Chombo Software Package for AMR Applications Design Document

M. Adams
P. Colella
D. T. Graves
J. N. Johnson
H. S. Johansen
N. D. Keen
T. J. Ligocki
D. F. Martin
P. W. McCorquodale
D. Modiano
P. O. Schwartz
T. D. Sternberg
B. Van Straalen

Applied Numerical Algorithms Group
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA

March 21, 2014

Disclaimer

Contents

1 Introduction **7**
- 1.1 Requirements . . . 9
- 1.2 Installation . . . 10
  - 1.2.1 Downloading Chombo . . . 10
  - 1.2.2 Configuring Chombo for a Particular System . . . 11
  - 1.2.3 Compiling Chombo's Libraries . . . 13
  - 1.2.4 Compiling and running Chombo's test programs . . . 14
  - 1.2.5 Compiling and running Chombo's example applications . . . 15
  - 1.2.6 Building an application using Chombo . . . 15
    - 1.2.6.1 Using the Chombo application makefiles . . . 16
    - 1.2.6.2 Using an existing application makefile . . . 17
  - 1.2.7 Building Chombo's Doxygen documentation . . . 18
  - 1.2.8 Namespace . . . 18

2 BaseTools and BoxTools **20**
- 2.1 AMR Spatial Discretization . . . 20
- 2.2 Points, Regions and Rectangular Arrays . . . 23
  - 2.2.1 Class `IntVect` . . . 23
  - 2.2.2 Class `Box` . . . 23
  - 2.2.3 Class IntVectSet . . . 25
  - 2.2.4 `Box` and `IntVectSet` Iterators . . . 26
  - 2.2.5 Class Interval . . . 26
  - 2.2.6 Rectangular arrays . . . 26
    - 2.2.6.1 Aliasing . . . 29
- 2.3 Class ProblemDomain . . . 29
- 2.4 Data on Unions of Rectangles . . . 36
  - 2.4.1 Introduction . . . 36
  - 2.4.2 Layouts . . . 36
    - 2.4.2.1 Class BoxLayout . . . 36
    - 2.4.2.2 Class DisjointBoxLayout . . . 38
  - 2.4.3 Templated Data Holders . . . 39
    - 2.4.3.1 Class LayoutData . . . 40

2.4.3.2 Class BoxLayoutData . . . 40
2.4.3.3 Class LevelData . . . 42
2.4.3.4 Aliasing . . . 43
2.4.4 Iterators . . . 44

3 AMRTools 46
3.1 Multilevel Operators. . . . 46
3.1.1 Interlevel Transfer Operators . . . 49
3.1.1.1 Conservative Averaging. . . . 49
3.1.1.2 Piecewise Constant Interpolation. . . . 49
3.1.1.3 Piecewise Linear Interpolation. . . . 49
3.1.2 Coarse-Fine Boundary Interpolation . . . 50
3.1.2.1 Piecewise Linear Interpolation . . . 50
3.1.2.2 Quadratic Coarse-Fine Boundary Interpolation . . . 51
3.1.2.3 Level Divergence, Composite Divergence, and Refluxing 54
3.2 C++ Classes for Two-Level Operators . . . 57
3.2.1 Class `CoarseAverage` . . . 57
3.2.2 Class `CoarseAverageFace` . . . 58
3.2.3 Class `FineInterp` . . . 58
3.2.4 Class `FineInterpFace` . . . 59
3.2.5 Class `PiecewiseLinearFillPatch` . . . 60
3.2.6 Class `PiecewiseLinearFillPatchFace` . . . 62
3.2.7 Class `QuadCFInterp` . . . 63
3.2.8 Class `LevelFluxRegister` . . . 64
3.2.9 Class `LevelFluxRegisterEdge` . . . 67
3.3 Class BRMeshRefine . . . 69
3.3.1 domainSplit . . . 75
3.4 Multilevel Utilities . . . 75
3.4.1 Function `computeSum` . . . 75
3.4.2 Function `computeNorm` . . . 76

4 AMRElliptic Algorithm and Implementation 78
4.1 Multigrid Algorithm . . . 78
4.2 The AMR Elliptic User Interface . . . 78
4.2.1 Overview . . . 81
4.3 Operator Interfaces . . . 82
4.3.1 Class `LinearOp` . . . 82
4.3.2 Class `MGLevelOp` . . . 83
4.3.3 Class `MGLevelOpFactory` . . . 84
4.3.4 Class `AMRLevelOp` . . . 84
4.3.5 Class `AMRLevelOpFactory` . . . 86
4.4 Solver Templates . . . 86

4.4.1 Class LinearSolver . . . 86
4.4.2 Class BiCGStabSolver . . . 87
4.4.3 Class MultiGrid . . . 88
4.4.4 Class AMRMultiGrid . . . 88
4.5 The MultilevelLinearOp<T> class . . . 89
4.6 Elliptic Examples . . . 92
4.6.1 AMRPoisson . . . 93
4.6.1.1 AMRPoisson Factory Interface . . . 93
4.6.1.2 Code fragment . . . 94
4.6.2 ResistivityOp . . . 95
4.6.2.1 ResistivityOp Factory Interface . . . 95
4.6.3 ViscousTensorOp . . . 96
4.6.3.1 ViscousTensorOp Factory Interface . . . 97
4.6.4 Boundary Condition Interface . . . 97
4.7 Parabolic Equations – the TGA scheme . . . 99
4.7.1 The TGAHelmOp and LevelTGAHelmOp classes . . . 100
4.7.2 The AMRTGA class . . . 101
4.7.3 The BaseLevelTGA and LevelTGA classes . . . 102
4.8 TGA addendum for EB and non-unity identity coefficients . . . 105
4.9 TGA with time-dependent operator coefficients . . . 106
4.9.1 Implicit TGA update for ϕ . . . 106
4.9.1.1 Embedded boundary considerations . . . 108
4.9.2 Conservative calculation of $L(\phi)$ . . . 108
4.9.3 Setting the time centering of a LevelTGAHelmOp . . . 108

5 AMRTimeDependent 110
5.1 Hyperbolic Systems of Conservation Laws . . . 110
5.2 Classes AMR and AMRLevel . . . 113
5.2.1 Class structure . . . 114
5.2.2 Class AMR . . . 114
5.2.3 Class AMRLevel . . . 116
5.2.4 Class AMRLevelFactory . . . 118
5.3 AMRTimeDependent Example: Advection-Diffusion . . . 119
5.3.1 AdvectionDiffusion-Specific Classes . . . 119
5.3.2 Usage Pattern . . . 120
5.3.3 AMRLevelAdvectDiffuseFactory Interface . . . 122
5.3.4 LevelAdvect Interface . . . 123

6 HDF5 I/O with Chombo 126
6.1 HDF5 I/O . . . 126
6.1.1 Class HDF5Handle . . . 126
6.1.2 Class HDF5HeaderData . . . 127

6.1.3 HDF5 I/O for BoxLayoutData . . . 128
6.1.4 HDF5 Out-Of-Core readers . . . 129
6.2 AMR I/O routines . . . 131
6.2.1 Function WriteAMRHierarchyHDF5 . . . 131
6.2.2 Function ReadAMRHierarchyHDF5 . . . 132
6.3 Other HDF5 I/O functions . . . 133
6.3.1 Functions writeFAB and writeFABname . . . 133
6.3.2 Functions writeLevel and writeLevelname . . . 134
6.3.3 Functions writeDBL and writeDBLname . . . 135

7 Using PETSc in Chombo 136
7.1 Building Chombo with PETSc . . . 136
7.2 PETSc Composite Grid Classes . . . 136
7.3 PETSc Solver Usage . . . 137
7.3.1 Making a PETSc Matrix . . . 137
7.3.2 Making a PETSc Solver . . . 138
7.3.3 Making an operator . . . 138

8 Parallel Programming with Chombo 140
8.1 Initialization and Scoping . . . 140
8.2 Overview of Chombo Data Parallelism . . . 141
8.3 Box-processor assignment . . . 141
8.4 LoadBalance . . . 142
8.5 Broadcast and Gather . . . 143
8.5.1 linearIn, linearOut, linearSize . . . 145

9 Chombo Fortran 146
9.1 Introduction . . . 146
9.2 ChF Fortran macros . . . 147
9.3 dimension-handling macros . . . 147
9.4 Declaration macros . . . 150
9.5 Access macros . . . 152
9.6 C++ macros . . . 153
9.7 Declaration macros . . . 154
9.8 Language support . . . 157
9.9 Examples . . . 158
9.9.1 Dot Product Example . . . 158
9.9.2 RealVect and IntVect Example . . . 159
9.9.3 Laplacian Example . . . 159
9.9.4 Internal Procedure Example . . . 161
9.10 Landmines . . . 162

10 Multidimensional Chombo **164**
10.1 Namespace implementation . . . 164
10.2 Transdimensional utilities . . . 164
10.2.1 struct `SliceSpec` . . . 164
10.2.2 Slicing . . . 165
10.2.3 Injection . . . 166
10.2.4 The `ReductionCopier` class . . . 167
10.2.5 The `SpreadingCopier` class . . . 168
10.2.6 The `SumOp` class . . . 169
10.2.7 The `SpreadingOp` class . . . 170
10.3 Phase field example . . . 173
10.3.1 Algorithm . . . 173
10.4 Building the multidim example . . . 173

11 Chombo Debugging and Performance Tools **175**
11.1 Overview of Chombo Debugging Tips . . . 175
11.2 Chombo Print Utilities . . . 176
11.3 Viewing data objects with VisIt from gdb . . . 177
11.4 pout() . . . 177
11.5 Memory Tracking . . . 178
11.6 TraceTimer . . . 178
11.6.1 Auto hierarchy . . . 180
11.6.2 Finer control . . . 180

12 Troubleshooting **182**

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 [6] 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 [7] [25] [4] [2] [22] [14] [1] [16] [23] [15] [10] .

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 [12] [24]. The Chombo package at the present time consists of the following components:

- The BaseTools Library contains helper classes and functionality which is indepen-

dent of spatial dimension, such as `Vector` and `List` container classes, etc. The BaseTools library need not be specified in the Chombo makefiles, and is always included and linked to when compiling using the Chombo make system. In earlier versions of Chombo, the contents of the BaseTools library resided in the BoxTools library.

- 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 MultiDim library includes functions which support Chombo in a multidimensional environment, as described in section 10.

- Support for embedded boundary (EB) discretizations and algorithms is found in the EBAMRElliptic, EBAMRTimeDependent, EBAMRTools, EBTools, and Workshop libraries, which are described in a separate set of EB design documents. These are only built if `EB=TRUE` is specified during the build process.

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, sparse irregular calculations have generally been implemented 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 [13] at UCSD/SDSC, and we expect ultimately that our parallel support will be built on top of KeLP.

- We are trying to enable use of parts of Chombo which others might find useful by pursuing a component-based design approach. For example, it is possible to use our implementation of the Berger-Rigoutsos [8] 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 discussing 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://www.hdfgroup.org/downloads/`

  We recommend using HDF5 version 1.6.x. We suggest that it be configured with the option `"--enable-production"`. Chombo does work with HDF5 1.8.x but needs to be compiled with the 1.6 compatibility flags from HDF5. One can either configure and build HDF5 with `--with-default-api-version=v16` or include `-DH5_USE_16_API` in the compilation flags during the Chombo build process as detailed in section 1.2.2.

- A functioning MPI-1.2 (or higher) 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. If hybrid parallelism is desired the compiler should also support OpenMP API. Chombo has been compiled and tested with
  - GNU g++ 3.32 or higher.
  - Intel icc v9 or higher on Linux. There seems to be some issues with v10 and floating point errors.
  - IBM xlC version 7 or later.
- At least a Fortran 77 compiler is required. Chombo has been compiled and tested with:
  - g95, gfortran, g77
  - Intel ifc on Linux
  - IBM xlf, xlf90 on AIX
  - Portland Group pgf77, pgf90
  - HP/Compaq/DEC f77,f90,f95
  - Sun f77,f90
- 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

1.2.1 Downloading Chombo

Previous versions of Chombo were distributed as compressed tar files. The Chombo team is now using a more continuous distribution process using branches and the Subversion revision control system.

To *check out* Chombo you will use the `svn` Subversion client. Subversion is already installed on most unix-like operating systems. In order to access the Chombo svn repository, first go to `https://anag-repo.lbl.gov` and register to establish a username and password. You will then be redirected to a page with specific download instructions. There is also a link on `https://anag-repo.lbl.gov` which has the same information without registering. Future changes to the Chombo release may be obtained through svn updates.

1.2.2 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`
- copy (or create a symbolic link to) the system-specific customization file from `Chombo/lib/mk/local` if you are using a computer we already use (Franklin, Hopper, Jaguar, etc.)

The first two option produces a customization file with no variables defined, but with the important variables documented in comments. **The resulting `Make.defs.local` file must be edited to set the HDFINCFLAGS and HDFLIBFLAGS variables (see below) if either of these options is used.** The system-specific files in the second 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 (=1, 2, 3, 4, 5, or 6). The default is 2. There is limited functionality for high dimensions (`DIM > 3`).

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

`DEBUG` determines whether to compile with a symbol table (=TRUE) or not (=FALSE). The default is TRUE.

`OPT` determines whether to compile with optimization (=TRUE) or not (=FALSE). The default is FALSE. OPT can also be set to HIGH in which case asserts are removed from the code and FArrayBox memory is initialized to zero (instead of a large positive value) during memory allocation. It is recommended that OPT=HIGH not be used unless absolutely necessary.

`PROFILE` determines whether to compile 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 gfortran, 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.

`OMP` determines if hybrid parallelism is in use (=TRUE). 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. For serial builds against a standard installation of HDF5. It is no longer necessary to use the `-DH5_USE_16_API` flag.

`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 OPT=TRUE 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 OPT=TRUE (default is blank)
`cxxdbgflags`	options for C++ compiler and linker when DEBUG=TRUE (default is -g)
`cxxoptflags`	options for C++ compiler and linker when OPT=TRUE (default is -O)
`fdbgflags`	options for Fortran compiler when DEBUG=TRUE (default is -g)
`foptflags`	options for Fortran compiler when OPT=TRUE (default is -O)
`lddbgflags`	options for linker only when DEBUG=TRUE (default is blank)
`ldoptflags`	options for linker only when OPT=TRUE (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

1.2.3 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@hpcrdm.lbl.gov>.

1.2.4 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.

1.2.5 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/releasedExamples
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.[1]

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). The command:

```
cd Chombo/releasedExamples
make usage
```

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

1.2.6 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] Exception: the IBM xlC compiler complains about multiple definitions. Our experience has been that it is safe to ignore these complaints as long as the executable files (*.ex) are created and run successfully.

```
## 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 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
```

Figure 1.1: Sample Chombo example makefile

1.2.6.1 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.

Figure 1.1 shows an example of a "GNUmakefile" that builds an application with all of its code contained in a single directory. It is based on the GNUmakefile in `Chombo/releasedExamples/AMRPoisson/execCell`, with unnecessary lines removed for simplicity.

The first line defines the CHOMBO_HOME variable, which tells the rest of the makefiles where the `Chombo/lib` 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 is application-specific, defining the primary build target. For the example 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. Source files are identified for compilation by using a set of suffix rules; any files in the directory with common suffixes like "cpp", "F",

etc. will be compiled.[2] The next line specifies which Chombo libraries are needed, which will vary depending on the application. Note that most linkers require that the libraries be listed in the order they will be searched. For Chombo, that generally means that the more basic libraries (like `BoxTools`) must come *after* libraries (like `AMRTools`) which reference them. The `XTRALIBFLAGS` variable should be defined to specify additional libraries if needed. 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 involved approach allows the Chombo makefile system to build an application with source code in multiple directories. The file

Chombo/releasedExamples/AMRGodunov/execPolytropic/GNUmakefile

demonstrates how this is done. In general, the GNUmakefile looks similar to the description above. The major difference is the addition of a `src_dirs` variable which lists the other directories containing source code, along with a `base_dir` variable (usually ".") which defines the directory where the source file containing `main` is located. The final line includes another Chombo makefile (`Chombo/lib/mk/Make.example`) 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.

1.2.6.2 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:

[2] The complete list of suffixes may be found in `Chombo/lib/mk/Make.rules`

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

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

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.

1.2.7 Building Chombo's Doxygen documentation

C++ header files in Chombo libraries are annotated with comments which are compatible with the Doxygen (`http://www.doxygen.org`) software documentation system. To create html documentation of the Chombo libraries,

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

which will install doxygen-generated html documentation in `Chombo/lib/doc/doxygen`.
To access it once built, point a web browser at `Chombo/lib/doc/doxygen/html/index.html`.
To include EB documentation, add `EB=TRUE` to `make doxygen`.

Doxygen documentation of the current release is also available online at `http://chombo.lbl.gov`.

1.2.8 Namespace

Chombo can be embedded in its own C++ namespace. While this was part of previous versions of the code, it is more relevant now that Chombo is part of larger development projects. This is controlled by the `NamespaceHeader.H` and `NamespaceFooter.H` files that are included in every header and source file in Chombo.

By default, Chombo builds with `NAMESPACE=FALSE` which turns these include files into empty files, and they have no effect. If you build Chombo with

```
>make NAMESPACE=TRUE
```

Then every class and function defined between `NamespaceHeader.H` and `NamespaceFooter.H` will be placed in the `Chombo` namespace. In practice, this means that one should place

```
#include "NamespaceHeader.H"
```

after all other include files, but before any C++ code in a file, and then

```
#include "NamespaceFooter.H"
```

after all C++ code in a file (but within any multiple-inclusion guards, if in a C++ header file). Alternatively,

```
#include "UsingNamespace.H"
```

will ensure that the C++ code in a given file is not embedded in a Chombo namespace, but will be able to use Chombo code which *is* in the Chombo namespace. This is most useful in an application code.

If instead a user uses

```
>make NAMESPACE=MULTIDIM
```

Then Chombo creates a dimension-dependent namespace `Chombo::XD` where X is the dimensionality in which Chombo is being built. This can be used to build Chombo in multiple dimensions and use them together in the same application. (see Chapter 10.)

Chapter 2

BaseTools and 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 $|\boldsymbol{i}| = \max_{d=0...\mathbf{D}-1}(|i_d|)$, $\Gamma + \boldsymbol{i}$ 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

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

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

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

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 $\frac{h^l}{h^{l+1}} = n^l_{ref}$. 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*

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

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., $\psi : \Omega^{l,\boldsymbol{v}} \to \mathbb{R}^m$. 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

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

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 $\varphi : \Gamma \to \mathbb{R}^m \Leftrightarrow$ 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 $\varphi : \Omega \to \mathbb{R}^m \Leftrightarrow$ 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 Class `IntVect`

`IntVects` represent points in the rectangular lattice $\mathbb{Z}^{\mathbf{D}}$.
Operations on `IntVect`s. In the following definitions $\boldsymbol{i}$, $\boldsymbol{j}$ are `IntVects` 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 `IntVects` 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 `min`$(\boldsymbol{i}, \boldsymbol{j})$, `max`$(\boldsymbol{i}, \boldsymbol{j})$ operators.

• 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 d 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 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, $0 \leq d < \mathbf{D}$.

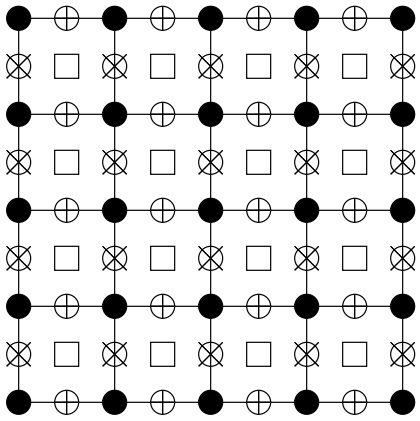

Figure 2.2: Vertex (•), cell (□) and face (⊕ and ⊗) sites on a grid.

• Constructors. $B(\boldsymbol{i}_1, \boldsymbol{i}_2, \boldsymbol{v} = \texttt{Zero})$ 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 Class 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 Class 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.

• Data Modification Functions. $\mathcal{A}$.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.

• Domain Modification Functions $\mathcal{A}$.shift$(\boldsymbol{i})$ changes the Box over which $\mathcal{A}$ is defined to $\mathcal{A}$.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.

Note: All FArrayBox objects are initialized upon creation, using setVal with an argument of 1.23456789e10, whether the Chombo libraries are compiled with debugging on or off.

• Pointwise Arithmetic Operators $\mathcal{A}_1 \oplus = \mathcal{A}_2$, $\oplus \in \{+, -, *, /\}$, 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. $\mathcal{A}.\mathtt{abs}()$ 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.

• Reduction Operators $\mathcal{A}.\mathtt{sum}(s)$, $\mathcal{A}.\mathtt{min}(s)$, $\mathcal{A}.\mathtt{max}(s)$, return real values containing the sum, minimum, and maximum of the values of the s-th component of $\mathcal{A}$. $\mathcal{A}.\mathtt{minIndex}(s)$, $\mathcal{A}.\mathtt{maxIndex}(s)$ 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.

• Mask Functions $\mathcal{A}.\mathtt{maskLT}(M, a, s)$ 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 $\mathcal{A}.\mathtt{dataPtr}()$. 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 aliased BaseFab can only have the same `Box` domain as the original.

