRefine(a_vectDomain[0], a_vectRefRatio,
                    a_fillRat, blockFactor, buffersize,
                            a_maxboxsize)
    meshRefine.regrid(VVBoxNew, tags, baseLevel, topLevel,
                      VVBoxOld);
  else
    VVBoxNew = VVBoxOld;

  a_vectGrids = VVBoxNew;
  return 0;
}
```

Figure 3.4: Sample code to show the use of BRMeshRefine to create lists of grids from tags.

- `const Vector<int>&`
  `refRatios() const;`

  Returns the vector of refinement ratios

- `const Real`
  `fillRatio() const;`

  Returns the FillRatio.

- `const int`
  `blockFactor() const;`

  Returns the blocking factor.

- `const int`
  `bufferSize() const;`

  Returns the proper nesting buffer size.

- `const int`
  `maxSize() const;`

  returns the maximum box length. A value of 0 means the maximum value means ∞ (no limit).

- `void`
  `refRatios(const Vector<int>& a_nRefVect);`

  Sets the vector of refinement ratios

- `void`
  `fillRatio(const Real a_fillRat);`

  Sets the FillRatio.

- `void`
  `blockFactor(const int a_blockFactor);`

  Sets the blocking factor.

- `void`
  `bufferSize(const int a_buffSize);`

  Sets the proper nesting buffer size.

- `void`
  `maxSize(const int a_maxSize);`

  Sets the maximum box length. An input value of 0 means the maximum value will be ∞ (no limit).

3.3.1 domainSplit

There are many times when the physical domain on the coarsest AMR level (level 0) is larger than the maximum desired block size. In this case, the solution is to split the domain into more than one piece. This is especially useful for parallel computations. To simplify this process, the stand-alone function `DomainSplit` is provided:

```
void
domainSplit(const ProblemDomain& a_domain,
            Vector<Box>&         a_vbox,
            const int            a_maxsize,
            const int            a_blockfactor=1);

void
domainSplit(const Box&   a_domain,
            Vector<Box>& a_vbox,
            const int    a_maxsize,
            const int    a_blockfactor=1);
```

Arguments:

- `a_domain` Physical domain
- `a_vbox` Vector of boxes which satisfy the blocking factor and maxsize requirements which make up the decomposed domain.
- `a_maxsize` Maximum allowable box size (0 means no limit).
- `a_blockfactor` Blocking factor; has the same definition as in `BRMeshRefine`.

Chapter 4

AMRTimeDependent

4.1 Hyperbolic Systems of Conservation Laws

In this section, we will describe a general framework for solving time-dependent problems using AMR, including refinement in time. In order to motivate that framework, we first describe in detail the AMR algorithm in [BC89] for solving systems of hyperbolic conservation laws.

We want to solve a system of equations of the form

$$\frac{\partial V}{\partial t} + \nabla \cdot \vec{\mathcal{F}} = 0$$
$$V = V(\boldsymbol{x}, t) \in \mathbb{R}^m$$
$$\vec{\mathcal{F}} = (\mathcal{F}^0(V), ..., \mathcal{F}^{\mathbf{D}-1}(V))$$
$$\mathcal{F}^d : \mathbb{R}^m \rightarrow \mathbb{R}^m$$

We assume that the system is hyperbolic, i.e., that the matrices $\sum_{d=0}^{\mathbf{D}-1} \xi_d \nabla_v \mathcal{F}^d$ have real eigenvalues and a complete set of eigenvectors for all $\xi \in \mathbb{R}^{\mathbf{D}}$. If the system is hyperbolic, we expect that specifying initial conditions of the form

$$V(\boldsymbol{x}, 0) = \Psi(\boldsymbol{x})$$

leads to a well-posed problem.

A variety of multiple-scale phenomena arise in solutions to hyperbolic systems of conservation laws arising from continuum mechanics problems which make AMR an attractive option. These include dynamics of shocks and interfaces, shock-shock intersections, and nonlinear wave focusing.

We assume that the underlying discretization of the above hyperbolic system of equations on a uniform grid is an explicit finite difference method in discrete conservation form.

$$U^{new} = U^{old} - \Delta t \; D\vec{F} \;\; \text{on } \Gamma$$

where $\vec{F}$ is a staggered-grid vector field on Γ, and D is the discrete divergence operator defined in section 3.1.2.3. $\vec{F}$ is function of U^{old}, with a finite domain of dependence: $F_d(U^{old})_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d}$ depends only on $\{U^{old}_{\boldsymbol{i}+\boldsymbol{s}}\}_{|\boldsymbol{s}-\frac{1}{2}\boldsymbol{e}^d|\leq p}$ where p is independent of the mesh spacing.

We extend this method to an adaptive mesh hierarchy using the Berger-Oliger algorithm. We define

$$\{U^l\}_{l=0}^{l_{max}}, U^l : \Omega^l \rightarrow \mathbb{R}^m$$

$U^l = U^l(t^l)$. Here $\{t^l\}$ are a collection of discrete times that satisfy the temporal analogue of proper nesting. $\{t^l\} = \{t^{l-1} + k\Delta t^l : 0 \leq k < n^l_{ref}\}$ The algorithm in [BC89] for advancing the solution in time is given in pseudo-code in figure 4.1. The discrete fluxes $\vec{F}$ are computed by using the piecewise linear interpolation function in section 3.1.1.3 to define an extended solution on

$$\tilde{\Omega} = \mathcal{G}(\Omega^l, p) \cap \Gamma^l \text{ , } \tilde{U} : \tilde{\Omega} \rightarrow \mathbb{R}^m$$

$$\tilde{U}_{\boldsymbol{i}} = \begin{cases} U^l_{\boldsymbol{i}}(t^l) & \text{for } \boldsymbol{i} \in \Omega^l \\ I_{pwl}((1-\alpha)U^{l-1}(t^{l-1}) + \alpha\, U^{l-1}(t^{l-1} + \Delta t^{l-1}))_{\boldsymbol{i}} & \text{otherwise} \end{cases}$$

$$\alpha = \frac{t^l - t^{l-1}}{\Delta t^{l-1}}$$

The regridding step is performed in these steps. First, one constructs $\mathcal{I}^l \subset \Omega^l, l = l_{base}, ..., l_{max}-1$ corresponding to those cells for which a user-specified measure of the error exceeds a specified tolerance. Second, one generates new grids $\Omega^{l,new}, l = l_{base}+1, ..., l_{max}$ on which the new solution be defined. These new grids should satisfy $\mathcal{C}_{n^l_{ref}}(\Omega^{l+1,new}) \supset \mathcal{I}^l$, and should be properly nested, as well as satisfying any other required nesting conditions. This imposes some constraints on $\mathcal{I}^l$ if $l_{base} > 0$. These constraints are met typically by reducing the size of the $\mathcal{I}^l$'s prior to the grid generation step. Finally, one initializes the new data $U^{l,new}(t^l)$. For $l = l_{base}, U^l = U^{l,new}$. For $l = l_{base}+1, ..., l_{max}$,

$$U^{l,new}(t_{regrid})_{\boldsymbol{i}} = \begin{cases} U^{l,new}(t_{regrid})_{\boldsymbol{i}} = U^l(t_{regrid})_{\boldsymbol{i}} & \boldsymbol{i} \in \Omega^l \cap \Omega^{l,new} \\ I_{pwl}(U^{l-1,new}(t_{regrid}))_{\boldsymbol{i}} & \text{otherwise.} \end{cases}$$

Refinement Criteria

Generally speaking, there are two approaches to determining which cells are to be tagged for refinement. One is to tag points at which some local function of the dependent variables or their derivatives exceeds some threshold. The second approach is to compute a local estimate of the truncation error, and tag points at which the magnitude of that error exceeds a given threshold. The first approach can be used very successfully in cases where the user can exploit application-specific information. The second approach is more

procedure advance (l)
$U^l(t^l + \Delta t^l) = U^l(t^l) - \Delta t D\vec{F}^l$ on Ω^l
if $l < l_{max}$
    $\delta F_d^{l+1} = -F_d^l$ on $\zeta_{+,d}^{l+1} \cup \zeta_{-,d}^{l+1}, d = 0, ..., \mathbf{D} - 1$
end if
if $l > 0$
    $\delta F_d^l := \frac{1}{n_{ref}^{l-1}} < F_d^l >$ on $\zeta_{+,d}^l \cup \zeta_{-,d}^l, d = 0, ..., \mathbf{D} - 1$
end if
for $q = 0, ..., n_{ref}^l - 1$
    advance$(l + 1)$
end for
$U^l(t^l + \Delta t^l) = Average(U^{l+1}(t^l + \Delta t^l), n_{ref}^l)$ on $\mathcal{C}_{n_{ref}^l}(\Omega^{l+1})$
$U^l(t^l + \Delta t^l) := U^l(t^l + \Delta t^l) - \Delta t^l D_R(\delta F^{l+1})$
$t^l := t^l + \Delta t^l$
$n_{step}^l := n_{step}^l + 1$
if $(n_{step}^l = 0$ mod $n_{regrid})$ and $(n_{step}^{l-1} \neq 0$ mod $n_{regrid})$
    regrid(l)
end if

Figure 4.1: Pseudo-code description of the Berger-Colella AMR algorithm for hyperbolic conservation laws.

general and more difficult to implement correctly. For that reason, we will discuss it in some detail here.

Let $L^h(\varphi^h) : \Gamma \rightarrow \mathbb{R}$ be a finite difference approximation on a uniform grid to a differential operator $\mathcal{L}$ defined for any $\varphi^h : \Gamma \rightarrow \mathbb{R}^m$. We define the truncation error for L to be

$$\tau^h_{\boldsymbol{i}} = L^h(\psi^h)_{\boldsymbol{i}} - \mathcal{L}(\psi)(\boldsymbol{x}_0 + \boldsymbol{i}h)$$

where $\psi^h = \psi(\boldsymbol{x}_0 + \boldsymbol{i}h)$, and ψ is a smooth function $\psi : \mathbb{R}^{\mathbf{D}} \rightarrow \mathbb{R}^m$. To compute the truncation error in a numerical solution to a system of PDE's, ψ would be the particular solution being computed. The difficulty is that we don't know the exact solution to the PDE. Instead, we can compute the Richardson estimate to the truncation error

$$\tau^{R,2h}_{\boldsymbol{i}} = (L^{2h}(Average(\varphi^h, 2))_{\boldsymbol{i}} - Average(L^h(\varphi^h), 2)_{\boldsymbol{i}})$$

for $\boldsymbol{i} \in \mathcal{C}_2(\Gamma)$. $\tau^{R,2h}_{\boldsymbol{i}} = C\tau^h_{\boldsymbol{i}} + O(h^{p+1})$ where p is the order of accuracy of the operator $L : \tau_{\boldsymbol{i}} = O(h^p)$. It is straightforward to extend the definition of τ^R to an AMR grid hierarchy. In that case, one can tag points to be refined based on whether $|\tau^R_{\boldsymbol{i}}|$ exceeds some threshold.

We can see two difficulties with this approach. The first is that finite difference operators often have a lower order accurate truncation error at the problem domain boundaries. In addition, the discussion in section 3.1 indicates that the truncation error is of lower order accuracy at coarse-fine boundaries as well. However, as indicated previously, the effect of the truncation error at boundaries on the solution error is typically much smaller than the magnitude of the former would indicate. To compute an error estimator that appropriately reflects this fact we rescale the tagging criterion at cells adjacent to the boundaries.

4.2 The Classes `AMR` and `AMRLevel`

The class `AMR` implements a framework for the Berger-Oliger adaptive mesh refinement (AMR) algorithm for time-dependent simulations [BO84]. The data is organized on a hierarchy of levels of refinement, stored as a collection of `AMRLevels`. The class `AMRLevel` is an abstract base class from which must be derived a concrete class which defines and contains the data representation for one level and implements the algorithms for advancing one level in time. The class `AMR` implements refinement of the time step Δt based on the refinement of the grid.

4.2.1 Class structure

The class `AMR` manages the entire hierarchy of levels. The levels are represented in the class `AMR` as a collection of `AMRLevels`. The class `AMRLevel` is an abstract base class from which the applications implementer must derive a concrete *physics class* which defines the

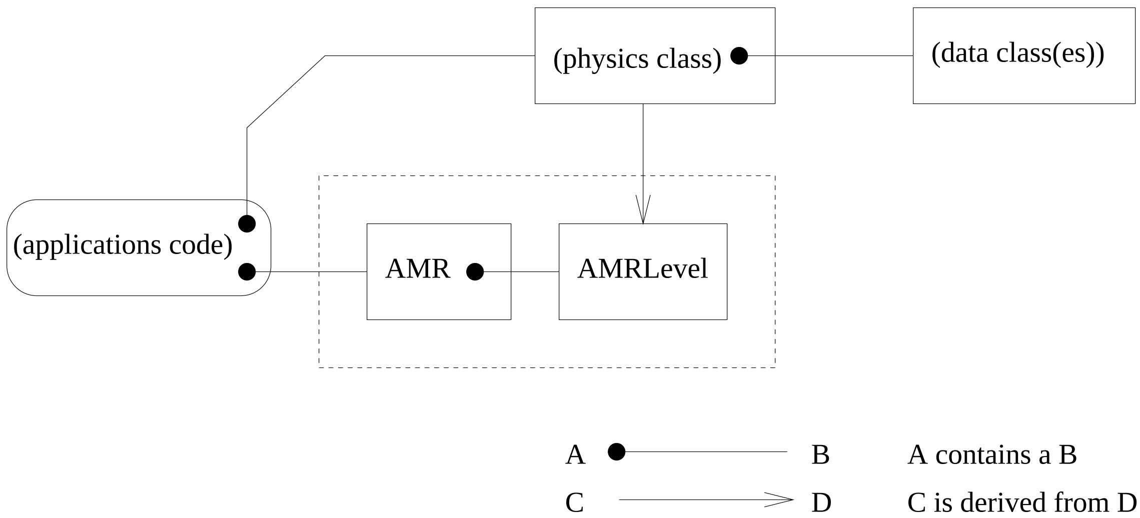

Figure 4.2: Class structure. `AMR` contains some `AMRLevels`. Physics class is derived from `AMRLevel` and contains the data representation.

form of the solution data and, for a single level, implements algorithms for advancement by Δt, calculation of a stable Δt, initialization, input and output, etc. The class `AMR` has no knowledge of the form of the solution data. The applications implementer must instantiate an object of type `AMR` and an object of the physics class type. The physics class object is required input to the definition of an `AMR`. See figure 4.2.

Applications code constructs and invokes public member functions of the class `AMR`. The class `AMR` controls the entire hierarchy of levels. It constructs and invokes member functions of the `AMRLevels`. It requires a physics class derived from `AMRLevelFactory` as input.

4.2.2 The Class `AMR`

The `AMR` class is a framework for Berger-Oliger timestepping for adaptive mesh refinement of time-dependent problems. It is applicable to both hyperbolic and parabolic problems. It represents a hierarchy of levels of refinement as a collection of `AMRLevels`. The usage pattern of this class is as follows:

- Call `define` to define the parameters that do not change throughout the run (maxlevel, refinement ratios, domain, and operator).
- Modify any parameters you like (blocking factor and so forth) using access functions.
- Call any one of the three setup functions so AMR can set up all its internal data structures.
- Call `run` to run the calculation.
- Call `conclude` to produce statistical output, e.g., how many cells were updated.

The important functions of the public interface for the AMR class are:

