chombo-discharge Documentation

Release latest

Robert Marskar

Apr 21, 2022

INTRODUCTION

Important: This is an alpha release. Development is still in progress, and various bugs may be present. User interfaces can and will change.

`chombo-discharge` is a modular and scalable research code for Cartesian two- and three-dimensional simulations of low-temperature plasmas in complex geometries. The code is hosted at GitHub together with the source files for this documentation.

`chombo-discharge` supports

- Fully written in C++.
- Support for complex geometries.
- Scalar advection-diffusion-reaction processes.
- Electrostatics with support for electrodes and dielectrics.
- Radiative transport as a diffusion or Monte Carlo process.
- Particle-mesh operations (like Particle-In-Cell)
- Parallel I/O with HDF5.
- Dual-grid simulations with individual load balancing of fluid and particles.
- Various multi-physics interfaces that use the above solvers.
- Various time integration schemes.

Numerical solvers are designed to run either on their own, or as a part of a larger application.

For scalability, `chombo-discharge` is built on top of Chombo 3, and therefore additionally features

- Cut-cell representation of multi-material geometries.
- Patch based adaptive mesh refinement.
- Weak and strong scalability to thousands of computer cores.

Our goal is that users will be able to use `chombo-discharge` without modifying the underlying solvers. There are interfaces for describing e.g. the plasma physics, boundary conditions, mesh refinement, etc. As `chombo-discharge` evolves, so will these interfaces. We aim for (but cannot guarantee) backward compatibility such that existing `chombo-discharge` models can be run on future versions of `chombo-discharge`.

CHAPTER

ONE

INTRODUCTION

1.1 Using this documentation

This documentation is the user documentation for `chombo-discharge`. It includes an explanation of the data structures, algorithms, and code design. It was built using *reStructuredText* with *Sphinx*, and is HTML browser-friendly.

1.1.1 PDF documentation

If you want to build a PDF version of this documentation, please navigate to the `sphinx/` subfolder and execute

```
make clean latexpdf
```

A PDF version of this documentation named `chombo-discharge.pdf` will appear in `sphinx/build/latex/`.

1.1.2 Doxygen documentation

A separate Doxygen documentation of the `chombo-discharge` code is found here.

1.2 `chombo-discharge` overview

1.2.1 History

`chombo-discharge` is aimed at solving discharge problems. It was originally developed at SINTEF Energy Research between 2015-2018, and aimed at elucidating discharge dynamics in high-voltage engineering. Further development was started in 2021, where much of the code was completely redesigned both for improved modularity and performance.

`chombo-discharge` is distributed via GitHub. The main branch is isolated, and changes to `chombo-discharge` are done through pull requests (after code review and passing a test suite).

1.2.2 Key functionalities

chombo-discharge uses a Cartesian embedded boundary (EB) grid formulation and adaptive mesh refinement (AMR) formalism where the grids frequently change and are adapted to the solution as simulations progress.

Important: chombo-discharge is **not** a black-box model for discharge applications. It is an evolving research code with occasionally expanded capabilities, API changes, and performance improvements. Although problems can be set up through our Python toos, users should nonetheless be willing to take time to understand how the code works. In particular, developers should invest some effort in understanding the data structures and Chombo basics (see *Chombo-3 basics*).

Key functionalities are provided in Table 1.2.1.

Table 1.2.1: Key capabilities in chombo-discharge

Capabilities	Supported?
Grids	**Fundamentally Cartesian**.
Parallelized?	**Yes**, using flat MPI.
Load balancing?	**Yes**, with support for individual particle and fluid load balancing.
Complex geometries?	**Yes**, using embedded boundaries (i.e., cut-cells).
Adaptive mesh refinement?	**Yes**, using patch-based refinement.
Subcycling in time?	**No**, only global time step methods.
Linear solvers?	**Yes**, using geometric multigrid in complex geometries.
Time discretizations?	**Mostly explicit**.
Parallel IO?	**Yes**, using HDF5.
Checkpoint-restart?	**Yes**, for both fluid and particles.

An early version of chombo-discharge used sub-cycling in time, but this has now been replaced with global time stepping methods. That is, all the AMR levels are advanced using the same time step. chombo-discharge has also incorporated many changes to the EB functionality supplied by Chombo. This includes much faster grid generation, support for polygon surfaces, and many performance optimizations (in particular to the EB formulation).

chombo-discharge supports both fluid and particle methods, and can use multply parallel distributed grids (see *Realm*) for individually load balancing e.g. the fluid and particle grids. Although many abstractions are in place so that user can describe a new set of physics, or write entirely new solvers into chombo-discharge and still use the embedded boundary formalism, chombo-discharge also provides several physics modules for describing various types of problems. These modules reside in $DISCHARGE_HOME/Physics and are also listed in *Implemented models*.

1.2.3 Design approach

A fundamental design principle in chombo-discharge is the division between the AMR core, geometry, solvers, and user applications. As an example, the fundamental time integrator class TimeStepper in chombo-discharge is a just an abstraction, i.e. it only presents an API which application codes must use. Because of that, TimeStepper can be used for solving completely unrelated problems. We have, for example, implementations of TimeStepper for solving radiative transfer equations, advection-diffusion problems, electrostatic problems, or for multiphysics plasma problems.

The division between computational concepts (e.g., AMR functionality and solvers) exists so that users will be able to solve problems across a range of geometries, add new solvers functionality, or write entirely new applications, without requiring fundamental changes to chombo-discharge. Fig. 1.2.1 shows the basic design sketch of the chombo-discharge code. To the right in this figure we have the AMR core functionality, which supplies the infrastructure for running solvers. In general, solvers may share features or be completely disjoint. For this reason

numerical solvers are asked to register their AMR needs. For example, elliptic solvers need functionality for interpolating ghost cells over the refinement boundary, but pure particle solvers have no need for such functionality. A consequence of this is that the numerical solvers are literally asked (during their instantiation) to register what type of AMR infrastructure they require. In return, the AMR core will allocate this infrastructure and return it to the solver, as illustrated in Fig. 1.2.1.

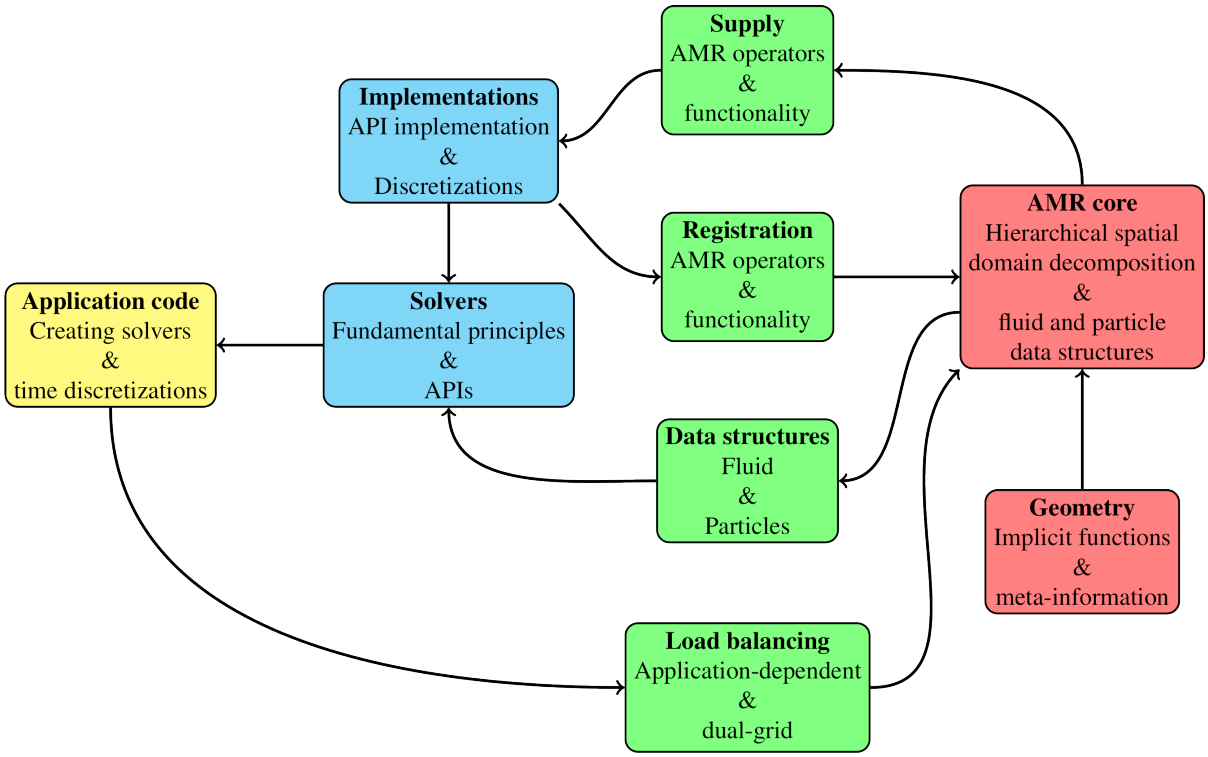

Fig. 1.2.1: Concept design sketch for `chombo-discharge`.

`chombo-discharge` also uses *loosely coupled* solvers as a foundation for the code design, where a *solver* indicates a piece of code for solving an equation. For example, solving the Laplace equation $\nabla^2\Phi = 0$ is encapsulated by one of the `chombo-discharge` solvers. In general, all solvers in `chombo-discharge` have a null-implemented API, i.e. we have enforced a strict separation of the solver interface and the solver implementation. This constraint exists because while new features may be added to a discretization, we do not want such changes to affect upstream application code. An example of this is the `FieldSolver`, which conceptualizes a numerical solver for solving for electrostatic field problems. The `FieldSolver` is an API with no fundamental discretization – it only contains high-level routines for understanding the type of solver being dealt with. Yet, it is the `FieldSolver` API which is used by application code (and not the implementation class).

All numerical solvers interact with a common AMR core that encapsulates functionality for running the solvers. All solvers are also compatible with mesh refinement and complex geometries, but they can only run through application codes, i.e. *physics modules*. These modules encapsulate the time advancement of either individual or coupled solvers. Solvers can only interact with one another through these modules, and these modules usually advance the equations of motion using the method-of-lines. Fundamentally, it is also possible to couple different application codes.

1.3 Installation

1.3.1 Obtaining chombo-discharge

chombo-discharge can be freely obtained from https://github.com/chombo-discharge/chombo-discharge. The following packages are *required*:

- Chombo, which is supplied with chombo-discharge.
- The C++ JSON file parser https://github.com/nlohmann/json.
- LAPACK and BLAS

The Chombo and nlohmann/json dependencies are automatically handled by chombo-discharge through git submodules.

Warning: Our version of Chombo is hosted at https://github.com/chombo-discharge/Chombo-3.3.git. chombo-discharge has made substantial changes to the embedded boundary generation in Chombo. It will not compile with other versions of Chombo than the one above.

Optional packages are

- HDF5, used for writing plot and checkpoint file.
- MPI, used for parallelization.
- VisIt visualization, used for visualization.

1.3.2 Cloning chombo-discharge

To compile chombo-discharge, the makefiles must be able to find both chombo-discharge and Chombo. In our makefiles the paths to these are supplied through the environment variables

- CHOMBO_HOME, pointing to your Chombo library
- DISCHARGE_HOME, pointing to the chombo-discharge root directory.

To clone chombo-discharge, set the environment variables and clone (using --recursive to fetch submodules):

```
export DISCHARGE_HOME=/home/foo/chombo-discharge
export CHOMBO_HOME=$DISCHARGE_HOME/Submodules/Chombo-3.3/lib

git clone --recursive git@github.com:chombo-discharge/chombo-discharge.git ${DISCHARGE_HOME}
```

Note that DISCHARGE_HOME must point to the root folder in the chombo-discharge source code, while CHOMBO_HOME must point to the lib/ folder in your Chombo root directory. When cloning with submodules, both Chombo and nlohmann/json will be placed in the Submodules folder in $DISCHARGE_HOME.

1.3.3 Test build

For a quick compilation test the user can use the GNU configuration file supplied with `chombo-discharge` by following the steps below.

1. Copy the GNU configuration file

```
cp $DISCHARGE_HOME/Local/Make.defs.GNU $CHOMBO_HOME/mk/Make.defs.local
```

2. If you do not have the GNU compiler suite, install it by

```
sudo apt install csh gfortran g++ libblas-dev liblapack-dev
```

   This will install

   - LAPACK and BLAS
   - GNU compilers for Fortran and C++

3. Compile `chombo-discharge`

```
cd $DISCHARGE_HOME
make -s -j4 discharge-lib
```

This will compile the `chombo-discharge` source code in serial and without HDF5 (using four cores for the compilation). If successful, `chombo-discharge` libraries will appear in `$DISCARGE_HOME/Lib`.

1.3.4 Advanced configuration

`chombo-discharge` is compiled using GNU Make, following the `Chombo`. Compilers, libraries, and configuration options are defined in a file `Make.defs.local` which resides in `$CHOMBO_HOME/mk`. Users need to supply this file in order to compile `chombo-discharge`. Typically, these steps include

- Specifying Fortran and C++ compilers
- Specifying configurations. E.g., serial or parallel builds, and compiler flags.
- Specifying library paths (in particular for HDF5).

Main settings

The main variables that the user needs to set are

- `DIM = 2/3` The dimensionality (must be 2 or 3).
- `DEBUG = TRUE/FALSE` This enables or disables debugging flags and code checks/assertions.
- `OPT = FALSE/TRUE/HIGH`. Setting `OPT=TRUE/HIGH` enables optimization flags that will speed up `Chombo` and `chombo-discharge`.
- `PRECISION = DOUBLE` Currently, `chombo-discharge` has not been wetted with single precision. Many algorithms (like conjugate gradient) depend on the use of double precision.
- `CXX = <C++ compiler>`
- `FC = <Fortran compiler>`
- `MPI = TRUE/FALSE` This enables/disables MPI.
- `MPICXX = <MPI compiler>`
- `CXXSTD=14` Sets the C++ standard - we are currently at C++14.

- `USE_EB=TRUE` Configures `Chombo` with embedded boundary functionality. This is a requirement.
- `USE_MF=TRUE` Configures `Chombo` with multifluid functionality. This is a requirement.
- `USE_HDF5 = TRUE/FALSE` This enables and disables HDF5 code.