2.3 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 appropriately 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 conditions 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 [5], Kohn and Baden [18], Rendleman, et. al. [24], and others have used this technique. Our API is derived from joint work with Baden to develop an abstract version of KeLP [13]. 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 processors. 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 Class 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:

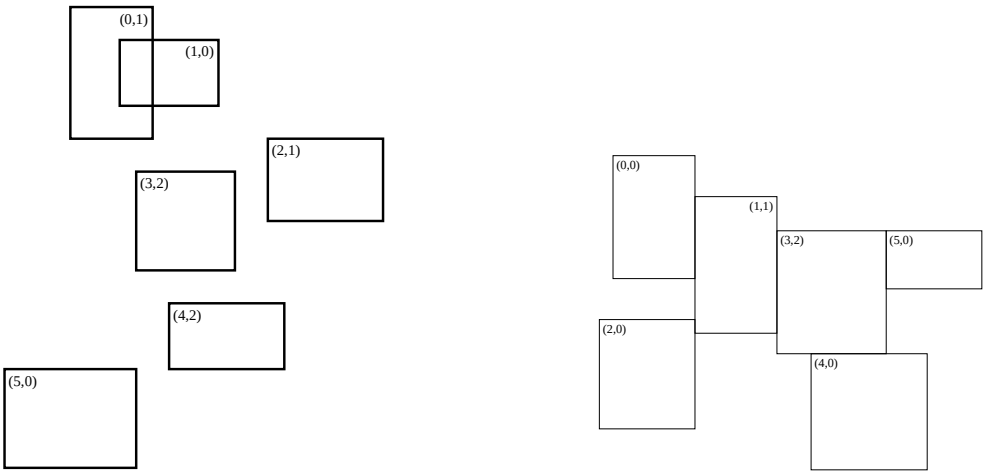

Figure 2.3: Left: Example of a BoxLayout. The first integer in the pair identifies the Box, and the second integer the processor ID. In this case we have the following assignments. Processor 0: B_1, B_5. Processor 1: B_0, B_2. Processor 2: B_3, B_4. Note that B_0 and B_1 have a non-empty intersection.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

• Construction

```
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 Class 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 Class 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 dataIterator() 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(). dataIterator 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 Class 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 Class 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 is associated to the Box grown from B 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();
```

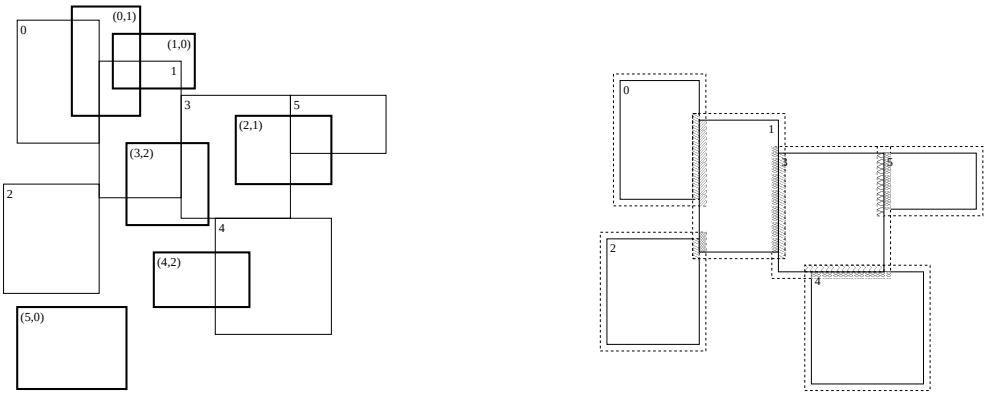

Figure 2.4: Left: `CopyTo` example. This figure illustrates copying from a `LevelData` built on the `DisjointBoxLayout` in figure 2.3 to a `BoxLayoutData` built on top of the `BoxLayout` in figure 2.3. A single call to `Copy` would perform the following data movements: Data from B_0 copied to B'_0. Data 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`.

• Construction. 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)).

• 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. `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.

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

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 $\Omega_k^l \in \mathcal{R}(\Omega^l)$

$$\begin{aligned} \varphi_{\boldsymbol{i}}^{l,ext} &= \varphi_{\boldsymbol{i}}^l \text{ on } \Omega_{valid}^l \\ &= I(\varphi^{comp}, \boldsymbol{x}_0^l + \boldsymbol{i}h^l) \text{ on } \mathcal{G}(\Omega_k^l, S) \cap \Gamma^l - \Omega_{valid}^l \end{aligned}$$

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

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

$$\psi^l = \psi(\boldsymbol{x}_0^l + \boldsymbol{i}h^l) \text{ on } \Gamma^l$$

and composite array

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

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

$$\begin{aligned} \tau_{\boldsymbol{i}} &\equiv L^{comp}(\psi^{comp})_{\boldsymbol{i}} - \mathcal{L}(\psi)(\boldsymbol{x}_0^l + \boldsymbol{i}h^l) \\ &= \sum_{|\boldsymbol{s}|\leq \mathcal{S}} c_{\boldsymbol{i},\boldsymbol{s}} \psi_{\boldsymbol{i}+\boldsymbol{s}} - \mathcal{L}(\psi)(\boldsymbol{x}_0^l + \boldsymbol{i}h^l) \\ &- \sum_{\substack{|\boldsymbol{s}|\leq \mathcal{S} \\ \boldsymbol{i}+\boldsymbol{s} \notin \Omega_{valid}^l}} c_{\boldsymbol{i},\boldsymbol{s}} (\psi_{\boldsymbol{i}+\boldsymbol{s}} - I(\psi^{comp}, \boldsymbol{x}_0^l + (\boldsymbol{i}+\boldsymbol{s})h^l)) \end{aligned}$$

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 $I(\varphi^{comp}, \boldsymbol{x}_0^l + \boldsymbol{i}h)$ 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.

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

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

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

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

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

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

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:

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

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 $\Omega^c : \varphi^c = Average(\varphi^f, n_{ref})$ 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).

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

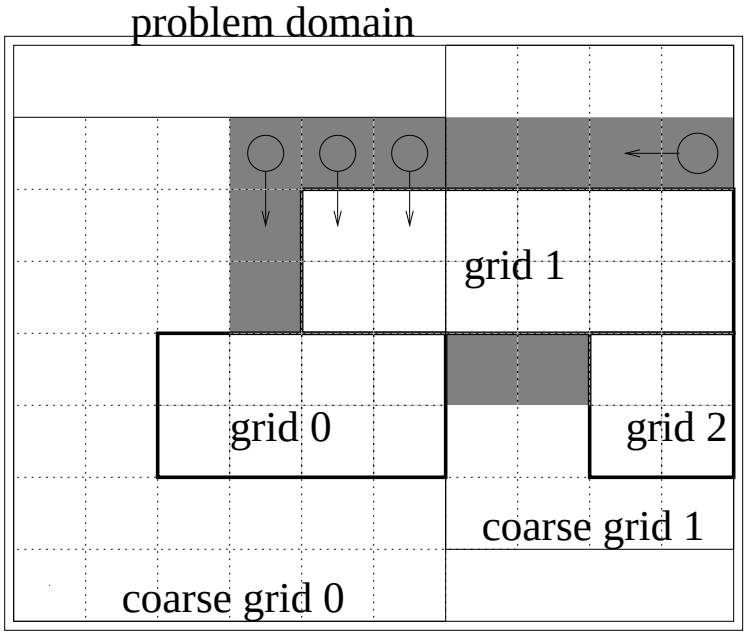

Figure 3.1: Interpolation on the coarse grid. One-sided differences are used at cells marked with circles.

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

(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

$$\psi^f_{\boldsymbol{i}} = \psi(\boldsymbol{x}^f_0 + \boldsymbol{i}h^f) \text{ on } \Omega^f$$
$$\psi^{c,valid}_{\boldsymbol{i}} = \psi(\boldsymbol{x}^c_0 + \boldsymbol{i}h^c) \text{ on } \Omega^{c,valid}$$

then

$$\tilde{\psi}^f_{\boldsymbol{i}^f} = \psi(\boldsymbol{x}^f_0 + \boldsymbol{i}h^f) + O(h^2) \text{ for } \boldsymbol{i} \in \tilde{\Omega}^f - \Omega^f$$

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 $O(h^2)$, 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 $\vec{G}^f = (G^f_0, \ldots G^f_{\mathbf{D}-1})$ 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.

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

Then, for each $\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d$ such that both $\boldsymbol{i}, \boldsymbol{i} + \boldsymbol{e}^d \in \tilde{\Omega}^f_k$ 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 $\tilde{\Omega}^f_k \cap \Omega^f$, the values for $\tilde{\varphi}$ will be given by $\tilde{\varphi}_{\boldsymbol{i}} = \varphi^f_{\boldsymbol{i}}$. The values for the remaining points in $\tilde{\Omega}^f_k - \Omega^f$ will be obtained by interpolating data from φ^f and φ^c.

To perform this interpolation, we first observe that, given $\boldsymbol{i} \in \tilde{\Omega}^f_k - \Omega^f$, there is a unique choice of $\pm$ and d, such that $\boldsymbol{i} \mp \boldsymbol{e}^d \in \Omega^f_k$. 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.

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

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

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

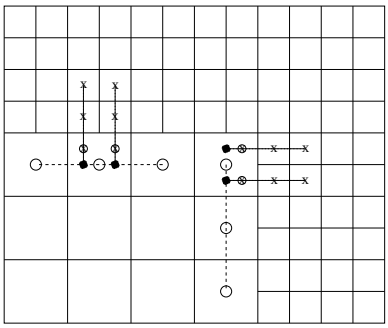

Figure 3.2: Interpolation at a coarse-fine interface. Left stencil is the usual stencil. Right stencil is the modified interpolation stencil; since the upper coarse cell is covered by a fine grid, use shifted coarse grid stencil (open circles) to get intermediate values (solid circles), then perform 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)

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

$$(D^{2,d'd''}\varphi^c)_{\boldsymbol{i}} = \begin{cases} \frac{1}{N_{valid}} \sum_{s'=\pm1}\sum_{s''=\pm1}(D^{d'd''}\varphi^c)_{\boldsymbol{i}+\frac{1}{2}s'\boldsymbol{e}^{d'}+\frac{1}{2}s''\boldsymbol{e}^{d''}} & \text{if } N_{valid} > 0 \\ 0 & \text{otherwise} \end{cases}$$

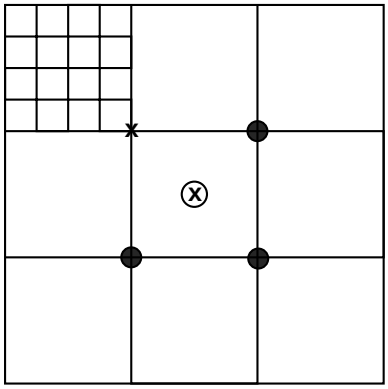

Figure 3.3: Mixed-derivative approximation illustration. The upper-left corner is covered by a finer level so the mixed derivative in the upper left (the uncircled x) has a stencil which extends into the finer level. We therefore average the mixed derivatives centered on the other corners (the filled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

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

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 $\Omega_{valid}^{c,\boldsymbol{e}^d}$, 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:

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

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 [17]. 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_{valid}^l, 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 construct 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 Class `CoarseAverage`

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

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

  Arguments:

  - `a_fine_domain` (not modified): the fine-level grids (valid region).
  - `a_crse_domain` (not modified): the coarse-level grids (valid region).
  - `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 Class `CoarseAverageFace`

Similar to `CoarseAverage`, but averaging face-centered data instead of cell-centered data. Data on coarse-level faces is computed as the average of the overlying fine-level faces. Both arithmetic and harmonic averaging are supported.

- `void define(const DisjointBoxLayout& a_fineGrids,`
  `int a_nComp, int a_nRef)`

- `void averageToCoarse(LevelData<FluxBox>& a_coarse_data,`
  `const LevelData<FluxBox>& a_fine_data)`

  averages fine-level data to coarse level using arithmetic averaging

- `void averageToCoarseHarmonic(LevelData<FluxBox>& a_coarse_data,`
  `const LevelData<FluxBox>& a_fine_data)`

  averages fine-level data to coarse level using harmonic averaging

3.2.3 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_fine_problem_domain);

  void
  define(const DisjointBoxLayout& a_fine_domain,
         int a_numcomps,
         int a_ref_ratio,
         const Box& a_fine_problem_domain)
  ```

  Arguments:

  - `a_fine_domain` (not modified): grids (valid region) on 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_fine_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, bool a_averageFromDest=false););`

  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.
  - `a_averageFromDest`: if true, first fill a projection of the fine grid with averaged values from `a_fine_data` before filling with coarse data and performing interpolation. This is often useful in operations like flattening an AMR hierarchy to a single resolution, in which the fine data may not be properly nested within the coarse data grids. Default is `false`.

- `void pwcinterpToFine(LevelData<FArrayBox>& a_fine_data, const LevelData<FArrayBox>& a_coarse_data, bool a_averageFromDest=false););`

  Replaces fine data by piecewise-constant 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.
  - `a_averageFromDest`: if true, first fill a projection of the fine grid with averaged values from `a_fine_data` before filling with coarse data and performing interpolation. This is often useful in operations like flattening an AMR hierarchy to a single resolution, in which the fine data may not be properly nested within the coarse data grids. Default is `false`.

3.2.4 Class `FineInterpFace`

This class fills face-centered data in the valid region of a level of data by piecewise linear interpolation from face-centered data on a coarser level of refinement. This interpolation is performed in two steps:

1. Data on fine-level faces which overlie coarse-level faces is interpolated using only the underlying co-planar coarse faces.

2. Data on fine-level faces which do not overlie coarse-level faces is computed using linear interpolation in the normal direction between the two nearest fine-level faces which overlie coarse-level faces (and were filled in the previous step)

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

  Arguments:

  - `a_fine_domain`: the fine level domain
  - `a_numcomps`: the number of components
  - `a_ref_ratio`: the refinement ratio
  - `a_fine_problem_domain`: problem domain

- `void define(const DisjointBoxLayout& a_fine_domain, // the fine level domain`
  `const int& a_numcomps, // the number of components`
  `const int& a_ref_ratio, // the refinement ratio`
  `const ProblemDomain& a_fine_problem_domain); // problem domain`

  Arguments:

  - `a_fine_domain`: the fine level domain
  - `a_numcomps`: the number of components
  - `a_ref_ratio`: the refinement ratio
  - `a_fine_problem_domain`: problem domain on finest level

- `void interpToFine(LevelData<FluxBox>& a_fine_data,`
  `const LevelData<FluxBox>& a_coarse_data);`
  `bool a_averageFromDest=false);`

3.2.5 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

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

This can be useful, for example, when one has coarse-level data at two times ($t^{c,old}$ and $t^{c,new}$) and needs interpolated ghost cell data at an intermediate time $t^{fine} = t^{c,old} + \alpha(t^{c,new} - t^{c,old})$.

Ghost cells which lie inside the valid region of another fine grid are not filled. Also, 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_crse_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_crse_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): grids on the fine level.
  - `a_coarse_domain` (not modified): grids on the coarse level.
  - `a_num_comps` (not modified): number of components of state vector.
  - `a_crse_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.

3.2.6 Class `PiecewiseLinearFillPatchFace`

The `PiecewiseLinearFillPatchFace` class is similar to the `PiecewiseLinearFillPatch` class, but computes interpolated face-centered data to fill "ghost faces" around fine grids by interpolating from coarse-level face-centered data. This interpolation is performed in two steps:

1. Data on fine-level faces which overlie coarse-level faces is interpolated using only the underlying co-planar coarse faces.
2. Data on fine-level faces which do not overlie coarse-level faces is computed using linear interpolation in the normal direction between the two nearest fine-level faces which overlie coarse-level faces (and were filled in the previous step)

- ```
  void
  define(const DisjointBoxLayout& a_fine_domain,
         const DisjointBoxLayout& a_coarse_domain,
         int a_num_comps,
         const ProblemDomain& a_crse_problem_domain,
         int a_ref_ratio,
         int a_interp_radius)
  ```

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

  - `a_fine_domain` (not modified): grids on the fine level.
  - `a_coarse_domain` (not modified): grids on the coarse level.
  - `a_num_comps` (not modified): number of components of state vector.
  - `a_crse_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 faces to fill by interpolation.

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

  Fills fine-level ghost faces by linear interpolation.
  Arguments:

  - a_fine_data (modified): fine data whose ghost faces 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.

3.2.7 Class QuadCFInterp

The class QuadCFInterp interpolates data onto the ghost cells along the coarse-fine interface 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 for 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 problem domain at the fine 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.

3.2.8 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 problem 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_coarseDataIndex,
                       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) coincides 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_coarseDataIndex]` 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_coarseDataIndex (not modified): DataIndex which corresponds to which box in the coarse-level DisjointBoxLayout (a_dblCoarse in the define function) over 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 which are incremented by the flux data. Should have the same size as a_srcInterval.
- 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) covers 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 fine-level DisjointBoxLayout (a_dbl in the define function) over 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 which are incremented by the flux data.
  - 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>` which is modified by the refluxing operation.
  - `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.

3.2.9 Class `LevelFluxRegisterEdge`

In Magnetohydrodynamics (among other fields), there is a class of numerical schemes commonly known as "constrained transport" schemes in which the solenoidal property of a field is maintained by making use of the identity $div(curl(u)) == 0$. In MHD, for example, one can write the evolution of the magnetic field $\vec{B}$ in terms of the curl of the electric field $\vec{E}$, which ensures that the $div(\vec{B}) = 0$ down to roundoff. This is generally accomplished on a staggered mesh, in which $\vec{B}$ is face-centered and $\vec{E}$ is edge-centered.

At coarse-fine interfaces, one must perform a correction analogous to the reflux-divergence operation in order to maintain a divergence-free magnetic field. This correction is described, for example, in [3]. The `LevelFluxRegisterEdge` is a class designed to manage and apply this correction, and is analogous to the `LevelFluxRegister`. While the `LevelFluxRegister` corrects a cell-centered field with a "reflux-divergence" of face-centered fluxes, the `LevelFluxRegisterEdge` corrects a face-centered field with a "reflux curl" of edge-centered "fluxes".

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

  Defines this object

- `void setToZero()`

  Sets all registers to zero.

- ```
  void incrementCoarse(FArrayBox& a_coarseFlux,
                       Real a_scale,
                       const DataIndex& a_coarseDataIndex,
                       const Interval& a_srcInterval,
                       const Interval& a_dstInterval)
  ```

  increments the register with data from coarseFlux, multiplied by `a_scale`. `a_coarseFlux` must contain the edge-centered (in 3d, node centered in 2d) coarse fluxes in the dir direction for the grid `m_coarseLayout[coarseDataIndex]`. By convention, only the low side flux is used to avoid double-counting at coarse-fine interfaces.

- ```
  void incrementFine(FArrayBox& a_fineFlux,
                     Real a_scale,
                     const DataIndex& a_fineDataIndex,
                     const Interval& a_srcInterval,
                     const Interval& a_dstInterval)
  ```

  increments the register with data from fineFlux (which is edge-centered in 3d, node-centered in 2d), multiplied by `a_scale`, for all coarse-fine face directions associated with the grid box `m_fineLayout[fineDataIndex]`

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

  increments the register with data from fineFlux (which is edge-centered in 3d, node-centered in 2d), multiplied by `a_scale`. `a_dir` is the normal of the coarse-fine interface, and `a_sd` determines whether we're looking at the high-side or the low-side for the grid box `m_fineLayout[fineDataIndex]`

- ```
  void refluxCurl(LevelData<FluxBox>& a_uCoarse,
                  Real a_scale)
  ```

  increments uCoarse with the reflux "CURL" of the contents of the flux register. This is done for all components so `a_uCoarse` has to have the same number of components as input `a_nComp`. This operation is global and blocking.

3.3 Class BRMeshRefine

BRMeshRefine is an object which produces a hierarchy of block-structured grids which obey proper-nesting requirements. See Berger and Colella [7] for an explanation of proper nesting. BRMeshRefine follows the algorithm of Berger and Rigoutsos [8] 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,
         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 be produced. So if the AMR hierarchy has 9 levels and one wants all of them to change except level 0, then 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 problem 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.
  ```

- ```
  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 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 (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 a_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 be produced. So if the AMR hierarchy has 9 levels and one wants all of them to change except level 0, then 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 problem 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.

  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 section 8.4.

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

  Returns the vector of refinement ratios

```
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 which have been defined on the base level.

- `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 is ∞ (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 (in `BRMeshRefine.H`):

```
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`.

3.4 Multilevel Utilities

In addition to the two-level `AMRTools` operator objects, there are a set of multilevel tools which have proved useful in various multilevel AMR codes.

3.4.1 Function `computeSum`

This function computes the volume-weighted sum of ϕ over an AMR hierarchy by including only valid-region data in the sum.

$$sum = \sum_{\ell=\ell_{base}}^{\ell max} \sum_{\Omega^{\ell}_{valid}} (h^{\ell})^{D} \phi^{\ell}_{\mathbf{i}} \tag{3.5}$$

- ```
  void
  Real computeSum(const Vector<LevelData<FArrayBox>* >& a_phi,
                  const Vector<int>&                    a_nRefFine,
                  const Real&                           a_dxCrse,
                  const Interval&                       a_comps = Interval(0,0),
                  const int&                            a_lBase = 0);
  ```

  function which returns the volume-weighted sum of a_phi (essentially the integral of a_phi) over the valid regions of all levels $\ell \geq$ a_lBase.
  Arguments:

  - a_phi (not modified): data to be summed.
  - a_nRefFine (not modified): Vector of refinement ratios, where a_refFine[i] is the refinement ratios between levels i and $i+1$.
  - a_dxCrse (not modified): cell spacing on level a_lBase.
  - a_comps (not modified): components of a_phi to be summed.
  - a_lBase (not modified): Coarsest level to be included in sum; sum will include all levels $\ell \geq$ a_lBase.

3.4.2 Function computeNorm

This function computes the norm of ϕ over an AMR hierarchy by including only valid-region data in the computation.