- ```
  void define(int a_max_level,
              const Vector<int>& a_ref_ratios,
              const ProblemDomain& a_prob_domain,
              const AMRLevelFactory* const a_amrLevelFact);

  void define(int a_max_level,
              const Vector<int>& a_ref_ratios,
              const Box& a_prob_domain,
              const AMRLevelFactory* const a_amrLevelFact);
  ```

  Defines this object. User must call a setup function before running.
  Arguments:

  - a_prob_domain (not modified): Problem domain on the base level.
  - ref_ratios (not modified): Refinement ratios. There must be at least a_max_level+1 elements or an error will result. Element zero is the base level.
  - a_max_level (not modified): The maximum level number allowed, where the base level is zero. There may be a total of a_max_level+1 levels, since level zero and level a_max_level can both exist. Note that it while this is the maximum possible level, it is possible that fewer levels are actually defined, depending on the problem and the method and tolerances used to tag cells for refinement.
  - a_amrLevelFact (not modified): Pointer to a physics class factory object. The object it points to is used to construct the collection of AMRLevels in this AMR as objects of the physics class type.

- `void setupForRestart(HDF5Handle& a_handle);`

  Sets up this object from checkpointed data. User must have previously called `define`. A users needs to call this or setupForNewAMRRun or setupforFixedHierarchyRun before she calls run.

- `void setupForNewAMRRun();`

  Sets up this object for cold start. User must have previously called `define`. Need to call this or setupForRestart or setupforFixedHierarchyRun before you run.

- ```
  void setupForFixedHierarchyRun(const Vector<Vector<Box>>& a_amr_grids,
                                 int a_proper_nest = 1);
  ```

  This sets the grid hierarchy and sets `regrid_intervals` to -1 (turns off regridding). If you want to keep regridding on, reset regridIntervals after this call.

- `void run(Real a_max_time, int a_max_step);`

  Run the calculation. User must have previously called both the define function and a setup function in order to call this.
  Arguments:

  - `a_max_time` Time to stop the calculation.
  - `a_max_step` Maximum number of iterations.

- `void conclude() const`: The user should call this last. It writes the last checkpoint file and performs other housekeeping functions.

There are also functions in the `AMR` class which allow the user to reset various parameters of the run (blocking factor, regridding intervals, checkpointing intervals, etc. See the reference manual for details. Two examples of applications which use the AMR class to implement an adaptive Godunov method for gas dynamics is given in `Chombo/example/AMRGodunovSplit` and `Chombo/example/AMRGodunovUnsplit`.

4.2.3 The Class `AMRLevel`

`AMRLevel` is an abstract base class for data at the same level of refinement within a hierarchy of levels. The concrete class derived from `AMRLevel` is called a *physics class*. The domain of a level is a disjoint union of rectangles in a logically rectangular index space. Data is defined within this domain. There is also a problem domain, which may be larger, within which data can, in theory, be interpolated from some coarser level.

`AMRLevel` is the interface that the class `AMR` uses to call the physics class. The important parts of the public interface to `AMRLevel` are:

- ```
  virtual void define (AMRLevel* a_coarser_level_ptr,
                        const ProblemDomain& a_problem_domain,
                        int a_level,
                        int a_ref_ratio);

  virtual void define (AMRLevel* a_coarser_level_ptr,
                        const Box& a_problem_domain,
                        int a_level,
                        int a_ref_ratio);
  ```

  Defines this AMRLevel.
  Arguments:

  - `coarser_level_ptr` (not modified?): Pointer to next coarser level object.
  - `problem_domain` (not modified): Problem domain of this level.
  - `level` (not modified): Index of this level. The base level is zero.
  ```

  - `ref_ratio` (not modified): The refinement ratio between this level and the next finer level.

- `virtual Real advance() = 0;`

  Advance this level by one time step. Return an estimate of the new time step at this level.

- `virtual void postTimeStep() = 0;`

  Do all operations that are required after a timestep is completed. Refluxing happens here.

- `virtual void tagCells(IntVectSet& a_tags) const = 0;`

  Create tagged cells for dynamic mesh refinement.

- `virtual void tagCellsInit(IntVectSet& a_tags) const = 0;`

  Creates tagged cells for mesh refinement at initialization.

- `virtual void regrid(const Vector<Box>& a_new_grids) = 0;`

  Redefines this level to have the specified grids as its defined union of rectangles.

- `virtual void postRegrid(int a_base_level);`

  Perform any post-regridding operations which are necessary. This is not a pure virtual function to preserve compatibility with earlier versions of AMRLevel. The AMRLevel::postRegrid() instantiation does nothing.

- `virtual void initialGrid(const Vector<Box>& a_new_grids) = 0;`

  Initialize this level to have the specified domain `a_new_grids`.

- `virtual void initialData() = 0;`

  Initialize the data.

- `virtual void postInitialize() = 0;`

  Do any operations that are required just after initialization.

- `virtual Real computeDt() = 0;`

  Returns maximum stable time step for this level.

- `virtual Real computeInitialDt() = 0;`

  Returns maximum stable time step for this level with initial data.

- `virtual void writeCheckpointHeader(HDF5Handle& a_handle) const = 0;`

  Write the header to the checkpoint file handle.

- `virtual void writeCheckpointLevel(HDF5Handle& a_handle) const = 0;`
  Write checkpoint data for this level.
- `virtual void readCheckpointHeader(HDF5Handle& a_handle) = 0;`
  Reads checkpoint header.
- `virtual void readCheckpointLevel(HDF5Handle& a_handle) = 0;`
  Reads checkpoint data for this level.
- `virtual void writePlotHeader(HDF5Handle& a_handle) const = 0;`
  Writes plot file header for this level.
- `virtual void writePlotLevel(HDF5Handle& a_handle) const = 0;`
  Write plot file data for this level.

These are all the pure virtual functions of the `AMRLevel` interface and therefore all the functions that the user **must** define for her application. There are other ancillary functions in the interface that have reasonable defaults. Most of these involve data member access and modification.

4.2.4 The Class `AMRLevelFactory`

The class `AMRLevelFactory` is a pure virtual base class, with only one member function:

- `virtual AMRLevel* new_amrlevel() const = 0;`

  This is the only member function of `AMRLevelFactory`, and it must be defined by the user in a derived class. The derived function will return a pointer to a physics-specific class derived from `AMRLevel`.

A pointer to an object of this class is passed to the define function of an `AMR` object, which uses it to construct the various `AMRLevel` objects that it requires.

Chapter 5

AMRElliptic Algorithm and Implementation

5.1 Multigrid Algorithm

We want to solve the equation

$$L^{comp}\varphi^{comp} = \rho^{comp}$$

on an AMR hierarchy $\{\Omega^l\}_{l=0}^{l_{max}}$ satisfying the nesting conditions described in ([BC89]). The algorithm we use here is a natural extension of multigrid iteration. The particular version we describe here [MC96, Mar98] is a linear version of the algorithm used in [TF89] to compute steady incompressible flow, and has been used in a variety of settings [ABC94, ABC$^+$98, Bet98, CDW99].

A pseudo-code description of the algorithm is given in figure (5.3). The operators $Average$ and I_{pwc} are described in section 3.1.1, and the operator L^{nf} is a two-level discretization of the Laplacian:

$$L^{nf}(\psi^f, \psi^{c,valid}) = D(\vec{G}^f(\psi^f, \psi^{c,valid}))$$

It computes a uniform grid $2\mathbf{D}+1$ point discretization of the Laplacian applied to an extension of ψ^f obtained using the quadratic interpolation procedure in section 3.1.2.2. The smoothing operator mgRelax(φ^f, R^f, r) performs a multigrid V-cycle iteration on φ^f for the operator L^{nf}, assuming the coarse-grid values required for the boundary conditions are identically zero.

5.2 The AMR Elliptic User Interface

The implementation of the AMRElliptic package follows the algorithm specification in section 5.1. The principal components of the user interface are

```
procedure mgRelax(φ^f, R^f, r)
{
    for i = 1, ..., NumSmoothDown
        LevelGSRB(φ^f, R^f)
    end for
    if (r > 2) then
        δ^c := 0
        R^c := Average(R^f − L^{nf}(φ^f, φ^c ≡ 0))
        mgRelax(δ^c, R^c, r/2)
        φ^f := φ^f + I_{pwc}(δ^c)
        for i = 1, ..., NumSmoothUp
            LevelGSRB(φ^f, R^f)
        end for
    end if
}
```

Figure 5.1: Recursive relaxation procedure.

```
procedure LevelGSRB(φ^f, R^f)
{
    φ^f := φ^f + λ(L^{nf}(φ^f, φ^c ≡ 0) − R^f) on Ω^{BLACK}
    φ^f := φ^f + λ(L^{nf}(φ^f, φ^c ≡ 0) − R^f) on Ω^{RED}
}
```

Figure 5.2: Gauss-Seidel relaxation with red-black ordering. Here λ is the relaxation parameter.