HDF5

If using HDF5, one must also set the following flags:

- `HDFINCFLAGS = -I<path to hdf5-serial>/include` (for serial HDF5).
- `HDFLIBFLAGS = -L<path to hdf5-serial>/lib -lhdf5 -lz` (for serial HDF5)
- `HDFMPIINCFLAGS = -I<path to hdf5-parallel>/include` (for parallel HDF5)
- `HDFMPILIBFLAGS = -L<path to hdf5-parallel>/lib -lhdf5 -lz` (for parallel HDF5).

Compiler flags

Compiler flags are set through

- `cxxoptflags = <C++ compiler flags`
- `foptflags = <Fortran compiler flags`
- `syslibflags = <system library flags>`

Note that LAPACK and BLAS are requirements in `chombo-discharge`. Linking to these can often be done using

- `syslibflag = -llapack -lblas` (for GNU compilers)
- `syslibflag = -mkl=sequential` (for Intel compilers)

Pre-defined configuration files

Some commonly used configuration files are found in `$DISCHARGE_HOME/Local`. `chombo-discharge` can be compiled in serial or with MPI, and with or without HDF5. The user need to configure the `Chombo` makefile to ensure that the `chombo-discharge` is properly configured. Below, we include brief instructions for compilation on a Linux workstation and for a cluster.

GNU configuration for workstations

Here, we provide a more complete installation example using GNU compilers for a workstation. These steps are intended for users that do not have installed MPI or HDF5. If you already have installed MPI and/or HDF5, the steps below might require modifications.

1. Ensure that `$DISCHARGE_HOME` and `$CHOMBO_HOME` point to the correct locations:

```
echo $DISCHARGE_HOME
echo $CHOMBO_HOME
```

2. Install GNU compiler dependencies by

```
sudo apt install csh gfortran g++ libblas-dev liblapack-dev
```

This will install

- LAPACK and BLAS

- GNU compilers for Fortran and C++

3. To also install OpenMPI and HDF5:

```
sudo apt install libhdf5-dev libhdf5-openmpi-dev openmpi-bin
```

This will install

- OpenMPI
- HDF5, both serial and parallel.

Both serial and parallel HDF5 will be installed, and these are *usually* found in folders

- `/usr/lib/x86_64-linux-gnu/hdf5/serial/` for serial HDF5
- `/usr/lib/x86_64-linux-gnu/hdf5/openmpi/` for parallel HDF5 (using OpenMPI).

Before proceeding further, the user need to locate the HDF5 libraries (if building with HDF5).

4. After installing the dependencies, copy the desired configuration file to `$CHOMBO_HOME/mk`:

- **Serial build without HDF5**:

```
cp $DISCHARGE_HOME/Local/Make.defs.GNU $CHOMBO_HOME/mk/Make.defs.local
```

- **Serial build with HDF5**:

```
cp $DISCHARGE_HOME/Local/Make.defs.HDF5.GNU $CHOMBO_HOME/mk/Make.defs.local
```

- **MPI build without HDF5**:

```
cp $DISCHARGE_HOME/Local/Make.defs.MPI.GNU $CHOMBO_HOME/mk/Make.defs.local
```

- **MPI build with HDF5**:

```
cp $DISCHARGE_HOME/Local/Make.defs.MPI.HDF5.GNU $CHOMBO_HOME/mk/Make.defs.local
```

5. Compile the `chombo-discharge`

```
cd $DISCHARGE_HOME
make -s -j4 discharge-lib
```

This will compile the `chombo-discharge` source code using the configuration settings set by the user. To compile `chombo-discharge` in 3D, do `make -s -j4 DIM=3 discharge-lib`. If successful, `chombo-discharge` libraries will appear in `$DISCARGE_HOME/Lib`.

Configuration on clusters

To configure `chombo-discharge` for executation on a cluster, use one of the makefiles supplied in `$DISCHARGE_HOME/Local` if it exists for your computer. Alternatively, copy `$DISCHARGE_HOME/Local/Make.defs.local.template` to `$CHOMBO_HOME/mk/Make.defs.local` and set the compilers, optimization flags, and paths to HDF5 library.

On clusters, MPI and HDF5 are usually already installed, but must usually be loaded (e.g. as modules) before compilation.

1.3.5 Running an example application

In chombo-discharge, applications are set up so that they use the chombo-discharge source code and one chombo-discharge physics module. To run one of the applications that use a particular chombo-discharge physics module, we will run a simulation of a positive streamer (in air).

The application code is located in $DISCHARGE_HOME/Examples/CdrPlasma/DeterministicAir and it uses the convection-diffusion-reaction plasma module (located in $DISCHARGE_HOME/Physics/CdrPlasma).

First, compile the application by

```
cd $DISCHARGE_HOME/Examples/CdrPlasma/DeterministicAir
make -s -j4 DIM=2 program
```

This will provide an executable named program2d.<bunch_of_options>.ex. If one compiles for 3D, i.e. DIM=3, the executable will be named program3d.<bunch_of_options>.ex.

To run the application do:

Serial build

```
./program2d.<bunch_of_options>.ex positive2d.inputs
```

Parallel build

```
mpirun -np 8 program2d.<bunch_of_options>.ex positive2d.inputs
```

If the user also compiled with HDF5, plot files will appear in the subfolder plt.

1.3.6 Troubleshooting

If the prerequisites are in place, compilation of chombo-discharge is usually straightforward. However, due to dependencies on Chombo and HDF5, compilation can sometimes be an issue. Our experience is that if Chombo compiles, so does chombo-discharge.

If experiencing issues, try cleaning chombo-discharge by

```
cd $DISCHARGE_HOME
make pristine
```

Note: Do not hesitate to contact us at GitHub regarding installation issues.

Recommended configurations

Production runs

For production runs, we generally recommend that the user compiles with DEBUG=FALSE and OPT=HIGH. These settings can be set directly in Make.defs.local. Alternatively, they can be included directly on the command line when compiling problems.

Debugging

If you believe that there might be a bug in the code, one can compile with `DEBUG=TRUE` and `OPT=TRUE`. This will turn on some assertions throughout `Chombo` and `chombo-discharge`.

Common problems

- Missing library paths:

  On some installations the linker can not find the HDF5 library. To troubleshoot, make sure that the the environment variable `LD_LIBRARY_PATH` can find the HDF5 libraries:

  ```
  echo $LD_LIBRARY_PATH
  ```

  If the path is not included, it can be defined by:

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/<path_to_hdf5_installation>/lib
  ```

1.4 Visualization

`chombo-discharge` output files are written to HDF5 files in the format `<simulation_name>.step#.dimension.hdf5` and the files will be written to the directory specified by *Driver* runtime parameters. Currently, we have only used VisIt for visualizing the plot files.

1.5 Controlling `chombo-discharge`

In this chapter we give a brief overview of how to run a `chombo-discharge` simulation and control its behavior through input scripts or command line options.

1.5.1 Setting up applications

To set up an application from an existing physics template, navigate to the desired physics module and use the Python setup tools. The modules are located in `$DISCHARGE_HOME/Physics` and the Python setup tools will set up the main C++ file, makefile, and a templated inputs file. Use the documentation for each module to see how to set up applications (see *Implemented models*).

1.5.2 Compiling and running

To run simulations, the user must first compile his application. Once the application has been setup, the user can compile by

```
make -s -j 32 DIM=N <application_name>
```

where *N* may be 2 or 3, and <application_name> is the name of the file that holds the `main()` function. This will compile an executable whose name depends on your application name and compiler settings. Please refer to the `Chombo` manual for explanation of the executable name. You may, of course, rename your application.

Compilation options

chombo-discharge can compile with various code guards enabled, to spot bugs or potential errors in the code. To compile with these guards turned on, compile with DEBUG=TRUE, e.g. make -s -j32 DIM=2 DEBUG=TRUE <application_name>.

To compile for production runs, chombo-discharge should generally speaking be compiled with DEBUG=FALSE and OPT=HIGH, for example

```
make -s -j32 DIM=2 OPT=HIGH DEBUG=FALSE <application_name>
```

Recall also that default settings for the dimension (DIM), optimization level (OPT), and debug mode (DEBUG) can be set in Make.defs.local, see *Installation*.

Running in serial

Next, if the application was compiled for serial execution one runs it with:

```
./<application_executable> <input_file>
```

where <input_file> is your input file.

Running with MPI

If the executable was compiled with MPI, one executes with e.g. mpirun:

```
mpirun -np 32 <application_executable> <input_file>
```

On clusters, this is a little bit different and usually requires passing the above command through a batch system.

You may also pass input parameters through the command line. For example, running

```
mpirun -np 32 <application_executable> <input_file> Driver.max_steps=10
```

will set the Driver.max_steps parameter to 10. Command-line parameters override definitions in the input file. Moreover, parameters parsed through the command line become static parameters, i.e. they are not run-time configurable (see *Run-time configurations*). Also note that if you define a parameter multiple times in the input file, the last definition is canon.

1.5.3 Simulation inputs

chombo-discharge simulations take their input from a single simulation input file (possibly appended with overriding options on the command line, as in the example above). Simulations may consist of several hundred possible switches for altering the behavior of a simulation, and all physics models in chombo-discharge are therefore equipped with Python setup tools that collect all such options into a single file when setting up a new application. Generally, these input parameters are fetched from the options file of component that is used in a simulation. Simulation options usually consist of a prefix, a suffix, and a configuration value. For example, the configuration options that adjusts the number of time steps that will be run in a simulation is

```
Driver.max_steps = 100
```

Likewise, for controlling how often plot are written:

```
Driver.plot_interval = 5
```

1.5.4 Simulation outputs

Mesh data from chombo-discharge simulations is by default written to HDF5 files, and if HDF5 is disabled chombo-discharge will not write any plot or checkpoint files. In addition to plot files, MPI ranks can output information to separate files so that the simulation progress can be tracked.

Note: If users wish to write or output other types of data, they must supply code for this themselves.

chombo-discharge comes with controls for adjusting output. Through the *Driver* class the user may adjust the option Driver.output_directory to specify where output files will be placed. This directory is relative to the location where the application is run. If this directory does not exist, chombo-discharge will create it. It will also create the following subdirectories:

- output_directory/chk contains all checkpoint files (these are used for restarting simulations from a specified time step).
- output_directory/crash contains plot files written if a simulation crashes.
- output_directory/geo contains plot files for geometries (if you run with Driver.geometry_only = true).
- output_directory/mpi contains information about individual MPI ranks, such as computational loads or memory consumption per rank.
- output_directory/plt contains all plot files.
- output_directory/regrid contains plot files written during regrids (if you run with Driver.write_regrid_files).
- output_directory/restart contains plot files written during restarts (if you run with Driver.write_regrid_files).

The reason for this structure is that chombo-discharge can end up writing thousands of files per simulation and we feel that having a directory structure helps us navigate simulation data.

Fundamentally, there are only two types of HDF5 files written:

1. Plot files, containing plots of simulation data.
2. Checkpoint files, which are binary files used for restarting a simulation from a given time step.

The *Driver* class is responsible for writing output files at specified intervals, but the user is responsible for specifying what goes into those files. Since not all variables are always of interest, solver classes have options like plt_vars that specify which output variables in the solver will be written to the output file. For example, one of our convection-diffusion-reaction solver classes have the following output options:

```
CdrGodunov.plt_vars = phi vel dco src ebflux # Plot variables. Options are 'phi', 'vel', 'dco', 'src', 'ebflux'
```

where phi is the state density, vel is the drift velocity, dco is the diffusion coefficient, src is the source term, and ebflux is the flux at embedded boundaries. If you only want to plot the density, then you should put cdr_gdnv.plt_vars = phi. An empty entry like cdr_gdnv.plt_vars = will lead to run-time errors, so if you do not want a class to provide plot data you may put cdr_gdnv.plt_vars = none.

1.5.5 Controlling parallel processor verbosity

By default, Chombo will write a process output file *per MPI process* and this file will be named pout.n where n is the MPI rank. These files are written in the directory where you executed your application, and are *not* related to plot files or checkpoint files. However, chombo-discharge prints information to these files as simulations advance (for example by displaying information of the current time step, or convergence rates for multigrid solvers). While it is possible to monitor the evolution of chombo-discharge for each MPI rank, most of these files contain redundant information. To adjust the number of files that will be written, Chombo can read an environment variable CH_OUTPUT_INTERVAL. For example, if you only want the master MPI rank to write pout.0, you would do

```
export CH_OUTPUT_INTERVAL=999999999
```

If you run simulations at high concurrencies, you *should* turn off the number of process output files since they impact the performance of the file system.

1.5.6 Restarting simulations

Restarting simulations is done in exactly the same way as running simulations, although the user must set the Driver.restart parameter. For example,