- ```
  void
  Real computeNorm(const Vector<LevelData<FArrayBox>* >& a_phi,
                   const Vector<int>&                    a_nRefFine,
                   const Real&                           a_dxCrse,
                   const Interval&                       a_comps = Interval(0,0),
                   const int                             a_p = 2,
                   const int&                            a_lBase = 0);
  ```

  function which returns the $p-$norm of a_phi over the valid regions of all levels $\ell \geq$ a_lBase.
  Arguments:

  - a_phi (not modified): data to be summed.
  - a_nRefFine (not modified): Vector of refinement ratios, where a_refFine[i] is the refinement ratios between levels i and $i+1$.
  - a_dxCrse (not modified): cell spacing on level a_lBase.
  - a_comps (not modified): components of a_phi to be summed.

- a_p (not modified): type of norm to be computed. a_p=0 is the max (infinity) norm.
- a_lBase (not modified): Coarsest level to be included in sum; sum will include all levels $\ell \geq$ a_lBase.

Chapter 4

AMRElliptic Algorithm and Implementation

4.1 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 [7]. The algorithm we use here is a natural extension of multigrid iteration. The particular version we describe here [20, 21] is a linear version of the algorithm used in [25] to compute steady incompressible flow, and has been used in a variety of settings [2, 1, 9, 10].

A pseudo-code description of the algorithm is given in figure (4.3). 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.

4.2 The AMR Elliptic User Interface

The implementation of the AMRElliptic package follows the algorithm specification in section 4.1.

```
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 4.1: 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 4.2: 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}
    $R^{l-1} := Average(R^l - L^{nf}(e^l, e^{l-1}))$ 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 4.3: Pseudo-code description of the AMR multigrid algorithm.

4.2.1 Overview

Code reuse is facilitated by using a *Template Method* design pattern. The purpose of the *Template Method* design pattern is to define an algorithm as a fixed sequence of steps but have one or more of the steps be variable. In our case, we have a hierarchy of algorithms that we wish to re-use across a family of applications. The hierarchy of algorithms is defined by the specific *variable steps* that an application must provide to complete the algorithm.

Our variable steps are supplied by a hierarchy of *Operator Interfaces*. In C++, a *variable step* is represented as a *virtual function*. Our algorithms are in the form of *Solver Templates*. Each Solver Template requires virtual functions provided by its corresponding Operator Interface. The data type used in these algorithms is supplied by a template parameter.

Various specific solvers derive from the appropriate interface class to utilize the desired solver algorithm.

An overview of our class structure for this design is presented here. Indentation implies inheritance

`LinearOp<T>`: User can utilize solvers that implement the `LinearSolver<T>` interface

> `MGLevelOp<T>` : Users can utilize solvers that implement `LinearSolver<T>` and `MultiGrid<T>` interfaces.
>
> > `AMRLevelOp<T>` Users can utilize solvers that implement `LinearSolver<T>`, `MultiGrid<T>` and `AMRMultiGrid<T>` interfaces. Examples of instantiations of these interfaces are:
> >
> > * PoissonOp (template data type `FArrayBox`). A single-level solver
> > * AMRPoissonOp (template data type `LevelData<FArrayBox>`) cell-centered AMR Poisson solver
> > * VCAMRPoissonOp (template data type `LevelData<FArrayBox>`) cell-centered variable-coefficient AMR Poisson and Helmholtz solver.
> > * EBPoissonOp (template data type `LevelData<EBCellFAB>`)
> > * EBAMRPoissonOp (template data type `LevelData<EBCellFAB>`)
> > * AMRNodeOp (template data type `LevelData<NodeFArrayBox>`) Node-centered AMRPoisson and Helmholtz solver.
> > * ResistivityOp (template data type `LevelData<FArrayBox>`) cell-centered variable-coefficient operator to solve variable coefficient resistivity operator.
> > * ViscousTensorOp (template data type `LevelData<FArrayBox>`) cell-centered variable-coefficient operator to solve variable coefficient viscous tensor operator.

`LinearSolver<T>`

- uses LinearOp<T> interface functions to implement the algorithm's variable steps.
- some example implementations of this algorithm interface are:
  - BiCGStabSolver<T>
  - RelaxSolver<T>

MultiGrid<T>

- calls the MGLevelOp<T> interface
- uses a LinearSolver<T> as bottom solver.

AMRMultiGrid<T>

- uses AMRLevelOp<T> and MGLevelOp interfaces
- combines AMR coarse-fine operations with MultiGrid<T> and LinearSolver<T> operations.

4.3 Operator Interfaces

The variable steps of our *Template Method* are supplied through classes that implement the Operator Interfaces.

4.3.1 Class LinearOp

LinearOp is an operator class for representing L when solving $L(\phi) = \rho$ This interface class serves two main purposes. First, It acts as a factory class for the template data type. Second, it provides the variable steps necessary for the family of LinearSolver<T> classes in Chombo.

- `virtual void residual( T& lhs, const T& phi, const T& rhs, bool homogeneous = false) = 0;`
  Compute the residual. For example, if solving `L(phi) = rhs`, then set `lhs = L(phi) - rhs`. If `homogeneous` is true, evaluate the operator using homogeneous boundary conditions.
- `virtual void preCond( T& cor, const T& residual) = 0;`
  Given the current state of the residual and correction, apply your preconditioner to `cor`.
- `virtual void applyOp( T& lhs, const T& phi, bool homogeneous = false) = 0;`
  In the context of solving `L(phi) = rhs`, set `lhs = L(phi)`. If `homogeneous` is true, evaluate the operator using homogeneous boundary conditions.

- `virtual void create( T& lhs, const T& rhs) = 0;`
  Create data holder `lhs` that mirrors `rhs`. You do not need to copy the data of `rhs`, just make a holder the same size.

- `virtual void clear( T& lhs);`
  Perform any operations required before `lhs` is destructed. In general, this function only needs to be defined if the `create` function called `new`. There is a default implementation of this function, which does nothing (which means that in most cases, classes derived from `LinearOp` will not need to define this function).

- `virtual void assign( T& lhs, const T& rhs) = 0;`
  Set `lhs` equal to `rhs`.

- `virtual Real dotProduct(const T& a1, const T& a2) = 0;`
  Compute and return the dot product of `a1` and `a2`. In most contexts, this will return the sum over all data points of `a1*a2`.

- `virtual void incr ( T& lhs, const T& x, Real scale) = 0;`
  Increment by scaled amount (`lhs += scale*x`).

- `virtual void axby( T& lhs, const T& x, const T& y, Real a, Real b) = 0;`
  Compute a scaled sum (`lhs = a*x + b*y`).

- `virtual void scale( T& lhs, const Real& scale) = 0;`
  Multiply the input by a given scale `lhs *= scale`).

- `virtual Real norm(const T& rhs, int ord) = 0;`
  Return the norm of `rhs`. If `ord == 0`, compute max norm, If `ord == 1`, compute L_1 norm: sum(abs(rhs)). Otherwise, compute L_{ord} norm.

- `virtual void setToZero(T& lhs) = 0;`
  Set `lhs` to zero.

4.3.2 Class `MGLevelOp`

Class `MGLevelOp` handles the additional tasks of coordinating operations between this level and the next coarser 'level'. `MGLevelOp` provides the coarsening and interlevel operations needed for algorithms that implement the `MultiGrid<T>` solver class.

- `virtual void createCoarser(T& coarse, const T& fine, bool ghosted) = 0;` Create a coarsened (by two) version of the input data container. This does not include averaging the data. So, if `fine` is over a Box of $(0,0,0)$ $(63,63,63)$, `coarse` should be over a Box of $(0,0,0)$ $(31,31,31)$.

- `virtual void relax(T& correction, const T& residual, int iterations) = 0 ;` Apply relaxation operator to remove the high frequency wave numbers from the correction. A point relaxation scheme, for example, takes the form `correction -= lambda*(L(correction) - residual)`.

- `virtual void restrictResidual(T& resCoarse, T& phiFine, const T& rhsFine) = 0;` calculate restricted residual
  `resCoarse[2h] = I[h->2h] (rhsFine[h] - L[h](phiFine[h])`

- `virtual void prolongIncrement(T& phiThisLevel, const T& correctCoarse) = 0;` correct the fine solution based on coarse correction
  `phiThisLevel += I[2h->h](correctCoarse)`

4.3.3 Class `MGLevelOpFactory`

Factory class for generating `MGLevelOps`.

- `virtual MGLevelOp<T>* MGnewOp(const ProblemDomain& FineindexSpace, int depth, bool homoOnly = true) = 0;`
  Create an operator at an index `space = coarsen(fineIndexSpace, 2d̂epth)`. Return NULL if no such Multigrid level can be created at this `depth`. If `homoOnly = true`, then only homogeneous boundary conditions will be needed.

4.3.4 Class `AMRLevelOp`

The `AMRLevelOp` interface adds variable steps required by the AMRMultiGrid class of solvers. These pertain to operations between multigrid levels that do not form complete covering sets, and therefore require information from multiple levels simultaneously for coarse-fine boundary conditions.

- `virtual int refToCoarser() = 0;`
  Return the refinement ratio to next coarser level.
  Return 1 when there are no coarser AMR levels.

- ```
  virtual void AMRResidual(T& residual,
                           const T& phiFine,
                           const T& phi,
                           const T& phiCoarse,
                           const T& rhs,
                           bool homogeneousDomBC,
                           AMRLevelOp<T>*  finerOp) = 0;
  ```
  Compute the residual: `residual = rhs - L(phiFine, phi, phiCoarse)`.
  ```

- ```
  virtual void AMRResidualNF(T& residual,
                             const T& phi,
                             const T& phiCoarse,
                             const T& rhs,
                             bool homogeneousBC) = 0;
  ```

  Compute `residual = rhs - L(phi, phiCoarse)` assuming no finer level.

- ```
  virtual void AMRResidualNC(T& residual,
                             const T& phiFine,
                             const T& phi,
                             const T& rhs,
                             bool homogeneousBC,
                             AMRLevelOp<T>* finerOp) = 0;
  ```

  Compute `residual = rhs - L(phiFine, phi)` assuming no coarser AMR level.

- ```
  virtual void AMRRestrict(T& resCoarse,
                           const T& residual,
                           const T& correction,
                           const T& coarseCorrection) = 0;
  ```

  Set `resCoarse = I[h-2h]( residual - L(correction, coarseCorrection))` .

- ```
  virtual void AMRProlong(T& correction,
                          const T& coarseCorrection) = 0;
  ```

  Set `correction += I[2h->h](coarseCorrection)`

- ```
  virtual void AMRUpdateResidual(T& residual,
                                 const T& correction,
                                 const T& coarseCorrection) = 0;
  ```

  Set `residual = residual - L(correction, coarseCorrection)` .

- ```
  virtual Real AMRNorm(const T&   coarseResid,
                       const T&   fineResid,
                       const int& refRat,
                       const int& ord) = 0;
  ```

  Compute norm over all cells on coarse not covered by finer AMR levels.

- ```
  virtual void createCoarsened(T&         lhs,
                               const T&   rhs,
                               const int& refRat) = 0;
  ```

  Set the output to a coarsened (by the input refinement ratio) version of the finer.
  ```

4.3.5 Class `AMRLevelOpFactory`

Factory interface for `AMRLevelOp` generation.

- `virtual AMRLevelOp<T>* AMRnewOp(const ProblemDomain& indexSpace)=0;`
  Return a new operator object. This is done with a call to `new`; caller is responsible for deletion.

4.4 Solver Templates

Solver Template requires virtual functions provided by its corresponding Operator Interface. The data type used in these algorithms is supplied by a template parameter.

Various specific solvers derive from the appropriate interface class to utilize the desired solver algorithm.

4.4.1 Class `LinearSolver`

`LinearSolver` represents both a *Solver Algorithm* and an interface class. It is a Solver Algorithm with respect to the variable steps provided by the *Operator Interfaces*. Given an instantiation of a `LinearSolver`, any operator that implements the `LinearOp` interface can make invocations to its `define` and `solve` functions. It is also an interface, in that `LinearSolver` does not provide a default implementation, but instead is an interface to a variety of linear solver algorithms. `MultiGrid` and `AMRMultiGrid` provide a default implementation of geometric multigrid. Generic linear solver templates`BiCGStab` and others are built on top of this.

- ```
  virtual void define(LinearOp<T>* operator,
                      bool homogeneous = false) = 0;
  ```
  Define the operator and whether it is a homogeneous solver or not. The `LinearSolver` does not take over ownership of this `operator` object. It does not call delete on it when the `LinearSolver` is deleted. It is meant to be like a late-binding reference. If you created `operator` with new, you should call delete on it after `LinearSolver` is deleted if you want to avoid memory leaks.

- `virtual void solve(T& phi, const T& rhs) = 0;`
  Solve `L(phi) = rhs (phi = L^-1 (rhs))`.

- `virtual void setConvergenceMetrics(Real metric, Real tolerance) = 0;`
  If appropriate, sets a metric to judge convergence, along with the solver tolerance. If not set, use default convergence metrics. This can be useful when using a `LinearSolver` as a bottom solver, since one may want to propagate the convergence metric and solver tolerance from the outer solver in to the bottom solver.
  ```

4.4.2 Class `BiCGStabSolver`

Elliptic solver using the `BiCGStab` algorithm.

- `virtual void define(LinearOp<T>* op, bool homogeneous);`
  Define the solver.
  - `op` is the linear operator.
  - `homogeneous` is whether the solver uses homogeneous boundary conditions.
- `virtual void solve(T& phi, const T& rhs);`
  Solve the equation.
- `bool m_homogeneous`
  public member data: whether the solver is restricted to homogeneous boundary conditions.
- `LinearOp<T>* m_op:`
  public member data: operator to solve.
- `int m_imax;`
  public member data: maximum number of iterations.
- `int m_verbosity;`
  public member data: how much screen output the user wants. `set = 0` for no output.
- `Real m_eps;`
  public member data: solver tolerance
- `Real m_hang;`
  public member data: minimum rate that norm of solution should charge per iteration
- `Real m_small;`
  public member data: what the algorithm should consider "close to zero"
- `int m_numRestarts;`
  public member data: number of times the algorithm can restart
- `int m_normType;`
  public member data: norm to be used when evaluation convergence.
  0 is max norm, 1 is`L(1)`, 2 is `L(2)` and so on.

4.4.3 Class `MultiGrid`

`MultiGrid` is a class which executes a v-cycle of geometric multigrid. This class is not meant to be particularly user-friendly, and a good option for people who want something a tad less raw is to use `AMRMultigrid` instead.

- ```
  virtual void define(MGLevelOpFactory<T>& factory,
                      LinearSolver<T>* bottomSolver,
                      const ProblemDomain& domain,
                      int  maxDepth = -1);
  ```

  Function to define a `MultiGrid` object:

  - `factory` is the factory for generating operators.
  - `bottomSolver` is called at the bottom of v-cycle.
  - `domain` is the problem domain at the top of the v-cycle.
  - `maxDepth` defines the location of the bottom of the v-cycle.

  The v-cycle will terminate (hit bottom) when the factory returns NULL for a particular depth if `maxdepth = -1`. Otherwise the v-cycle terminates at maxdepth.

- `virtual void solve(T& e, const T& res);`
  Execute ONE v-cycle of multigrid. If you want the solution to converge, you will probably need to iterate this. See AMRMultiGrid for a more automatic solve() function. This operates residual-correction form of equation so all boundary conditions are assumed to be homogeneous. `L(e) = res`

- `int m_depth, m_pre, m_post, m_cycle,m_numMG;`
  Public solver parameters. `m_pre` and `m_post` are the ones that usually get set and are the number of relaxations performed before and after multigrid recursion. See AMRMultiGrid for a more user-friendly interface.

- `Vector< MGLevelOp<T>* > getAllOperators();`
  For changing coefficients — not for the faint of heart.

4.4.4 Class `AMRMultiGrid`

Class to execute geometric multigrid over an AMR hierarchy a-la Martin and Cartwright. [20]

- ```
  virtual void define(const ProblemDomain& coarseDomain,
                      AMRLevelOpFactory<T>& factory,
                      LinearSolver<T>* bottomSolver,
                      int maxAMRLevels);
  ```

  Define the solver.

- coarseDomain is the index space on the coarsest AMR level.
- factory is the operator factory through which all special information is conveyed.
- bottomSolver is the solver to be used at the termination of multigrid coarsening.
- numLevels is the number of AMR levels.

- ```
  virtual void solve(Vector<T*>& phi,
                     const Vector<T*>& rhs,
                     int l_max,
                     int l_base);
  ```

  Solve $L(\phi) = \rho$ from l_base to l_max. To solve over all levels, l_base = 0, l_max = max_level = numLevels-1.

- ```
  void setSolverParameters(const int&   pre,
                           const int&   post,
                           const int&   bottom,
                           const int&   numMG,
                           const int&   iterMax,
                           const Real&  eps,
                           const Real&  hang,
                           const Real&  normThresh);
  ```

  Set parameters of the solve.

  - pre is the number of smoothings before averaging.
  - post is the number of smoothings after averaging.
  - bottom is the number of smoothings at the bottom level.
  - numMG = 1 for v-cycle, 2 for w-cycle and so on (in most cases, use 1).
  - itermax is the max number of v cycles.
  - hang is the minimum amount of change per vcycle.
  - eps is the solution tolerance.
  - normThresh is how close to zero eps*resid is allowed to get.

4.5 The MultilevelLinearOp<T> class

The MultilevelLinearOp<T> class is a LinearOperator<Vector<LevelData<T>* > > designed to support multilevel composite operators. This is useful when using a LinearSolver such as BiCGStabSolver to solve elliptic equations over a hierarchy of AMR levels. The MultilevelLinearOp is derived from LinearOp<Vector<LevelData<T>* > >.

Defining a MultilevelLinearOp requires an AMRLevelOpFactory<LevelData<T> > which is used to define AMRLevelOps for each AMR level, which are then used to evaluate the multilevel operator, etc. An example which uses the MultilevelLinearOp<FArrayBox> in conjunction with the VCAMRPoissonOp to solve the variable-coefficient Helmholtz equation using multigrid-preconditioned BiCGStab is in
Chombo/example/AMRPoisson/variableCoefficientExec/VCPoissonSolve.cpp
Public member functions and member data:

- MultilevelLinearOp()
  - default constructor – leaves object in undefined state
- void define(const Vector<DisjointBoxLayout>& vectGrids,
  const Vector<int>& refRatios,
  const Vector<ProblemDomain>& domains,
  const Vector<RealVect>& vectDx,
  RefCountedPtr<AMRLevelOpFactory<LevelData<T> > >& opFactory,
  int lBase)
  - full define function
    - vectGrids – AMR hierarchy of grids
    - refRatios – refinement ratios; refRatios[0] is refinement ratio between levels 0 and 1.
    - domains – problem domains for each AMR level.
    - vectDx – cell-spacing on each AMR level
    - opFactory – AMRLevelOpFactory used to define operators for performing multilevel linear solves.
    - lBase – base level
- virtual void residual(Vector<LevelData<T>* >& lhs,
  const Vector<LevelData<T>* > >& phi,
  const Vector<LevelData<T>* > >& rhs,
  bool homogeneous = false)
  - compute residual = L(phi) - rhs
    - lhs – residual
    - phi – current approximation to the solution
    - rhs – rhs when solving L(phi) = rhs
    - homogeneous – if true, evaluate using homogeneous form of physical domain boundary conditions

- `virtual void preCond(Vector<LevelData<T>* >& cor, const Vector<LevelData<T>* >& residual)`

  – Apply preconditioner to problem which is already in residual-correction form. If `m_use_multigrid_preconditioner` is `true` (default case), then the preconditioner is `m_num_mg_iterations` AMR V-cycles using an `AMRMultiGrid` solver. Otherwise, use whatever preconditioner is provided by the `AMRLevelOp<LevelData<T> >`.

  - `cor` – correction (modified by the preconditioner)
  - `residual` – residual (= L(phi) - rhs)

- `virtual void applyOp(Vector<LevelData<T>* >& lhs, const Vector<LevelData<T>* >& phi, bool homogeneous = false)`

  – Evaluate the operator, setting lhs = L(phi) using the `AMRLevelOp<T>::AMROperator` functions. If `homogeneous` is true, evaluate the operator using homogeneous physical domain boundary conditions.

- `virtual void create(Vector<LevelData<T>* >& lhs, const Vector<LevelData<T>* >& rhs)`

  – Create data holder `lhs` that mirrors `rhs`, using the appropriate `AMRLevelOp::create` functionality. Does not copy the data of `rhs`, just makes a holder the same size.

- `virtual void clear(Vector<LevelData<T>* >& lhs) const`

  – Clear memory in data holder `lhs`. Note that the `MultilevelLinearOp` class requires this function be implemented because the `create` function calls `new` when allocating the `Vector<LevelData<T>* >`.

- `virtual void assign(Vector<LevelData<T>* >& lhs, const Vector<LevelData<T>* >& rhs)`

  – Set `lhs` equal to `rhs`.

- `virtual Real dotProduct(const Vector<<LevelData<T>* >& a_1, const Vector<LevelData<T>* >& a_2)`

  – Compute and return the volume-weighted AMR dot product of `a_1` and `a_2`. Does this by calling the `AMRLevelOp dotProduct` functions for each AMR level and then scaling appropriately.

- `virtual void incr(Vector<LevelData<T>* >& lhs, const Vector<LevelData<T>* >& x, Real scale)`

  – Increment by scaled amount (lhs += scale*x)

- `virtual void axby(Vector<LevelData<T>* >& lhs,`
  `const Vector<LevelData<T>* >& x,`
  `const Vector<LevelData<T>* >& y,`
  `Real a,`
  `Real b)`

  – Set input lhs to a scaled sum (lhs = a*x + b*y).

- `virtual void scale(Vector<LevelData<T>* >& lhs,`
  `const Real& scale)`

  – Multiply the input by a given scale (lhs *= scale).

- `virtual Real norm(const Vector<<LevelData<T>* >& rhs,`
  `int ord)`

  – Return the AMR norm of rhs (only counts valid regions for each level).

  – `ord` – norm type: 0 is max norm, 1 is L_1 norm – sum(abs(rhs)), otherwise, L_{ord} norm.

- `virtual void setToZero(Vector<LevelData<T>* >& lhs)`

  – Set lhs to zero.

- `bool m_use_multigrid_preconditioner`

  – if true (default value), use AMRMultiGrid multigrid V-cycles for preconditioner

- `int m_num_mg_iterations`

  – number of multigrid v-cycles to do in preconditioner (only relevant if `m_use_multigrid_preconditioner` is `true`).

- `int m_num_mg_smooth`

  – parameter for `AMRMultiGrid` – number of smoothing passes for each multigrid relaxation step. (only relevant if `m_use_multigrid_preconditioner` is `true`).

4.6 Elliptic Examples

We provide several examples of elliptic operators that conform to the `AMRLevelOp` interface

- `AMRPoissonOp` is used to solve Poisson's equation with constant coefficients.

- `ResistivityOp` is used to solve the variable-coefficient resistivity equations that arise from MHD.

- `ViscousTensorOp` is used to solve the variable-coefficient elliptic equations that arise when solving the compressible Navier-Stokes equations with variable viscosity.

All of these examples use the boundary condition interface described in section 4.6.4.

4.6.1 AMRPoisson

We provide an `AMRPoisson`, an example of an `AMRLevelOp` class. This class is designed to solve

$$(\alpha I + \beta\Delta)\phi = \rho \tag{4.1}$$

where α and β are constants. The discretization of the Laplacian is the standard centered-difference approximation

$$(\Delta^h\phi)_{\boldsymbol{i}} = \frac{1}{h^2}\sum_{d=0}^{\mathbf{D}-1}(\phi_{\boldsymbol{i}+\boldsymbol{e}^d} + \phi_{\boldsymbol{i}-\boldsymbol{e}^d} - 2\phi_{\boldsymbol{i}})$$

Domain boundary conditions are enforced by setting ghost cell values as described in section 4.6.4. Fluxes through coarse-fine interfaces are needed for the Martin-Cartwright algorithm. The flux F through a face at $\boldsymbol{i} + \frac{1}{2}\boldsymbol{e}^d$ is given by

$$F_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = \frac{\beta}{h}(\phi_{\boldsymbol{i}+\boldsymbol{e}^d} - \phi_{\boldsymbol{i}})$$

4.6.1.1 AMRPoisson Factory Interface

An AMRPoissonOpFactory needs to be defined using the following function.