$R := \rho - L(\varphi)$
while $(||R|| > \epsilon||\rho||)$
    AMRVCycleMG(l^{max})
    $R := \rho - L(\varphi)$
end while

Procedure AMRVCycleMG(level l):
if $(l = l^{max})$ then $R^l := \rho^l - L^{nf}(\varphi^l, \varphi^{l-1})$
if $(l > 0)$ then
    $\varphi^{l,save} := \varphi^l$ on Ω^l
    $e^l := 0$ on Ω^l
    mgRelax(e^l, R^l, n_{ref}^{l-1})
    $\varphi^l := \varphi^l + e^l$
    $e^{l-1} := 0$ on Ω^{l-1}
    $R^{l-1} := Average(R^{l-1} - L^{nf}(e^l, e^{l-1}))$ on $\mathcal{C}_{n_{ref}^{l-1}}(\Omega^l)$
    $R^{l-1} := \rho^{l-1} - L^{comp,l-1}(\varphi)$ on $\Omega^{l-1} - \mathcal{C}_{n_{ref}^{l-1}}(\Omega^l)$
    AMRVCycleMG($l-1$)
    $e^l := e^l + I_{pwc}(e^{l-1})$
    $R^l := R^l - L^{nf,l}(e^l, e^{l-1})$
    $\delta e^l := 0$ on Ω^l
    mgRelax($\delta e^l, R^l, n_{ref}^{l-1}$)
    $e^l := e^l + \delta e^l$
    $\varphi^l := \varphi^{l,save} + e^l$
else
    solve $L^{nf}(e^0) = R^0$ on Ω^0.
    $\varphi^0 := \varphi^0 + e^0$
end if

Figure 5.3: Pseudo-code description of the AMR multigrid algorithm.

- `AMRSolver` , which manages the AMR / multigrid solution to the elliptic equation on multiple levels;
- `LevelSolver` , which manages the multigrid solution of the elliptic equation on a union of rectangles, with boundary conditions defined in part by a solution defined on a coarser-level grid satisfying certain proper nesting conditions;
- `LevelOp` , an abstract base class that defines the interface between cell-centered discretizations of elliptic PDE's and the solvers.

An example of the use of this package to solve Poisson's equation is included in the Chombo example directory. Examples of classes derived from `LevelOp` for the case of the Poisson and Helmholtz operators are in the AMRElliptic source directory.

5.2.1 The Class `AMRSolver`

`AMRSolver` manages the AMR/multigrid solution to the elliptic equation on a multiple level grid that satisfies certain proper nesting conditions. It can be used either as a solver, or to apply the AMR / multigrid V-cycle as a preconditioner for some other iterative method, such as a Krylov method.
The API for the class `AMRSolver` is defined as follows.

- ```
  void define(
      const Vector<DisjointBoxLayout>& gridsLevel,
      const Vector<ProblemDomain>&     domainLevel,
      const Vector<Real>&              dxLevel,
      const Vector<int>&               refRatio,
      int                              numlevels,
      int                              lBase,
      const LevelOp*                   opin_a,
      int                              ncomp = 1)
  ```
- ```
  void define(
      const Vector<DisjointBoxLayout>& gridsLevel,
      const Vector<Box>&               domainLevel,
      const Vector<Real>&              dxLevel,
      const Vector<int>&               refRatio,
      int                              numlevels,
      int                              lBase,
      const LevelOp*                   opin_a,
      int                              ncomp = 1)
  ```

  Builds internal state for `AMRSolver`.
  Inputs:

- gridsLevel The grids at all levels. Each element in the vector is a level's worth of grids. gridsLevel[0] are grids for level 0 and so forth. Vector index corresponds to level number.
- domainLevel The domains at all levels. Each element in the vector is a level's domain. The second form of "define" with boxes for domains is only included for backward compatibility and should **not** be used. domainLevel[0] is the domain for level 0 and so forth. Vector index corresponds to level number.
- dxLevel The grid spacing at all levels. Each element in the vector is a level's Δx. dxLevel[0] is Δx for level 0 and so forth. Vector index corresponds to level number.
- ref_ratio The refinement ratio between all levels. ref_ratio[0] is the refinement ratio between level 0 and level 1; Vector index corresponds to level number.
- numlevels The number of AMR levels in the calculation. The length of the Vector s has to be at least numlevels.
- lBase - coarsest level on which solution is to be computed. This needs to be set at the time of definition, in order to build the bottom LevelSolver.
- opin_a The LevelOp to use in the solution.
- ncomp - number of components in solution and RHS

- ```
  void solveAMR(
      Vector<LevelData<FArrayBox>*>&       phiLevel,
      const Vector<LevelData<FArrayBox>*>& rhsLevel)
  ```

  Solves the elliptic equation over the hierarchy of levels lBase ... lMax where $lMax = numlevels - 1$. If lBase > 0, then the data at level lBase - 1 is used to interpolate boundary conditions at boundary cells that are not adjacent to the domain boundary.
  Inputs:

  - phiLevel - pointers to current guess at the solution values for levels (lMin = max(lBase-1,0)) ... lMax. Vector index corresponds to level number.
  - rhsLevel - pointers to right-hand side for levels lmin ... lMax. Vector index corresponds to level number.

  Outputs:

  - phiLevel - LevelData<FArrayBox>s pointed to for levels lMin,...,lMax are updated in place.

- ```
  void AMRVCycleMG(
      Vector<LevelData<FArrayBox>*>       corrLevel,
  ```

```
    const Vector<LevelData<FArrayBox>*> residLevel)
```

This class applies a single AMR multigrid v-cycle. It is assumed that the problem is in residual-correction form (so a homogeneous form of the physical boundary conditions will be used).
Inputs:

- `corrLevel` - pointers to current guess at the correction values for levels `lMin = max(lBase-1,0) ... lMax`. Vector index corresponds to level number.
- `residLevel` - pointers to residual for levels `lMin ... lMax`. Vector index corresponds to level number.

Outputs:

- `phiLevel` - `LevelData<FArrayBox>`s pointed to for levels `lMin,...,lMax` are updated in place.

- ```
  Real computeResidualNorm(
      Vector<LevelData<FArrayBox>*>       phiLevel,
      const Vector<LevelData<FArrayBox>*> rhsLevel,
      int                                 normType)
  ```

  This class computes the norm of the residual for the multilevel operator. Cells which are covered by finer grids are not included in the norm calculation. Integral norms are weighted by the area of the cells.
  Inputs:

  - `phiLevel` - pointers to current guess at the solution values for levels `lMin=max(lBase-1,0) ... lMax`. Vector index corresponds to level number.
  - `rhsLevel` - pointers to right-hand side for levels `lMin ... lMax`. Vector index corresponds to level number.
  - `normType` - type of norm required. `normtype = 0` is max norm, `normtype = r > 0` is L^r norm.

  Return Value: return value is the value of the norm of the residual for levels `lMin ... lMax`.

- ```
  void applyAMROperator(
      Vector<LevelData<FArrayBox>*> phiLevel,
      LevelData<FArrayBox>&         LOfPhi,
      int                           lev)
  ```

  Applies multilevel AMR operator, with inhomogeneous physical boundary conditions.
  Inputs:

- phiLevel - pointers to current guess at the solution values for levels lMin = max(lBase-1,0) ... lMax.
- lev - level on which operator is to be applied.

Outputs:

- LOfPhi - Value of operator applied to phiLevel on valid region of level lev.

- ```
  void applyAMROperatorHphys(
      Vector<LevelData<FArrayBox>*> phiLevel,
      LevelData<FArrayBox>&         LOfPhi,
      int                           lev)
  ```

  Applies multilevel AMR operator, with *homogeneous* physical boundary conditions.
  Inputs:

  - phiLevel - pointers to current guess at the solution values for levels lMin = max(lBase-1,0) ... lMax.
  - lev - level on which operator is to be applied.

  Outputs:

  - LOfPhi - Value of operator applied to phiLevel on valid region of level lev.

- ```
  void computeAMRResidual(
      Vector<LevelData<FArrayBox>*>       phiLevel,
      const Vector<LevelData<FArrayBox>*> rhsLevel,
      LevelData<FArrayBox>&               res,
      int                                 lev)
  ```

  Computes residual on valid region at a level.
  Inputs:

  - phiLevel - pointers to current guess at the solution values for levels lMin = max(lBase-1,0) ... lMax. Vector index corresponds to level number.
  - rhsLevel - pointers to right-hand side for levels lMin ... lMax. Vector index corresponds to level number.
  - lev - level for which the residual is to be calculated.

  Outputs:

  - res - LevelData<FArrayBox> into which residual on valid region of level lev is written.

- `void setNumVCyclesBottom(int a_numVCyclesBottom)`

  Set the number of v-cycles performed from the base level to the maximum coarsening of the base level during one overall v-cycle on the AMR hierarchy. The default is 1.
  Inputs:

  – `a_numVCyclesBottom` - number of v-cycles performed from the base level to the maximum coarsening of the base level during one overall v-cycle on the AMR hierarchy.

5.2.2 The Class `LevelSolver`

The class `LevelSolver` manages the solution to the elliptic equation on a union of rectangles, using for boundary conditions a combination of data on a coarser level that satisfies appropriate proper nesting conditions, and physical boundary conditions on a rectangular domain. This is essentially a wrapper around `LevelMG`. The API is given as follows.

- ```
  void define(
      const DisjointBoxLayout& grids,
      const DisjointBoxLayout* gCoarse,
      const ProblemDomain&     domain,
      Real                     dxLevel,
      int                      nRefine,
      const LevelOp*           lop,
      int                      ncomp = 1)
  ```

- ```
  void define(
      const DisjointBoxLayout& grids,
      const DisjointBoxLayout* gCoarse,
      const Box&               domain,
      Real                     dxLevel,
      int                      nRefine,
      const LevelOp*           lop,
      int                      ncomp = 1)
  ```

  Defines a `LevelSolver` object for the case where some of the boundary conditions will be specified by interpolation from variables defined on `*gCoarse`.
  Inputs:

  – `grids` - Union of rectangles on which the solution is to be defined.

  – `gCoarse` - Pointer to a union of rectangles on next coarser level. Set equal to `(DisjointBoxLayout*) NULL` if there is no coarser level, i.e. the union of the boxes in `grids` is equal to `domain`.

- domain - physical domain at this level. The second form of "define" with a box for a domain is only included for backward compatibility and should **not** be used.
- dxLevel - mesh spacing corresponding to grids.
- lop - LevelOp which produces the correct levelop to use in the solution
- nRefine - refinement ratio between grids and gCoarse. Set to zero if there is no coarser level.
- ncomp - number of components in solution and RHS.

- ```
  void levelSolve(
      LevelData<FArrayBox>&       phi,
      const LevelData<FArrayBox>* phiCoarse,
      const LevelData<FArrayBox>& rho)
  ```

  Solves $L(\varphi) = \rho$, on a level, with coarse-fine boundary conditions computed using phiCoarse.
  Inputs:

  - rho - right-hand side.
  - phiCoarse - pointer to data on next coarser level.

  Outputs:

  - phi - solution on a level.

- ```
  void levelSolveH(
      LevelData<FArrayBox>&       phi,
      const LevelData<FArrayBox>& rho)
  ```

  Solves $L(\varphi) = \rho$ for case of homogeneous BCs. Assumes that problem is already in residual-correction form.

- `void setTolerance(Real tol)`

  - resets tolerance in the solver.