```
mpirun -np 32 <application_executable> <input_file> Driver.restart=10
```

will restart from step 10.

Specifying anything but an integer is an error. When a simulation is restarted, chombo-discharge will look for a checkpoint file with the Driver.output_names variable and the specified restart step. It will look for this file in the subfolder /chk relative to the execution directory.

If the restart file is not found, restarting will not work and chombo-discharge will abort. You must therefore ensure that your executable can locate this file. This also implies that you cannot change the Driver.output_names or Driver.output_directory variables during restarts, unless you also change the name of your checkpoint file and move it to a new directory.

Note: If you set Driver.restart=0, you will get a fresh simulation.

1.5.7 Run-time configurations

chombo-discharge reads input parameters before the simulation starts, but also during run-time. This is useful when your simulation waited 5 days in the queue on a cluster before starting, but you forgot to tweak one parameter and don't want to wait another 5 days.

Driver re-reads the simulation input parameters after every time step. The new options are parsed by the core classes Driver, TimeStepper, AmrMesh, and CellTagger through special routines parseRuntimeOptions(). Note that not all input configurations are suitable for run-time configuration. For example, increasing the size of the simulation domain does not make sense but changing the blocking factor, refinement criteria, or plot intervals do. To see which options are run-time configurable, see *Driver*, *AmrMesh*, or the TimeStepper and CellTagger that you use.

CHAPTER

TWO

DISCRETIZATION

2.1 Spatial discretization

2.1.1 Cartesian AMR

`chombo-discharge` uses patch-based structured adaptive mesh refinement (AMR) provided by `Chombo` [4]. In patch-based AMR the domain is subdivided into a collection of hierarchically nested grid levels. With Cartesian AMR each patch is a Cartesian block of grid cells. A *grid level* is composed of a union of grid patches sharing the same grid resolution, with the additional requirement that the patches on a grid level are *non-overlapping*. With AMR, such levels can be hierarchically nested; finer grid levels exist on top of coarser ones. In patch-based AMR there are only a few fundamental requirements on how such grids are constructed. For example, a refined grid level must exist completely within the bounds of it's parent level. In other words, grid levels $l-1$ and $l+1$ are spatially separated by a non-zero number of grid cells on level l.

The resolution on level $l+1$ is typically finer than the resolution on level l by an integer (usually power of two). However,

Important: `chombo-discharge` only supports refinement factors of 2 and 4.

2.1.2 Embedded boundaries

`chombo-discharges` uses an embedded boundary (EB) formulation for describing complex geometries. With EBs, the Cartesian grid is directly intersected by the geometry. This is fundamentally different from unstructured grid where one generates a volume mesh that conforms to the surface mesh of the input geometry. Since EBs are directly intersected by the geometry, there is no fundamental need for a surface mesh for describing the geometry. Moreover, Cartesian EBs have a data layout which remains (almost) fully structured. The connectivity of neighboring grid cells is still trivially found by fundamental strides along the data rows/columns, which allows extending the efficiency of patch-based AMR to complex geometries. Figure Fig. 2.1.1 shows an example of patch-based grid refinement for a complex surface.

Since EBs are directly intersected by the geometry, pathological cases can arise where a Cartesian grid cell consists of multiple volumes. One can easily envision this case by intersecting a thin body with a Cartesian grid, as shown in Fig. 2.1.2. This figure shows a thin body which is intersected by a Cartesian grid, and this grid is then coarsened. At the coarsened level, one of the grid cells has two cell fragments on opposite sides of the body. Such multi-valued cells (a.k.a *multi-cells*) are fundamentally important for EB applications. Note that there is no fundamental difference between single-cut and multi-cut grid cells. This distinction exists primarily due to the fact that if all grid cells were single-cut cells the entire EB data structure would fit in a Cartesian grid block (say, of $N_x \times N_y \times N_z$ grid cells). Because of multi-cells, EB data structures are not purely Cartesian. Data structures need to live on more complex graphs that describe support multi-cells and, furthermore, describe the cell connectivity. Without multi-cells

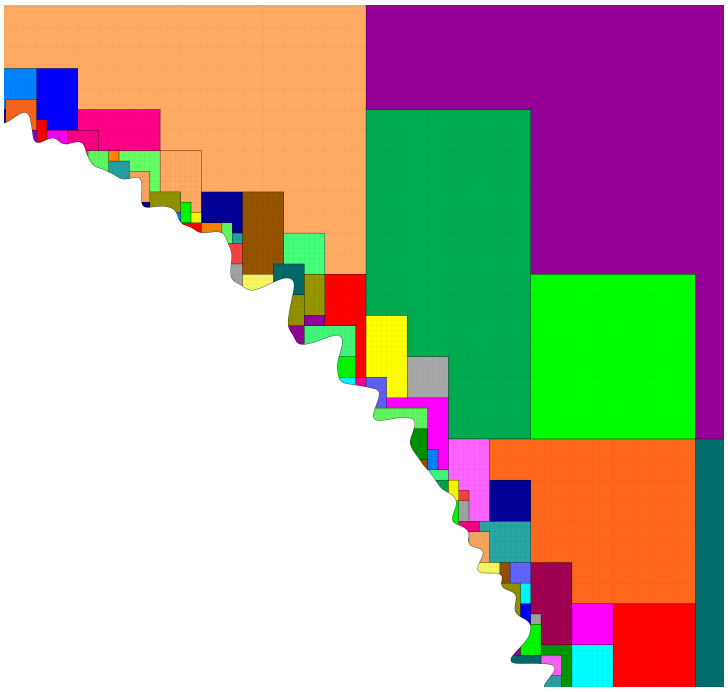

Fig. 2.1.1: Patch-based refinement (factor 4 between levels) of a complex surface. Each color shows a patch, which is a rectangular computational unit.

it would be impossible to describe most complex geometries. It would also be extremely difficult to obtain performant geometric multigrid methods (which rely on this type of coarsening).

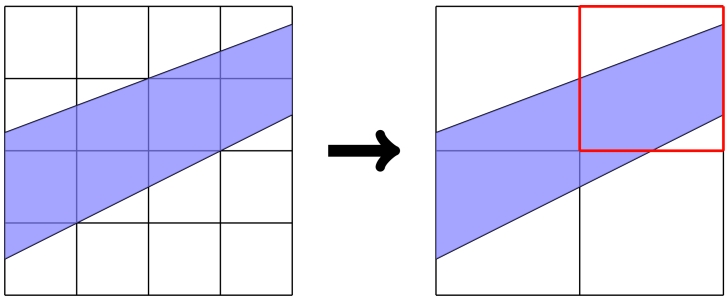

Fig. 2.1.2: Example of how multi-valued cells occur during grid coarsening. Left: Original grid. Right: Coarsened grid.

2.1.3 Geometry representation

`chombo-discharge` uses (approximations to) signed distance functions (SDFs) for describing geometries. Signed distance fields are functions $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ that describe the distance from the object. These functions are also *implicit functions*, i.e. $f(\mathbf{x}) = 0$ describes the surface of the object, $f(\mathbf{x}) > 0$ decribes a point inside the object and $f(\mathbf{x}) < 0$ describes a point outside the object.

Many EB applications only use the implicit function formulation, but `chombo-discharge` requires (an approximation to) the signed distance field. There are two reasons for this:

1. The SDF can be used for robustly load balancing the geometry generation with orders of magnitude speedup over naive approaches.

2. The SDF is useful for resolving particle collisions with boundaries, using e.g. simple ray tracing of particle paths.

To illustrate the difference between an SDF and an implicit function, consider the implicit functions for a sphere at the origin with radius R:

$$d_1(\mathbf{x}) = R - |\mathbf{x}|, \tag{2.1.1}$$

$$d_2(\mathbf{x}) = R^2 - \mathbf{x} \cdot \mathbf{x}. \tag{2.1.2}$$

Here, only $d_1(\mathbf{x})$ is a signed distance function.

In `chombo-discharge`, SDFs can be generated through analytic expressions, constructive solid geometry, or by supplying polygon tesselation. NURBS geometries are, unfortunately, not supported. Fundamentally, all geometric objects are described using `BaseIF` objects from `Chombo`, see *BaseIF*. Support for polygon surfaces differ from analytic functions only in how we implement the `BaseIF::value` function (see *BaseIF*).

Constructive solid geometry (CSG)

Constructive solid geometry can be used to generate complex shapes from geometric primitives. For example, to describe the union between two SDFs $d_1(\mathbf{x})$ and $d_2(\mathbf{x})$:

$$d(\mathbf{x}) = \min(d_1(\mathbf{x}), d_2(\mathbf{x}))$$

Note that the resulting is an implicit function but is *not* an SDF. However, the union typically approximates the signed distance field quite well near the surface. `Chombo` natively supports many ways of performing CSG.

Polygon surfaces

While functions like $R - |\mathbf{x}|$ are quick to compute, a polygon surface may consist of hundreds of thousands of primitives (e.g., triangles). Generating signed distance function from polygon tesselations is quite involved as it requires computing the signed distance to the closest feature, which can be a planar polygon (e.g., a triangle), edge, or a vertex. `chombo-discharge` supports such functions.

Warning: The signed distance function for a polygon surface is only well-defined if it is manifold-2, i.e. it is watertight and does not self-intersect. `chombo-discharge` should nonetheless compute the distance field as best as it can, but the final result may not make sense in an EB context.

Searching through all features (faces, edge, vertices) is unacceptably slow, and `chombo-discharge` therefore uses a bounding volume hierarchy for accelerating these searches. The bounding volume hierarchy is top-down constructed, using a root bounding volume (typically a cube) that encloses all triangles. Using heuristics, the root bounding volume is then subdivided into two separate bounding volumes that contain roughly half of the primitives each. The process is then recursed downwards until specified recursion criteria are met. Additional details are provided in *Complex geometries*.

Some options are available for tuning this process:

- The user can choose between Cartesian bounding volumes or bounding spheres.
- The implementation uses polymorphic lambdas for subdividing primitives into new bounding volumes. Users are free to construct new types of bounding volumes (e.g., object-oriented boxes).
- The implementation uses the same type of polymorphic functions for supplying stopping criteria (e.g., terminating at a certain depth).

Users can choose to use the default implementations for both sub-division and stopping. For example, three default partitioning functions are supported. One can sub-divide primitives into two sub-volumes based on their centroids, or such that the volume between the two halves are minimized.

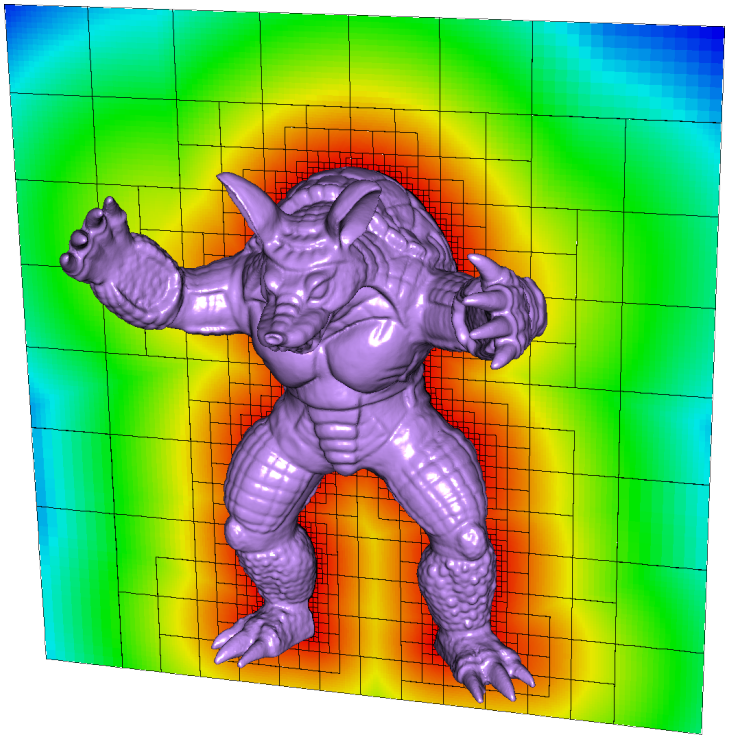

Fig. 2.1.3: Example of an SDF reconstruction and cut-cell grid from a surface tesselation in `chombo-discharge`.

Note: The polygon functionality is currently limited to PLY files. Contributions that provide plugins to other file formats are encouraged.

2.1.4 Geometry generation

`Chombo` approach

The default geometry generation method in `Chombo` is to locate cut-cells on the finest AMR level first. This scan looks through all cells on that level, so for a domain which is effectively $N \times N \times N$ cells there are at least N^3 implicit function queries (in 2D, the complexity is N^2). Note that as N becomes large, say $N = 10^5$, geometric queries of this type become a bottleneck. For example, even if the SDF evaluation for an arbitrary point $\mathbf{x}$ only took 10^{-7} seconds to evaluate, and the application was parallelized over 10,000 cores, it would take at least twenty hours to figure out where the cut-cells are located.

`chombo-discharge` pruning

`chombo-discharge` has made modifications to the geometry generation routines in `Chombo`, resolving a few bugs and, most importantly, using the signed distance function for load balancing the geometry generation step. This modification to `Chombo` is motivated by the reduction of the N^3 scaling in the native `Chombo` grid generation to an N^2 scaling in `chombo-discharge`. Typically, we find that this makes geometry generation computationally trivial (in the sense that it becomes extremely fast).

The SDF satisfies the Eikonal equation

$$|\nabla f| = 1, \tag{2.1.3}$$

and so it is well-behaved for all $\mathbf{x}$, and can be used to prune large regions in space where cut-cells don't exist. For example, consider a Cartesian grid patch with cell size Δx and cell-centered grid points $\mathbf{x_i} = \left(\mathbf{i} + \frac{\mathbf{1}}{\mathbf{2}}\right)\Delta x$ where $\mathbf{i} \in \mathbb{Z}^3$ are grid cells in the patch, like that shown in Fig. 2.1.4. We know that cut cells do not exist in the grid patch if $|f(\mathbf{x_i})| > \frac{1}{2}\Delta x$ for all $\mathbf{i}$ in the patch. One can use this to perform a quick scan of the SDF on a *coarse* grid level first, for example on $l = 0$, and recurse deeper into the grid hierarchy to locate cut-cells on the other levels. Typically, a level is decomposed into Cartesian subregions, and each subregion can be scanned independently of the other subregions (i.e. the problem is embarassingly parallel). Subregions that can't contain cut-cells are designated as *inside* or *outside*, depending on the sign of the SDF. There is no point in recursively refining these to look for cut-cells at finer grid levels, owing to the nature of the SDF they can be safely pruned from subsequent scans at finer levels. The subregions that did contain cut-cells are refined and decomposed into sub-subregions. This procedure recurses until $l = l_{\text{max}}$, at which point we have determined all sub-regions in space where cut-cells can exist (on each AMR level), and pruned the ones that don't. This process is shown in Fig. 2.1.4. Once all the grid patches that contain cut-cells have been found, these patches can be distributed (i.e. load balanced) to the various MPI ranks or threads for computing the discrete information.

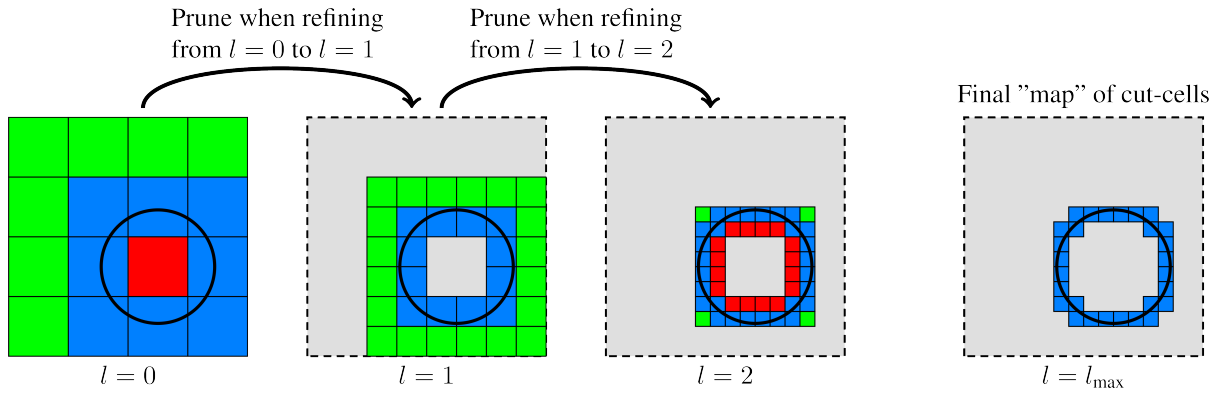

Fig. 2.1.4: Pruning cut-cells with the signed distance field. Red-colored grid patches are grid patches entirely contained inside the EB. Green-colored grid patches are entirely outside the EB, while blue-colored grid patches contain cut-cells.

The above load balancing strategy is very simple, and it reduces the original $O(N^3)$ complexity in 3D to $O(N^2)$ complexity (in 2D the complexity is reduced from $O(N^2)$ to $O(N)$). The strategy works for all SDFs although, strictly speaking, an SDF is not fundamentally needed. If a well-behaved Taylor series can be found for an implicit function, the bounds on the series can also be used to infer the location of the cut-cells, and the same algorithm can be used. For example, generating compound objects with CSG are typically sufficiently well behaved (provided that the components are SDFs). However, implicit functions like $d(\mathbf{x}) = R^2 - \mathbf{x} \cdot \mathbf{x}$ must be used with caution.

2.1.5 Mesh generation

`chombo-discharge` offers two algorithm for AMR grid generation, the classical Berger-Rigoutsos algorithm and a *tiled* algorithm. Both algorithms work by taking a set of flagged cells on each grid level and generating new boxes that cover the flags.

Berger-Rigoutsos algorithm

The Berger-Rigoustous grid algorithm is implemented in `Chombo` and is called by `chombo-discharge`. The classical Berger-Rigoustous algorithm is inherently serial in the sense that is collects the flagged cells onto each MPI rank and then generates the boxes. Typically, it is not used at large scale in 3D due to its memory consumption.

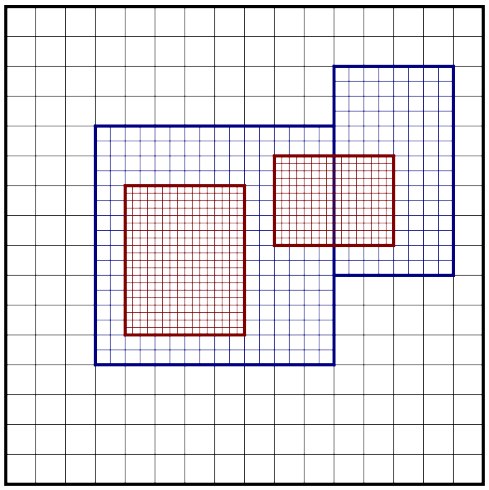

Fig. 2.1.5: Classical cartoon of patch-based refinement. Bold lines indicate entire grid blocks.

Tiled mesh refinement

`chombo-discharge` also supports a tiled algorithm where the grid boxes on each block are generated according to a predefined tiled pattern. If a tile contains a single tag, the entire tile is flagged for refinement. The tiled algorithm produces grids that are visually similar to octrees, but is more general since it also supports refinement factors other than 2 and is not restricted to domain extensions that are an integer factor of 2 (e.g. 2^{10} cells in each direction). Moreover, the algorithm is extremely fast and has low memory consumption even at large scales.

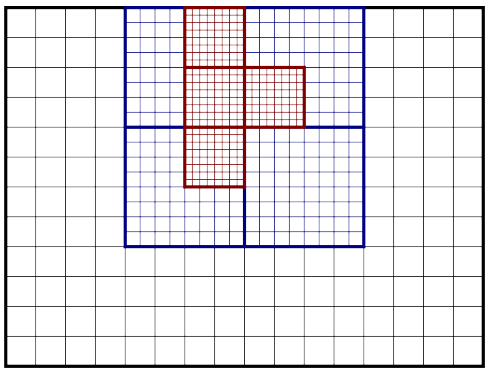

Fig. 2.1.6: Classical cartoon of tiled patch-based refinement. Bold lines indicate entire grid blocks.

2.1.6 Cell refinement philosophy

chombo-discharge can flag cells for refinement using various methods:

1. Refine all embedded boundaries down to a specified refinement level.
2. Refine embedded boundaries based on estimations of the surface curvature in the cut-cells.
3. Manually add refinement flags (by specifying boxes where cells will be refined).
4. Physics-based or data-based refinement where the user fetches data from solver classes (e.g., discretization errors, the electric field) and uses that for refinement.

The first two cases are covered by the Driver class in chombo-discharge (see *Driver*). In the first case the Driver class will simply fetch arguments from an input script which specifies the refinement depth for the embedded boundaries. In the second case, the Driver class will visit every cut-cell and check if the normal vectors in neighboring cut-cell deviate by more than a specified threshold angle. Given two normal vectors $\mathbf{n}$ and $\mathbf{n}'$, the cell is refined if

$$\mathbf{n} \cdot \mathbf{n}' \geq \cos\theta_c,$$

where θ_c is a threshold angle for grid refinent.

The other two cases are more complicated, and are covered by the *GeoCoarsener* and *CellTagger* classes.

2.2 Chombo-3 basics

To fully understand this documentation the user should be familiar with Chombo. This documentation uses class names from Chombo and the most relevant Chombo data structures are summarized here. What follows is a *very* brief introduction to these data structures, for in-depth explanations please see the Chombo manual.

2.2.1 Real

Real is a typedef'ed structure for holding a single floating point number. Compiling with double precision will typedef Real as double, otherwise it is typedef'ed as float.

2.2.2 RealVect

RealVect is a spatial vector. It holds two Real components in 2D and three Real components in 3D. The RealVect class has floating point arithmetic, e.g. addition, subtraction, multiplication etc.

Most of chombo-discharge is written in dimension-independent code, and for cases where RealVect is initialized with components the constructor uses Chombo macros for expanding the correct number of arguments. For example