```
void define(const ProblemDomain& coarseDomain,
            const Vector<DisjointBoxLayout>& grids,
            const Vector<int>& refRatios,
            const Real&        coarsedx,
            BCFunc bc,
            Real   alpha = 0.,
            Real   beta  = 1.);
```

- `coarseDomain` is the domain at the coarsest level.
- `grids` is the AMR hierarchy.
- `refRatios` are the refinement ratios between levels. The ratio lives with the coarser level so `refRatios[4]` is the ratio between AMR levels 4 and 5.
- `coarseDx` is the grid spacing at the coarsest level.

- BCFunc holds the boundary conditions.
- alpha is the coefficient of the identity.
- beta is the coefficient of the Laplacian.

4.6.1.2 Code fragment

For those of us who find code easier to read than documents, we provide a simplified example of how to use AMRMultiGrid and AMRPoissonOp. In the example below, we start with a known AMR hierarchy and a right-hand side and we solve (4.1). For a description of the boundary condition routine along with its code fragment, see subsection 4.6.4. Complete examples (along with convergence tests) can be found in Chombo/example/AMRPoisson.

```
/**
   solveElliptic solves (alpha I + beta Laplacian) phi = rhs
   using AMRMultiGrid and AMRPoissonOp

   Inputs:
   rhs:  Right-hand side of the solve over the level.
   grids: AMRHierarchy of grids
   refRatio:  refinement ratios
   level0Domain:  domain at the coarsest AMR level
   coarsestDx: grid spacing at the coarsest level
   alpha: identity  coefficient
   beta:  Laplacian coefficient

   Outputs:
   phi = (alpha I + beta Lapl)^{-1}(rhs)
 */
void solveElliptic( Vector<LevelData<FArrayBox>* >&      phi,
                    const Vector<LevelData<FArrayBox>* > rhs,
                    const Vector<DisjointBoxLayout>&     grids,
                    const Vector<int>&                   refRatios,
                    const ProblemDomain&                 level0Domain,
                    Real alpha, Real beta, Real coarsestDx)

{
  int numlevels = rhs.size();

  //define the operator factory
  AMRPoissonOpFactory opFactory;
  opFactory.define(level0Domain,
                   grids, refRatios, coarsestDx,
                   &ParseBC, alpha, beta);

  //this is the solver we shall use
  AMRMultiGrid<LevelData<FArrayBox> > solver;


  //this is the solver for the bottom of the muligrid v-cycle
  BiCGStabSolver<LevelData<FArrayBox> >   bottomSolver;
  //bicgstab can be noisy
  bottomSolver.m_verbosity = 0;

  //define the solver
  solver.define(level0Domain, opFactory, &bottomSolver, numlevels);

  //we want to solve over the whole hierarchy
```

```
  int lbase = 0;
  //so solve already.
  solver.solve(phi, rhs, numlevels-1, lbase);
}
```

4.6.2 ResistivityOp

ResistivityOp is the AMRLevelOp-derived class which solves the variable-coefficient equation For notation's sake, what we are solving is

$$L\vec{B} = \alpha\vec{B} + \beta\nabla \cdot F = \rho.$$

α and β are constants and F is given by

$$F = \eta(\nabla\vec{B} - \nabla\vec{B}^T + I\nabla \cdot \vec{B})$$

where I is the identity matrix and $\eta = \eta(\vec{x}) \geq 0$. The discretization of the flux divergence is as follows.

$$(\nabla \cdot F)^h_{\boldsymbol{i}} = \frac{1}{h}\sum_{d=1}^{D}(F_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d}F\phi_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^d})$$

We discretize normal components of the face-centered gradient using an average of cell-centered gradients for tangential components and a centered-difference approximation to the normal gradient.

$$(\nabla\vec{B})^{d'}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = \begin{pmatrix} \frac{1}{h}(\vec{B}_{\boldsymbol{i}+\boldsymbol{e}^d} - \vec{B}_{\boldsymbol{i}}) \text{ if } d = d' \\ \frac{1}{2}((\nabla\vec{B})^{d'}_{\boldsymbol{i}+\boldsymbol{e}^d} + (\nabla\vec{B})^{d'}_{\boldsymbol{i}}) \text{ if } d \neq d' \end{pmatrix}$$

where

$$(\nabla\vec{B})^d_{\boldsymbol{i}} = \frac{1}{2h}(\vec{B}_{\boldsymbol{i}+\boldsymbol{e}^d} - \vec{B}_{\boldsymbol{i}-\boldsymbol{e}^d}).$$

4.6.2.1 ResistivityOp Factory Interface

An ResistivityOpFactory needs to be defined using the following constructor.

```
ResistivityOpFactory(const Vector<DisjointBoxLayout>&                      grids,
                     const Vector<RefCountedPtr<LevelData<FluxBox> > >&    eta,
                     Real                                                  alpha,
                     Real                                                  beta,
                     const Vector<int>&                                    refRatios
                     const ProblemDomain&                                  domainCoa
                     const Real&                                           dxCoar,
                     BCFunc                                                bc);
```

- domainCoar is the domain at the coarsest level.

- `grids` is the AMR hierarchy.
- `refRatios` are the refinement ratios between levels. The ratio lives with the coarser level so `refRatios[4]` is the ratio between AMR levels 4 and 5.
- `dxCoar` is the grid spacing at the coarsest level.
- `bc` holds the boundary conditions.
- `alpha` is the coefficient of the identity.
- `beta` is the coefficient of the divergence of the flux.
- `eta` is the variable coeff function. This ought to to be positive if you expect multigrid to converge.

4.6.3 ViscousTensorOp

`ViscousTensorOp` is the `AMRLevelOp`-derived class which solves the variable-coefficient equation For notation's sake, what we are solving is

$$L\vec{B} = \alpha\vec{B} + \beta\nabla \cdot F = \rho.$$

$\alpha = \alpha(\vec{x})$ and $\beta = \beta(\vec{x})$. F is given by

$$F = \eta(\nabla\vec{B} + \nabla\vec{B}^T) + \lambda(I\nabla \cdot \vec{B})$$

where I is the identity matrix $\eta = \eta(\vec{x})$, and $\lambda = \lambda(\vec{x})$. The discretization of the flux divergence is as follows.

$$(\nabla \cdot F)^h_{\boldsymbol{i}} = \frac{1}{h}\sum_{d=1}^{D}(F_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} - F_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^d})$$

We discretize normal components of the face-centered gradient using an average of cell-centered gradients for tangential components and a centered-difference approximation to the normal gradient.

$$(\nabla\vec{B})^{d'}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^d} = \begin{pmatrix} \frac{1}{h}(\vec{B}_{\boldsymbol{i}+\boldsymbol{e}^d} - \vec{B}_{\boldsymbol{i}}) \text{ if } d = d' \\ \frac{1}{2}((\nabla\vec{B})^{d'}_{\boldsymbol{i}+\boldsymbol{e}^d} + (\nabla\vec{B})^{d'}_{\boldsymbol{i}}) \text{ if } d \neq d' \end{pmatrix}$$

where

$$(\nabla\vec{B})^d_{\boldsymbol{i}} = \frac{1}{2h}(\vec{B}_{\boldsymbol{i}+\boldsymbol{e}^d} - \vec{B}_{\boldsymbol{i}-\boldsymbol{e}^d}).$$

4.6.3.1 ViscousTensorOp Factory Interface

An `ViscousTensorOpFactory` needs to be defined using the following constructor.

```
ViscousTensorOpFactory(const Vector<DisjointBoxLayout>&                      a_grids
                       const Vector<RefCountedPtr<LevelData<FluxBox> > >&    a_eta,
                       const Vector<RefCountedPtr<LevelData<FluxBox> > >&    a_lambd
                       Real                                                  a_alpha
                       const Vector<RefCountedPtr<LevelData<FArrayBox> > >& a_beta,
                       const Vector<int>&                                    a_refRa
                       const ProblemDomain&                                  a_domai
                       const Real&                                           a_dxCoa
                       BCFunc                                                a_bc);
```

- `domainCoar` is the domain at the coarsest level.
- `grids` is the AMR hierarchy.
- `refRatios` are the refinement ratios between levels. The ratio lives with the coarser level so `refRatios[4]` is the ratio between AMR levels 4 and 5.
- `dxCoar` is the grid spacing at the coarsest level.
- `bc` holds the boundary conditions.
- `alpha` is the coefficient of the identity.
- `beta` is the coefficient of the divergence of the flux.
- `eta` is the variable coeff function that multiplies the gradient.
- `lambda` is the variable coeff function that multiplies the divergence.

4.6.4 Boundary Condition Interface

The value of the boundary condition is described by the `BCValueFunc` function interface.

```
/* Given
   pos [x,y,z] position on center of cell edge
   int dir direction, x being 0
   int side -1 for low, +1 = high,
   fill in the values array */
typedef void(*BCValueFunc)(Real*           pos,
                           int*            dir,
                           Side::LoHiSide* side,
                           Real*           value);
```

The boundary condition function must conform to the `BCFunc` specification. We provide both Dirichlet and Neumann boundary condition examples.

```
   /* Function interface for ghost cell boundary conditions
   of EBAMRPoissonOp.   If you are using Neumann or Dirichlet
   boundary conditions,   it is easiest to use the functions
   provided. */
typedef void(*BCFunc)(FArrayBox&           state,
                      const Box& valid,
                      const ProblemDomain& domain,
                      Real                 dx,
                      bool                 homogeneous);
```

A simple example of a custom boundary condition is given below.

```
/* this is the bc value func */
void ParseValue(Real* pos,
                int* dir,
                Side::LoHiSide* side,
                Real* values)
{
  ParmParse pp;
  Real bcVal;
  pp.get("bc_value",bcVal);
  values[0]=bcVal;
}


/*Use ParmParse to select boundary condtions */
/* this is the bcfunc */
void ParseBC(FArrayBox& state,
             const Box& valid,
             const ProblemDomain& domain,
             Real dx,
             bool homogeneous)
{
  if(!domain.domainBox().contains(state.box()))
    {
      std::vector<int>  bcLo, bcHi;
      ParmParse pp;
      pp.getarr("bc_lo", bcLo, 0, SpaceDim);
      pp.getarr("bc_hi", bcHi, 0, SpaceDim);

      Box valid = valid;
      for(int i=0; i<CH_SPACEDIM; ++i)
        {
          Box ghostBoxLo = adjCellBox(valid, i, Side::Lo, 1);
          Box ghostBoxHi = adjCellBox(valid, i, Side::Hi, 1);
          if(!domain.domainBox().contains(ghostBoxLo))
            {
              if(bcLo[i] == 1)
                {
                  pout() << "const neum bcs lo for direction " << i << endl;
                  NeumBC(state,
                         valid,
                         dx,
                         homogeneous,
                         ParseValue,
                         i,
                         Side::Lo);
                }
              else if(bcLo[i] == 0)
                {
                  pout() << "const diri bcs lo for direction " << i << endl;
                  DiriBC(state,
                         valid,
                         dx,
```

```
                    homogeneous,
                    ParseValue,
                    i,
                    Side::Lo);

            }
          else
            {
              MayDay::Error("bogus bc flag lo");
            }
        }

      if(!domain.domainBox().contains(ghostBoxHi))
        {
          if(bcHi[i] == 1)
            {
              pout() << "const neum bcs hi for direction " << i << endl;
              NeumBC(state,
                     valid,
                     dx,
                     homogeneous,
                     ParseValue,
                     i,
                     Side::Hi);
            }
          else if(bcHi[i] == 0)
            {
              pout() << "const diri bcs hi for direction " << i << endl;
              DiriBC(state,
                     valid,
                     dx,
                     homogeneous,
                     ParseValue,
                     i,
                     Side::Hi);
            }
          else
            {
              MayDay::Error("bogus bc flag hi");
            }
        }
    }
  }
}
```

4.7 Parabolic Equations – the TGA scheme

In this section, we describe our approach to solving parabolic equations of the form:

$$\frac{\partial \phi}{\partial t} = L(\phi) + S, \tag{4.2}$$

where L is a second-order linear elliptic operator, and S is a source term.

To evolve ϕ in time from time t^n to time $t^{n+1} = t^n + \Delta t$, we use a variant of the L_0-stable Runge-Kutta scheme presented by Twizell, Gumel, and Arigu (TGA) [26], which is second-order in time. Our variation is based on time-centering the source term S at time $t^{n+\frac{1}{2}} = t^n + \frac{\Delta t}{2}$, and is also described in [19].

Following [26], we discretize (4.2) in time:

$$\phi^{n+1} = (I - \mu_1 L)^{-1}(I - \mu_2 L)^{-1}\big[(I + \mu_3 L)\phi^n + \Delta t(I + \mu_4 L)S^{n+\frac{1}{2}}\big], \tag{4.3}$$

where $\phi^n = \phi(n\Delta t)$, $S^{n+\frac{1}{2}} = S\big((n+\frac{1}{2})\Delta t\big)$, and the coefficients $\mu_1, \mu_2, \mu_3, \mu_4$ are the values suggested in [26]:

$$\begin{aligned}
\mu_1 &= \frac{2a-1}{a + discr}\Delta t, \\
\mu_2 &= \frac{2a-1}{a - discr}\Delta t, \\
\mu_3 &= (1-a)\Delta t, \\
\mu_4 &= (\frac{1}{2} - a)\Delta t \\
a &= 2 - \sqrt{2} - \epsilon, \\
discr &= \sqrt{a^2 - 4a + 2},
\end{aligned}$$

where ϵ is a small quantity (we use 10^{-8}). We use this to define the operator $L^{TGA}(\phi^n, S^{n+\frac{1}{2}})$ as follows:

$$\begin{aligned}
L^{TGA}(\phi^n, S^{n+\frac{1}{2}}) &\equiv \frac{\phi^{n+1} - \phi^n}{\Delta t} - S^{n+\frac{1}{2}} \\
&\approx (L\phi)\Big((n+\frac{1}{2})\Delta t\Big) + O(\Delta t^2).
\end{aligned} \tag{4.4}$$

where $\phi^{n+1} = \phi^{n+1}(\phi^n, S^{n+\frac{1}{2}})$ is defined to be the expression (4.3).

To simplify the use of the TGA scheme, Chombo includes two classes which implement L^{TGA} – one for a single AMR level, and one for an entire AMR hierarchy. Note that computing L^{TGA} requires solving the heat equation and applying the operator L, these classes require that there be suitable `AMRLevelOp`-derived operator classes which discretize the appropriate Helmholtz operator.

4.7.1 The `TGAHelmOp` and `LevelTGAHelmOp` classes

Since implementing the TGA algorithm requires some additional functionality in addition to that specified in the `AMRLevelOp<T>` class, we have the `TGAHelmOp<T>` and `LevelTGAHelmOp<T, TFlux>` classes, which are general Helmholtz-type operators like $(\alpha + \nabla \cdot \beta \nabla)$ publicly derived from `AMRLevelOp<T>`. The multilevel TGA implementation uses `TGAHelmOp`. The single-level TGA implementation requires two functions not needed by the multilevel solve, so it uses `LevelTGAHelmOp<T,TFlux>`, which is publicly derived from `TGAHelmOp<T>` and includes the extra required functions. The `AMRPoissonOp`, `ResistivityOp`, and `ViscousTensorOp` classes are derived from `LevelTGAHelmOp`, and so may be used by both of the TGA solvers.

Additional functions required by `TGAHelmOp<T>` and `LevelTGAHelmOp<T,TFlux>`:

- `virtual void setAlphaAndBeta(const Real& a_alpha, const Real& a_beta) = 0`

  – Set the Helmholtz equation constants in the operator.

- `virtual void diagonalScale(T& a_rhs) = 0`
  `virtual void diagonalScale(T& a_rhs, bool a_kappaWeighted) = 0`

  – Set the diagonal scaling of the operator. For example, if solving $\rho(x)\frac{\partial\phi}{\partial t} = L(\phi)$, the diagonalScale would be ρ. In EB applications, even for constant coefficients, it means multiplication by κ.

- `virtual void applyOpNoBoundary(T& a_ans, const T& a_phi) = 0`

  – Apply operator without any boundary or coarse-fine boundary conditions and no finer level. This implies that we've set ghost-cell values outside the operator and want to use them in the operator evaluation.

Additional functions required by `LevelTGAHelmOp<T,TFlux>`:

- `virtual void fillGrad(const T& a_phi)`

  – This a function used in operators of fairly complex flux functions in which the gradient of the solution is computed and stored separately. This function is to signal the operator to do said computation with the input data. The default implementation does nothing.

- ```
  virtual void getFlux(TFlux&           a_flux,
                       const T&         a_data,
                       const Box&       a_grid,
                       const DataIndex& a_dit,
                       Real             a_scale) =0
  ```

  – compute face-centered flux.

  - `flux` – face-centered dataholder into which to place flux
  - `data` – cell-centered data used to compute the flux
  - `grid` – cell-centered box over which to compute face-centered flux.
  - `dit` – DataIndex of grid box we're working with
  - `scale` – scaling factor to multiply flux.

4.7.2 The `AMRTGA` class

The `AMRTGA<T>` class is a templated implementation of the TGA algorithm designed to advance an entire multilevel hierarchy of AMR grids by one (non-subcycled) timestep where `T` is the data holder for a single AMR level (like a `LevelData<FArrayBox>`.
Public member functions:
  ```

- AMRTGA(const RefCountedPtr<AMRMultiGrid<T> >& a_solver,
  const AMRLevelOpFactory<T>& a_factory,
  const ProblemDomain& a_level0Domain,
  const Vector<int>& a_refRat,
  int a_numLevels = -1,
  int a_verbosity = 0)
  - constructor
    - solver – AMRMultiGrid solver used to do the elliptic solves. This should be predefined with the appropriate TGAHelmOp operators.
    - factory – Factory which can be used to create TGAHelmOps for this problem.
    - level0Domain – problem domain on the coarsest level
    - refRat – vector of refinement ratios
    - numLevels – number of AMR levels
    - verbosity – how much output is written out. Higher number is more verbose (default is 0).
- void oneStep(Vector<T*>& a_phiNew,
  Vector<T*>& a_phiOld,
  Vector<T*>& a_source,
  const Real& a_dt,
  int a_lbase,
  int a_lmax)
  - advances a parabolic PDE one timestep using TGA on a non-moving domain.
    - phiNew – new-time solution computed using TGA.
    - phiOld – old-time solution
    - source – source term at the half-time
    - dt – timestep
    - lbase – coarsest level to be advanced
    - lmax – finest level to be advanced

4.7.3 The BaseLevelTGA and LevelTGA classes

The BaseLevelTGA<T, TFlux, TFR> class is a pure-virtual base class which implements the basic TGA scheme on a DisjointBoxLayout, which corresponds to solving on a single AMR level. The BaseLevelTGA class is templated on T, a (cell-centered) data holder over the entire DisjointBoxLayout (which is the same template type T used by the AMRLevelOp-derived operator, and which is the datatype of the solution on a single

level), TFlux, a face-centered data holder on a single patch, and TFR, the appropriate FluxRegister type of object to store diffusive fluxes along any coarse-fine interfaces.