- `void setSmoother(int numSmoothUp, int numSmoothDown)`

  - resets number of local smoother sweeps in the solver.

- `const LevelOp& getLevelOperator()`

  - returns a reference to the LevelOp corresponding to the problems being solved by LevelSolver.

5.2.3 The LevelOp Interface

- `virtual LevelOp* new_levelop()`

  The class `LevelOp` is a pure virtual base class which provides an interface for evaluating elliptic operators on a level in an adaptive hierarchy. This function returns a pointer to a class derived from `LevelOp` which has all its internal data unrelated to `LevelOp` defined. This gets around the "no virtual constructor" rule.

- ```
  virtual void define(
      const DisjointBoxLayout& Ba,
      const DisjointBoxLayout* base_ba,
      Real                     DxLevel,
      int                      refratio,
      const ProblemDomain&     domain,
      bool                     homogeneousOnly,
      int                      ncomp = 1) = 0
  ```

- ```
  virtual void define(
      const DisjointBoxLayout& Ba,
      const DisjointBoxLayout* base_ba,
      Real                     DxLevel,
      int                      refratio,
      const Box&               domain,
      bool                     homogeneousOnly,
      int                      ncomp = 1) = 0
  ```

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

  - `Ba` - grids at this level.
  - `base_ba` - Pointer to the grids at the next coarser level in the AMR hierarchy. This is set equal to `(DisjointBoxLayout *) NULL` if there is no coarser level. In that case, the the union of the boxes making up `Ba` must equal `domain`.
  - `DxLevel` - Grid spacing at this level.
  - `refratio` - Refinement ratio between this level and the next coarser level.
  - `domain` - The domain at this level. The second form of "define" with a box for a domain is only included for backward compatibility and should **not** be used.
  - `homogeneousOnly` - Whether the `LevelOp` is able to execute inhomogeneous coarse-fine boundary conditions.
  - `ncomp` - number of components for which operator will be applied.

- ```
  virtual void define(
      const LevelOp* opfine,
      int            reftofine) = 0
  ```

  Full define function. Makes all coarse-fine information and sets internal variables from finer LevelOp. Any LevelOp defined using this constructor is not able to execute inhomogeneous boundary conditions at the coarse-fine interface.
  Inputs:

  - opfine - The finer level operator.
  - reftofine - The refinement ratio between this LevelOp and the next finer.

Data Manipulation Functions

- ```
  virtual void CFInterp(
      LevelData<FArrayBox>&       phi,
      const LevelData<FArrayBox>& phiC) const = 0
  ```

  Coarse-fine interpolation. Fill the ghost cells of phi by interpolating between phi and phiC
  Inputs:

  - phi - The solution on the level.
  - phiC - The solution at the next coarser level

  Outputs:

  - phi - The smoothed solution on the level.

- ```
  virtual void smooth(
      LevelData<FArrayBox>&       phi,
      const LevelData<FArrayBox>& rhs) const = 0
  ```

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

  - phi - The solution on the level.
  - rhs - The right-hand side on the level.

  Outputs:

  - phi - The smoothed solution on the level.
  ```

- ```
  virtual void applyOpI(
      LevelData<FArrayBox>&       phi,
      const LevelData<FArrayBox>* phiCoarse,
      LevelData<FArrayBox>&       LOfPhi) const = 0
  ```

  Evaluate operator using inhomogeneous coarse-fine boundary conditions and inhomogeneous physical boundary conditions.
  Inputs:

  - `phi` - The solution on the level.
  - `phiCoarse` - The solution on the next coarser level.

  Outputs:

  - `LOfPhi` - $L(\varphi)$ on the level.

- ```
  virtual void applyOpIcfHphys(
      LevelData<FArrayBox>&       phi,
      const LevelData<FArrayBox>* phiCoarse,
      LevelData<FArrayBox>&       LOfPhi) const = 0
  ```

  Evaluate operator using inhomogeneous coarse-fine boundary conditions and homogeneous physical (domain) boundary conditions.
  Inputs:

  - `phi` - The solution on the level.
  - `phiCoarse` - The solution on the next coarser level.

  Outputs:

  - `LOfPhi` - $L(\varphi)$ on the level.

- ```
  virtual void applyOpH(
      LevelData<FArrayBox>& phi,
      LevelData<FArrayBox>& LOfPhi) const = 0
  ```

  Evaluate Operator using homogeneous coarse-fine boundary conditions and homogeneous physical boundary conditions.
  Inputs:

  - `phi` - The solution at the current level.

  Outputs:

  - `LOfPhi` - $L(\varphi)$ on the level.
  ```

- ```
  virtual void applyOpHcfIphys(
      LevelData<FArrayBox>& phi,
      LevelData<FArrayBox>& LOfPhi) const = 0
  ```

  Evaluate Operator using homogeneous coarse-fine boundary conditions and inhomogeneous physical boundary conditions.
  Inputs:

  – phi - The solution at the current level.

  Outputs:

  – LOfPhi - $L(\varphi)$ on the level.

- ```
  virtual void bottomSmoother(
      LevelData<FArrayBox>&       phi,
      const LevelData<FArrayBox>& rhs) const = 0
  ```

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

  – phi - The solution at the current level.

  – rhs - The right-hand side on the current level.

  Outputs:

  – phi - The smoothed solution at the current level.

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

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

  – a_phihat – the result of the preconditioning

  – a_rhshat – the right-hand-side of the preconditioning

- ```
  void getFlux(
      FArrayBox&        flux
      const FArrayBox&  data,
      int               dir)
  ```

Fill face-centered `FArrayBox`flux computed from cell-centered `FArrayBox`data. The `FArrayBox`flux gets resized in the routine to be the interior faces of data, Nothing happens if there are no interior faces of data.
Inputs:

- `data` - The data from which to generate the flux (if flux is the gradient of φ, this is φ).
- `dir` - Direction of the face.

Outputs:

- `flux` - Flux `FArrayBox`. `flux` lives on the cell EDGES in direction dir. The flux `FArrayBox`gets resized inside the routine.

Any operator that is to be used within this software framework must provide functions for each of these interfaces. The Laplacian operator which implements the one-sided differenced operator described in the text is provided.

Chapter 6

HDF5 I/O with Chombo

6.1 HDF5 I/O

We have developed a user interface for file I/O based on version 5 of the Hierarchical Data Format library (HDF5) developed at The National Center for Supercomputing Applications (NCSA). HDF5 provides efficient and flexible mechanisms for handling I/O of large scientific datasets, and is becoming a standard in the scientific community for binary portable data files. We exploit a number of features provided by HDF5, including the portability of data across platforms, the ability to read and write files on distributed memory parallel systems. HDF5 also has a number of useful utilities, such as `h5dump`, which produces a human-readable formatted ASCII output of an HDF5 file.

HDF5 has three main user abstractions: *group, dataset, attribute*. Group abstracts the notion of the location in a file, while dataset and attribute are different types of data that can be stored in an HDF5 file. We provide an API for creating HDF5 files and for reading from and writing into such files. These are implemented using two classes, plus a collection of stand-alone functions.

6.1.1 Class `HDF5Handle`

HDF5Handle is a class that manages accessing of and navigation within an HDF5 file.

• Constructors.

```
HDF5Handle();
HDF5Handle(const std:string& a_filename, mode);
int open(const std:string& a_filename, mode);
bool isOpen();
void close();
enum mode {CREATE,OPEN_RDONLY,OPEN_RDWR}
```

A, `HDF5Handle` requires a `a_filename` supplied either at construction using the second constructor, or by a call to `open`. `filename` follows the semantics of `fopen` from the

<stdio.h> of libc. It is an error if a file has already been opened by the HDF5Handle. It is also illegal to open a single file using two different HDF5Handles. The enumeration class mode specifies the access permissions. If mode = CREATE, the a file is created, deleting the previously existing copy of that file if necessary. If mode = OPEN_RDONLY, an existing file is opened with read-only access. If mode = OPEN_RDWR an existing file is opened with read-write access. In the latter two cases, if the file doesn't exist, then the open operation fails: there is no file bound to the HDF5Handle, and a call to isOpen would return false. HDF5Handle objects must be explicitly closed by the user, just like file pointers in standard C. This is done with the close function. You can inquire whether a handle is open or closed with the isOpen() function. Once close has been called, it is possible open a new file with the same HDF5Handle using open.

- File Navigation.

```
int HDF5Handle::setGroup(const std::string& a_groupAbsPath);
const std::string& HDF5Handle::getGroup() const;
```

The function setGroup sets the group (i.e., the location in the file) to be that labeled with the string a_groupAbsPath. If such a group does not yet exist, setGroup creates such a group. The function getGroup returns the string corresponding to the group to which the HDF5Handle is currently set. The input and output strings to which the groups are set are assumed to be of the form of an Unix absolute directory path, e.g., "/foo", "/level_1/info", etc. There is a distinguished root group "/" to which the HDF5Handle is initialized when a file is opened. setGroup can be thought of as analogous to a Unix "cd" command. getGroup can be thought of as analogous to the Unix "pwd" command. We should emphasize that the setting of the group in an HDF5Handle is usually unrelated to the actual physical file layout. It just represents an evocative and convenient notation for navigating within an HDF5 file. setGroup returns 0 on success, a negative number if HDF5 had an error. If the group doesn't already exist, then it is created if the file is write-enabled (CREATE or OPEN_RDWR). In the event of error, file remains open and setGroup can be called again, but HDF5Handle object is not capable of processing reads or writes until a successful setGroup has been performed. (except immediately after file opening when the root group is valid for writing).

6.1.2 Class HDF5HeaderData

The class HDF5HeaderData provides an interface for writing collections of reals, integers, strings, Boxes and IntVectSets. In this interface, one must associate a name (in the form of a character string) for each object. The internal treatment of this data assumes that these are small collections of "metadata", where the efficiency of storage is not a serious concern.

```
class HDF5HeaderData
{
```

```
public:
  int writeToFile(HDF5Handle& a_handle) const;
  int readFromFile(HDF5Handle& a_handle);
  void clear();

  map<std::string, Real>        m_real;
  map<std::string, int>         m_int;
  map<std::string, std::string> m_string;
  map<std::string, IntVect>     m_intvect;
  map<std::string, RealVect>    m_realvect;
  map<std::string, Box>         m_box;

};
```

Once an HDF5HeaderData object is created, the user adds objects to to be stored by adding values to the STL maps that are contained as member data. For example,

```
HDF5HeaderData metaData;
metaData.m_real["mesh spacing"] = dx;
```

If there is already a value in the map corresponding to the string "mesh spacing", the value is overwritten. One queries to see if an if an attribute is entered in one of the maps as follows.

```
bool ghost_exists =
  (metaData.m_intvect.find("ghost") != metaData.m_intvect.end());
```

Finally, one deletes an attribute from a map as follows:

```
metaData.m_real.erase("mesh spacing");
```

Once the user finishes filling in an HDF5HeaderData object, the member functions are writeToFile and readFromFile write and read group attributes from the group currently pointed to by a_handle.

6.1.3 HDF5 I/O for BoxLayoutData

We provide a set of function interfaces for writing out data defined on unions of rectangles. There are two sets of functions: one for reading and writing the unions of rectangles, the second for reading and writing BoxLayoutData objects.