```
RealVect v(D_DECL(vx, vy, vz));
```

will expand to RealVect v(vx,vy) in 2D and RealVect v(vx, vy, vz) in 3D.

2.2.3 IntVect

`IntVect` is an integer spatial vector, and is used for indexing data structures. It works in much the same way as `RealVect`, except that the components are integers.

2.2.4 Box

The `Box` object describes a box in Cartesian space. The boxes are indexed by the low and high corners, both of which are an `IntVect`. The `Box` may be cell-centered or face-centered. To turn a cell-centered `Box` into a face-centered box one would do

```
Box bx(IntVect::Zero, IntVect::Unit); // Default constructor give cell centered boxes
bx.surroundingNodes():                // Now a cell-centered box
```

This will increase the box dimensions by one in each coordinate direction.

2.2.5 EBCellFAB and FArrayBox

The `EBCellFAB` object is an array for holding cell-centered data in an embedded boundary context. The `EBCellFAB` has two data structures: An `FArrayBox` that holds the data on the cell centers, and a additional data structure that holds data in cells that are multiply cut. Doing arithmetic with `EBCellFAB` usually requires one to iterate over all the cell in the `FArrayBox`, and then to iterate over the *irregular cells* (i.e. cut-cells) later. A `VoFIterator` is such as object; it can iterate over cut-cells. Usually, code for doing anything with the `EBCellFAB` looks like this:

```
// Call Fortran code
FORT_DO_SOMETHING(....)

// Iterate over cut-cells
for (VoFIterator vofit(...); vofit.ok(); ++vofit){
   (...)
}
```

2.2.6 Vector

`Vector<T>` is a one-dimensional array with constant-time random access and range checking. It uses `std::vector` under the hood and can access the most commonly used `std::vector` functionality through the public member functions. E.g. to obtain an element in the vector

```
Vector<T> my_vector(10, T());

T& element = my_vector[5];
```

Likewise, `push_back`, `resize` etc works in much the same way as for `std::vector`.

2.2.7 RefCountedPtr

`RefCountedPtr<T>` is a pointer class in `Chombo` with reference counting. That is, when objects that hold a reference to some `RefCountedPtr<T>` object goes out of scope the reference counter is decremented. If the reference counter reaches zero, the object that `RefCountedPtr<T>` points to it deallocated. Using `RefCountedPtr<T>` is much preferred over using a raw pointer `T*` to 1) avoid memory leaks and 2) compress code since no explicit deallocations need to be called.

In modern C++-speak, `RefCountedPtr<T>` can be thought of as a *very* simple version of `std::shared_ptr<T>`.

2.2.8 DisjointBoxLayout

The `DisjointBoxLayout` class describes a grid on an AMR level where all the boxes are *disjoint*, i.e. they don't overlap. `DisjointBoxLayout` is built upon a union of non-overlapping boxes having the same grid resolution and with unique rank-to-box ownership. The constructor is

```
Vector<Box> boxes(...);  // Vector of disjoint boxes
Vector<int> ranks(...);  // Ownership of each box

DisjointBoxLayout dbl(boxes, ranks);
```

In simple terms, `DisjointBoxLayout` is the decomposed grid on each level in which MPI ranks have unique ownership of specific parts of the grid.

The `DisjointBoxLayout` view is global, i.e. each MPI rank knows about all the boxes and the box ownership on the entire AMR level. However, ranks will only allocate data on the part of the grid that they own. Data iterators also exist, and the most common is to use iterators that only iterate over the part of the `DisjointBoxLayout` that the specific MPI ranks own:

```
DisjointBoxLayout dbl;
for (DataIterator dit(dbl); dit.ok(); ++dit){
   // Do something
}
```

Each MPI rank will then iterate *only* over the part of the grid where it has ownership.

Other data iterators exist that iterate over all boxes in the grid:

```
for (LayoutIterator lit = dbl.layoutIterator(); dit.ok(); ++dit){
   // Do something
}
```

This is typically used if one wants to do some global operation, e.g. count the number of cells in the grid. However, trying to use `LayoutIterator` to retrieve data that was allocated locally on a different MPI rank is an error.

2.2.9 LevelData

The `LevelData<T>` template structure holds data on all the grid patches of one AMR level. The data is distributed with the domain decomposition specified by `DisjointBoxLayout`, and each patch contains exactly one instance of `T`. `LevelData<T>` uses a factory pattern for creating the `T` objects, so if you have new data structures that should fit the in `LevelData<T>` structure you must also implement a factory method for `T`.

The `LevelData<T>` object provides the domain decomposition method in `Chombo` and `chombo-discharge`. Often, `T` is an `EBCellFAB`, i.e. a Cartesian grid patch that also supports EB formulations.

To iterate over `LevelData<T>` one will use the data iterator above:

```
LevelData<T> myData;
for (DataIterator dit(dbl); dit.ok(); ++dit){
   T& = myData[dit()];
}
```

`LevelData<T>` also includes the concept of ghost cells and exchange operations.

2.2.10 EBISLayout and EBISBox

The `EBISLayout` holds the geometric information over one `DisjointBoxLayout` level. Typically, the `EBISLayout` is used for fetching the geometric moments that are required for performing computations near cut-cells. `EBISLayout` can be thought of as an object which provides all EB-related information on a specific grid level. The EB information consists of e.g. cell flags (i.e., is the cell a cut-cell?), volume fractions, etc. This information is stored in a class `EBISBox`, which holds all the EB information for one specific grid patch. To obtain the EB-information for a specific grid patch, one will call:

```
EBISLayout ebisl;
for (DataIterator dit(dbl); dit.ok(); ++dit){
   EBISBox& ebisbox = ebisl[dit()];
}
```

where `EBISBox` contains the geometric information over only one grid patch. One can thus think of the `EBISLayout` as a `LevelData<EBISBox>` structure.

As an example, to iterate over all the cut-cells defined for a cell-centered data holder an AMR-level one would do:

```
constexpr int comp = 0;

// Assume that these exist.
LevelData<EBCellFAB> myData;
EBISLayout ebisl;

// Iterate over all the patches on a grid level.
for (DataIterator dit(dbl); ++dit){
   const Box  cellBox   = dbl[dit()];
   EBCellFAB& patchData = myData[dit()];
   EBISBox&   ebisbox   = ebisl [dit()];

   // Get all the cut-cells in the grid patch
   const IntVectSet& ivs = ebisbox.getIrregIVS(cellBox);
   const EBGraph&        = ebisbox.getEBGraph();

   // Define a VoFIterator for the cut-cells and iterate over all the cut-cells.
   for (VoFIterator vofit(ivs, ebgraph); vofit.ok(); ++vofit){
      const VolIndex& vof = vofit();

      patchData(vof, comp) = ...
   }
}
```

Here, `EBGraph` is the graph that describes the connectivity of the cut cells.

2.2.11 BaseIF

The `BaseIF` is a `Chombo` class which encapsulates an implicit function (recall that all SDFs are also implicit functions, see *Geometry representation*). `BaseIF` is therefore used for fundamentally constructing a geometric object. Many examples of `BaseIF` are found in `Chombo` itself, and `chombo-discharge` includes additional ones.

To implement a new implicit function, the user must inherit from `BaseIF` and implement the pure function

```
virtual Real BaseIF::value(const RealVect& a_point) const = 0;
```

The implemention should return a positive value if the point `a_point` is inside the object and a negative value otherwise.

2.3 Mesh data

Mesh data structures in chombo-discharge are derived from a class EBAMRData<T> which holds a T in every grid patch across the AMR hiearchy. Internally, the data is stored as a Vector<RefCountedPtr<LevelData<T> > >. Here, the Vector holds data on each AMR level; the data is allocated with a smart pointer called RefCountedPtr which points to a LevelData template structure, see *Chombo-3 basics*. The first entry in the Vector is base AMR level and finer levels follow later in the Vector.

The reason for having class encapsulation of mesh data is due to *Realm*, so that we can only keep track on which Realm the mesh data is defined. Users will not have to interact with EBAMRData<T> directly, but will do so primarily through application code, or interacting with the core AMR functionality in *AmrMesh* (such as computing gradients, interpolating ghost cells etc.). AmrMesh (see *AmrMesh*) has functionality for defining most EBAMRData<T> types one a Realm, and EBAMRData<T> itself it typically not used anywhere elsewhere within chombo-discharge.

A number of explicit template specifications also exist, and these are outlined below:

```
typedef EBAMRData<EBCellFAB>         EBAMRCellData;  // Cell-centered single-phase data
typedef EBAMRData<EBFluxFAB>         EBAMRFluxData;  // Face-centered data in all coordinate direction
typedef EBAMRData<EBFaceFAB>         EBAMRFaceData;  // Face-centered in a single coordinate direction
typedef EBAMRData<BaseIVFAB<Real> >  EBAMRIVData;    // Data on irregular data centroids
typedef EBAMRData<DomainFluxIFFAB>   EBAMRIFData;    // Data on domain phases
typedef EBAMRData<BaseFab<bool> >    EBAMRBool;      // For holding bool at every cell

typedef EBAMRData<MFCellFAB>         MFAMRCellData;  // Cell-centered multifluid data
typedef EBAMRData<MFFluxFAB>         MFAMRFluxData;  // Face-centered multifluid data
typedef EBAMRData<MFBaseIVFAB>       MFAMRIVData;    // Irregular face multifluid data
```

For example, EBAMRCellData is a Vector<RefCountedPtr<LevelData<EBCellFAB> > >, describing cell-centered data across the entire AMR hierarchy. There are many more data structures in place, but the above data structures are the most commonly used ones. Here, EBAMRFluxData is precisely like EBAMRCellData, except that the data is stored on *cell faces* rather than cell centers. Likewise, EBAMRIVData is a typedef'ed data holder that holds data on each cut-cell center across the entire AMR hierachy. In the same way, EBAMRIFData holds data on each face of all cut cells.

2.3.1 Allocating mesh data

To allocate data over a particular Realm, the user will interact with AmrMesh (see *AmrMesh*):

```
int nComps = 1;
EBAMRCellData myData;
m_amr->allocate(myData, "myRealm", phase::gas, nComps);
```

Note that it *does* matter on which Realm and on which phase the data is defined. See *Realm* for details.

The user *can* specify a number of ghost cells for his/hers application code directly in the AmrMesh::allocate routine, like so:

```
int nComps = 1;
EBAMRCellData myData;
m_amr->allocate(myData, "myRealm", phase::gas, nComps, 5*IntVect::Unit);
```

If the user does not specify the number of ghost cells when calling AmrMesh::allocate, AmrMesh will use the default number of ghost cells specified in the input file.

2.3.2 Iterating over data

To iterate over data in an AMR hierarchy, you will first iterate over levels and the patches in levels:

```
for (int lvl = 0; lvl < myData.size(); lvl++){
   LevelData<EBCellFAB>& levelData = *myData[lvl];

   const DisjointBoxLayout& levelGrids = levelData.disjointBoxLayout();

   for (DataIterator dit = levelGrids.dataIterator(); dit.ok(); ++dit){
      EBCellFAB& patchData = levelData[dit()];
   }
}
```

Iterating over the cells in a patch data holder (like the `EBCellFAB`) can be done with a `VoFIterator`. The `VoFIterator` can iterate through cells on an `EBCellFAB` that are not covered by the geometry, but it can be slow to define so the typical iteration structure consists of a loop that iterates through all cells first, and then the irregular cells later. For example:

```
const int component = 0;

for (int lvl = 0; lvl < myData.size(); lvl++){
   LevelData<EBCellFAB>& levelData = *myData[lvl];

   const DisjointBoxLayout& levelGrids = levelData.disjointBoxLayout();

   for (DataIterator dit = levelGrids.dataIterator(); dit.ok(); ++dit){
      const Box bx = levelGrids[dit()];

      EBCellFAB& patchData       = levelData[dit()];
      BaseFab<Real>& regularData = patchData.getSingleValuedFab(); // Regular data

      // Iterate through all cells in bx
      for (BoxIterator bit(bx); bit.ok(); ++bit){
         const IntVect iv = bit();

         regularData(iv, component) = ....
      }

      // Iterate through irregular cells later
      for (VoFIterator vofit(...); vofit.ok(); ++vofit){
         const VolIndex& vof = vofit();

         patchData(vof, component)  = ...
      }
   }
}
```

2.3.3 Computing gradients

In `chombo-discharge` gradients are computed using a standard second-order stencil based on finite differences. This is true everywhere except near the refinement boundary and EB where the coarse-side stencil will avoid using the coarsened data beneath the fine level. This is shown in Fig. 2.3.1 which shows the typical 5-point stencil in regular grid regions, and also a much larger and more complex stencil.

In Fig. 2.3.1 we have shown two regular 5-point stencils (red and green). The coarse stencil (red) reaches underneath the fine level and uses the data defined by coarsening of the fine-level data. The coarsened data in this case is just an average of the fine-level data. Likewise, the green stencil reaches over the refinement boundary and into one of the ghost cells on the coarse level.

Fig. 2.3.1 also shows a much larger stencil (blue stencil). The larger stencil is necessary because computing the y component of the gradient using a regular 5-point stencil would have the stencil reach underneath the fine level and into coarse data that is also irregular data. Since there is no unique way (that we know of) for coarsening the cut-cell fine-level data onto the coarse cut-cell without introducing spurious artifacts into the gradient, we reconstruct the gradient using a least squares procedure. In this case we fetch a sufficiently large neighborhood of cells for computing a least squares minimization of a local solution reconstruction in the neighborhood of the coarse cell. In order to avoid

fetching potentially badly coarsened data, this neighborhood of cells only uses *valid* grid cells, i.e. the stencil does not reach underneath the fine level at all. Once this neighborhood of cells is obtained, we compute the gradient using the procedure in *Least squares*.

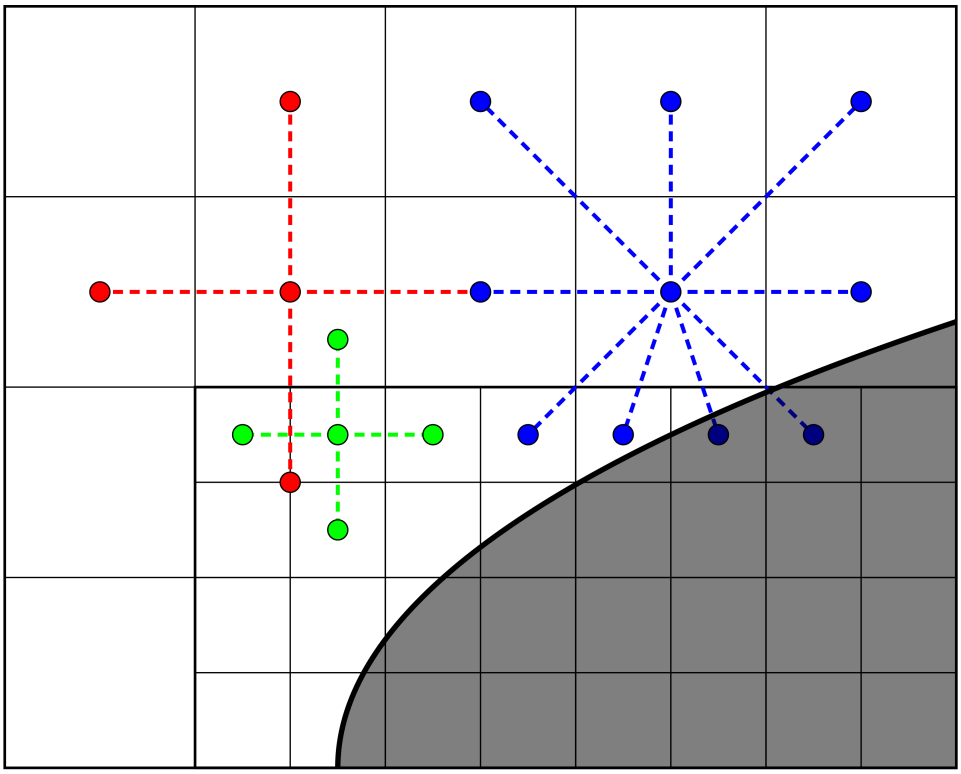

Fig. 2.3.1: Example of stencils for computing gradients near embedded boundaries. The red stencil shows a regular 5-point stencil for computing the gradient on the coarse side of the refinement boundary; it reaches into the coarsened data beneath the fine level. The green stencil shows a similar 5-point stencil on the fine side of the refinement boundary; the stencil reaches over the refinement boundary and into one ghost cell. The blue stencils shows a much more complex stencil which is computed using a least squares reconstruction procedure.

To compute gradients of a scalar, one can simply call `AmrMesh::computeGradient(...)`. See *AmrMesh* for details.

2.3.4 Common data operations

DataOps

We have prototyped functions for many common data operations in a static class `DataOps` (see `$DISCHARGE_HOME/Source/Utilities/CD_DataOps.H`). For example, setting the value of a data holder can be done with

```
EBAMRFluxData cellData;
EBAMRFluxData fluxData;
EBAMRIVData   irreData;

DataOps::setValue(cellData, 0.0);
DataOps::setValue(fluxData, 1.0);
DataOps::setValue(irreData, 2.0);
```

Many functions are available in `DataOps`. Common data operations should always be put in this class.

Copying data

To copy data, one may use the `EBAMRData<T>::copy(...)` function *or* `DataOps`. These differ in the sense that `DataOps` will always do a local copy, and thus the data that is copied *must* be defined on the same realm. Runtime errors will occur otherwise. If you call `EBAMRData<T>::copy(...)`, the data holders will first check if they are both defined on the same realm. If they are, a purely local copy is perform. Communication copies involving MPI are performed otherwise.

Coarsening

Conservative coarsening data is done using the `averageDown(...)` function in `AmrMesh`, with signatures as follows:

```
// Conservatively coarsen multifluid cell-centered data
void averageDown(MFAMRCellData& a_data, const std::string a_realm) const;

// Conservatively coarsen multifluid face-centered data
void averageDown(MFAMRFluxData& a_data, const std::string a_realm) const;

// Conservatively coarsen cell-centered data
void averageDown(EBAMRCellData& a_data, const std::string a_realm, const phase::which_phase a_phase) const;

// Conservatively coarsen face-centered data
void averageDown(EBAMRFluxData& a_data, const std::string a_realm, const phase::which_phase a_phase) const;

// Conservatively coarsen EB-centered data
void averageDown(EBAMRIVData& a_data, const std::string a_realm, const phase::which_phase a_phase) const;
```

See *AmrMesh* for additional details.

Ghost cells

Filling ghost cells is done using the `interpGhost(...)` function in `AmrMesh`.

```
// Interpolate ghost cells for multifluid cell-centered data.
void interpGhost(MFAMRCellData& a_data, const std::string a_realm) const;

// Interpolate ghost cells for multifluid cell-centered data
void interpGhost(EBAMRCellData& a_data, const std::string a_realm, const phase::which_phase a_phase) const;
```

This will fill the specified number of ghost cells using data from the coarse level only, using piecewise linear interpolation (with limiters).

As an alternative, one can interpolate a single layer of ghost cells using the multigrid interpolator (see *Ghost cell interpolation*). In this case only a single layer of ghost cells are filled in regular regions, but additional ghost cells (up to some specified range) are filled near the EB. This is often required when computing gradients (to avoid reaching into invalid cut-cells). See *Computing gradients*.

2.4 Particles

`chombo-discharge` supports computational particles using native `Chombo` particle data. The source code for the particle functionality resides in `$DISCHARGE_HOME/Source/Particle`.

2.4.1 ParticleContainer

The `ParticleContainer<P>` is a template class that

1. Stores computational particles of type `P` over an AMR hierchy.
2. Provides infrastructure for mapping and remapping.

`ParticleContainer<P>` uses the `Chombo` structure `ParticleData<P>` under the hood, and therefore has template constraints on `P`. The simplest way to use `ParticleContainer` for a new type of particle is to let `P` inherit from the `Chombo` class `BinItem`. `BinItem` only has a single member variable which is its position, but derived classes will contain more and must therefore also add new linearization functions if the new member variables should be communicated. Please refer to the `Chombo` design document for complete specification on the template constraints of `P`, or see some of the examples in `chombo-discharge`.

2.4.2 Data structures

List<P> and ListBox<P>

At the lowest level the particles are always stored in a linked list `List<P>`. The class can be simply be through of as a regular list of `P`.

The `ListBox<P>` consists of a `List<P>` *and* a `Box`. The latter specifies the grid patch that the particles are assigned to.

To get the list of particles from a `ListBox<P>`:

```
ListBox<P> myListBox;

List<P>& myList = myListBox.listItems();
```

ListIterator<P>

In order to iterate over particles, you will use an iterator `ListIterator<P>` (which is not random access):

```
List<P> myParticles;
for (ListIterator<P> lit(myParticles); lit.ok(); ++lit){
   P& p = lit();

   // ... do something with this particle
}
```

ParticleData<P>

On each grid level, `ParticleContainer<P>` stores the particles in a `Chombo` class `ParticleData`.

```
template <class P>
ParticleData<P>
```

where `P` is the particle type. `ParticleData<P>` can be thought of as a `LevelData<P>`, although it actually inherits from `LayoutData<ListBox<P> >`. Each grid patch therefore contains a `ListBox<P>` of particles.

AMRParticles<P>

AMRParticles<P> is our AMR version of `ParticleData<P>`. It is a simply a typedef of a vector of pointers to `ParticleData<P>` on each level:

```
template <class P>
using AMRParticles = Vector<RefCountedPtr<ParticleData<P> > >;
```

Again, the `Vector` indicates the AMR level and the `ParticleData<P>` is a distributed data holder that holds the particles on each AMR level.

2.4.3 Basic use

The public member functions for `ParticleContainer` can be found in `chombo-discharge/Source/Particle/CD_ParticleContainer.H`. Here, we give some examples of basic use of `ParticleContainer`.

Getting the particles

To get the particles from a `ParticleContainer<P>` one can call `AMRParticles<P>& ParticleContainer<P>::getParticles()` which will provide the particles:

```
ParticleContainer<P> myParticleContainer;

AMRParticles<P>& myParticles = myParticleContainer.getParticles();
```

Alternatively, one can fetch directly from a specified grid level by calling `ParticleContainer<P> ParticleContainer<P>::operator[](int)`, e.g.