The simplest example of a BaseLevelTGA-derived class is the LevelTGA class, which is a publicly-derived BaseLevelTGA<LevelData<FArrayBox>, FluxBox, LevelFluxRegister>. Note that the BaseLevelTGA objects are defined over the entire AMR hierarchy at once, and then advance a single-level as specified.

Public member functions:

- ```
  BaseLevelTGA(const Vector<DisjointBoxLayout>&            a_grids,
               const Vector<int>&                          a_refRat,
               const ProblemDomain&                        a_level0Domain,
               RefCountedPtr<AMRLevelOpFactory< T > >&     a_opFact,
               const RefCountedPtr<AMRMultiGrid<T > >&     a_solver)
  ```

  – constructor

  - grids – grids over the entire AMR hierarchy
  - refRat – refinement ratios
  - level0domain – problem domain on coarsest level
  - opFact – factory used to create LevelTGAHelmOps for this problem.
  - solver – AMR Multigrid solver over entire AMR hierarchy, pre-defined with the appropriate LevelTGAHelmOp-derived operators.

- ```
  void updateSoln(T&                  a_phiNew,
                  T&                  a_phiOld,
                  T&                  a_src,
                  LevelData<TFlux>&   a_flux,
                  TFR*                a_FineFluxRegPtr,
                  TFR*                a_CrseFluxRegPtr,
                  const T*            a_crsePhiOldPtr,
                  const T*            a_crsePhiNewPtr,
                  Real                oldTime,
                  Real                crseOldTime,
                  Real                crseNewTime,
                  Real                dt,
                  int                 a_level,
                  bool                a_zeroPhi = true,
                  int                 a_fluxStartComponent = 0);
  ```

  – update ALL components of phi

  - phiNew – updated solution at oldTime + dt
  - phiOld – solution at oldTime

- src – source term at (oldTime + 0.5*dt)
- flux –
- FineFluxRegPtr – flux register for diffusive fluxes along coarse-fine interface between this level (level) and the next-finer level (level+1).
- CrseFluxRegPtr – flux register for diffusive fluxes along coarse-fine interface between this level (level) and the next-coarser level (level-1).
- crsePhiOldPtr – pointer to old-time coarse data (for coarse-fine boundary conditions)
- crsePhiNewPtr – pointer to new-time coarse data (for coarse-fine boundary conditions)
- oldTime – time centering of phiOld
- crseOldTime – time centering of crsePhiOldPtr
- crseNewTime – time centering of crsePhiNewPtr
- dt – timestep
- level – AMR level which we're advancing
- zeroPhi –
- fluxStartComponent – component at which to place first component of flux

- ```
  void computeDiffusion(T&                a_DiffusiveTerm,
                        T&                a_phiOld,
                        T&                a_src,
                        LevelData<TFlux>& a_flux,
                        TFR*              a_FineFluxRegPtr,
                        TFR*              a_crseFluxRegPtr,
                        const T*          a_crsePhiOldPtr,
                        const T*          a_crsePhiNewPtr,
                        Real              a_oldTime,
                        Real              a_crseOldTime,
                        Real              a_crseNewTime,
                        Real              a_dt,
                        int               a_level);
  ```

  – compute time-centered $L^{TGA}(\phi, S)$ for use in subsequent update operations. In this case, we do solve for phiNew, then subtract source and phiOld back out to get L(phi). Function arguments have the same meanings as for the updateSoln function.

  - DiffusiveTerm – $L^{TGA}(\phi, S)$ as computed by this function.
  ```

4.8 TGA addendum for EB and non-unity identity coefficients

Consider the partial differential equation

$$\begin{aligned} a\frac{\partial \phi}{\partial t} &= L\phi + S \\ a &= a(x), \quad a > 0. \end{aligned} \tag{4.5}$$

We can extend the TGA algorithm by defining

$$L_a\phi \equiv \frac{L\phi}{a}$$

$$S_a\phi \equiv \frac{S}{a}$$

The solution is given by

$$(I - \mu_1 L_a)(I - \mu_2 L_a)\phi^{n+1} = (I + \mu_3 L_a)\phi^n + \Delta t(I + \mu_4 L_a)S_a$$

Because a (though non-zero), can be quite small, we are reluctant to have it in the denominator. We use the handy identity $(AB)^{-1} = B^{-1}A^{-1}$ to get

$$(I - L)^{-1} = (aI - aL)^{-1}(a) \tag{4.6}$$

to derive

$$\phi^{n+1} = (aI - \mu_1 L)^{-1}(a(aI - \mu_2 L)^{-1}[(aI + \mu_3 L)\phi^n + \Delta t(aI + \mu_4 L)\frac{S}{a}]). \tag{4.7}$$

Note that we still have to divide the source term by the coefficient but, since this is typically density, it should be floored reasonably above zero.

Finally, with embedded boundaries, we must also multiply out factors of the volume fraction κ. L is divided by κ and κ can be arbitrarily small. We can reuse the above identity to obtain the following.

$$\phi^{n+1} = (\kappa aI - \kappa\mu_1 L)^{-1}\kappa a\left[(\kappa aI - \kappa\mu_2 L)^{-1}[(\kappa aI + \kappa\mu_3 L)\phi^n + \Delta t(\kappa aI + \kappa\mu_4 L)\frac{S}{a}])\right].$$

Mercifully, we never have to divide by the volume fraction.

To accommodate these variable coefficient issues `LevelTGAHelmOp` needs some extra functions.

- `virtual void  diagonalScale(T& a_rhs, bool a_kappaWeighted)`

  This function multiplies in place the input by the identity coefficient. In the case of EB, setting `a_kappaWeighting` to `true` means the input gets multiplied by κa.

`virtual void  diagonalScale(T& a_rhs)`

This version of `diagonalScale` assumes that κ weighting is desired in EB applications.

- `virtual void  divideByIdentityCoef(T& a_S)`

  This function divides the input in place by the a coefficient.

4.9 TGA with time-dependent operator coefficients

Consider a parabolic partial differential equation in which a quantity $a\phi$ is conserved and in both a and ϕ are time dependent:

$$\begin{aligned} \frac{\partial(a\phi)}{\partial t} &= L\phi + S \\ a &= a(\vec{x}, t), \quad a > 0. \end{aligned} \tag{4.8}$$

For the purposes of this discussion, we assume that a can be updated explicitly, and that a^n and a^{n+1} are known and may be used in the solution of (4.8) to obtain ϕ^{n+1} from ϕ^n.

Equations of this form arise in the diffusion of energy and momentum in compressible hydrodynamic flows. In fluid flows with heat conduction, for example, the energy per unit volume E is related to both the mass density ρ and the temperature T by the expression

$$E = c_p \rho T \tag{4.9}$$

where c_p is the specific heat as measured under constant pressure. If we assume that c_p is constant within a material, the transfer of energy by heat conduction throughout a body with thermal conductivity K is described by the conservation equation

$$c_p \frac{\partial(\rho T)}{\partial t} = \nabla \cdot (K \nabla T) + S \tag{4.10}$$

where S is an external source of energy. The total energy is conserved in this process.

4.9.1 Implicit TGA update for ϕ

We begin by expressing (4.8) in a form that more closely resembles (4.5). The time change in $a\phi$ is

$$\frac{\partial(a\phi)}{\partial t} = \left[\left(\frac{\partial a}{\partial t} \right) \phi + a \left(\frac{\partial \phi}{\partial t} \right) \right]. \tag{4.11}$$

Thus, we can express the evolution equation for ϕ as

$$a\frac{\partial \phi}{\partial t} = L(\phi) + S - \left(\frac{\partial a}{\partial t}\right)\phi \tag{4.12}$$

Since we have explicitly updated a and have a^{n+1} as well as a^n, we may explicitly estimate the second term on the right hand side of (4.12) as a source using the first-order finite difference

$$\left(\frac{\partial a}{\partial t}\right)\phi \approx \frac{(a^{n+1} - a^n)}{\Delta t}\phi^n. \tag{4.13}$$

Because this expression appears on the right hand side of (4.12), it is then multiplied by a factor of Δt and rendered second-order accurate.

In order to conserve $a\phi$, it is crucial that we use values of a at the correct time centerings in the TGA update for ϕ. We define

$$L_a(t) \equiv \frac{L(\phi, t)}{a(t)} \tag{4.14}$$

$$S_a(t) \equiv \frac{S(t) - \phi^n(a^{n+1} - a^n)/\Delta t}{a(t)} \tag{4.15}$$

where we have added t as a parameter to all time-dependent quantities to emphasize their time dependence. The TGA update for ϕ is then written

$$\begin{aligned}\phi^{n+1} = \quad & [I - \mu_1 L_a(t^{n+1})]^{-1}[I - \mu_2 L_a(t^*)]^{-1} \times \\ & \left([I + \mu_3 L_a(t^n)]\phi^n + \Delta t[I + \mu_4 L_a(t^{n+\frac{1}{2}})]S_a(t^{n+\frac{1}{2}})\right).\end{aligned} \tag{4.16}$$

where t^* is the "intermediate" time used in the TGA update. Hereafter, we use abbreviated superscripts on a, L, and S to refer to their time centerings.

Using the identity (4.6) and multiplying the last two terms by a^n/a^n and $a^{n+\frac{1}{2}}/a^{n+\frac{1}{2}}$ respectively, we obtain the expression

$$\begin{aligned}\phi^{n+1} = \quad & [a^{n+1}I - \mu_1 L^{n+1}]^{-1}a^{n+1}[a^* I - \mu_2 L^*]^{-1}a^* \times \\ & \left(\tfrac{1}{a^n}[a^n I + \mu_3 L^n]\phi^n + \tfrac{\Delta t}{a^{n+\frac{1}{2}}}[a^{n+\frac{1}{2}}I + \mu_4 L^{n+\frac{1}{2}}][\tfrac{S^{n+\frac{1}{2}} - \phi^n(a^{n+1} - a^n)/\Delta t}{a^{n+\frac{1}{2}}}]\right).\end{aligned} \tag{4.17}$$

This expression is more complicated than (4.7) because the various factors of a are evaluated at different times and thus do not cancel.

4.9.1.1 Embedded boundary considerations

The same analysis can be carried out in the presence of irregular cells near embedded boundaries, but since the volume fraction κ is constant in time, the same factors appear as in (4.7). In the Chombo implementation of this algorithm, factors of a and L in the numerator each include a factor of κ, but factors of a in the denominator do not, so we must take special care in computing ϕ^{n+1}.

To illustrate how these factors must be treated in Chombo, we include all factors of κ instead of accounting for cancellations. In this case the expression for ϕ^{n+1} within irregular cells with the appropriate factors of κ is

$$\phi^{n+1} = [\kappa a^{n+1} I - \mu_1 \kappa L^{n+1}]^{-1} \kappa a^{n+1} [\kappa a^* I - \mu_2 \kappa L^*]^{-1} \kappa a^* \times \left(\frac{1}{\kappa a^n} [\kappa a^n I + \mu_3 \kappa L^n] \phi^n + \frac{\Delta t}{\kappa a^{n+\frac{1}{2}}} [\kappa a^{n+\frac{1}{2}} I + \mu_4 \kappa L^{n+\frac{1}{2}}] [\frac{S^{n+\frac{1}{2}} - \phi^n (a^{n+1} - a^n)/\Delta t}{a^{n+\frac{1}{2}}}] \right). \tag{4.18}$$

The factors of κ in the denominator cancel with the factor, say, in front of a^*. We can implement this cancellation by calling `diagonalScale` with a `a_kappaWeighted=false` to indicate that a^* should *not* be multiplied by κ. Our final expression for ϕ^{n+1} for operators with time-dependent coefficients in irregular cells is

$$\phi^{n+1} = [\kappa a^{n+1} I - \mu_1 \kappa L^{n+1}]^{-1} \kappa a^{n+1} [\kappa a^* I - \mu_2 \kappa L^*]^{-1} a^* \times \left(\frac{1}{a^n} [\kappa a^n I + \mu_3 \kappa L^n] \phi^n + \frac{\Delta t}{a^{n+\frac{1}{2}}} [\kappa a^{n+\frac{1}{2}} I + \mu_4 \kappa L^{n+\frac{1}{2}}] [\frac{S^{n+\frac{1}{2}} - \phi^n (a^{n+1} - a^n)/\Delta t}{a^{n+\frac{1}{2}}}] \right). \tag{4.19}$$

4.9.2 Conservative calculation of $L(\phi)$

After ϕ^{n+1} is computed, we obtain a conservative value for $L(\phi)$ by calculating

$$L(\phi) = \frac{(a^{n+1}\phi^{n+1} - a^n \phi^n)}{\Delta t} - S. \tag{4.20}$$

4.9.3 Setting the time centering of a `LevelTGAHelmOp`

(4.17) is correct for any time-dependent entities $a(t)$ and $L(t)$, but we need a way to obtain these properly-centered values in practice. Since our algorithm is second-order accurate, we can compute a time-centered value for $a^{n+\mu}$ or $L^{n+\mu}$ by interpolating linearly between the beginning-of-step values a^n, L^n and their end-of-step values a^{n+1}, L^{n+1} *if we know their values at t^{n+1} beforehand*. The TGA algorithm sets the time centering of a `LevelTGAHelmOp` by calling the `setTime` method:

```
virtual void setTime(Real a_oldTime, Real a_mu, Real a_dt)
```

Here, a_oldTime is the beginning-of-step time t^n, a_mu is the fraction of the time step that has elapsed, and a_dt is the step size. With these three parameters, one can compute $t^{n+\mu}$ and can interpolate a or L between t^n and t^{n+1} as needed. For example, the time-interpolated value for a at a fraction μ through the step is

$$\begin{aligned} a^{n+\mu} &= a^n + \mu\Delta t \left(\frac{a^{n+1} - a^n}{\Delta t} \right) \\ &= (1-\mu)a^n + \mu a^{n+1} \end{aligned} \tag{4.21}$$

Implementation details

Currently, the TGA solver assumes that a LevelTGAHelmOp is time independent unless otherwise specified. If you wish to create a time-dependent subclass of LevelTGAHelmOp, that subclass must call the single-argument LevelTGAHelmOp constructor that specifies that it is time-dependent:

```
LevelTGAHelmOp(bool a_isTimeDependent)
```

This allows the TGA solver to query the operator via its isTimeDependent method:

```
bool isTimeDependent() const
```

Note that LevelTGAHelmOp::isTimeDependent is *not* a virtual method– it simply returns the flag passed to the above constructor. The time dependence of an operator object must be established at the time of its construction.

Additionally, if the operator needs to interpolate any of its coefficients over the time step, it should store the beginning-of-step and end-of-step values for such coefficients and ensure that they are valid before the TGA solve.

Chapter 5

AMRTimeDependent

5.1 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 [7] 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 that 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 a 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 [6]. We define

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

where $U^l = U^l(t^l)$. 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 [7] for advancing the solution in time is given in pseudo-code in figure 5.1. 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

$$\tilde{\Omega} = \mathcal{G}(\Omega^l, p) \cap \Gamma^l,$$

namely $\tilde{U} : \tilde{\Omega} \to \mathbb{R}^m$, where

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

$$\text{where} \quad \alpha = \frac{t^l - t^{l-1}}{\Delta t^{l-1}}.$$

Regridding is performed in the following steps.

1. Construct $\mathcal{I}^l \subset \Omega^l, l = l_{base}, ..., l_{max} - 1$ corresponding to those cells for which a user-specified measure of the error exceeds a specified tolerance.

2. Generate new grids $\Omega^{l,new}, l = l_{base} + 1, ..., l_{max}$ on which the new solution is to 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. If $l_{base} > 0$, these conditions impose some constraints on $\mathcal{I}^l$, which are met typically by reducing the size of the $\mathcal{I}^l$'s prior to the grid generation step.

3. Initialize the new data $U^{l,new}(t^l)$. For $l = l_{base}, U^l = U^{l,new}$. For $l = l_{base} + 1, ..., l_{max}$,

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

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 $l > 0$
    $\delta F_d^l := \delta F_d^l + \frac{1}{n_{ref}^{l-1}} \langle F_d^l \rangle$ 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 $(n_{step}^l = 0 \bmod n_{regrid})$ and $(n_{step}^{l-1} \neq 0 \bmod n_{regrid})$
    regrid(l)
end if

Figure 5.1: Pseudo-code description of the Berger–Colella AMR algorithm for hyperbolic conservation laws.

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 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

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

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,

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

for $\boldsymbol{i} \in \mathcal{C}_2(\Gamma)$. Here $\tau_{\boldsymbol{i}}^{R,2h} = C\tau_{\boldsymbol{i}}^h + O(h^{p+1})$ 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 $|\tau_{\boldsymbol{i}}^R|$ 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.

5.2 Classes `AMR` and `AMRLevel`

The class `AMR` implements a framework for the Berger–Oliger adaptive mesh refinement (AMR) algorithm for time-dependent simulations [6]. 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

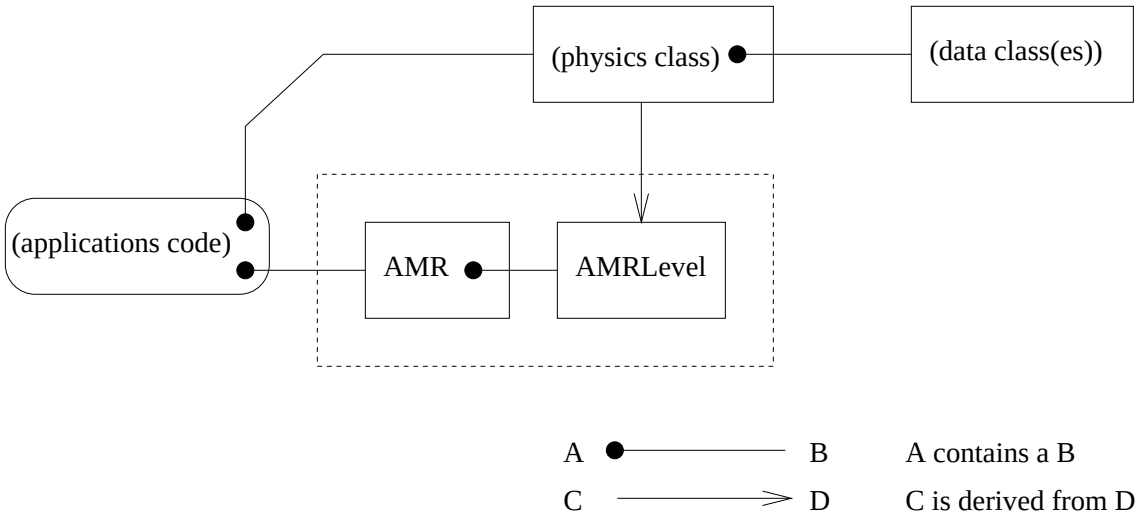

Figure 5.2: Class structure. `AMR` contains some `AMRLevels`. Physics class is derived from `AMRLevel` and contains the data representation.

level in time. The class `AMR` implements refinement of the time step Δt based on the refinement of the grid.

5.2.1 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 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 5.2.

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.

5.2.2 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 (max level, 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_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.
  - 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_prob_domain (not modified): Problem domain on the base level.
  - 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 user needs to call either this function 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 function 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. Examples of applications that use the `AMR` class to implement an adaptive Godunov method are given in `Chombo/example/AMRGodunov`.

5.2.3 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;`

  Create tagged cells for mesh refinement at initialization.