- BoxLayout I/O.

```
int write(HDF5Handle& a_handle, const BoxLayout& a_layout);
int read(HDF5Handle& a_handle, Vector<Box>& boxes);
```

The `write` function writes out the union of boxes corresponding to all of the `BoxLayoutData` objects to be written to that group. Consequently, one can only write one `BoxLayout` object for that group. The read function is not symmetric with the write function. The reason for this is that processor assignment is not written out to the file with the BoxLayout. The file is considered *parallel neutral*. Since a BoxLayout is a combination of Boxes *and* processor assignment, the read function does not have enough information to build a BoxLayout. It is the users responsibility to invoke the appropriate load balancing function after the boxes have been read in, and build a BoxLayout object

- BoxLayoutData I/O.

```
template <class T>
int write(HDF5Handle& a_handle,
          const LevelData<T>& a_data,
          const std::string& a_name);

template <class T>
int read(HDF5Handle& a_handle,
         LevelData<T>& a_data,
         const std::string& a_name,
         const DisjointBoxLayout& a_layout,
         bool redefineData = true);
```

`write` writes the collection of `T` objects in a_data into an HDF5 dataset, linearizing each object into a into a set of 1D Arrays. The default implementation is to use the linearization function `T::linearOut` that is required to define the `LevelData<T>` class, but that will output data in terms of bytes, and in general will not be portable across platforms. The user may provide a more detailed linearization interface, in which case the HDF5 files can be made portable across platforms. Such an interface has been provided in Chombo for the case of `T` = `FArrayBox`. `read` reads a `LevelData<T>` object that had been previously written by the `write` function. On input, a_layout is a null-constructed `LevelData`, which is then defined inside `read`. If `redfineData == false`, then the user takes responsibility for calling `define` is the correct manner for the `LevelData<T> a_data` argument. The a_layout argument must consist of the same collection of `Boxes` as that used to define a_data, but may have a different mapping of boxes to processors than that of the data as it was written.

There are also versions of these functions for the case of `BoxLayoutData<T>`.

6.1.4 HDF5 Out-Of-Core readers

Frequently, a user will generate a data file from a simulation run (particularly in parallel) that will exceed a single computers available RAM. This can make auxiliary post-processing programs impossible to run unless they too are programmed in parallel. The nature of many post-processing operation, however, are more like data-mining than a full simulations. For those cases a simpler approach can be used where only a subset of the data file is read in and operated on.

```
template <class T>
int readLevel(
         HDF5Handle& a_handle,
         const int& a_level,
         LevelData<T>& a_data,
         Real& a_dx,
         Real& a_dt,
         Real& a_time,
         Box& a_domain,
         int& a_refRatio,
         const Interval& a_comps = Interval(),
         const IntVect& ghost = IntVect::Zero,
         bool setGhost = false);

int readBoxes(
         HDF5Handle& a_handle,
     Vector<Vector<Box> >& boxes);



int readFArrayBox(
         HDF5Handle& a_handle,
     FArrayBox&  a_fab,
     int a_level,
     int a_boxNumber,
     const Interval& a_components,
     const std::string& a_dataName = "data" );
```

The first function defaults to reading the entire range of components for a given level of data. If the user specifies `Interval a_comps` (currently defaulting to the entire data range) then a user can select out just a particular range of components.

`readFArrayBox` is even more specific, in that it reads individual FArrayBox data's from the file by

$$level, index$$

reference.

6.2 AMR I/O routines

WriteAMRHierarchyHDF5 and ReadAMRHierarchyHDF5 are convenient global functions used in the AMR codes developed by ANAG. There are three main reasons for their use:

1. It relieves the user from having to learn about the HDF5 interface code.
2. It places the data into a format that can subsequently be read successfully by the ChomboVis and ChomboPlot post-processing tools.
3. They are symmetric and can be used for efficient checkpoint file generation.

6.2.1 function WriteAMRHierarchyHDF5

```
void
WriteAMRHierarchyHDF5(const string& filename,
                      const Vector<DisjointBoxLayout>& a_vectGrids,
                      const Vector<LevelData<FArrayBox>* >& a_vectData,
                      const Vector<string>& a_vectNames,
                      const Box& a_domain,
                      const Real& a_dx,
                      const Real& a_dt,
                      const Real& a_time,
                      const Vector<int>& a_refRatio,
                      const int& a_numLevels)
```

Arguments:
No arguments are modified.

- filename : file to output to.
- a_vectGrids : grids at each level.
- a_vectData : data at each level.
- a_vectNames: names of variables.
- a_domain : domain at coarsest level.
- a_dx : grid spacing at coarsest level.
- a_dt : time step at coarsest level.
- a_time : time.
- a_vectRatio : refinement ratio at all levels (ith entry is refinement ratio between levels i and i + 1).

- a_numLevels : number of levels to output.

filename is created if it doesn't already exist, and overwritten if it does exist. a_vectGrids must match the grids that a_vectData is defined over. a_vectNames are the names you wish to be associated with the components of the a_vectData. a_domain is the covering domain box at the coarsest level. a_numLevels is the number of levels, starting at level 0 that the user wishes to be output.

6.2.2 function ReadAMRHierarchyHDF5

```
int
ReadAMRHierarchyHDF5(const string& filename,
                     Vector<DisjointBoxLayout>& a_vectGrids,
                     Vector<LevelData<FArrayBox>* > & a_vectData,
                     Vector<string>& a_vectNames,
                     Box& a_domain,
                     Real& a_dx,
                     Real& a_dt,
                     Real& a_time,
                     Vector<int>& a_refRatio,
                     int& a_numLevels,
                     const IntVect& ghostVector)
```

Arguments:

- filename : file to input from.
- a_vectGrids : grids at each level.
- a_vectData : data at each level.
- a_vectNames: names of variables.
- a_domain : domain at coarsest level.
- a_dx : grid spacing at coarsest level.
- a_dt : time step at coarsest level.
- a_time : time.
- a_vectRatio : refinement ratio at all levels.
- a_numLevels : number of levels to read.
- a_ghostVector : IntVect used to define a_vectData

return codes:

```
0: success
-1: number of levels <= 0
-2: number of components <= 0
-3: error in readlevel function
-4: file open failed
```

the argument notes are the same as for `WriteAMRHierarchyHDF5`, with the addition of `ghostVector`. `ghostVector` is passed in the argument list and is used in the definition of the `LevelData<FArrayBox>` definition of `a_vectData`. This was most useful for data moving between a simulation code and a post-processing code where the ghost cell requirements can be different.

6.3 Other HDF5 I/O functions

To aid in debugging and in visualizing intermediate data, the functions `writeFAB`, `writeFABname`, `writeLevel`, and `writeLevelname` may be used. These functions are designed to output a single `FArrayBox` or a `LevelData<FArrayBox>` into a file which can then be read by ChomboVis. These functions may be called from debuggers such as gdb during the debugging process. The usual way these functions are used is as follows:

1. place the line
   `#include ``AMRIO.H''`
   in the file which contains the function `main()`.

2. Run the code in the debugger, calling the needed IO function as needed.

3. in a shell environment, start ChomboVis to look at the output file

6.3.1 functions writeFAB and writeFABname

The functions `writeFAB` and `writeFABname` write a single FArrayBox into a file which mimics the plotfile format of WriteAMRHierarchyHDF5, and which can then be viewed with ChomboVis (called separately). The writeFAB function writes the input FArrayBox into a file named fab.out, while the writeFABname allows the user to specify the name of the output file. Note that the data is passed using a pointer to an FArrayBox.

```
void
writeFAB(const FArrayBox* a_data)
```

Arguments:
No Arguments are modified.

- a_data : data to be written

A file named fab.out is created if it doesn't already exist and overwritten if it does exist.

```
void
writeFABname(const FArrayBox* a_data, const char* a_filename)
```

Arguments:
No Arguments are modified.

- a_data : data to be written
- a_filename : name of file into which data is written

a_filename is created if it doesn't already exist, and overwritten if it does exist.

6.3.2 functions writeLevel and writeLevelname

The functions writeLevel and writeLevelname write the data from a single LevelData<FArrayBox> into a file which mimics the plotfile format of WriteAMRHierarchyHDF5, and which can then be viewed with ChomboVis (called separately). The writeLevel function writes the input LevelData<FArrayBox> into a file named LDF.out, while the writeLevelname allows the user to specify the name of the output file.

Notes:

- Data is passed using a pointer to a LevelData<FArrayBox>.
- All data is written on an FArrayBox by FArrayBox basis, including any and all ghost cells.

```
void
writeLevel(const LevelData<FArrayBox>* a_data)
```

Arguments:
No Arguments are modified.

- a_data : data to be written

A file named LDF.out is created if it doesn't already exist and overwritten if it does exist.

```
void
writeLevelname(const LevelData<FArrayBox>* a_data, const char* a_filename)
```

Arguments:
No Arguments are modified.

- a_data : data to be written
- a_filename : name of file into which data is written

a_filename is created if it doesn't already exist, and overwritten if it does exist.

Chapter 7

Parallel Programming with Chombo

7.1 Initialization and Scoping

Chombo provides no special MPI initialization function. This is done intentionally to make it easier for Chombo users to interface with other parallel packages (which may provide their own special initialization routines) in the same code that they use with Chombo.

If one is using Chombo in a parallel code, one must somehow call MPI_Init before instantiating any Chombo data objects (whether that is in the context of some other package's initialization routines or not). One must also call MPI_Finalize after all Chombo data objects have gone out of scope. To make sure these get done in the correct order, some care in scoping is required. Some Chombo classes, by necessity, call MPI functions in their destructors. One must be careful to make certain that all Chombo classes go out of scope *before* MPI_Finalize gets called. A simple way to do this is shown here:

```
int main(int argc, char* argv[])
{
#ifdef MPI
MPI_Init(&argc, &argv);
#endif
//this sets the beginning of the scope of Chombo objects
{

LevelData<FArrayBox> phi, rhs;
...do a bunch of calculations...

//this sets the end of  the scope of Chombo objects
}
#ifdef MPI
MPI_Finalize();
#endif
return(0);
```

```
}
```

In this example, MPI_Init and MPI_Finalize get used in the normal sense but the braces between them force Chombo destructors to be called before MPI_Finalize is called.

7.2 Overview of Chombo Data Parallelism

Our parallel model is box-based SPMD parallel programming. Distributed data always lives in containers (LayoutData, BoxLayoutData, and LevelData are the containers). All processors execute the same code. All processors have access to the unions of rectangles (the BoxLayouts and the DisjointBoxLayouts) and what processors each rectangle's data lives upon. The data associated with the rectangles is distributed among processors. We use smart iterators over our data objects which stop at the boxes which live on the current processor. To be complete, broadcast and gather functions are included for situations where parallelism cannot be hidden by iterators.

As in the serial case, the class BoxLayout represents an arbitrary union of rectangles and LayoutData and BoxLayoutData both represent data built upon such a union. The class DisjointBoxLayout represents a disjoint union of rectangles and LevelData represents data built upon a disjoint union. The data in these data holders is distributed among processors according to the boxes that the data lives upon. A box's worth of data is considered atomic in this model.

Also, as in the serial case, the data holders have very different communication patterns even though all the holders distribute their data among processors. LayoutData is a distributed object that can not be involved in communication (it can be neither the source nor destination in copyTo or exchange). A BoxLayoutData object may be only the destination of a copyTo function and exchange is not defined for BoxLayoutData. A LevelData object, because it is built upon a disjoint layout, may be involved in any of our forms of data communication. Chombo also contains two templated communication functions that sometimes cannot be avoided in parallel applications: broadcast and gather. See section 7.5 for details.

7.3 Box-processor assignment

A BoxLayout is a set of boxes and processor assignments. We construct the layout with two matching lists: a Vector of boxes and a Vector of integers which represent the processor into which data over the box will be distributed. Chombo does provide a load balancing function (see section 7.4 for details) which can generate these processor assignments. This function is not integrated into the BoxLayout class for the express purpose of providing a user the ability to use her own load balancing algorithm to generate processor assignments.

For now, assume we have a vbox = Vector<Box> which represents the grids from which we generate a BoxLayout and we have a vint = Vector<int> which represents the processor mapping we desire (we want data which resides on `vbox[i]` to reside on processor `vint[i]`). A `BoxLayout` can be constructed either incrementally:

```
BoxLayout boxlayout; //layout
Vector<Box> vbox;  //grids
Vector<int> vint;  //processor assignments
for(int ibox = 0; ibox < vint.size(); ibox++)
   boxlayout.addBox(vbox[ibox], vint[ibox]);
boxlayout.close();
```

or all at once:

```
Vector<Box> vbox;  //grids
Vector<int> vint;  //processor assignments
BoxLayout boxlayout(vbox, vint);
boxlayout.close();
```

Note that the `close` function must be called in either case after all the boxes are added. A `DisjointBoxLayout` is constructed in exactly the same way. If the boxes which are added to a `DisjointBoxLayout` are not disjoint (i.e. they have some nontrivial intersection) a runtime error is raised when `close()` is called.