```
int lvl;
ParticleContainer<P> myParticleContainer;

ParticleData<P>& levelParticles = myParticleContainer[lvl];
```

Iterating over particles

To do something basic with the particle in a `ParticleContainer<P>`, one will typically iterate over the particles in all grid levels and patches.

The code bit below shows a typical example of how the particles can be moved, and then remapped onto the correct grid patches and ranks if they fall off their original one.

```
ParticleContainer<P> myParticleContainer;

// Iterate over grid levels
for (int lvl = 0; lvl <= m_amr->getFinestLevel(); lvl++){

   // Get the grid on this level.
   const DisjointBoxLayout& dbl = m_amr->getGrids(myParticleContainer.getRealm())[lvl];

   // Get the distributed particles on this level
   ParticleData<P>& levelParticles = myParticleContainer[lvl]

   // Iterate over grid patches on this level
   for (DataIterator dit(dbl); dit.ok(); ++dit){

      // Get the particles in the current patch.
      List<P>& patchParticles = levelParticles[dit()].listItems();

      // Iterate over the particles in the current patch.
      for (ListIterator<P> lit(patchParticles); lit.ok(); ++lit){
```

(continues on next page)

(continued from previous page)

```
            P& p = lit();

            // Move the particle
            p.position() = ...
        }
    }
}

// Remap particles onto new patches and ranks (they may have moved off their original ones)
myParticleContainer.remap();
```

2.4.4 Sorting particles

Sorting by cell

The particles can also be sorted by cell by calling `void ParticleContainer<P>::sortParticleByCell()`, like so:

```
ParticleContainer<P> myParticleContainer;

myParticleContainer.sortParticlesByCell();
```

Internally in `ParticleContainer<P>`, this will place the particles in another container which can be iterated over on a per-cell basis. This is different from `List<P>` and `ListBox<P>` above, which contained particles stored on a per-patch basis with no internal ordering of the particles.

The per-cell particle container is a `Vector<RefCountedPtr<LayoutData<BinFab<P> > > >` type where again the `Vector` holds the particles on each AMR level and the `LayoutData<BinFab>` holds one `BinFab` on each grid patch. The `BinFab` is also a template, and it holds a `List<P>` in each grid cell. Thus, this data structure stores the particles per cell rather than per patch. Due to the horrific template depth, this container is typedef'ed as `AMRCellParticles<P>`.

To get cell-sorted particles one can call

```
AMRCellParticles<P>& cellSortedParticles = myParticleContainer.getCellParticles();
```

Iteration over cell-sorted particles is mostly the same as for patch-sorted particles, except that we also need to explicitly iterate over the grid cells in each grid patch:

```
const int comp = 0;

// Iterate over all AMR levels
for (int lvl = 0; lvl <= m_amr->getFinestLevel(); lvl++){

   // Get the grids on this level
   const DisjointBoxLayout& dbl = m_amr->getGrids(myParticleContainer.getRealm())[lvl];

   // Iterate over grid patches on this level
   for (DataIterator dit(dbl); dit.ok(); ++dit){

      // Get the Cartesian box for the current grid aptch
      const Box cellBox = dbl[dit()];

      // Get the particles in the current grid patch.
      BinFab<P>& cellSortedBoxParticles = (*cellSortedParticles[lvl])[dit()];

      // Iterate over all cells in the current box
      for (BoxIterator bit(cellBox); bit.ok(); ++bit){
         const IntVect iv = bit();

         // Get the particles in the current grid cell.
         List<P>& cellParticles = cellSortedBoxParticles(iv, comp);

         // Do something with cellParticles
         for (ListIterator<P> lit(cellParticles); lit.ok(); ++lit){
```

(continues on next page)

(continued from previous page)

```
            P& p = lit();
          }
        }
    }
}
```

Sorting by patch

If the particles need to return to patch-sorted particles:

```
ParticleContainer<P> myParticleContainer;

myParticleContainer.sortParticlesByPatch();
```

Important: If particles are sorted by cell, calling `ParticleContainer<P>` member functions that fetch particles by patch will issue an error. This is done by design since the patch-sorted particles have been moved to a different container. Note that remapping particles also requires that the particles are patch-sorted. Calling `remap()` with cell-sorted particles will issue a run-time error.

2.4.5 Allocating particles

`AmrMesh` has a very simple function for allocating a `ParticleContainer<P>`:

```
template <typename P>
void allocate(ParticleContainer<P>& a_container, const int a_pvrBuffer, const std::string a_realm);
```

which will allocate a `ParticleContainer` on realm `a_realm` with a buffer zone of `a_pvrBuffer`. This buffer zone adjusts if particles on the fine side of a refinement boundary map to the coarse grid or the fine grid (see *Mapping and remapping*).

2.4.6 Mapping and remapping

Mapping particles with ParticleValidRegion

The `ParticleValidRegion` (PVR) allows particles to be transferred to coarser grid levels if they are within a specified number of grid cells from the refinement boundary. There are two reasons why such a functionality is useful:

1. Particles that live in the first strip of cells on the fine side of a refinement boundary have deposition clouds that hang over the boundary and into ghost cells. This mass must be added to the coarse level.
2. Deposition and interpolation kernels can be entirely contained within a grid level. It might be useful to keep the kernel on a specific AMR level for a certain number of time step.

The PVR is automatically allocated through the particle constructor by specifying the `a_pvrBuffer` flag. If you do not want to use PVR functionality, simply set `a_pvrBuffer = 0` for your `ParticleContainer<P>`. In this case the particles will live on the grid patch that contains them.

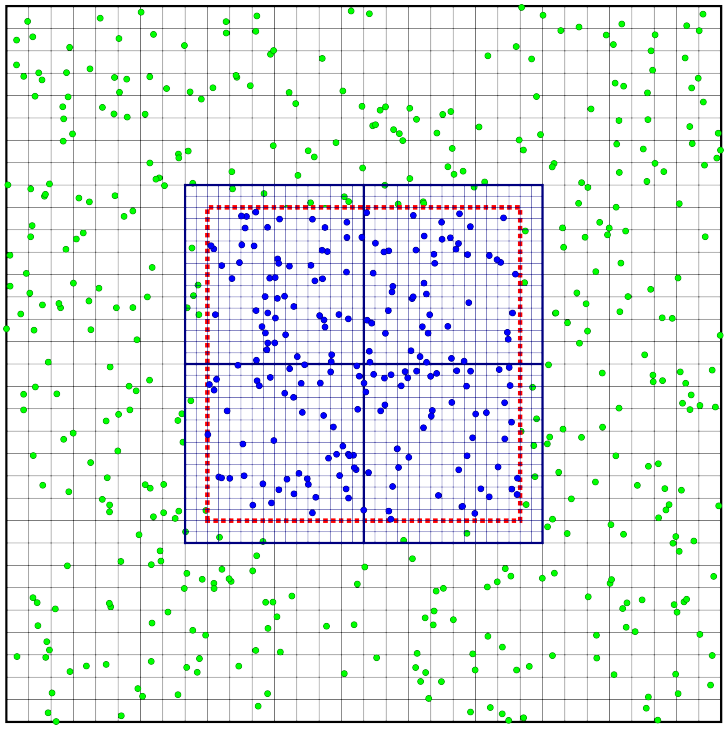

Fig. 2.4.1: The `ParticleValidRegion` allows particles whose position fall into a fine grid patch to be moved to a coarser level if they are within a specified distance from the refinement boundary. In this case, the green particles that overlap with the fine-level grid are remapped to the coarse level.

Remapping particles

Particles that move off their original grid patch must be remapped in order to ensure that they are assigned to the correct grid. The remapping function for `ParticleContainer<P>` is `void ParticleContainer<P>::remap()`, which is simply used as follows:

```
ParticleContainer<P> myParticles;

myParticles.remap();
```

Note that if a PVR region is set, the particle container remapping will respect it.

2.4.7 Regridding

`ParticleContainer<P>` is comparatively simple to regrid, and this is done in two steps:

1. Each MPI rank collects *all* particles on a single `List<P>` by calling `void ParticleContainer<P>::preRegrid(int a_base)`. This will pull the particles off their current grids and collect them in a single list (on a per-rank basis).
2. When `ParticleContainer<P>` regrids, each rank adds his `List<P>` back into the internal particle containers.

The use case typically looks like this:

```
ParticleContainer<P> myParticleContainer;

// Each rank caches his particles
```

(continues on next page)

(continued from previous page)

```
const int baseLevel = 0;
myParticleContainer.preRegrid(0);

// Driver does a regrid.
.
.
.

// After the regrid we fetch grids from AmrMesh:
Vector<DisjointBoxLayout> grids;
Vector<ProblemDomain> domains;
Vector<Real> dx;
Vector<int> refinement_ratios;
int base;
int newFinestLevel;

myParticleContainer.regrid(grids, domains, dx, refinement_ratios, baseLevel, newFinestLevel);
```

Here, `baseLevel` is the finest level that didn't change and `newFinestLevel` is the finest AMR level after the regrid.

2.4.8 Masked particles

`ParticleContainer<P>` also supports the concept of *masked particles*, where one can fetch a subset of particles that live only in specified regions in space. Typically, this "specified region" is the refinement boundary, but the functionality is generic and might prove useful also in other cases.

When *masked particles* are used, the user can provide a boolean mask over the AMR hierarchy and obtain the subset of particles that live in regions where the mask evaluates to true. This functionality is for example used for some of the particle deposition methods in `chombo-discharge` where we deposit particles that live near the refinement boundary with special deposition functions.

To fill the masked particles, ``ParticleContainer<P>` has members functions for copying the particles into internal data containers which the user can later fetch. The function signatures for these are

```
using AmrMask = Vector<RefCountedPtr<LevelData<BaseFab<bool> > > >;

template <class P>
void copyMaskParticles(const AmrMask& a_mask) const;

template <class P>
void copyNonMaskParticles(const AmrMask& a_mask) const;
```

The argument `a_mask` holds a bool at each cell in the AMR hierarchy. Particles that live in cells where `a_mask` is true will be copied to an internal data holder in `ParticleContainer<P>` which can be retried through a call

```
AMRParticles<P>& maskParticles = myParticleContainer.getMaskParticles();
```

Note that `copyNonMaskParticles` is just like `copyMaskParticles` except that the bools in `a_mask` have been flipped.

Note that the mask particles are *copied*, and the original particles are left untouched. After the user is done with the particles, they should be deleted through the functions `void clearMaskParticles()` and `void clearNonMaskParticles`, like so:

```
AmrMask myMask;
ParticleContainer<P> myParticles;

// Copy mask particles
myParticles.copyMaskParticles(myMask);

// Do something with the mask particles
AMRParticles<P>& maskParticles = myParticleContainer.getMaskParticles();

// Release the mask particles
myParticles.clearMaskParticles();
```

Registering particle masks

`AmrMesh` can register a *halo* mask with a specified width:

```
void registerMask(const std::string a_mask, const int a_buffer, const std::string a_realm);
```

where `a_mask` is `"s_particle_halo"`.

This will register a mask which is false everywhere except in coarse-grid cells that are within a distance a_buffer from the refinement boundary.

2.4.9 Embedded boundaries

`ParticleContainer<P>` is EB-agnostic and has no information about the embedded boundary. This means that particles remap just as if the EB was not there. Interaction with the EB is done via the implicit function, and modifications in the interpolation and deposition steps.

Signed distance function

When signed distance functions are used, one can always query how far a particle is from a boundary:

```
List<P>& particles;
BaseIF distanceFunction;

for (ListIterator<P> lit(particles); lit.ok(); ++lit){
   const P& p          = lit();
   const RealVect& pos = p.position();

   const Real distanceToBoundary = distanceFunction.value(pos);
}
```

If the particle is inside the EB then the signed distance function will be positive and the particle can be removed from the simulation. The distance function can also be used to detect collisions between particles and the EB.

2.4.10 Deposition and interpolation

Particle depositon

To deposit particles on the mesh, the user can call the templated function `AmrMesh::depositParticles` which has a signature

```
template <class P, const Real&(P::*particleScalarField)() const>
void depositParticles(EBAMRCellData&                a_meshData,
                      const std::string&            a_realm,
                      const phase::which_phase&     a_phase,
                      const ParticleContainer<P>& a_particles,
                      const DepositionType          a_depositionType,
                      const CoarseFineDeposition    a_coarseFineDeposition,
                      const bool                    a_forceIrregNGP);
```

Here, the template parameter `P` is the particle type and the template parameter `particleScalarField` is a C++ pointer-to-member-function. This function must have the indicated signature `const Real& P::particleScalarField() const` *or* the signature `Real P::particleScalarField() const`. The pointer-to-member `particleScalarField` indicates the variable to be deposited on the mesh. This function pointer does not need to return a member in the particle class - derived variables are also fine. For example, if the particle type `P` needs to deposit a computational mass on the mesh, the particle class will at least contain the following member functions:

```
class P : public BinItem {
public:

   const Real& mass() const {
      return m_mass;
   }

   Real mass2() const {
      return m_mass*m_mass.
   }

protected:

   Real m_mass;
};
```

Here, we have included an extra member function `mass2()` which returns the squared mass. Note that the function does not return a member variable but an r-value. When depositing the mass on the mesh the user will e.g. call

```
RefCountedPtr<AmrMesh> amr;

amr->depositParticles<P, &P::mass >(...).
amr->depositParticles<P,  P::mass2>(...).
```

Next, the input arguments to `depositParticles` are the output mesh data holder (must have exactly one component), the realm and phase where the particles live, and the particles themselves (`a_particles`). The enum `DepositionType` and input argument `a_depositionType` indicates the deposition method. Valid arguments are

- `DepositionType::NGP` (Nearest grid-point).
- `DepositionType::CIC` (Cloud-In-Cell).
- `DepositionType::TSC` (Triangle-Shaped Cloud).
- `DepositionType::W4` (Fourth order weighted).

The input argument `a_coarseFineDeposition` determines how coarse-fine deposition is handled. Valid input arguments are

- `CoarseFineDeposition::PVR` This uses a standard PVR formulation. When the particles near the refinement boundary deposit on the mesh, some of the mass from the coarse-side particles will end up underneath the fine grid. This mass is interpolated to the fine grid using piecewise constant interpolation. If the fine-level particles also have particle clouds that hang over the refinement boundary, the hanging mass will be added to the coarse level.
- `CoarseFineDeposition::Halo` This uses a what we call *halo* particles. Instead of interpolating the mass from the invalid coarse region onto the fine level, the particles near the refinement boundary (i.e., the *halo* particles) deposit directly into the fine level but with 2x or 4x the particle width. So, if a coarse-level particle lives right next to the fine grid and the refinement factor between the grids is r, it will deposit both into the fine grid with r times the particle width compared to the coarse grid. Again, if the fine-level particles also have particle clouds that hang over the refinement boundary, the hanging mass will be added to the coarse level.

Finally, the flag `a_forceIrregNGP` permits the user to enforce nearest grid-point deposition in cut-cells. This option is motivated by the fact that some applications might require hard mass conservation, and the user can ensure that mass is not deposited into covered regions.

Particle interpolation

To interpolate a field onto a particle position, the user can call the `AmrMesh` member functions