- ```
  virtual void preRegrid(int a_base_level,
                         const Vector<Vector<Box> >& a_new_grids);
  ```

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

- `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;`

  Read checkpoint header.

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

  Read checkpoint data for this level.

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

  Write 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.

5.2.4 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.

5.3 AMRTimeDependent Example: Advection-Diffusion

We provide an example which uses the AMRTimeDependent infrastructure to solve the advection-diffusion equation. Given an analytic velocity field $\vec{v}$ and a diffusion coefficient ν, we evolve an advected and diffused scalar ϕ using the advection-diffusion equation:

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\vec{v}\phi) = \nu \Delta \phi.$$

The algorithm is mostly as described in previous sections. The solution to the Riemann problem is given by simple upwinding. F_d, the advective flux in the d direction, is given by $F_d = v^d \phi$. The characteristic values are the velocity components.

The main difference between the algorithm used in this example and that in Figure 5.1 is in the approach taken to flux correction during the synchronization step. The total flux is the advective flux minus the diffusive flux: $F_d^{tot} = v^d \phi - \nu \nabla \phi$. Because the fluxes used to update the solution include both advective and diffusive fluxes, simply performing the explicit refluxing correction step:

$$\phi^l(t^l + \Delta t^l) := \phi^l(t^l + \Delta t^l) - \Delta t^l D_R(\delta F^{l+1})$$

will be unstable because it represents a forward-Euler update for the heat equation (which requires that $\Delta t \approx O(h^2)$ for stability). To preserve stability, we instead apply the flux correction implicitly, as described in [19]:

We first solve a Helmholtz equation for a correction:

$$(I - \Delta t^{\ell_{base}} \nu \Delta^{comp}) \delta\phi = \Delta t^\ell D_R(\delta \vec{F}^{tot,\ell+1}) \quad \text{for} \quad \ell \geq \ell_{base} \tag{5.1}$$

Then, the correction is added to the solution:

$$\phi^\ell := \phi^\ell + \delta\phi^\ell \quad \text{for} \quad \ell \geq \ell_{base}. \tag{5.2}$$

5.3.1 AdvectionDiffusion-Specific Classes

The AMR advection-diffusion application in `Chombo/example/AMRGodunov` uses the AMRTimeDependent infrastructure extensively. These are the classes that are specific to this application.

- `AdvectPhysics` is the `GodunovPhysics`-derived class. It provides `PatchGodunov` the physics-specific information needed to advance the solution.

- `LevelAdvect` is used to advance an advection-diffusion solution one step:

$$\phi^{new} = \phi^{old} - \Delta t(\nabla \cdot (\vec{v}\phi))^{n+\frac{1}{2}}.$$

  In the case of non-zero diffusion, we use this advance as an input to a semi-implicit advance using the `LevelTGA` class described in section 4.7. In the pure advection case, we can just use this update.

- `AMRLevelAdvectDiffuse` is the `AMRLevel`-derived class that drives the calculation.

- `AMRLevelAdvectDiffuseFactory` is the factory class that the user sends to `AMR`. This class generates `AMRLevelAdvectDiffuse` objects for `AMR`.

5.3.2 Usage Pattern

In broad strokes, the driver for the `AMRLevelAdvectDiffuse` example looks something like this:

```
// read inputs
ParmParse ppgodunov;

//define the domain of computation
ProblemDomain prob_domain;
getProblemDomain(prob_domain);

//define the initial and boundary conditions
RefCountedPtr<AdvectTestIBC> ibc;
getAdvectTestIBC(ibc);

//define the AdvectPhysics object
AdvectPhysics advPhys;
advPhys.setPhysIBC(&(*ibc));

//pick an advection velocity function
AdvectionVelocityFunction velFunc;
getAdvectionVelocityFunction(velFunc);

//define the factory object
RefCountedPtr<AMRLevelAdvectDiffuseFactory>  amrg_fact;
getAMRLADFactory(amrg_fact, velFunc, advPhys);

//define the AMR object
AMR amr;
defineAMR(amr, amrg_fact, prob_domain, a_refRat);

//sets all the amr parameters (as opposed to a fixed grid run)
```

```
  setupAMRForAMRRun(amr);

  // run
  amr.run(a_stopTime,a_nstop);

  // output last pltfile and statistics
  amr.conclude();
```

To use these functions directly (all of these functions exist in `AdvectDiffuseUtils`), one must have an input file that looks something like this:

```
##cfl number and its initial time counterpart
cfl = 0.9
initial_cfl = 0.1
##nu
diffusion_coef = 0.001
##how much output one wants from the application
verbosity = 1
##max number of time steps
max_step = 100
##final solution time
max_time = 100.0
##number of buffer cells
tag_buffer_size = 3
##tagging threshold for undivided difference of solution
refine_thresh = 0.05
##how often we regrid
regrid_interval = 2 2 2 2 2 2
##how fast time step can grow and how far out of bounds it can get
max_dt_growth = 1.1
dt_tolerance_factor = 1.1
## size of the domain
domain_length = 1.0
## distance (in level l cells) between levels l+1 and l-1
grid_buffer_size = 1
##do periodic boundary conditions or not
periodic_bc = 1 1 1
##fixed time step size.  negative if not fixing time step
fixed_dt   = -1.0
##maximum AMR level number
max_level = 2
##coarsest AMRLevel grid size
n_cell =  64 64 64
##refinement ratios between levels
ref_ratio = 2 2 2 2
##blocking factor
```

```
block_factor =  4
##maximum size of grids
max_grid_size = 64
##fill ratio for Berger-Rigoutsos
fill_ratio = 0.75
## how often to checkpoint.  negative if no checkpoints
checkpoint_interval = -1
## how often to dump plotfile.  negative if no plotfiles.
plot_interval = 10
##prefix of plotfiles
plot_prefix  = plt
##prefix of checkpoint files
chk_prefix = chk
blob_center = 0.75 0.5 0.5
blob_radius = 0.1
#0 = constant
#1 = rotating
advection_vel_type = 0
use_limiting = true
amrmultigrid.num_smooth  = 8
amrmultigrid.tolerance   = 1.0e-10
amrmultigrid.num_mg      = 1
amrmultigrid.norm_thresh = 1.0e-10
amrmultigrid.hang_eps    = 1.0e-10
amrmultigrid.max_iter    = 100
amrmultigrid.verbosity   = 1
##1 = neumann, 0 = dirichlet
bc_lo   = 1 1 1
bc_hi   = 1 1 1
bc_value = 0.
```

5.3.3 `AMRLevelAdvectDiffuseFactory` Interface

We use the following interface to create an `AMRLevelAdvectDiffuseFactory`.

```
AMRLevelAdvectDiffuseFactory(const AdvectPhysics&        gphys,
                             AdvectionVelocityFunction   advFunc,
                             BCHolder                    bcFunc,
                             const Real&                 cfl,
                             const Real&                 domainLength,
                             const Real&                 refineThresh,
                             const int&                  tagBufferSize,
                             const Real&                 initialDtMultiplier,
                             const bool&                 useLimiting,
                             const Real&                 nu);
```

- gphys – The AdvectPhysics used to create the advection term.
- advFunc – The function that provides the advection velocity. This function must be of the form

```
typedef Real (*AdvectionVelocityFunction)(const RealVect& point,
                                          const int&      velComp);
```

- bcFunc – The boundary condition class sent to solve the diffusion equation. See section 4.6.4 for details.
- cfl – The CFL number.
- domainLength – The physical length of the domain.
- refineThresh – Undivided gradient size over which a cell will be tagged for refinement.
- tagBufferSize – Number of buffer cells around each tagged point that will also be tagged.
- initialDtMultiplier – CFL number at the beginning of the calculation.
- useLimiting – Whether to use vanLeer limiting. Turn this to true unless you are doing a convergence test.
- nu – Diffusion coefficient.

5.3.4 LevelAdvect Interface

LevelAdvect is used to advance an advection-diffusion solution one step. In the case of non-zero diffusion, we use this advance as an input to a semi-implicit advance using LevelTGA. In the pure advection case, we can just use this update. If a user is simply using the AMRLevelAdvectDiffuse application, this class is entirely internal to AMRLevelAdvectDiffuse. This discussion is provided for the benefit for those who want to use this object for other applications.

We define a LevelAdvect with the following interface

```
void define(const AdvectPhysics&        gphys,
            const DisjointBoxLayout&    thisDisjointBoxLayout,
            const DisjointBoxLayout&    coarserDisjointBoxLayout,
            const ProblemDomain&        domain,
            const int&                  refineCoarse,
            const bool&                 useLimiting,
```

```
            const Real&                 dx,
            const bool&                 hasCoarser,
            const bool&                 hasFiner);
```

- gphys – The AdvectPhysics used to create the advection term.
- domain – The domain of the computation.
- refineCoarse – The refinement ratio to the next coarser level.
- useLimiting – Whether to use vanLeer limiting. Turn this to true unless you are doing a convergence test.
- dx – The grid spacing.
- hasCoarser – Whether there is a coarser AMR level.
- hasFiner – Whether there is a finer AMR level.

We present the interface function that advances the solution.

$$\phi^{new} = \phi^{old} - \Delta t(\nabla \cdot (\vec{v}\phi))^{n+\frac{1}{2}}$$

```
Real step(LevelData<FArrayBox>&         U,
          LevelFluxRegister&            finerFluxRegister,
          LevelFluxRegister&            coarserFluxRegister,
          LevelData<FluxBox>&           advectionVelocity,
          const LevelData<FArrayBox>&   S,
          const LevelData<FArrayBox>&   UCoarseOld,
          const Real&                   TCoarseOld,
          const LevelData<FArrayBox>&   UCoarseNew,
          const Real&                   TCoarseNew,
          const Real&                   time,
          const Real&                   dt);
```

- U – The solution. Input is the old solution ϕ^{old}. Input is the new solution ϕ^{new}.
- finerFluxRegister – The flux register with the next finer level.
- coarserFluxRegister – The flux register with the next coarser level.
- advectionVelocity – The holder for the advection velocity. This is held in a LevelData to make this object of somewhat more general utility.
- S – Source term. In the case of advection-diffusion, $S = \nu\Delta\phi$.
- UCoarseOld – Solution at next coarser level at the old time.

- `TCoarseOld` – Time at which `UCoarseOld` exists.
- `UCoarseNew` – Solution at next coarser level at the new time.
- `TCoarseNew` – Time at which `UCoarseNew` exists.
- `time` – Time at which U exists before advance happens (t^{old}).
- `dt` – Time step.

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 and 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 the 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 VisIt post-processing and visualization 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 VisIt. 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 ''FABView.H''`[1]
   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 VisIt 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 uses the plotfile format of WriteAMRHierarchyHDF5, and which can then be viewed with VisIt (called separately). The writeFAB function writes the input FArrayBox into a file named fab.hdf5, 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)
```

[1]The actual function declarations are in `AMRIO.H`, but `FABView.H` contains fake "calls" to the various functions to ensure that they are all included in the executable file.

Arguments:
No Arguments are modified.

- a_data : data to be written

A file named fab.hdf5 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 uses the plotfile format of WriteAMRHierarchyHDF5, and which can then be viewed with VisIt (called separately). The writeLevel function writes the input LevelData<FArrayBox> into a file named LDF.hdf5, 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.hdf5 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.

6.3.3 Functions writeDBL and writeDBLname

The functions writeDBL and writeDBLname write the data from a single DisjointBoxLayout into a file which uses the plotfile format of WriteAMRHierarchyHDF5, and which can then be viewed with VisIt (called separately). It does this by creating a LevelData<FArrayBox> using the input DisjointBoxLayout with the data initialized to the processor ID of each grid, and then calling the associated writeLevel functions. The writeDBL function writes the input DisjointBoxLayout into a file named DBL.hdf5, while the writeDBLname allows the user to specify the name of the output file.

Notes:

- Data is passed using a pointer to a DisjointBoxLayout.

```
void
writeDBL(const DisjointBoxLayout* a_data)
```

Arguments:
No Arguments are modified.

- a_data : data to be written

A file named DBL.hdf5 is created if it doesn't already exist and overwritten if it does exist.

```
void
writeDBLname(const DisjointBoxLayout* 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

Using PETSc in Chombo

PETSc (Portable, Extensible Toolkit for Scientific Computation) is a library of of tools for computing with structured and unstructured grids. Its primary functionality is a time stepping class (TS), which has a nonlinear solver (SNES), which has a linear solver (KSP). Chombo provides support for constructing PETSc composite grid matrices (`class PetscCompGrid`) from Chombo's block structured hierarchy of grids. Applications can then use these sparse (unstructured "AIJ") matrices in PETSc's (time,nonlinear,linear) solvers.

This approach to solvers in Chombo is motivated by applications that have operators that are not solved well with Chombo's geometric multigrid solver (§4.4.3). The first application to use this functionality is the BISICLES ice sheet modeling code.

7.1 Building Chombo with PETSc

Follow the instructions on the PETSc web page and document to build PETSc if there is not a PETSc build on your machine. This involves setting the make variables PETSC_DIR and PETSC_ARCH in you environment or as an argument to make. Build your Chombo application with `USE_PETSC=TRUE`.

7.2 PETSc Composite Grid Classes

Chombo provides a base class `PetscCompGrid` that provides a framework for constructing PETSc matrices from a Chombo AMR hierarchy of grids. Specific operators are derived from this base class (e.g., `PetscCompGridPois`). An example of the usage to solve a Laplacian is in `lib/test/AMRElliptic/testPetscCompGrid.cpp`.

7.3 PETSc Solver Usage

There are three components to building a PETSc solver in Chombo: 1) making a matrix (§7.3.1), 2) make a solver (§7.3.2), and 3) make the operator (§7.3.3).

7.3.1 Making a PETSc Matrix

Chombo currently has a constant coefficient Laplacian class (`PetscCompGridPois`) and a variable coefficient viscous tensor operator class `PetscCompGridVTO`. Figure 7.1 shows an example making a matrix for the constant coefficient Laplacian.

```
#include "PetscCompGridPois.H"
....
PetscCompGridPois petscop(0.,-1.,2); /* alpha, beta, and order */
RefCountedPtr<ConstDiriBC> bcfunc = RefCountedPtr<ConstDiriBC>(new
ConstDiriBC(1,petscop.getGhostVect()));
BCHolder bc(bcfunc);
petscop.setCornerStencil(true);
petscop.define(cdomain,grids,refratios,bc,cdx*RealVect::Unit);
petscop.setVerbose(1);
Mat A; /* linear system matrix */
A = petscop.getMatrix();
```

Figure 7.1: Example of creating a matrix from `testPetscCompGrid.cpp`

This example has a simple operator "define" process: α and β, in the matrix for the operator $\alpha u + \nabla \cdot \beta \nabla u$, and the order of the discretization are arguments to the constructor. The type of stencil is set with an operator specific method. The actual `define` method is a base class method and provides the grid hierarchy, domain, and boundary condition class. See `lib/src/AMRElliptic/PetscCompGrid.H` for the current version and Figure 7.2 for reference. The class `ConstDiriBC` implements homogenous Dirichlet boundary

```
virtual void define(
 const ProblemDomain &a_cdomain,
 Vector<DisjointBoxLayout> &a_grids,
 Vector<int> &a_refratios,
 BCHolder a_bc,
 const RealVect &a_cdx,
 int a_ibase=0,
 int a_maxLev=-1);
```

Figure 7.2: `PetscCompGrid::define`

conditions. Other boundary conditions can be implemented by deriving from the base class CompBC in lib/src/AMRElliptic/PetscCompGrid.H

7.3.2 Making a PETSc Solver

Figure 7.3 shows an example of a PETSc linear solver.

```
// solve A phi = rhs
Vec x, b; /* approx solution, RHS */
KSP ksp; /* linear solver context */
ierr = MatGetVecs(A,&x,&b); CHKERRQ(ierr);
ierr = petscop.putChomboInPetsc(rhs,b); CHKERRQ(ierr);
ierr = KSPCreate(PETSC_COMM_WORLD, &ksp); CHKERRQ(ierr);
ierr = KSPSetOperators(ksp, A, A, DIFFERENT_NONZERO_PATTERN);
CHKERRQ(ierr);
ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
ierr = KSPSolve(ksp, b, x); CHKERRQ(ierr);
ierr = KSPDestroy(&ksp); CHKERRQ(ierr);
ierr = petscop.putPetscInChombo(x,phi); CHKERRQ(ierr);
```

Figure 7.3: Example of a linear solver from testPetscCompGrid.cpp

The PetscCompGrid class provides helper methods to move vector data between PETSc and Chombo data structures (putChomboInPetsc and putPetscInChombo). The rest of this solver code is standard PETSc and one can refer to the PETSc documentation for more information.

7.3.3 Making an operator

The PetscCompGrid class has one pure virtual method: PetscErrorCode createOpStencil(IntVect, int, DataIterator, StencilTensor &) = 0;. Figure 7.4 shows an example of a simple 5 and 7 point stencil for the constant coefficient Laplacian. Note, the first two arguments are the components of a multilevel cell index, which can be used to construct the multilevel cell index object IndexML. The third argument is a data iterator that could be use to, for instance, index into a coefficient data structure; this example does not use this argument. Chombo provides some build-in operators and users are encouraged to implement their own as needed.

```
PetscErrorCode PetscCompGridPois::createOpStencil(
    IntVect a_iv,
    int a_ilev,
    DataIterator a_dit,
    StencilTensor &a_sten)
 {
     Real dx=m_dxs[a_ilev][0],idx2=1./(dx*dx);
     PetscFunctionBeginUser;
     StencilTensorValue &v0 = a_sten[IndexML(a_iv,a_ilev)];
     v0.define(1);
     v0.setValue(0,0,m_alpha - m_beta*2.*SpaceDim*idx2);
     for (int dir=0; dir<CH_SPACEDIM; ++dir)
       {
         for (SideIterator sit; sit.ok(); ++sit)
         {
           int isign = sign(sit());
           IntVect jiv(a_iv); jiv.shift(dir,isign);
           StencilTensorValue &v1 = a_sten[IndexML(jiv,a_ilev)];
           v1.define(1);
           v1.setValue(0,0,m_beta*idx2);
         }
       }
     PetscFunctionReturn(0);
 }
```

Figure 7.4: Example of an operator from `testPetscCompGridPois.cpp`

Chapter 8

Parallel Programming with Chombo

8.1 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.

8.2 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 of release 3.2 it is also possible to use hybrid parallelism with OpenMP. When using Chombo's native iterators, it suffices to modify looping over the boxes. We use the same iterators, however, the boxes are indexed explicitly to let OpenMP know that they are parallelizable. The lower level code has been made thread-safe and examples of hybrid parallelism are available in `AMRTimeDependent` and `AMRElliptic/AMRPoissonOp.*`. Note that the memory tracking feature of Chombo is not yet usable with hybrid parallelism.

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 8.5 for details.

8.3 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 8.4 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.

8.4 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 [11]. The interface to LoadBalance is given by

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

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

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 8.1.

8.5 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 8.5.1.

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.

8.5.1 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 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 9

Chombo Fortran

9.1 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 and 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 designed to create abstractions which avoid these problems. Chombo Fortran allows the C++-Fortran programmer many advantages.

- The complicated data structures (classes) provided by Chombo in C++ can be passed to and used in Fortran routines.
- 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).

9.2 ChF Fortran macros

There are three classes of Fortran macros in ChF: array declaration, array access and dimension-handling. The array declaration macros are used to specify arguments to Fortran subroutines that will be called from C++. The array access macros are used to reference these arguments in the body of Fortran subroutines. The dimension-handling macros are used in the body of the Fortran subroutines to create dimension-independent code.

9.3 dimension-handling macros

The dimension-handling macros are:

- `CHF_DDECL` and `CHF_AUTODECL` for declaring variables and creating argument lists
- `CHF_DTERM` for choosing multiple expressions or statements based on dimension
- `CHF_DINVTERM` for choosing multiple expressions or statements based on dimension and reversing their order
- `CHF_DSELECT` for choosing one expression or statement based on dimension
- `CHF_AUTOID` for setting multiple variables based on dimension
- `CHF_MULTIDO` and `CHF_AUTOMULTIDO` for handling nested DO loops
- `CHF_ENDDO` goes with `CHF_MULTIDO` and `CHF_AUTOMULTIDO`

`CHF_DDECL[arg0;arg1;arg2]` translates to arg0, arg1, arg2 (in three dimensions). 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.

This is also used in creating argument lists for calling other routines. Using the previous example, to call a routine named `FOO` that expects SpaceDim arguments, one would write the call as

```
call FOO( CHF_DDECL[ u; v; w ] )
```

The CHF_AUTODECL macro performs a similar function by expanding a root: CHF_AUTODECL[arg] will expand to arg0, arg1, arg2 (in three dimensions). This can result in more compact code, especially for code intended to support higher dimensionality.

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. For example:

```
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.