7.4 LoadBalance

Chombo provides a load balancing function (called `LoadBalance`) to compute an assignment of boxes to processors for an AMR mesh hierarchy. The assignment is made to balance the computation workload on each processor (i.e., make it as even as possible). The meshes in the AMR hierarchy are represented using `Vector< Vector< Box > >`. The computational workload is a real number for each box in the hierarchy, represented as a `Vector< Vector <Real> >`. This is an input which the user may prescribe to her own needs. Load determination is far too application-specific to permit any kind of general solution. The resulting assignment is an integer for each box in the hierarchy, which gives the processor number (starting at zero) on which each box will reside. `LoadBalance` uses the Kernighan-Lin algorithm for solving knapsack problems. This algorithm has been used quite successfully for load balancing parallel AMR calculations [Cru91]. The interface to `LoadBalance` is given by

```
int LoadBalance(Vector<int>& procAssignments,
          const Vector<Box>& boxes).
```

Here `boxes` are the input grids and `procAssignments` are the processor numbers that go with each box. The load for a given used by this version of `LoadBalance` is the number of points in the box. There is a more elaborate version of `LoadBalance` in Chombo which allows the user to input the loads for each box. See the reference manual for details. The return value of `LoadBalance` is an error code. If `LoadBalance` exited without error, 0 is returned. If anything other than zero is returned, the output values are undefined. An example of how to use `LoadBalance` is shown in 7.1.

```
void setGrids(Vector<DisjointBoxLayout>& vectGrids,
              const Vector<int>&         vectRefRatio,
              const Vector<Vector<Box>&  VVBoxNew,
              const int& numlevels)
{
  for(int ilev = 0; ilev < numlevels; ilev++)
   {
        const Vector<Box>& levelGrids = VVBoxNew[ilev];
        Vector<int> procAssign;
        int eekflag = LoadBalance(procAssign, levelGrids);
        assert(eekflag == 0);
        vectGrids[ilev].define(levelGrids, procAssign);
        vectGrids[ilev].close();
   }
}
```

Figure 7.1: Code snippet to show how LoadBalance is used to transform a list of boxes into `DisjointBoxLayouts`.

7.5 Broadcast and Gather

Chombo also contains two templated communication functions that sometimes cannot be avoided in parallel applications: broadcast and gather. Consider the following example: suppose one has a `LevelData<FArrayBox>` called `resid` and one wants to calculate the max norm of the data in this container. A naive way to do this would be:

```
//naive routine to calculate max norm of resid at varNum component
Real maxNorm(LevelData<FArrayBox>& resid, int varNum)
{
  Real maxnorm = 0;
  DataIterator dit = resid.iterator();
  for (dit.reset(); dit.ok(); ++dit)
  {
    maxnorm = Max(maxnorm, resid[dit()].norm(0, varNum, 1);
  }
  return maxnorm;
}
```

This code is correct in serial and incorrect in parallel. In parallel, every processor will have a different value of maxnorm. To make this code correct, we must *gather* all the values of maxnorm, calculate the maximum value, and *broadcast* this value to all processors.

The interface to the templated gather function is as follows:

```
///gather a_input into  a_outVec on a_dest
template <class T>
void gather(Vector<T>& a_outVec, const T& a_input, int a_dest);
```

This function gathers a_input from every processor into Vector<T> a_outVec on processor a_dest. a_outVec is a vector of length nProc() long with the value of a_input on every processor in its elements.

The interface to the templated broadcast function is as follows:

```
///broadcast a_inAndOut to every processor from a_src
template <class T>
void broadcast(T& a_inAndOut,  int a_src);
```

This function broadcasts a_inAndOut from processor a_src to all processors for both broadcast<T> and gather<T>. There are some restrictions on T, which are explained in section 7.5.1.

Here is how to make the previous example work in parallel:

```
//correct routine to calculate max norm of resid at varNum variable
Real maxNorm(LevelData<FArrayBox>& resid, int varNum)
{
  Real maxnormLocal = 0;
  DataIterator dit = resid.iterator();
  for (dit.reset(); dit.ok(); ++dit)
  {
    maxnormLocal = Max(maxnormLocal, resid[dit()].norm(0, varNum, 1);
  }
  //gather all maxnormLocals onto processor 0
  int srcProc = 0;
  Vector<Real> allMaxNorm(numProc());
  gather(allMaxNorm, maxnormLocal, srcProc);
  Real maxnorm = 0;
  if(procID() == srcProc)
  {
    for(int ivec = 0; ivec < numProc(); ivec++)
      maxnorm = Max(maxnorm, allMaxNorm[ivec]);
  }

  //broadcast the right answer to all procs
  broadcast(maxnorm, srcProc);
  return maxnorm;
}
```

This example will work in both the serial and parallel cases.

7.5.1 linearIn, linearOut, linearSize

By "linearize," we mean "to convert a data structure into a contiguous block of memory." For either gather<T> or broadcast<T> to work, T must have have the following template functions:

- int linearSize<T>(const T& inputT)

  Return the linear size of the object inputT in bytes.

- void linearIn<T>(T& outputT,
  const void* const inBuf)

  Initialize the object outputT from the byte stream in inBuf.

- void linearOut<T>(void* const outBuf,
  const T& inputT)

  Output the object inputT into the byte stream outBuf. The memory for the buffer is assumed to be allocated elsewhere.

Chombo provides these functions for Box, IntVectSet, Real, int and a templated function for any Vector<T> (as long as T has the three functions itself).

Chapter 8

Chombo Fortran

8.1 Introduction

The Chombo library is built with the ability to call Fortran routines from C++. There are many reasons to want to do this. For example, one many want to use the more complex data structures that C++ supports but may not want to forfeit the superior floating-point performance that Fortran offers. The details of mixed language programming, however can be complex can be both compiler and platform-dependent. Another complication is that C++ can be written in a dimension-independent form but the syntax of Fortran is intrinsically dimension-dependent. Array access, declaration and looping all require knowledge of the dimensionality of the problem. Chombo Fortran is a package designed to create abstractions which avoid these problems. Chombo Fortran allows the C++-Fortran programmer many advantages.

- The name-mangling differences between Fortran and C++ are handled automatically and cleanly.
- Type checking of arguments in calls to Fortran from C++ is handled automatically by the C++ compiler. This makes mixed language code far less error-prone.
- Dimension-independent Fortran code is made possible. This eliminates the maintenance problems associated with having to maintain separate Fortran kernels for simulation codes which differ only in the number of spatial dimensions.
- Very long Fortran argument lists and declarations (due to array specification) are greatly reduced by the Chombo Fortran macros. This makes Chombo Fortran less error-prone and easier to read.

The basic usage pattern is this. One uses Chombo Fortran to declare her subroutine argument lists and local floating point arguments. ChF interprets these macros in the context of the input dimensionality and precision and creates a Fortran file. ChF also creates a prototype file to be included in the C++ calling file which unravels the compiler and platform-dependence of the Fortran name mangling (so C++ will be able to find the function).

8.2 ChF Fortran macros

There are two classes of Fortran macros in ChF: declaration and access. The declaration macros are used to specify arguments to Fortran subroutines that will be called from C++. The access macros are used to reference these arguments in the body of Fortran subroutines. There are also the two macros that do not fit into either category: `CHF_DDECL` and `CHF_DTERM`.

8.3 CHF_DDECL and CHF_DTERM

`CHF_DDECL[arg0;arg1;arg2]` translates to arg0, arg1, arg2. This is useful when one needs to declare variables that only exist in a dimension-dependent context. Say, for example, one has SpaceDim components of velocity called `(u,v,w)` in three dimensions. Since in two dimensions, the third component is not used in the code, one could declare these variables as

```
    integer CHF_DDECL[u;v;w]
```

to avoid "unused variable" compiler warnings. This macro will respect carriage returns and other white space.

Similarly `CHF_DTERM[arg0;arg1;arg2]` translates to `arg0arg1arg2` in three dimensions and `arg0arg1` in two dimensions. This is useful if one has code that is dimension-dependent. One example is this:

```
  integer CHF_DDECL[ii;jj;kk]

  CHF_DTERM[
  ii = CHF_ID(0,idir);
  jj = CHF_ID(1,idir);
  kk = CHF_ID(2,idir)]
```

This macro will respect carriage returns and other white space.

8.4 CHF_MULTIDO and CHF_ENDDO

`CHF_MULTIDO` is used to iterate over a box in a dimension independent fashion by setting up nested Fortran do loops and `CHF_ENDDO` is used to terminate those do loops correctly. Specifically, `CHF_MULTIDO[box;i;j;k]` will generate a do loop for `i` nested inside a do loop for `j` and, in 3D, this will be nested inside a do loop for `k`. The `i` loop will go from first element of the low corner of box to the first element of the high corner of box. Similarly, the j loop will use the second element and, in 3D, the k loop will use the third element. CHF_ENDDO will end all the do loops set up by CHF_MULTIDO. Here is an example using these macros:

```
subroutine LOOP(CHF_FRA1[array],CHF_BOX[box])

integer CHF_DDECL[i;j;k]
integer productsum

productsum = 0
CHF_MULTIDO[box;i;j;k]
  productsum = productsum + i*j*k
  array(CHF_IX[i;j;k]) = productsum
CHF_ENDDO

return
end
```

The other sections contain exact definitions of the other macros used in the this example.

8.5 Declaration macros

The declaration macros are used inside Fortran SUBROUTINE statements (in the argument list) to specify the types of the arguments to the subroutine.

The ChF system automatically generates type declaration statements for the variables named in ChF declaration macros so explicit declarations statements for these variables are unnecessary and will likely cause compilation errors.

The declaration macros can be used to declare variables of the basic data types (INTEGER and REAL_T) and variables corresponding to Chombo C++ classes (Box, FArrayBox and Vector). Variables of the basic types can be scalars or 1D arrays (CHF*1D macros). Variables of FArrayBox type can have single or multiple components (CHF*F*1 macros).

The macros automatically create and declare all the extra arguments related to array sizes that are needed. The ChF access macros can be used to access these variables. For example, the macro CHF_LBOUND[A;1] would return the lower bound of the A variable in the first dimension.

The "_CONST" qualifier in the macro names indicates that the variable named in the macro is not modified in the Fortran subroutine. This has no direct effect on the Fortran code or its execution, but it does affect the C++ code that calls the Fortran subroutine and the C++ prototype that is automatically-generated by ChF.

The following is the complete list of ChF Fortran macros and their uses.
Declaration macros:

- CHF_INT[<*arg*>] Declare a scalar integer argument.
- CHF_REAL[<*arg*>] Declare a scalar floating point argument.
- CHF_CONST_REALVECT[<*arg*>] Declare a constant real vector of spacedim length argument (goes from 0 to SpaceDim-1).
- CHF_REALVECT[<*arg*>] Declare a real vector of spacedim length argument (goes from 0 to SpaceDim-1).
- CHF_CONST_INTVECT[<*arg*>] Declare a constant integer vector of spacedim length argument (goes from 0 to SpaceDim-1).
- CHF_INTVECT[<*arg*>] Declare an integer vector of spacedim length argument (goes from 0 to SpaceDim-1).
- CHF_CONST_INT[<*arg*>] Declare a read-only scalar integer argument.
- CHF_CONST_REAL[<*arg*>] Declare a read-only scalar floating point argument.
- CHF_FIA[<*arg*>] Declare a multi-component integer C++ BaseFab argument.
- CHF_FRA[<*arg*>] Declare a multi-component floating point BaseFab argument.
- CHF_FIA1[<*arg*>] Declare a single-component integer BaseFab argument.
- CHF_FRA1[<*arg*>] Declare a single-component floating point BaseFab argument.
- CHF_CONST_FIA[<*arg*>] Declare a read-only multi-component integer BaseFab argument.
- CHF_CONST_FRA[<*arg*>] Declare a read-only multi-component floating point BaseFab argument.
- CHF_CONST_FIA1[<*arg*>) Declare a read-only single-component integer BaseFab argument.
- CHF_CONST_FRA1[<*arg*>] Declare a read-only single-component floating point BaseFab argument.
- CHF_BOX[<*arg*>] Declare a Box argument Boxs are always read-only.