```
template <class P, Real&(P::*particleScalarField)()>
void interpolateParticles(ParticleContainer<P>&       a_particles,
                          const std::string&          a_realm,
                          const phase::which_phase&   a_phase,
                          const EBAMRCellData&        a_meshScalarField,
                          const DepositionType        a_interpType,
                          const bool                  a_forceIrregNGP) const;

template <class P, RealVect&(P::*particleVectorField)()>
void interpolateParticles(ParticleContainer<P>&       a_particles,
                          const std::string&          a_realm,
                          const phase::which_phase&   a_phase,
                          const EBAMRCellData&        a_meshVectorField,
                          const DepositionType        a_interpType,
                          const bool                  a_forceIrregNGP) const;
```

The function signature for particle interpolation is pretty much the same as for particle deposition, with the exception of the interpolated field. The template parameter `P` still indicates the particle type, but the user can interpolate onto either a scalar particle variable or a vector variable. For example, in order to interpolate the particle acceleration, the particle class (let's call it `MyParticleClass`) will typically have a member function `RealVect& acceleration()`, and in this case one can interpolate the acceleration by

```
RefCountedPtr<AmrMesh> amr;

amr->interpolateParticles<MyParticleClass, &MyParticleClass::acceleration>(...)
```

Note that if the user interpolates onto a scalar variable, the mesh variable must have exactly one component. Likewise, if interpolating a vector variable, the mesh variable must have exact `SpaceDim` components.

2.5 Realm

`Realm` is a class for centralizing EBAMR-related grids and operators for a specific AMR grid. For example, a `Realm` consists of a set of grids (i.e. a `Vector<DisjointBoxLayout>`) as well as *operators*, e.g. functionality for filling ghost cells or averaging down a solution from a fine level to a coarse level. One may think of a `Realm` as a fully-fledged AMR hierarchy with associated multilevel operators, i.e. how one would usually do AMR.

2.5.1 Dual grid

The reason why `Realm` exists at all is due to individual load balancing of algorithmic components. The terminology *dual grid* is used when more than one `Realm` is used in a simulation, and in this case the user/developer has chosen to solve the equations of motion over a different set of `DisjointBoxLayout` on each level. This approach is very useful when using computational particles since users can quickly generate separate Eulerian sets of grids for fluids and particles, and the grids can then be load balanced separately. Note that every `Realm` consists of the same boxes, i.e. the physical domain and computational grids are the same for all realms. The difference lies primarily in the assignment of MPI ranks to grids; i.e. the load-balancing and domain decomposition.

Fig. 2.5.1 shows an example of a dual-grid approach. In this figure we have a set of grid patches on a particular grid level. In the top panel the grid patches are load-balanced using the grid patch volume as a proxy for the computational load. The numbers in each grid patch indicates the MPI rank ownership of the patches. In the bottom panel we have introduced computational particles in some of the patches. For particles, the computational load is better defined by the number of computational particles assigned to the patch, and so using the number of particles as a proxy for the load yields different rank ownership over the grid patches.

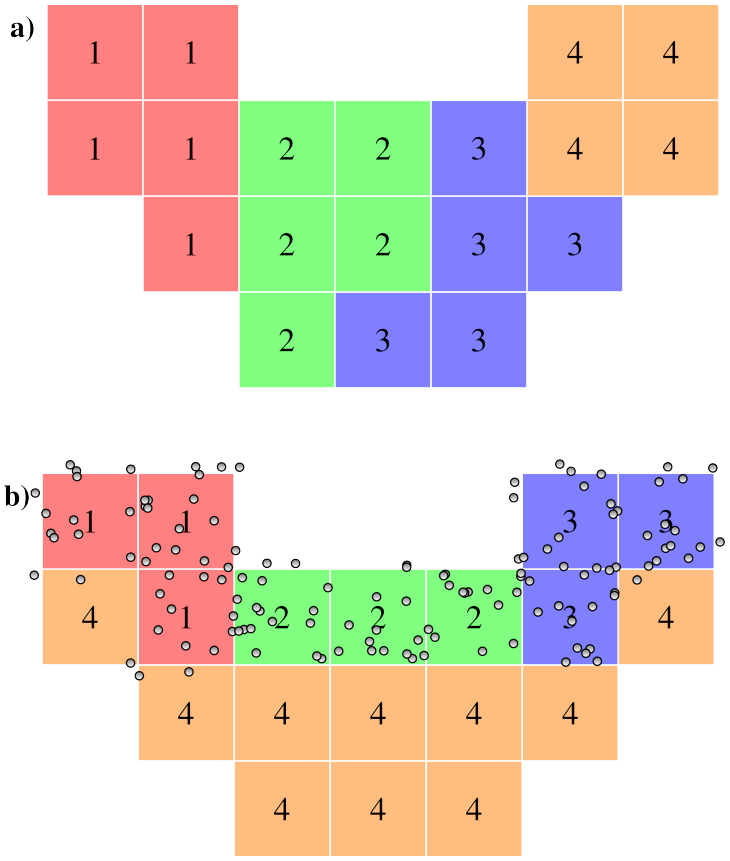

Fig. 2.5.1: Sketch of dual grid approach. Each rectangle represents a grid patch. a) Load balancing with the number of grid cells. b) Load balancing with the number of particles.

2.5.2 Realm registration

To register a `Realm`, users will have `TimeStepper` allocate the desired number of realms in the pure routine `registerRealms()`, as follows:

```
void myTimeStepper::registerRealms(){
   m_amr->registerRealm(Realm::Primal);
   m_amr->registerRealm("particleRealm");
   m_amr->registerRealm("otherParticleRealm");
}
```

Since at least one realm is required, `Driver` will *always* register the realm `"Primal"`.

During regrid, all realms are initially load balanced with the grid patch volume as the load proxy. However, users can change load balancing individually for each realm through the load balancing routines in *TimeStepper*.

2.5.3 Operator registration

Internally, an instantiation of `Realm` contains the grids and the geometric information (e.g. `EBISLayout`), as well as any operators that the user has seen fit to *register*. Various operators are available for e.g. gradient stencils, conservative coarsening, ghost cell interpolation, filling a patch with interpolation data, redistribution, and so on. Since operators always incur overhead and not all applications require *all* operators, they must be *registered*. If a solver needs an operator for, say, piecewise linear ghost cell interpolation, the solver needs to *register* that operator through the `AmrMesh` public interface:

```
m_amr->registerOperator(s_eb_pwl_interp,   m_realm, m_phase);
```

Once an operator has been registered, `Realm` will define those operators during initialization e.g. regrids. Run-time aborts with error messages are issued if an AMR operator is used, but has not been registered.

More commonly, `chombo-discharge` solvers will contain a routine that registers the operators that the solver needs. A valid `TimeStepper` implementation *must* register all required operators in the function `registerOperators()`, which is mostly as simple as:

```
FieldSolver myPoissonSolver;
CdrSolver myCdrSolver;

void myTimeStepper::registerOperators(){
   myPoissonSolver->registerOperators();
   myCdrSolver->registerOperators();
}
```

This will register the operators needed for `FieldSolver` and `CdrSolver` solver classes. Note that if the solvers register the same operators, these operators are only defined once in *AmrMesh*.

2.5.4 Available operators

The current operators are currently available:

1. Gradient `s_eb_gradient`.
2. Irregular cell centroid interpolation, `s_eb_irreg_interp`.
3. Coarse grid conservative coarsening, `s_eb_coar_ave`.
4. Piecewise linear interpolation (with slope limiters), `s_eb_fill_patch`.
5. Linear ghost cell interpolation, `s_eb_pwl_interp`.
6. Flux registers, `s_eb_flux_reg`.
7. Redistribution registers, `s_eb_redist`.
8. Non-conservative divergence stencils, `s_eb_noncons_div`.
9. Multigrid interpolators, `s_eb_multigrid` (used for multigrid).
10. Signed distance function defined on grid, `s_levelset`.
11. Particle-mesh support, `s_eb_particle_mesh`.

Solvers will typically allocate a subset of these operators, but for multiphysics code that use both fluid and particles, most of these will be in use.

2.5.5 Interacting with realms

Users will not interact with `Realm` directly. Every `Realm` is owned by `AmrMesh`, and the user will only interact with realms through the public `AmrMesh` interface, for example by fetching operators for performing AMR operations. In addition, data that is defined on one realm can be copied to another; `EBAMRData<T>` takes care of this. You will simply call a copier:

```
EBAMRCellData realmOneData;
EBAMRCellData realmTwoData;

realmOneData.copy(realmTwoData);
```

The rest of the functionality uses the public interface of *AmrMesh*. For example for coarsening of multifluid data:

```
std::string multifluidRealm;
MFAMRCellData multifluidData;
AmrMesh amrMesh;