The CHF_DINVTERM[arg0;arg1;arg2] macro is a variation on CHF_DTERM which reverses the chosen arguments. It translates to arg2arg1arg0 in three dimensions and arg1arg0 in two dimensions. This is useful if one has indexing loops in code that is dimension-dependent. For example:

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

CHF_DINVTERM[
do ii = 0,10;
do jj = 0,10;
do kk = 0,10]
```

Like CHF_DTERM, this macro respects carriage returns and other white space.

The CHF_DSELECT macro is a variation on CHF_DTERM. Instead of choosing the arguments from 1 to SpaceDim, it chooses *only* the SpaceDim'th argument. This is useful for expressions that are different for each dimension. For example:

```
rho = CHF_DSELECT[ cos(x) ; sin(x*y) ; cos(x*z)*sin(y*z) ]
```

Like CHF_DTERM, this macro respects carriage returns and other white space.

The macro CHF_AUTOID[ii;idir] generates

```
CHF_DTERM[
ii0 = CHF_ID(0,idir);
ii1 = CHF_ID(1,idir);
ii2 = CHF_ID(2,idir)]
```

and can also be called with an additional optional argument, where CHF_AUTOID[ii;idir;factor] generates

```
CHF_DTERM[
ii0 = factor*CHF_ID(0,idir);
ii1 = factor*CHF_ID(1,idir);
ii2 = factor*CHF_ID(2,idir)]
```

CHF_MULTIDO and CHF_AUTOMULTIDO are used to iterate over a box in a dimension-independent fashion by setting up nested Fortran DO loops. 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 the 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.

CHF_MULTIDO can also be used to iterate with a stride. The syntax for this is CHF_MULTIDO[box;i;j;k;2], where the "2" could be any integer constant except 0. A negative stride will make the loop iterate backward in each dimension (from the high corner to the low corner). Be warned that using a variable name instead of an integer constant will not produce the desired result because ChomboFortran will just think you've coded a 4-dimensional loop so it will ignore the last variable.

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 + CHF_DTERM[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 this example.

The CHF_AUTOMULTIDO macro also sets up nested loops, but constructs the indices of the loops based on a root. Specifically, CHF_AUTOMULTIDO[box;i] is the same as CHF_MULTIDO[box;i0;i1;i2]. Strides are also supported, so CHF_AUTOMULTIDO[box;i;2] is the same as CHF_MULTIDO[box;i0;i1;i2;2] (note that the indices start with 0 instead of 1, which is consistent with the conventions elsewhere in ChomboFortran).

Here is the previous example written using the AUTO macros:

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

```
integer CHF_AUTODECL[i]
integer productsum

productsum = 0
CHF_AUTOMULTIDO[box;i]
  productsum = productsum + CHF_DTERM[i0;*i1;*i2]
  array(CHF_AUTOIX[i]) = productsum
CHF_ENDDO

return
end
```

9.4 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 IntVect, RealVect, 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* 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 lowest index of the array A in the second dimension (dimensions are counted starting at 0). As a special case, CHF_UBOUND[V] is the same as CHF_UBOUND[V;0] and is used with Vectors and 1D arrays of basic data types.

The "_CONST" qualifier in the macro names indicates that the variable named in the macro is not modified in the Fortran subroutine. This form of the macros should be used when the C++ variable is declared 'const'. 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 declaration macros and their uses.

- CHF_INT[<*arg*>] Declare a scalar integer argument.
- CHF_CONST_INT[<*arg*>] Declare a read-only scalar integer argument.
- CHF_REAL[<*arg*>] Declare a scalar floating point argument.

- CHF_CONST_REAL[<*arg*>] Declare a read-only scalar floating point argument.
- CHF_COMPLEX[<*arg*>] Declare a scalar complex argument.
- CHF_CONST_COMPLEX[<*arg*>] Declare a read-only scalar complex argument.
- CHF_REALVECT[<*arg*>] Declare a real vector of SpaceDim length argument (indices go from 0 to SpaceDim-1).
- CHF_CONST_REALVECT[<*arg*>] Declare a constant real vector of SpaceDim length argument (").
- CHF_INTVECT[<*arg*>] Declare an integer vector of SpaceDim length argument (").
- CHF_CONST_INTVECT[<*arg*>] Declare a constant integer vector of SpaceDim length argument (").
- CHF_I1D[<*arg*>] Declare a C array of integers (indices go from 0 to CHF_UBOUND[<*arg*>]).
- CHF_CONST_I1D[<*arg*>] Declare a read-only C array (").
- CHF_R1D[<*arg*>] Declare a C array of reals (").
- CHF_CONST_R1D[<*arg*>] Declare a read-only C array of reals (").
- CHF_VI[<*arg*>] Declare a Chombo Vector<int>.
- CHF_CONST_VI[<*arg*>] Declare a read-only Chombo Vector<int>.
- CHF_VR[<*arg*>] Declare a Chombo Vector<Real>.
- CHF_CONST_VR[<*arg*>] Declare a read-only Chombo Vector<Real>.
- CHF_VC[<*arg*>] Declare a Chombo Vector<Complex>.
- CHF_CONST_VC[<*arg*>] Declare a read-only Chombo Vector<Complex>.
- CHF_FIA[<*arg*>] Declare a multi-component integer C++ BaseFab argument.
- CHF_CONST_FIA[<*arg*>] Declare a read-only multi-component integer BaseFab argument.
- CHF_FRA[<*arg*>] Declare a multi-component floating point BaseFab argument.
- CHF_CONST_FRA[<*arg*>] Declare a read-only multi-component floating point BaseFab argument.
- CHF_FIA1[<*arg*>] Declare a single-component integer BaseFab argument.

- CHF_CONST_FIA1[<*arg*>) Declare a read-only single-component integer BaseFab argument.
- CHF_FRA1[<*arg*>] Declare a single-component floating point 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.

Chombo Fortran preprocessing of arguments can be disabled by adding the comment ! NO_CHF to the end of the line with the subroutine statement. In a ".ChF" file, this should only be used where absolutely necessary (an example would be an internal procedure as shown in section 9.9.4). Ordinary Fortran subroutines should normally be placed in a separate ".F" file.

9.5 Access macros

- CHF_LBOUND[<*arg*>;<*dim*>] Access the lower bound of a BaseFab or Box <*arg*> in constant dimension <*dim*>. Returns an integer variable.
- CHF_UBOUND[<*arg*>;<*dim*>] Access the upper bound of a BaseFab or Box <*arg*> in constant dimension <*dim*>. Returns an integer variable. Also used to access the upper bound of a 1D array or Chombo Vector, in which case <*dim*> need not be specified.[1]
- CHF_NCOMP[<*arg*>] Access the number of components in the BaseFab <*arg*>. Returns an integer. Note that the components in Fortran code are numbered from 0 to CHF_NCOMP(<*arg*>)-1 to be consistent with the requirements of C++.

[1] the upper bound of a 1D array is always one less than the dimension specified in the C++ call to CHF_I1D or CHF_R1D.

- CHF_IX[<*index0*>;<*index1*>;<*index2*>] Access an element of an array declared with one of the F*A* macros.
- CHF_AUTOIX[<*indexRoot*>] Access an element of an array declared with one of the F*A* macros. Expands the indexRoot so that CHF_AUTOIX[i] is the same as CHF_IX[i0;i1;i2].
- CHF_OFFSETIX[<*indexRoot*>;<*offsetRoot*>] Access an element of an array declared with one of the F*A* macros. Similar to CHF_AUTOIX, but with an offset. For example, CHF_OFFSETIX[i;-ii] expands to be the same as CHF_IX[i0-ii0; i1-ii1; i2-ii2].
- 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 isn't really a macro—it is a 6x6 (to support up to 6D) 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, Box, 1D array or Chombo Vector macros.
- The <*dim*> macro argument must be an integer constant in the range 0...CH_SPACEDIM-1.
- The <*dim1*> and <*dim2*> macro arguments must be integer variables or constants 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.

9.6 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.[2]

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) in C++ and 1 in Fortran,
- the 1D array macros (CHF_*1D) take 2 arguments (array, length) in C++ and 1 in Fortran,
- for each Fortran subroutine <*proc*>, a C++ macro FORT_<*proc*> is defined.

9.7 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++ declaration macros and their uses.

- CHF_INT(<*arg*>) Pass a scalar int variable.
- CHF_CONST_INT(<*arg*>) Pass a const scalar int variable.
- CHF_REAL(<*arg*>) Pass a scalar Real variable.
- CHF_CONST_REAL(<*arg*>) Pass a const scalar Real variable.

[2]application code should never use the CHFp_ macros directly.

- CHF_COMPLEX(<*arg*>) Pass a scalar Complex variable.
- CHF_CONST_COMPLEX(<*arg*>) Pass a const scalar Complex variable.
- CHF_REALVECT(<*arg*>) Pass a RealVect variable.
- CHF_CONST_REALVECT(<*arg*>) Pass a constant RealVect variable.
- CHF_INTVECT(<*arg*>) Pass a IntVect variable.
- CHF_CONST_INTVECT(<*arg*>) Pass a constant IntVect variable.
- CHF_I1D(<*arg*>,<*len*>) Pass a 1D array of ints of length <*len*>.
- CHF_CONST_I1D(<*arg*>,<*len*>) Pass a constant 1D array of ints of length <*len*>.
- CHF_R1D(<*arg*>,<*len*>) Pass a 1D array of Reals of length <*len*>.
- CHF_CONST_R1D(<*arg*>,<*len*>) Pass a constant 1D array of Reals of length <*len*>.
- CHF_VI(<*arg*>) Pass a Vector<int>.
- CHF_CONST_VI(<*arg*>) Pass a constant Vector<int>.
- CHF_VR(<*arg*>) Pass a Vector<Real>.
- CHF_CONST_VR(<*arg*>) Pass a constant Vector<Real>.
- CHF_VC(<*arg*>) Pass a Vector<Complex>.
- CHF_CONST_VC(<*arg*>) Pass a constant Vector<Complex>.
- CHF_FIA(<*arg*>) Pass a BaseFab<int> .
- CHF_CONST_FIA(<*arg*>) Pass a const BaseFab<int> .
- CHF_FRA(<*arg*>) Pass a BaseFab<Real> .
- CHF_CONST_FRA(<*arg*>) Pass a const BaseFab<Real>.
- CHF_FIA1(<*arg*>,<*comp*>) Pass a single component of a BaseFab<int>.
- CHF_CONST_FIA1(<*arg*>,<*comp*>) Pass a single const component of a BaseFab<int>.
- CHF_FRA1(<*arg*>,<*comp*>) Pass a single component of a BaseFab<Real>.
- 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.

The macros CHF(_CONST)_FIA, CHF(_CONST)_FRA, CHF(_CONST)_FIA1, and CHF(_CONST)_FRA1 all come in a version with the appendix _SHIFT and take an extra IntVect describing the amount to shift the associated box before passing the argument to Fortran. This can be used to align box data and/or change the reference frame. No change is required to the corresponding Fortran declaration macro and using the shift macro has no effect on the original data. In other words, there is no need to un-shift. A common use is to shift all boxes to the positive quadrant so that coarsening can be achieved by simple integer division. In the C++ code:

```
{
  const int refRatio = 2;
  Box crBox (-IntVect::Unit, IntVect::Unit);
  Box fnBox = refine(crBox, refRatio);
  FArrayBox crFRA(crBox, 1);
  crFRA.setVal(0.);
  FArrayBox fnFRA(fnBox, 1);
  fnFRA.setVal(1.);

  const IntVect crShiftToZero = crBox.smallEnd();
  const IntVect fnShiftToZero = scale(crShiftToZero, refRatio);
  FORT_SUMFINE(CHF_FRA1_SHIFT(crFRA, 0, crShiftToZero),
               CHF_BOX_SHIFT(fnBox, fnShiftToZero),
               CHF_CONST_FRA1_SHIFT(fnFRA, 0, fnShiftToZero),
               CHF_CONST_INT(refRatio));
}
```

In the Fortran code:

```
      subroutine SUMFINE(
     &   CHF_FRA1[crFRA],
     &   CHF_BOX[fnBox],
     &   CHF_CONST_FRA1[fnFRA],
     &   CHF_CONST_INT[refRatio])

      integer CHF_AUTODECL[iFn]
      integer CHF_AUTODECL[iCr]

      CHF_AUTOMULTIDO[fnBox;iFn]
c        Coarsen iFn
         CHF_DTERM[
```

```
        iCr0 = iFn0/refRatio;
        iCr1 = iFn1/refRatio;
        iCr2 = iFn2/refRatio;
        iCr3 = iFn3/refRatio;
        iCr4 = iFn4/refRatio;
        iCr5 = iFn5/refRatio;]
     crFRA(CHF_AUTOIX[iCr]) = crFRA(CHF_AUTOIX[iCr]) +
&       fnFRA(CHF_AUTOIX[iFn])
 CHF_ENDDO

 return
 end
```

9.8 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 ChomboFortran subroutine 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.

9.9 Examples

9.9.1 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
```

9.9.2 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
```

9.9.3 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
```

This can also be expressed in a simpler way making full use of the AUTO and OFFSET macros:

```
    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_AUTODECL[i], CHF_AUTODECL[ii]

    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_AUTOMULTIDO[region; i]

    lphi = zero
    do idir = 0, CH_SPACEDIM-1
       CHF_AUTOID[ii;idir]

       lphi = lphi +
&           ( (phi(CHF_OFFSETIX[i;+ii],n)
&           -  phi(CHF_AUTOIX[i],n))
&           - (phi(CHF_AUTOIX[i],n)
&           -  phi(CHF_OFFSETIX[i;-ii],n))
&           )*(dxinv)
    enddo

    lofphi(CHF_AUTOIX[i],n) =  lphi

    CHF_ENDDO
 enddo

 return
 end
```

9.9.4 Internal Procedure Example

This example demonstrates use of an internal procedure in a "*.ChF" file. Ordinary Fortran arguments could have also been declared for the internal procedure.

```
 subroutine testNoChFCall(
&   CHF_BOX[box])

 integer CHF_AUTODECL[i]
 integer c

 c = 0
 call countCells
 write(*,*) c
 call countCells
 write(*,*) c

 return

 contains

 subroutine countCells ! NO_CHF
```

```
   CHF_AUTOMULTIDO[box;i]
      c = c+1
   CHF_ENDDO
end subroutine

end
```

9.10 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, with the exception of internal procedures as shown in the example of section 9.9.4. 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" extensions as Fortran and files with ".ChF" extensions 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 exception to this is the 2nd argument to the `CHF_*1D` macros, which can be a literal constant.

- 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 9.9.3 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.

- Chombo `Vectors` and 1D arrays *always* start at index 0. You *cannot* call `CHF_LBOUND` on a `Vector` or 1D array. The value returned from `CHF_UBOUND` on a 1D array will always be one less than the length value passed as the 2nd argument in the C++ call to `CHF_*1D`.

Chapter 10

Multidimensional Chombo

This section describes the Chombo support for solving multidimensional systems of equations. "Multidimensional" in this sense means a system in which part of the solution approach is in an $N-$dimensional space, while another part is in an $M-$dimensional space, where $N \neq M$.

10.1 Namespace implementation

To implement multidimensional Chombo, we make use of namespaces to encapsulate each dimensionality. If Chombo is compiled with multidimensional support, then a version of the Chombo libraries is complied for each dimension and placed in a corresponding namespace. This allows the different dimensionalities to co-exist in a single build while avoiding naming collisions. Note that in order for Chombo's multidimensional build system to work correctly with your application code, you *must* include the namespace headers and footers in your source and header files (see Section 1.2.8). The lone exception is for code in which inter-dimensional transfers take place, which must explicitly reference each dimensionality by name.

10.2 Transdimensional utilities

To move between the different dimensionalities, we will need slicing and injection capability. This functionality is found in `/Chombo/lib/src/MultiDim`

10.2.1 **struct** `SliceSpec`

A `SliceSpec` specifies a slice in space, using a direction d and a position p. The `SliceSpec` class is actually a dimensional object, and is a part of the `BoxTools` library. A `DIM`-dimensional `SliceSpec` defines a `(DIM-1)`-dimensional slice through `DIM`-Dimensional space, with the d-th coordinate index set to p. When referring to an `IntVect`,

the d index defines which element will be removed. For a Box and related objects, d refers to the dimension which will be removed during a slicing operation.

10.2.2 Slicing

To move to a lower-dimensional space, we use the slicing functionality found in Slicing.H.transdim, which supports moving from the D space to the $D-1$ space. The Slicing code is heavily templated to work in a multidimensional setting. If DLo is the lower-dimension namespace and DHi is the higher dimension namespace (for example, DLo = D2 and DHi = D3), then the following functions are available.

- ```
  void sliceIntVect(DLo::IntVect& a_to,
                    const DHi::IntVectT&    a_from,
                    const DHi::SliceSpec& a_spec )
  ```

  Makes a_to the indicated (by a_spec) slice of a_from; a_from must be of dimension one larger then a_to.

- ```
  void sliceBox(DLo::Box& a_to,
                const DHi::Box&  a_from,
                const DHi::SliceSpec& a_slicespec,
                bool* a_outofbounds=0 )
  ```

  Defines a_to as a_from with its a_slicespec.direction-th coordinate missing; a_to is one dimension lower, e.g. if a_from is a D3::Box, then a_to is a D2::Box. If a_outofbounds isn't NULL, we set it to true if a_slicespec.position is outside of a_from, along the a_slicespec.direction-th axis.

- ```
  DLo::ProblemDomain
  sliceDomain(const DHl::ProblemDomain& DHi::a_from,
              const DHi::SliceSpec& a_slicespec,
              bool* a_outofbounds=0 );
  ```

  Functions similarly to the sliceBox function, but also maintains periodicity information appropriately.

- ```
  template<T>
  void sliceBaseFab(DLo::BaseFab<T>& a_to,
                const DHi::BaseFab<T>& a_from,
                const DHi::SliceSpec& a_slicespec )
  ```

  Sets a_to to be a slice of a_from, along the a_slicespec.direction-th axis at the a_slicespec.position-th position. Data from the slice of a_from is copied to the destination BaseFab> a_to. As usual, a_to is one dimension lower than a_from; if a_from is a D3::BaseFab, a_to is a D2::BaseFab. It is an error if a_slicespec.position is outside of a_from.

- ```
  void
  sliceDisjointBoxLayout(DLo::DisjointBoxLayout& a_to,
                         const DHi::DisjointBoxLayout& a_from,
                         const DHi::SliceSpec& a_slicespec )
  ```

- ```
  template<class T>
  void
  sliceLevelData(DLo::LevelData<T>& a_to,
                 const DHi::LevelData<T>& a_from,
                 const DHi::SliceSpec& a_slicespec )
  ```

- ```
  void
  sliceLevelFlux( DLo::LevelData<FluxBox>& a_to,
                  const DHi::LevelData<FluxBox>& a_from,
                  const DHi::SliceSpec& a_slicespec )
  ```

  Template specialization for `LevelData<FluxBox>` Makes `a_from` a slice of `a_to` at `a_slicespec.position` along the `a_slicespec.direction`-th axis. If `a_to` is defined, we use its `DisjointBoxLayout`, otherwise we create an appropriate new `DisjointBoxLayout` for it.

10.2.3 Injection

To move from the D space to the $D+1$ space, we use the injection functionality in `Injection.H.transdim`. In general, the inject functions simply define a higher-dimensional version of the lower-dimensional input, with the "new" dimension defined by the `SliceSpec::direction` argument d. For the case of an IntVect, the value of the newly added dimension is set to `SliceSpec::position` p. For a Box and containers defined on a `Box` or `DisjointBoxLayout`, the higher-dimensonal destination is defined to be one-cell wide in the new direction, and is located at $i_d = p$.

- ```
  void injectIntVect(DHi::IntVectT& a_to,
                     const DLo::IntVect& a_from,
                     const DHi::SliceSpec& a_spec )
  ```

  Makes `a_to` like `a_from`, only one dimension higher. The "new" transverse dimension is `a_spec.direction`, and the value at that dimension is `a_spec.position`.

- ```
  void injectBox(DHi::BoxT& a_to,
                 const DLo::Box& a_from,
                 const DLo::SliceSpec& a_slicespec )
  ```

  Sets `a_to` to be `a_from` with an extra dimension – `a_slicespec.direction` – in which it's just one cell thick and has coordinate value `a_slicespec.position`.
  ```

- ```
  template<typename T>
  void injectBaseFab(DHi::BaseFab<T>& a_to,
                     const DLo::BaseFab<T>& a_from,
                     const DLo::SliceSpec& a_slicespec )
  ```

  Sets a_to to be a_from with an extra dimension – a_slicespec.direction – in which it's just one cell thick and has coordinate value a_slicespec.position.

- ```
  void injectDisjointBoxLayout(DHi::DisjointBoxLayout& a_to,
                               const DLo::DisjointBoxLayout& a_from,
                               const DLo::SliceSpec& a_slicespec)
  ```

  Sets a_to to be a_from with an extra dimension – a_slicespec.direction – in which it's just one cell thick and has coordinate value a_slicespec.position. Data values are copied from a_from to fill a_to.

- ```
  template <class T>
  void injectLevelData(DHi::LevelData<T>& a_to,
                       const DLo::LevelData<T>& a_from,
                       const DLo::SliceSpec& a_slicespec )
  ```

  Sets a_to to be a_from with an extra dimension – a_slicespec.direction – in which it's just one cell thick and has coordinate value a_slicespec.position. If a_to is defined, we use its DisjointBoxLayout, otherwise we give it a DisjointBoxLayout with the same assignment-to-processors as a_from has. Data values are copied from a_from to fill a_to.

- ```
  void injectLevelFlux(DHi::LevelData<FluxBox>& a_to,
                       const DLo::LevelData<FluxBox>& a_from,
                       const DLo::SliceSpec& a_slicespec )
  ```

  Version of injection for LevelData<FluxBox>. Sets a_to to be a_from with an extra dimension – a_slicespec.direction – in which it's just one cell thick and has coordinate value a_slicespec.position. If a_to is defined, we use its DisjointBoxLayout, otherwise we give it a DisjointBoxLayout with the same assignment-to-processors as a_from has. Data values are copied from a_from to fill a_to.

10.2.4 The `ReductionCopier` class

The ReductionCopier is a specialized Copier designed to support a reduction from a higher-dimensional DisjointBoxLayout to a smaller-dimensional one by copying all of the data in the transverse direction to the destination boxLayout. It is assumed that this will be used with a different sort of LDOperator (such as the SumOp described below), since a simple copy operation wouldn't make much sense.

Essentially, the ReductionCopier computes and stores the intersection list required to do a copyTo operation which copies all of the data in src into dest, assuming that an intersection can be found by shifting the location of a data point in src in the transverse direction.

Note that this class operates entirely in a single dimensional namespace. The usage pattern is to perform the reduction operation in the higher-dimensional space, then use a slicing operation to move to a lower-dimensional space. A simple usage example is shown in figure 10.1, with a corresponding code example in figure 10.2.