So a typical subroutine declaration would look like this:

```
      subroutine TYPICAL(
     &     CHF_FRA[fab],
     &     CHF_CONST_FRA[constfab],
     &     CHF_BOX[region],
     &     CHF_CONST_REAL[dx],
     &     CHF_INT[intflag])
```

This routine takes two floating point BaseFabs (one constant), a box, a constant floating point scalar and an integer. Keep in mind that this is still Fortran. All arguments are still being sent as pointers so they can be changed in the Fortran code. The CONST modifier of the declaration just adds a const to the C++ prototype to allow the user to send const C++ variables without the C++ compiler complaining.

8.6 Access macros

- CHF_LBOUND[<*arg*>;<*dim*>] Access the lower bound of a BaseFab or Box <*arg*> in dimension <*dim*>. Returns an integer variable.
- CHF_UBOUND[<*arg*>;<*dim*>] Access the upper bound of a BaseFab or Box <*arg*> in dimension <*dim*>. Returns an integer variable.
- CHF_NCOMP[<*arg*>] Access the number of components in the BaseFab, Vector or 1D array <*arg*>. Returns an integer. Note that the components are numbered from 0 to CHF_NCOMP(<*arg*>)-1 in Fortran code, to be consistent with the requirements of C++.
- CHF_IX[<*index0*>;<*index1*>;<*index2*>] Access an element of an array declared with one of the F*A* macros.
- CHF_ID(<*dim1*>,<*dim2*>) Return 1 when the arguments have the same value. Used with CHF_IX for accessing "nearby" array elements. Notice that CHF_ID uses parentheses instead of square brackets and a comma instead of a semicolon. Simply put, CHF_ID is isn't really a macro—it is a 3x3 identity matrix which gets declared in every subroutine the parentheses are consistent with array access in Fortran,

Notes:

- The <*arg*> macro argument must be a variable that was declared with one of the BaseFab or Box macros.
- The <*dim*> macro argument must be an integer variable or constant in the range 0...CH_SPACEDIM-1.
- Only SUBROUTINEs can be called from C++. FUNCTIONs are not supported.
- The dimensions values are 0-based as in C++, not 1-based as is the default for Fortran.
- CHF_ID is a local variable that gets declared in every routine. If it is not used, minor compiler warnings can result.

8.7 C++ macros

The ChF C++ macros are intended to be used in C++ code that calls Fortran subroutines that have been declared using the ChF Fortran macros. The prototype header file that is automatically generated by the ChF Fortran macros must be #included in any file where the ChF C++ macros are used to call a Fortran subroutine. The name of this header file is of the form "<*fortran_file_basename*>_F.H", where <*fortran_file_basename*> is the name of the Fortran source code file without the extension. Every Fortran subroutine that is called from C++ must appear in one and only one included prototype header file.

There are two aspects to using the ChF macros to call Fortran subroutines: specifying the name of the Fortran subroutine and specifying the arguments to the Fortran subroutine.

Fortran subroutines must be called from C++ by prefixing the name of the subroutine with FORT_ and always using uppercase. For example, the Fortran subroutine named "FOO" must be called from C++ using the name "FORT_FOO". Attempts to access the Fortran name directly will fail on some systems because of compiler-dependent inter-language calling conventions.

The C++ prototypes for Fortran subroutines with no arguments will be generated with the keyword "void" in the argument list.

All arguments to a Fortran subroutine called from C++ must be specified in ChF declaration macros. The macro names indicate the data type of the argument and allow the ChF system to generate appropriate dimension-independent code. The macros used in C++ application code should match the macros that appear in the prototypes provided in the *_F.H header files, except that macros in application code should use the CHF_ prefix where the macros used in the prototypes use the CHFp_ prefix.

Most of the declaration macros come in a CONST and non-CONST form. The CONST form should be used to declare arguments that are not modified by the Fortran subroutine. The Box macro does not have a CONST form because Boxes are assumed to be constant always.

The ChF C++ declaration macros are almost identical in syntax and usage to the Fortran declaration macros. The differences are:

- the C++ macros are case-sensitive,
- the single-component BaseFab macros (CHF_*F{I|R}1()) take 2 arguments: (BaseFab, component_number),
- for each Fortran subroutine <*proc*>, a C++ macro FORT_<*proc*> is defined.

8.8 Declaration macros

The C++ declaration macros are those that the application programmer uses to pass variables to Fortran routines from C++.

The following is the complete list of ChF C++ macros and their uses.

- CHF_INT(<*arg*>) Pass a scalar int variable.
- CHF_REAL(<*arg*>) Pass a scalar Real variable.
- CHF_CONST_REALVECT(<*arg*>) Pass a constant RealVect variable.
- CHF_REALVECT(<*arg*>) Pass a RealVect variable.
- CHF_INTVECT(<*arg*>) Pass a constant IntVect variable.
- CHF_CONST_INT(<*arg*>) Pass a const scalar int variable.
- CHF_CONST_REAL(<*arg*>) Pass a const scalar Real variable.
- CHF_FIA(<*arg*>) Pass an BaseFab<int> .
- CHF_FRA(<*arg*>) Pass a BaseFab<Real> .
- CHF_FIA1(<*arg*>,<*comp*>) Pass a single component of an BaseFab<int>.
- CHF_FRA1(<*arg*>,<*comp*>) Pass a single component of a BaseFab<Real>.
- CHF_CONST_FIA(<*arg*>) Pass a const BaseFab<int> .
- CHF_CONST_FRA(<*arg*>) Pass a const BaseFab<Real>.
- CHF_CONST_FIA1(<*arg*>,<*comp*>) Pass a single const component of an BaseFab<int>.
- CHF_CONST_FRA1(<*arg*>,<*comp*>) Pass a single const component of a BaseFab<Real>.
- CHF_BOX(<*arg*>) Pass a Box. Boxes are always const.
- FORT_<*proc*>(...) Call the Fortran subroutine <*proc*> with the arguments specified.

8.9 Language support

Chombo Fortran supports the Fortran standard language with a few exceptions. The exceptions include standard Fortran features that are not supported and an extension to the standard that is required.
Chombo Fortran does not support the following features of the Fortran standard:

- REAL, DOUBLE PRECISION, COMPLEX datatypes. The only floating point datatype that is supported is REAL_T. REAL_T is a Chombo Fortran extension to the Fortran standard.
- Appending "*<*length*>" to a datatype is not supported. This is not standard Fortran, but is a common extension.

- Non-void functions are not supported by Chombo Fortran. Only subroutine statements are supported and those are only allowed with Chombo Fortran macros as arguments.

The code generated by the Chombo Fortran preprocessor conforms to the Fortran standard (ISO/IEC 1539:1991, ANSI X3.198-1992) with the following exceptions:

- The code produced by ChF may violate the Fortran standard maximum number of continuation lines in a statement (19). If this occurs, it will be necessary to provide a compiler option to increase the limit or change the original Fortran code so that it produces fewer continuation lines, usually by breaking a single statement into several separate statements.

- Chombo Fortran does not support input and output to the standard units (i.e., 5,6,"*") on all combinations of C++ and Fortran compilers. Input and output to files should work correctly in all systems. This problem is a fundamental one of mixed-language programming and cannot be solved in any kind of a general way. A special subroutine is provided which allows the Fortran code to print a special message and terminate execution of the program. This subroutine interfaces with the `MayDay` class in the Chombo C++ library. The subroutine has two versions, named `MAYDAY_ERROR` and `MAYDAY_ABORT`.

- The code generated for any Fortran subroutine that is not declared will contain an `IMPLICIT NONE` statement so this statement should not be used in the source code. As a result, all variables used in the subroutine must be explicitly declared else the code will not compile successfully.

8.10 Examples

8.10.1 Dot Product Example

This routine multiplies each point of one BaseFab with the corresponding point the other BaseFab over the input Box and puts the result in the input Real.

```
      subroutine DOTPRODUCT(
     &     CHF_REAL[dotprodout],
     &     CHF_CONST_FRA[afab],
     &     CHF_CONST_FRA[bfab],
     &     CHF_BOX[region])

      integer CHF_DDECL[i;j;k]
      integer nv,ncomp

      ncomp = CHF_NCOMP[afab]
```

```
      if(ncomp .ne. CHF_NCOMP[bfab]) then
          call MAYDAY_ERROR()
      endif

      dotprodout = zero
      do nv = 0, ncomp-1
         CHF_MULTIDO[region; i; j; k]

         dotprodout = dotprodout +
     &        afab(CHF_IX[i;j;k],nv)*
     &        bfab(CHF_IX[i;j;k],nv)

         CHF_ENDDO
      enddo

      return
      end
```

8.10.2 RealVect and IntVect Example

```
      subroutine realVectTest(CHF_REALVECT[foo])

      CHF_DTERM[
      foo(0) = 1.0;
      foo(1) = 2.0;
      foo(2) = 3.0]

      return
      end
      subroutine intVectTest(CHF_INTVECT[foo])

      CHF_DTERM[
      foo(0) = 1;
      foo(1) = 2;
      foo(2) = 3]

      return
      end
```

8.10.3 Laplacian Example

This subroutine produces a standard (3 point in one dimension, 5 point in two dimensions, and 7 point in three dimensions) discrete Laplacian of the input BaseFab over the input box.

```
      subroutine OPERATORLAP(
     &     CHF_FRA[lofphi],
     &     CHF_CONST_FRA[phi],
     &     CHF_BOX[region],
     &     CHF_CONST_REAL[dx])


      REAL_T dxinv,lphi
      integer n,ncomp,idir

      integer CHF_DDECL[ii,i;jj,j;kk,k]

      ncomp = CHF_NCOMP[phi]
      if(ncomp .ne. CHF_NCOMP[lofphi]) then
         call MAYDAY_ERROR()
      endif

      dxinv = one/(dx*dx)
      do n = 0, ncomp-1
         CHF_MULTIDO[region; i; j; k]

         lphi = zero
         do idir = 0, CH_SPACEDIM-1
         CHF_DTERM[
            ii = CHF_ID(idir, 0);
            jj = CHF_ID(idir, 1);
            kk = CHF_ID(idir, 2)]

            lphi = lphi +
     &           ( (phi(CHF_IX[i+ii;j+jj;k+kk],n)
     &           -  phi(CHF_IX[i   ;j   ;k   ],n))
     &           - (phi(CHF_IX[i   ;j   ;k   ],n)
     &           -  phi(CHF_IX[i-ii;j-jj;k-kk],n))
     &           )*(dxinv)
         enddo

         lofphi(CHF_IX[i;j;k],n) =  lphi

         CHF_ENDDO
      enddo

      return
      end
```

8.11 Landmines

This section is intended to point out some known uses of Chombo Fortran that will result in errors.

- Be aware that using C++ and Fortran together defeats most bounds checkers. If you step out of bounds in a Fortran, as a rule, your bounds checker will not save you. This holds for both Fortran and Chombo Fortran.

- Combining Fortran and Chombo Fortran in the same file is a bad idea. The Chombo Fortran parser keys on the word "subroutine," and dissects the argument list as described above. If ordinary Fortran subroutines are put into a Chombo Fortran file, the parser will fail to produce correct code. To use both Fortran and Chombo Fortran in the same application, put them into separate files. The Chombo makefile system recognizes files with ".F" extentions as Fortran and files with ".ChF" extentions as Chombo Fortran files.