- ```
  ReductionCopier(const DisjointBoxLayout& a_level,
                  const BoxLayout& a_dest,
                  const IntVect& a_destGhost,
                  int a_transverseDir,
                  bool  a_exchange = false);

  void define(const DisjointBoxLayout& a_level,
              const BoxLayout& a_dest,
              const IntVect& a_destGhost,
              int a_transverseDir,
              bool  a_exchange = false);
  ```

  Full constructor and define functions (constructor simply calls define)

  - a_level – source DisjointBoxLayout
  - a_dest – destination boxes
  - a_destGhost – number of ghost cells to be filled around the destination grids
  - a_transverseDir – the direction of the reduction operation
  - a_exchange – if true, this copier is being defined for an exchange, rather than a copyTo operation.

10.2.5 The SpreadingCopier class

The SpreadingCopier is a specialized Copier designed to support a reduction from a lower-dimensional DisjointBoxLayout to a higher-dimensional one by copying all of the data in the transverse direction to the destination boxLayout. It is assumed that this will be used with an appropriate LDOperator such as the SpreadingOp described below.

Essentially, the SpreadingCopier computes and stores the intersection list required to do a copyTo operation which copies the data in src into dest, assuming that an intersection can be found by shifting the location of a data point in src in the transverse direction.

Like the ReductionCopier (only in reverse) this class operates entirely in a single dimensional namespace. The usage pattern is to use an injection operation to move the
  ```

data from the lower-dimensional space to the higher-dimensional space, then perform the spreading operation in the higher-dimensional space.

- ```
  SpreadingCopier(const DisjointBoxLayout& a_level,
                  const BoxLayout& a_dest,
                  const IntVect& a_destGhost,
                  int a_transverseDir,
                  bool  a_exchange = false);

  void define(const DisjointBoxLayout& a_level,
              const BoxLayout& a_dest,
              const IntVect& a_destGhost,
              int a_transverseDir,
              bool  a_exchange = false);
  ```

  Full constructor and define functions (constructor simply calls define)

  - `a_level` – source `DisjointBoxLayout`
  - `a_dest` – destination boxes
  - `a_destGhost` – number of ghost cells to be filled around the destination grids
  - `a_transverseDir` – the direction of the reduction operation
  - `a_exchange` – if true, this copier is being defined for an exchange, rather than a copyTo operation.

10.2.6 The `SumOp` class

The `SumOp` class is an instance of `LDOperator<FarrayBox>` which performs a summing operation of the data in src in the `summingDir` direction, multiplies by the scale, and places the sum in the corresponding location in `dest`. Note that `scale` is a public member function, and so may be set without an access function.

- `SumOp(int a_summingDir);`

  Constructor – sets scale to one.

  - `a_summingDir` – direction in which to sum

- ```
  void op(FarrayBox& dest,
          const Box& RegionFrom,
          const Interval& Cdest,
          const Box& RegionTo,
          const FArrayBox& src,
          const Interval& Csrc) const;
  ```

  Computes componentwise sum of `src` over the `RegionFrom` in the `summingDir` direction, multiplies by `scale` and places the result in `dest` in the `RegionTo`.

- dest – destination FAB
- RegionFrom – Region in src over which to compute sum
- Cdest – Interval in dest into which sum will be computed.
- RegionTo – Region in dest into which the sum will be placed.
- src – source FAB
- Csrc – interval in src over which sum will be computed (must be the same size as Cdest).

- ```
  virtual void linearIn(FArrayBox& arg,
                        void* buf,
                        const Box& R,
                        const Interval& comps) const;
  ```

  Linearization function.

10.2.7 The SpreadingOp class

The SpreadingOp class spreads the data in src along the summingDir direction, multiplying by scale, and placing the resulting values in the corresponding locations in dest. Essentially the inverse of the SumOp class. Like the SumOp, scale is a public data member.

- `SpreadingOp(int a_spreadingDir);`

  Defining constructor. Sets default value for scale to 1.0.

  - a_spreadingDir – direction in which to spread data.

- ```
  void op(FArrayBox& dest,
          const Box& RegionFrom,
          const Interval& Cdest,
          const Box& RegionTo,
          const FArrayBox& src,
          const Interval& Csrc) const;
  ```

  Perform spreading operation – for each IntVect in RegionFrom, scale value in src by scale and then place scaled value into dest for each IntVect in RegionTo which corresponds to a translation from the source location along the spreadingDir direction. This is a componentwise operation, so the scaled values of the first component of Csrc will be placed in the first component of Cdest, etc.

  - dest
  - RegionFrom
  - Cdest – Interval in dest over which operation is performed.

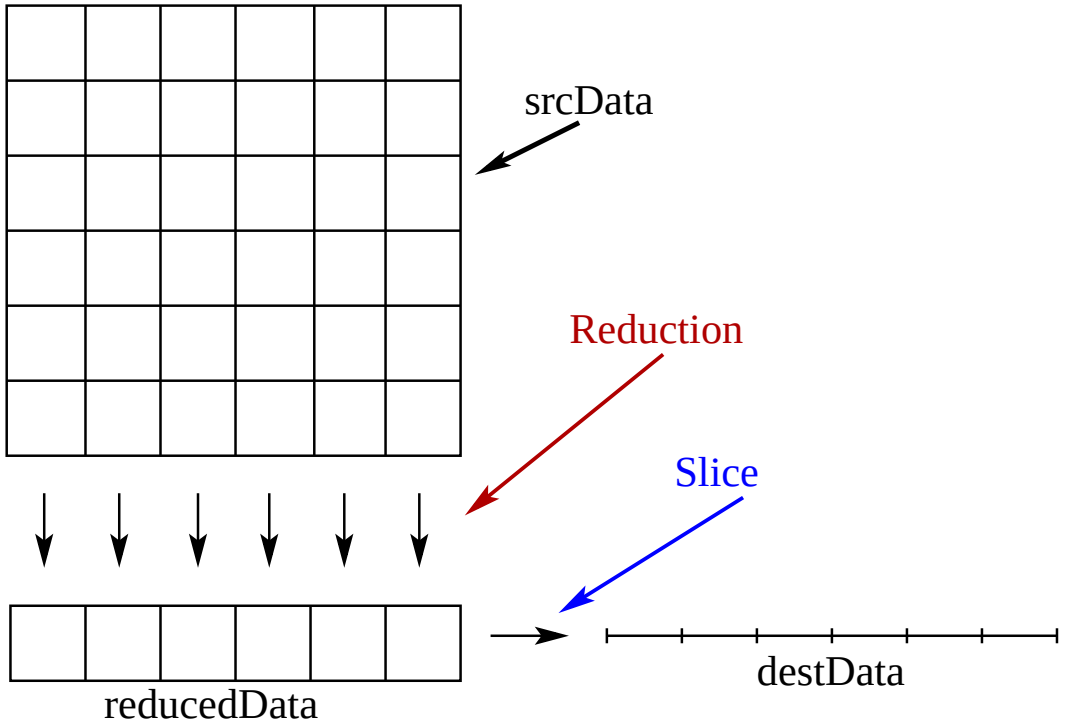

Figure 10.1: schematic of reduction operations carried out by the code in Figure 10.2

  - `RegionTo`
  - `src`
  - `Csrc` – Interval in `src` over which operation is performed. Must have the same size as `Cdest`.

- ```
  virtual void linearIn(FArrayBox& arg,
                        void* buf, const Box& R,
                        const Interval& comps) const;
  ```

  Linearization function.
  ```

```
// function to reduce a 2D LevelData<FArrayBox> to 1D by
// summing over the y-direction
void reduce2DTo1D(D1::LevelData<D1::FArrayBox> >& destData,
                  D2::LevelData<D2::FarrayBox> >& srcData)
{
  // create 2D version of 1D boxes
  D2::LevelData<D2::FarrayBox> sliceGrids;
  D2::SliceSpec slice(1,0);

  injectDisjointBoxLayout(sliceGrids, srcData.getBoxes(), slice);

  // define sliced data holder
  D2::LevelData<D2::FArrayBox> reducedData(sliceGrids, srcData.nComp);

  // define ReductionCopier to compute intersections (sum in the y-direction)
  int transverseDir = 1;
  ReductionCopier reduceCopier(srcData.getBoxes(), sliceGrids, 1);

  SumOp op(transverseDir);
  op.scale = 1.0;

  // do summing operation -- sums data in srcData along lines in the
  // y-direction and places sum in reducedData
  srcData.copyTo(srcData.interval(),
                 reducedData, reducedData.interval(),
                 reduceCopier, op);

  // finally, take the data in reducedData (which is a 2D object)
  // and slice to 1D
  sliceLevelData(destData, reducedData, slice);
}
```

Figure 10.2: Code fragment illustrating use of ReductionCopier

10.3 Phase field example

As a simple example of a multidimensional application, we solve the phase space example from [27]:

$$\frac{\partial f}{\partial t} + \frac{\partial v f}{\partial x} + \frac{\partial a f}{\partial v} = 0 \tag{10.1}$$

$$a = -\frac{\partial \phi}{\partial x},$$

$$\frac{\partial^2 \phi}{\partial x^2}(x,t) = 4\pi G \int f(x,v,t) dv, \tag{10.2}$$

where G is the gravitational constant (π for the example in [27]). This example may be found in `releasedExamples/MultiDimPhase` in the Chombo distribution.

10.3.1 Algorithm

At time t we have $f_{ij}^n = f(x_i, v_j, t^n)$.

1. **Compute phase velocity** a
   (a) Compute rhs for Poisson solve: $\rho_i = \sum_{j=0}^{j<N_j} f_{i,j}^n \Delta v$
   (b) do 1D Poisson solve for $\phi(x,t)$: Solve $L_i^{1D}\phi = \rho_i$
   (c) Compute a in 1D: $a_i = G_i^{1D,CC}\phi$
   (d) Spread a to 2D mesh: $a_{ij} = a_i$.
2. **Advect f**
   (a) predict $f_{face}^{n+\frac{1}{2}}$ using AMRGodunovUnsplit predictor
   (b) update f:

$$f_{ij}^{n+1} = f_{ij}^n - \frac{\Delta t}{\Delta x}(v_j f_{i+\frac{1}{2},j}^{n+\frac{1}{2}} - v_j f_{i-\frac{1}{2},j}^{n+\frac{1}{2}}) - \frac{\Delta t}{\Delta v}(a_i f_{i,j+\frac{1}{2}}^{n+\frac{1}{2}} - a_i f_{i,j-\frac{1}{2}}^{n+\frac{1}{2}})$$

10.4 Building the multidim example

Building a multidim Chombo application like the `MultiDimPhase` example is somewhat different from the standard Chombo build process, since it relies on a build shell script to orchestrate the build. This script has been incorporated into the build system rules, so building is fairly straightforward, but does require two separate `GNUmakefiles`, as may be found in `releasedExamples/MultiDimPhase/exec`. The primary makefile (`GNUmakefile` in the `MultiDimPhase` example) merely specifies the location of `CHOMBO_HOME`, the min and max dimensions used by the given compilation, and the name of the second makefile.

The second makefile (GNUmakefile.multidim in the MultiDimPhase example, specifies which directories should be compiled in which dimensionalities (including the "multiDim" dimensionality which is where any inter-dimensional transfers happen and which is compiled without a particular SpaceDim) are defined in the 1dsrc_dirs, 2dsrc_dirs, 3dsrc_dirs, 4dsrc_dirs, 5dsrc_dirs, 6dsrc_dirs and mdsrc_dirs (for the multidimensional source files) variables in the local GNUmakefile.multidim in the exec directory. At this point, the only real difference when invoking the makefile is that no dimensionality should be specified (you can, but it won't do anything):

```
obtain Chombo
edit Chombo/lib/mk/Make.defs.local as usual
cd Chombo/releasedExamples/MultiDimPhase/exec
gmake all
(go get a cup of coffee, and maybe some lunch...)
./phase<config>.ex inputs
```

Note that the executable doesn't have a "<DIM>d" in its name – just look for a phase*.ex executable.

Chapter 11

Chombo Debugging and Performance Tools

11.1 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. To stop in Fortran subroutine `myfortranfunc`,

  ```
  <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.

11.2 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.

11.3 Viewing data objects with VisIt from gdb

There are two ways to use VisIt to help examine data during a debugging session. One may use the `writeFAB, writeFABname, writeLevel`, and `writeLevelname` functions described in section 6.3 to write data in a `FArrayBox` or `LevelData<FArrayBox>` to a file, and then call `VisIt` from a shell to view the data. Alternatively, the `viewFAB` and `viewLevel` functions allow `VisIt` to be called directly from the `gdb` session.

If one has an `FArrayBox fab` and a `LevelData<FArrayBox> ldf`, one may do the following:

```
<gdb prompt> call viewFAB(&fab)
<gdb prompt> call viewLevel(&ldf)
```

which will result in fab and ldf being written to separate temporary files, and then one VisIt processes being started with two windows controlling the different data objects.

11.4 pout()

In Chombo, the `pout()` function is used in place of the std::cout output stream object. defined in header parstream.H

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;
}
```

11.5 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 identify 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)
```

11.6 TraceTimer

TraceTimer class is a self-tracing code instrumentation system for Chombo (or any other package really). The user interface is specified by a small set of macros. The usage

model is that you just leave these timers in the code, for good. Initially, your application will have 'main' and a few heavy functions instrumented, and the lower level Chombo library instrumentation. As your tool or application matures, it will garner a larger set of instrumentation giving clear views of your code performance. After a routine has been cleverly and lovingly optimized, the timers are left in place to spot when some later bug fix or *improvement* undoes your previous labors.

You should never need to use or interact with the the classes TraceTimer or AutoStart. Use the macros. They call the right functions and classes for you.

The first macro is what people will use the most:

```
CH_TIME("label");
```

This is the simplest interface for TraceTimer. You place this macro call in a function you wish to be timed. It handles making the timer, calling 'start' when you enter the function, and calling 'stop' when you leave the function. A good idea is to use a 'label' specific enough to be unambiguous without being overwhelming. for instance:

```
void AMRLevelPolytropicGas::define(AMRLevel*            a_coarserLevelPtr,
                                   const ProblemDomain& a_problemDomain,
                                   int                  a_level,
                                   int                  a_refRatio)
{
   CH_TIME("AMRLevelPolytropicGas::define");
 .
 .
}
```

In this case, we have a class with many constructors and define functions that all funnel into a single general function. We can just call this 'define' and not worry about naming/instrumenting all the different overloaded instances. If you slip up and use the same label twice, that is not a real problem, the two locations will be timed and tracked properly (even if one is a sibling or parent of the other). The only place it will make things a little harder is in the output where you might have the same name show up and look confusing.

In serial, you will see a file called `time.table` (in parallel, you will get a `time.table.n` (where n is the rank number) files).

By default, Chombo compiles in the instructions for the timers wherever the macros appear. If the compiler macro `CH_TIMER` is defined, then all the `CH_TIME*` macros evaluate to empty expressions at compile time.

So, you put some `CH_TIME` calls in your code and ran it, and nothing happened: Chombo looks for the environment variable **CH_TIMER**. If it is set to anything (even if it is set to 'false' or 'no' or whatever) then the timers will be active and reporting will happen. If this environment variable is not set, then all the timers check a bool and return after doing nothing.

One point of interest with using the environment variable: In parallel jobs using mpich, only processor 0 inherits the environment variables from the shell where you invoke 'mpirun', the rest read your .cshrc (.bashrc, etc.) file to get their environment. To time all your processes, you need to make sure the **CH_TIMER** environment variable gets to all your processes.

11.6.1 Auto hierarchy

The timers automatically figure out their parent/child relationships. They also can be placed in template code. This has some consequences. First, if you have a low level function instrumented that has no timers near it in the code call stack, you will see it show up as a child of a high level timer. the root timer "main" will catch all orphaned timers. So, even though you might make no call to, say, 'exchange' in your 'main' function, you might very well call a function, that calls a function, that calls 'exchange'. Since no code in between was instrumented, this exchange is accounted for at 'main'. This might look strange, but it should prove very powerful. An expensive orphan is exactly where you should consider some more timers, or reconsidering code design.

For performance reasons, child timers have only one parent. As a consequence each CH_TIME("label") label can show up at multiple places in your output. Each instance has it's own timer. So, each path through the call graph that arrives at a low-level function has a unique lineage, with it's own counter and time. Thus, I can instrument LevelData::copyTo once, but copyTo can appear in many places in the time.table file.

11.6.2 Finer control

The next level up in complexity is the set of *four* macros for when you want sub-function resolution in your timers. For instance, in a really huge function that you have not figured out how to re-factor, or built with lots of bad cut n paste code 're-use'.

```
  CH_TIMERS("parent");
  CH_TIMER("child1", t1);
  CH_TIMER("child2", t2);
  CH_START(t1);
//some code go here
  CH_STOP(t1);
  CH_START(t2);
//some other code go here
  CH_STOP(t2);
  CH_START(t1);
//can start something here again
  CH_STOP(t1);
```

One very good place to use the more sophisticated API is within loops. START and STOP are very fast compared to the timer declaration:

```
CH_TIMERS("parent"); // parent declared and started
CH_TIMER("t1", t1); // t1 declared and made child parent, not started
for(IVSIterator it(ivs); it.ok(); ++it){
       CH_START(t1);
      //some code go here
       CH_STOP(t1);
      //other stuff you don't want timed
}
```

CH_TIMERS has the same semantic as CH_TIME, except that you can declare an arbitrary number of children after it in the same function scope. The children here do not autostart and autostop, you have to tell them where to start and stop timing. The children can themselves be parents for timers in called functions, of course. The children obey a set of mutual exclusions. The following generate run time errors:

- double start called
- double stop called
- start called when another child is also started
- you leave the function with a child not stopped

the following will generate compile time errors:

- more than one CH_TIME macro in a function
- invoking CH_TIMER("child", t) without having first invoked CH_TIMERS
- re-using the timer handle ie. CH_TIMER("bobby", t1); CH_TIMER("sally", t1)
- mixing CH_TIME macro with CH_TIMER
- mixing CH_TIME macro with CH_TIMERS

You do not have to put any calls in your main routine to activate the clocks or generate a report at completion, this is handled with static initialization and an `atexit` function. The exception to this is for parallel reporting. Since `atexit` and `MPI_Finalize()` do not interact in an agreeable fashion, you need to explicitly call the Chombo macro `CH_TIMER_REPORT()` before the code reaches `MPI_Finalize`

There is a larger argument about manual instrumentation being detrimental to clean software. Profiling the code is supposed to tell you where to expend your optimization effort. Manual instrumentation opens the door to people wasting time *assuming* what parts of the code are going to take up lots of time and instrumenting them, before seeing any real performance data. Good judgment is needed. We have a body of knowledge about Chombo that will inform us about a good minimal first set of functions to instrument.

Chapter 12

Troubleshooting

1. `error: 'H5Pset_fapl_mpio' was not declared in this scope`

    - This is a compilation error. The user is trying to build the parallel version of Chombo but is not linking against an HDF5 build that has used `--enable-parallel` when it was `configured`
    - Most Chombo users need access to both a serial and a parallel version of their HDF5 libraries. These are distinguished in the Chombo makefiles as `HDFINCFLAGS` and `HDFMPIINCFLAGS` (these values should be different).

Bibliography

[1] 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.

[2] A. S. Almgren, T. Buttke, and P. Colella. A fast adaptive vortex method in three dimensions. *J. Comput. Phys.*, 113(2):177–200, 1994.

[3] D.S. Balsara. Divergence-free adaptive mesh refinement for magnetohydrodynamics. *J. Comput. Phys.*, 174(2):614–648, 2001.

[4] 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.

[5] M. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Trans. Comp.*, 1986.

[6] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53:484–512, March 1984.

[7] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

[8] 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.

[9] 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.

[10] P. Colella, M. Dorr, and D. Wake. Numerical solution of plasma-fluid equations using locally refined grids. *J. Comput. Phys.*, 152:550–583, 1999.

[11] W. Y. Crutchfield. Load balancing irregular algorithms. Technical Report UCRL-JC-107679, LLNL, July 1991.

[12] W. Y. Crutchfield and M. Welcome. Object-oriented implementation of adaptive mesh refinement algorithms. *Scientific Programming*, 2(4):145–156, 1993.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] Scott R. Kohn and Scott B. Baden. Irregular coarse-grain data parallelism under lparx. *J. Scientific Programming*, 1996.

[19] D. Martin, P. Colella, and D. T. Graves. A cell-centered adaptive projection method for the incompressible Navier-Stokes equations in three dimensions. *Journal of Computational Physics*, 227:1863–1886, 2008.

[20] D. F. Martin and K. L. Cartwright. Solving Poisson's equation using adaptive mesh refinement. *Technical Report UCB/ERI M96/66 UC Berkeley*, 1996.

[21] Daniel Francis Martin. *An Adaptive Cell-Centered Projection Method for the Incompressible Euler Equations*. PhD thesis, University of California, Berkeley, 1998.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] E.H. Twizell, A.B. Gumel, and M.A. Arigu. Second-order, l_0-stable methods for the heat equation with time-dependent boundary conditions. *Advances in Computational Mathematics*, 6:333–352, 1996.

[27] Paul R. Woodward. Piecewise-parabolic methods for astrophysical fluid dynamics. Technical report, Lawrence Livermore National Laboratory, 1983. prepared for Proceedings of the NATO Advanced Workshop in Astrophysical Radiation Hydrodynamics, Munich, West Germany, August 1982.