- Send constants to Chombo Fortran (or plain Fortran, for that matter) using temporary variables. The C++ macros in Chombo Fortran are precisely that–macros. If you insert an explicit constant in a Chombo Fortran call, the macro will simply try to take the address of the explicit constant, resulting in undefined behavior. Say you want to send the number four to a Chombo Fortran routine. Here are both the incorrect and correct ways to do so.

```
//error! gets tranlated to senseless:
//myfunc_(&4);
FORT_MYFUNC(CHF_CONST_INT(4));

//correct.  address sent to fortran is legal.  This gets translated to
//myfunc_(&ivar);
int ivar = 4;
FORT_MYFUNC(CHF_CONST_INT(ivar));
```

- The arguments of a ChF Fortran macro must be enclosed in square brackets and separated by semicolons. Commas between the brackets will pass through to Fortran, as in the example in section 8.10.3 where `CHF_DDECL[ii,i;jj,j;kk,k]` translates to `ii,i,jj,j,kk,k` or `ii,i,jj,j`. The one apparent exception is `CHF_ID(`<*dim1*>`,`<*dim2*>`)`, but as noted above, `CHF_ID` is a matrix, not a macro.

Chapter 9

Chombo Debugging Tips

9.1 Overview of Chombo Debugging Tips

Chombo contains many complex datatypes and it can be difficult to look at one's data with simple gdb print statements. This chapter is intended to provide some tools for the user to be able to debug her applications with more facility.

Here are some general points.

- All the examples provided use gdb.
- To avoid ugly printouts while in gdb, one puts in her `~/.gdbinit` :

  ```
  set print static-members off
  set print pretty on
  ```

- To stop in a Fortran function, one usually has to add an underscore to the end of the name.

  ```
  <gdb prompt> break myfortranfunc_
  ```

- gdb is most useful when run within emacs. To do this, type

  ```
  Meta-x gdb
  ```

  while the buffer is in a file in the same directory as the executable (the input file or the makefile are popular choices). The advantages of this are many:

  - One gets emacs's nifty color scheme.
  - One interacts with the code directly. This means that when her code segfaults at a particular point, she will be looking at the offending line of code.

- One can set breakpoints by just being at the line she wants to stop at and typing control-x spacebar.

- Avoid using more than one gdb session in an emacs session. It gets confusing. One should fire up another emacs if one needs to debug two codes at once.

- If one wants to view some complex datatype (say class BagODonuts) for which Chombo does not yet have a nifty print function, she can write her own print function and call it using

  ```
  <gdb prompt>  call printBagODonuts(&myBagODonuts)
  ```

  All such functions will work most reliably if she always writes them to take pointers and define them using `extern ‘‘C’’`. Then gdb will not get confused about copy constructors and demangling. Chombo provides many examples of such functions. They live in

  ```
  Chombo/lib/src/BoxTools/DebugOut.H
  ```

- These functions are not "safe." There is no type checking in gdb. Anything one sends to the function will be interpreted as a pointer and it will try to run with it. It is very easy to seg fault one's gdb session if she

  - Forgets the & so the address you are sending is nonsense.
  - Mismatches the call with the data type.

9.2 Chombo Print Utilities

Chombo provides a bunch of functions to print out various datatypes in the debugger. To use these functions, one includes `DebugDump.H` in her code. The prototypes for the functions are in are in `Chombo/lib/src/BoxTools/DebugOut.H`. The functions are used in gdb as follows:

```
<gdb prompt>  call dumpIVS(&myIVS).
```

A list of some of them and what they print out follows:

- `void dumpLDF(const LevelData<FArrayBox>*  memLDF);`

  Dump a level's worth of data to standard out. Only use this for small data sets.

- `void dumpDBL(const DisjointBoxLayout* a_dblInPtr);`

  Dump a `DisjointBoxLayout` to standard out.

- `void dumpIVS(const IntVectSet* a_ivsInPtr)`

  Dump the points of an `IntVectSet` to standard out.

9.3 ArrayView and DatasetServer

ArrayView is one of the ways that one can view arrays in Chombo. It may be used to visualize either a single box's worth of data or data over an entire level. ArrayView is much faster than using the equivalent chomboVis functions but only displays the spreadsheet.

DatasetServer is an application that waits on a socket for information and displays spreadsheet when it receives the data. The first step in using ArrayView is to simply type on the command line:

```
<unix prompt>  DatasetServer2d.ex &
```

if one is working in two dimensions or

```
<unix prompt>  DatasetServer3d.ex &
```

if one is working in three dimensions. Both can be running simultaneously. If there is already a DatasetServer of a given dimension running, a friendly error message happens and the new DatasetServer gracefully exits.

ArrayView has two interface functions

- ArrayView(const FArrayBox* fabPtr) displays a FArrayBox as a spreadsheet. This can
- MultiArrayViewFab(const LevelData<FArrayBox>* ldfPtr) displays the as a spreadsheet the LevelData<FArrayBox>.

These are stand-alone functions that can be used either programmatically or inside a debugger. To view data using ArrayView, one must at some point include ArrayView.H in her code. Say one is using gdb to debug a Chombo application which contains a FArrayBox called fab and a LevelData<FArrayBox> called ldf. To view fab using ArrayView, she simply types at the gdb prompt:

```
<gdb prompt> call ArrayView(&fab)
```

and this will display the data in a spreadsheet format. To view ldf using ArrayView, she simply types at the gdb prompt:

```
<gdb prompt> call MultiArrayViewFab(&ldf)
```

and this will display the data in a spreadsheet format. Programmatically, the ArrayView functions work as any other functions do. One simply includes the header and calls the function:

```
#include "ArrayView.H"

 LevelData<FArrayBox> ldf(...);
 ...
 MultiArrayFab(&ldf);
```

This will display an spreadsheet every time this code is hit. When using multiple processors, every processor will try to display a spreadsheet. We advise that one use ArrayViewLDF (described below) when debugging using multiple processors.

For parallel applications, there is one more ArrayView function to describe

- ArrayViewLDF(const LevelData<FArrayBox>* ldfPtr) first copies all data in ldfPtr to processor 0 then displays the data as a spreadsheet.

This function is designed for parallel debugging and can only be used programmatically. An example of how to use this function is described below:

```
#include "DatasetClient.H"

 LevelData<FArrayBox> ldf(...);
 ...
 ArrayViewLDF(&ldf);
```

This will pop up a spreadsheet with the data from ldf every time this bit of code is hit. Only processor 0 displays the spreadsheet.

9.4 pout()

In Chombo, the pout() function is used in place of the std::cout output stream object. defined in header parstreamḢ

In serial this function returns std::cout. In parallel, this creates a file called pout.n where n is the procID() of the given processor. Output is then directed to these files. This keeps the output from different processors from getting all jumbled up. Used just as one would use a standard output stream.

```
  if(verbose >= 3)
  {
    pout()<<''In such-and-such piece of code \n''
          <<''Value of var == ``<<var<<std::endl;
  }
```

9.5 Memory Tracking

Chombo provides a simply internal memory tracking facility.

from memtrack.H:

```
class Memtrack
{
public:
  static void ReportUnfreedMemory(ostream& os);

/// calls ReportUnfreedMemory with pout()
  static void UnfreedMemory();

  static void memTrackingOn();

  static void memtrackingOff();

  static void overallMemoryUsage(long long& currentTotal,
                                 long long& peak);

};
```

This is compiled into Chombo and turned on by default when the compiler macro `ENABLE_MEMORY_TRACKING` is defined.

`ReportUnfreedMemory` produces a breakdown of current memory usage broken down by Chombo class. This does not include the system image of the program itself, or stack usage. This will also not include memory allocated by the user, just memory allocated by Chombo functions. RTTI is used in some places to indentify the type of objects held in a Chombo Vector template container.

example output of an `UnfreedMemory()` call:

```
Vector 3Box: 56 bytes (0 Mb)
Vector 5Entry: 252 bytes (0 Mb)
Vector Ui: 28 bytes (0 Mb)
Vector i: 16 bytes (0 Mb)
---------------------------------
Total Unfreed : 352 bytes (0 Mb)
peak memory usage: 360 bytes (0 Mb)
```

Bibliography

[ABC94] A. S. Almgren, T. Buttke, and P. Colella. A fast adaptive vortex method in three dimensions. *J. Comput. Phys.*, 113(2):177–200, 1994.

[ABC$^+$98] A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. J. Welcome. A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations. *J. Comput. Phys.*, 142(1):1–46, May 1998.

[BB86] M. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Trans. Comp.*, 1986.

[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.

[BBSW94] J. B. Bell, M. J. Berger, J. S. Saltzman, and M. Welcome. A three–dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM Journal on Scientific Computing*, 15:127–138, 1994.

[BC89] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

[Bet98] Matthew Tyler Bettencourt. *A Block-Structured Adaptive Steady–State Solver for the Drift–Diffusion Equations*. PhD thesis, Dept. of Mechanical Engineering, Univ. of California, Berkeley, May 1998.

[BO84] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53:484–512, March 1984.

[BR91] M. J. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions Systems, Man, and Cybernetics*, 21(5):1278–1286, 1991.

[CDW99] P. Colella, M. Dorr, and D. Wake. Numerical solution of plasma-fluid equations using locally refined grids. *J. Comput. Phys.*, 152:550–583, 1999.

[Cru91] W. Y. Crutchfield. Load balancing irregular algorithms. Technical Report UCRL-JC-107679, LLNL, July 1991.

[CW93] W. Y. Crutchfield and M. Welcome. Object-oriented implementation of adaptive mesh refinement algorithms. *Scientific Programming*, 2(4):145–156, 1993.

[FBK96] Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Flexible communication schedules for block structured applications. In *Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, Santa Barbara, California, August 1996.

[HPC+99] L. H. Howell, R. B. Pember, P. Colella, J. P. Jessee, and W. A. Fiveland. A conservative adaptive-mesh algorithm for unsteady, combined-mode heat transfer using the discrete ordinates method. *Numerical Heat Transfer, Part B: Fundamentals*, 35:407–430, 1999.

[HT97] R. Hornung and J. A. Trangenstein. Adaptive mesh refinement and multilevel iteration for flow in porous media. *J. Comput. Phys.*, 136(2):522–545, September 1997.

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

[JFH+98] J. P. Jessee, W. A. Fiveland, L. H. Howell, P. Colella, and R. B. Pember. An adaptive mesh refinement algorithm for the radiative transport equation. *J. Comput. Phys.*, 139(2):380–398, January 1998.

[KB96] Scott R. Kohn and Scott B. Baden. Irregular coarse-grain data parallelism under lparx. *J. Scientific Programming*, 1996.

[Mar98] Daniel Francis Martin. *An Adaptive Cell-Centered Projection Method for the Incompressible Euler Equations*. PhD thesis, University of California, Berkeley, 1998.

[MC96] D. F. Martin and K. L. Cartwright. Solving Poisson's equation using adaptive mesh refinement. *Technical Report UCB/ERI M96/66 UC Berkeley*, 1996.

[PBC+95] R. B. Pember, J. B. Bell, P. Colella, W. Y. Crutchfield, and M. L. Welcome. An adaptive Cartesian grid method for unsteady compressible flow in irregular regions. *J. Comput. Phys.*, 120:278–304, 1995.

[PHB+98] R. B. Pember, L. H. Howell, J. B. Bell, P. Colella, W. Y. Crutchfield, W. A. Fiveland, and J. P. Jessee. An adaptive projection method for unsteady, low mach number combustion. *Combustion Science and Technology*, 140:123–168, 1998.

[RBL+99] Charles A. Rendleman, Vincent E. Beckner, Mike Lijewski, William Crutchfield, and John B. Bell. Parallelization of structured, hierarchical adaptive mesh refinement algorithms. *Computing and Visualization*, 1999.

[TF89] M. C. Thompson and J. H. Ferziger. An adaptive multigrid technique for the incompressible Navier-Stokes equations. *J. Comput. Phys.*, 82(1):94–121, May 1989.