amrMesh.averageDown(multifluidData, multifluidRealm);
```

2.6 Linear solvers

2.6.1 Helmholtz equation

The Helmholtz equation is represented by

$$\alpha a\left(\mathbf{x}\right)\Phi + \beta\nabla\cdot\left[b\left(\mathbf{x}\right)\nabla\Phi\right] = \rho$$

where α and β are constants and $a\left(\mathbf{x}\right)$ and $b\left(\mathbf{x}\right)$ are spatially dependent (and only piecewise smooth).

To solve the Helmholtz equation we solve it in the form

$$\kappa L\Phi = \kappa\rho,$$

where L is the Helmholtz operator above. The preconditioning by the volume fraction κ is done in order to avoid the small-cell problem encountered in finite-volume discretizations on EB grids.

Discretization and fluxes

The Helmholtz equation is solved by assuming that Φ lies on the cell-center. The $b\left(\mathbf{x}\right)$-coefficient lies on face centers and EB faces. In the general case the cell center might lie inside the embedded boundary, and the cell-centered discretization relies on the concept of an extended state. Thus, Φ does not satisfy a discrete maximum principle.

The finite volume update require fluxes on the face centroids rather than the centers. These are constructed by first computing the fluxes to second order on the face centers, and then interpolating them to the face centroids. For example, the flux F_3 in the figure above is

$$F_3 = \beta b_{i,j+1/2}\frac{\Phi_{i,j+1} - \Phi_{i,j}}{\Delta x}.$$

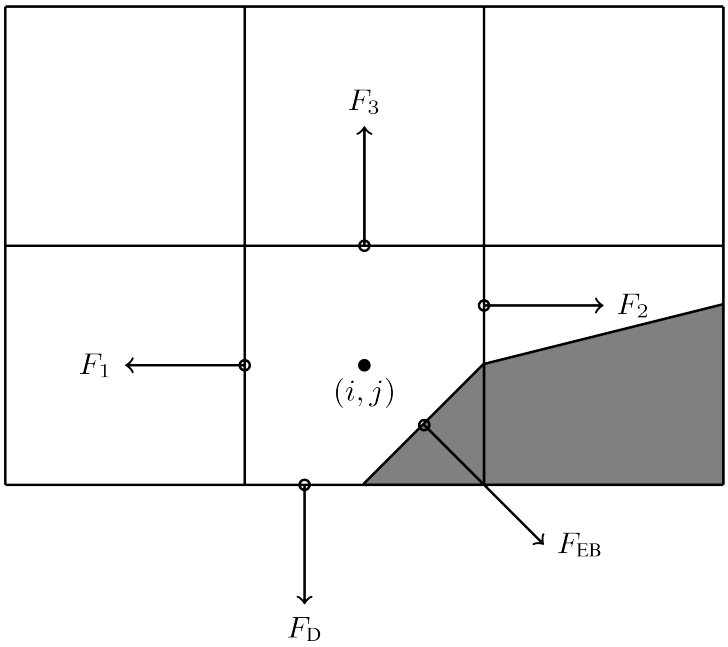

Fig. 2.6.1: Location of fluxes for finite volume discretization.

Boundary conditions

The finite volume discretization of the Helmholtz equation require fluxes through the EB and domain faces. Below, we discuss how these are implemented.

Note: `chombo-discharge` supports spatially dependent boundary conditions

Neumann

Neumann boundary conditions are straightforward since the flux through the EB or domain faces are specified directly.

From the above figure, the fluxes F_{EB} and F_{D} are specified.

Dirichlet

Dirichlet boundary conditions are more involved since only the value at the boundary is prescribed, but the finite volume discretization requires a flux. On the domain boundaries the fluxes are face-centered and we therefore use finite differencing for obtaining a second order accurate approximation to the flux at the boundary.

On the embedded boundaries the flux is more complicated to compute, and requires us to compute an approximation to the normal gradient $\partial_n \Phi$ at the boundary. Our approach is to approximate this flux by expanding the solution as a polynomial around a specified number of grid cells. By using more grid cells than there are unknown in the Taylor series, we formulate an over-determined system of equations up to some specified order. As a first approximation we include only those cells in the quadrant or half-space defined by the normal vector, see Fig. 2.6.2. If we can not find enough equations, the strategy is to 1) drop order and 2) include all cells around the cut-cell.

Once the cells used for the gradient reconstruction have been obtained, we use weighted least squares to compute the approximation to the derivative to specified order (for details, see *Least squares*). The result of the least squares

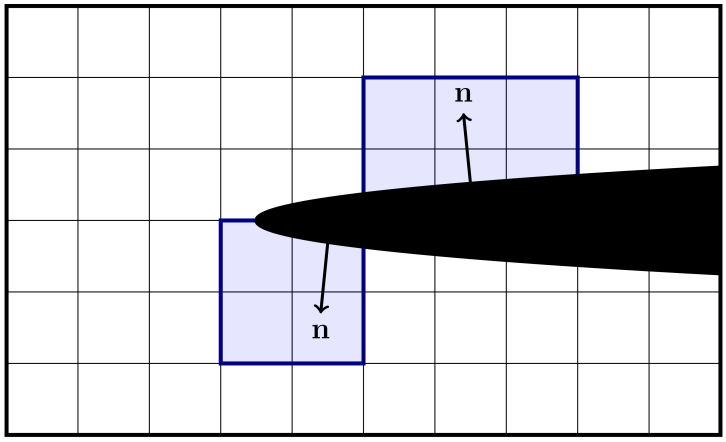

Fig. 2.6.2: Examples of neighborhoods (quadrant and half-space) used for gradient reconstruction on the EB.

computation is represented as a stencil:

$$\frac{\partial \Phi}{\partial n} = w_{\mathrm{B}} \Phi_{\mathrm{B}} + \sum_{\mathbf{i}} w_{\mathbf{i}} \Phi_{\mathbf{i}},$$

where Φ_{B} is the value on the boundary, the w are weights for grid points $\mathbf{i}$, and the sum runs over cells in the domain.

Note that the gradient reconstruction can end up requiring more than one ghost cell layer near the embedded boundaries. For example, Fig. 2.6.3 shows a typical stencil region which is built when using second order gradient reconstruction on the EB. In this case the gradient reconstruction requires a stencil with a radius of 2, but as the cut-cell lies on the refinement boundary the stencil reaches into two layers of ghost cells. For the same reason, gradient reconstruction near the cut-cells might require interpolation of corner ghost cells on refinement boundaries.

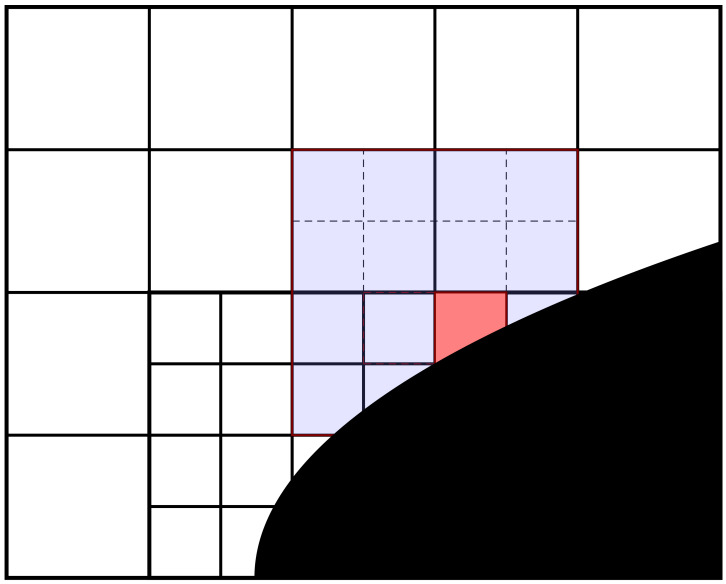

Fig. 2.6.3: Example of the region of a second order stencil for the Laplacian operator with second order gradient reconstruction on the embedded boundary.

Robin

Robin boundary conditions are in the form

$$A\partial_n\Phi + B\Phi = C,$$

where A, B, and C are constants. This boundary conditions is enforced through the flux

$$\partial_n\Phi = \frac{1}{A}\left(C - B\Phi\right),$$

which requires an evaluation of Φ on the domain boundaries and the EB.

For domain boundaries we extrapolate the cell-centered solution to the domain edge, using standard first order finite differencing.

On the embedded boundary, we approximate $\Phi\left(\mathbf{x}_{\mathrm{EB}}\right)$ by linearly interpolating the solution with a least squares fit, using cells which can be reached with a monotone path of radius one around the EB face (see *Least squares* for details). The Robin boundary condition takes the form

$$\partial_n\Phi = \frac{C}{A} - \frac{B}{A}\sum_{\mathbf{i}} w_{\mathbf{i}}\Phi_{\mathbf{i}}.$$

Currently, we include the data in the cut-cell itself in the interpolation, and thus also use unweighted least squares.

Ghost cell interpolation

With AMR, multigrid requires ghost cells on the refinement boundary. The interior stencils for the Helmholtz operator are first order and thus only require a single level of ghost cells (and no corner ghost cells). These ghost cells are filled using a finite-difference stencil, see Fig. 2.6.4.

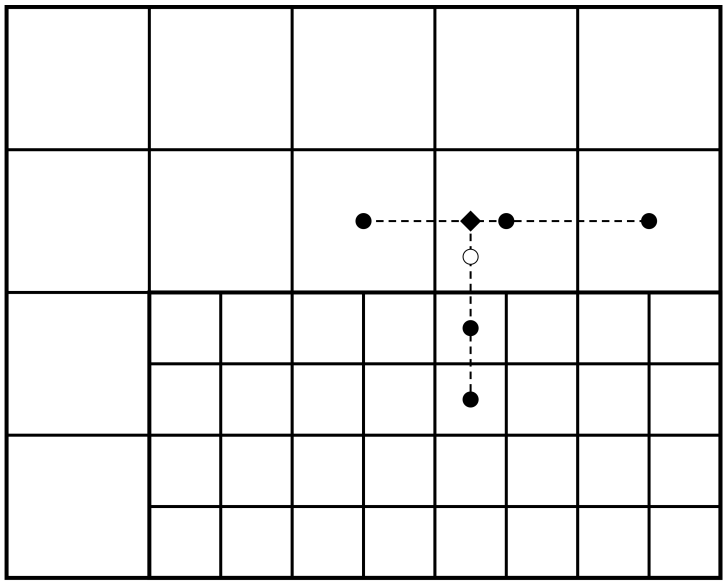

Fig. 2.6.4: Standard finite-difference stencil for ghost cell interpolation (open circle). We first interpolate the coarse-grid cells to the centerline (diamond). The coarse-grid interpolation is then used together with the fine-grid cells (filled circles) for interpolation to the ghost cell (open circle).

Embedded boundaries introduce many pathologies for multigrid:

1. Cut-cell stencils may have a large radius (see Fig. 2.6.3) and thus require more ghost cell layers.
2. The EBs cut the grid in arbitrary ways, leading to multiple pathologies regarding cell availability.

The pathologies mean that standard finite differencing fails near the EB, mandating a more general approach. Our way of handling ghost cell interpolation near EBs is to reconstruct the solution (to specified order) in the ghost cells, using the available cells around the ghost cell (see *Least squares* for details). As per conventional wisdom regarding multigrid interpolation, this reconstruction does *not* use coarse-level grid cells that are covered by the fine level.

Figure Fig. 2.6.5 shows a typical interpolation stencil for the stencil in Fig. 2.6.3. Here, the open circle indicates the ghost cell to be interpolated, and we interpolate the solution in this cell using neighboring grid cells (closed circles). For this particular case there are 10 nearby grid cells available, which is sufficient for second order interpolation (which requirse at least 6 cells in 2D).

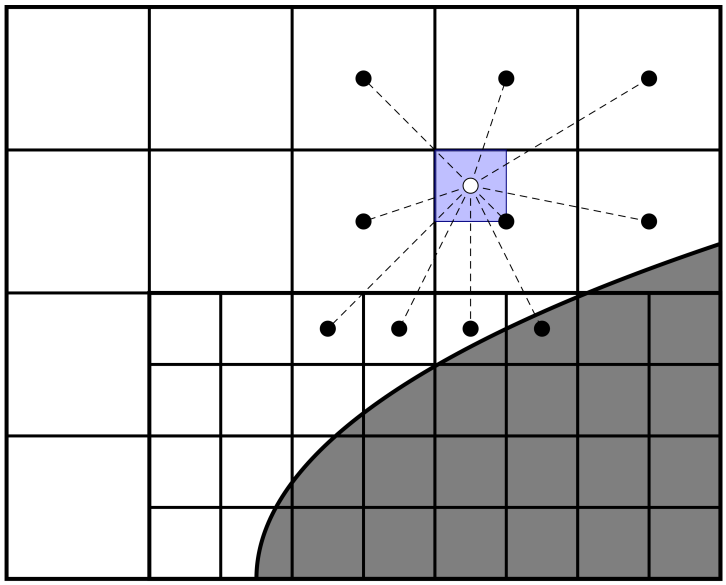

Fig. 2.6.5: Multigrid interpolation for refinement boundaries away from and close to an embedded boundary.

Note: `chombo-discharge` implements a fairly general ghost cell interpolation scheme near the EB. The ghost cell values can be reconstructed to specified order (and with specified least squares weights).

Relaxation methods

The Helmholtz equation is solved using multigrid, with various smoothers available on each grid level. The currently supported smoothers are:

1. Standard point Jacobi relaxation.
2. Red-black Gauss-Seidel relaxation in which the relaxation pattern follows that of a checkerboard.
3. Multi-colored Gauss-Seidel relaxation in which the relaxation pattern follows quadrants in 2D and octants in 3D.

Users can select between the various smoothers in solvers that use multigrid.

Note: Multi-colored Gauss-Seidel usually provide the best convergence rates. However, the multi-colored kernels are

twice as expensive as red-black Gauss-Seidel relaxation in 2D, and four times as expensive in 3D.

2.6.2 Multiphase Helmholtz equation

`chombo-discharge` also supports a *multiphase version* where data exists on both sides of the embedded boundary. The most common case is that involving discontinuous coefficients, e.g. for

$$\nabla \cdot \left[b\left(\mathbf{x}\right) \nabla \Phi \left(\mathbf{x}\right) \right] = 0.$$

where $b\left(\mathbf{x}\right)$ is only piecewise constant.

Jump conditions

For the case of discontinous coefficients there is a jump condition on the interface between two materials:

$$b_1 \partial_{n_1} \Phi + b_2 \partial_{n_2} \Phi = \sigma,$$

where b_1 and b_2 are the Helmholtz equation coefficients on each side of the interface, and $n_1 = -n_2$ are the normal vectors pointing away from the interface in each phase. σ is a jump factor.

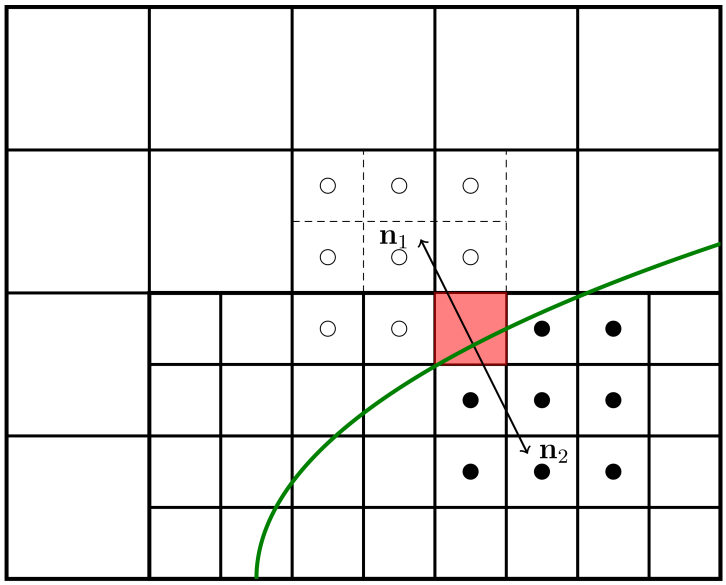

Fig. 2.6.6: Example of cells and stencils that are involved in discretizing the jump condition. Open and filled circles indicate cells in separate phases.

Discretization

To incorporate the jump condition in the Helmholtz discretization, we use a gradient reconstruction to obtain a solution to Φ on the boundary, and use this value to impose a Dirichlet boundary condition during multigrid relaxation. Recalling the gradient reconstruction $\frac{\partial \Phi}{\partial n} = w_{\mathrm{B}} \Phi_{\mathrm{B}} + \sum_{\mathbf{i}} w_{\mathbf{i}} \Phi_{\mathbf{i}}$, the matching condition (see Fig. 2.6.6) can be written as

$$b_1 \left[w_{\mathrm{B},1} \Phi_{\mathrm{B}} + \sum_{\mathbf{i}} w_{\mathbf{i},1} \Phi_{\mathbf{i},1} \right] + b_2 \left[w_{\mathrm{B},2} \Phi_{\mathrm{B}} + \sum_{\mathbf{i}} w_{\mathbf{i},2} \Phi_{\mathbf{i},2} \right] = \sigma.$$

This equation can be solved for the boundary value Φ_B, which can then be used to compute the finite-volume fluxes into the cut-cells.

Note: For discontinuous coefficients the gradient reconstruction on one side of the EB does not reach into the other (since the solution is not differentiable across the EB).

2.6.3 AMRMultiGrid

`AMRMultiGrid` is the `Chombo` implementation of the Martin-Cartwright multigrid algorithm. It takes an "operator factory" as an argument, and the factory can generate objects (i.e., operators) that encapsulate the discretization on each AMR level.

`chombo-discharge` runs its own operator, and the user can use either of:

1. `EBHelmholtzOpFactory` for single-phase problems.
2. `MFHelmholtzOpFactory` for multi-phase problems.

The source code for these are located in `$DISCHARGE_HOME/Source/Elliptic`.

Bottom solvers

Chombo provides (at least) three bottom solvers which can be used with `AMRMultiGrid`.

1. A regular smoother (e.g., point Jacobi).
2. A biconjugate gradient stabilized method (BiCGStab)
3. A generalized minimal residual method (GMRES).

The user can select between these for the various solvers that use multigrid.

CHAPTER

THREE

PHYSICS MODELS

3.1 Implemented models

Various models that use the `chombo-discharge` solvers and functionality have been implemented and are shipped with `chombo-discharge`. These models implement `TimeStepper` for advancing various types of equations, and have been set up both for unit testing and as building blocks for more complex applications. The models reside `$DICHARGE_HOME/Physics` and are supplemented with Python tools for quickly setting up new applications that use the same code. In general, users are encouraged to modify or copy these models and tailor them for their own applications.

The following models are currently supported:

1. *Advection diffusion* Used for solving pure advection-diffusion problems in 2D/3D with complex geometries.
2. *Brownian walker* Used for solving advection-diffusion problems using computational particles. This is, essentially, a Particle-In-Cell code that uses classical particles (rather than kinetic ones).
3. *CDR plasma* Implements `TimeStepper` for solving low-temperature discharge plasma problems using fluids. Used e.g. for streamer simulations.
4. *Electrostatics* Implements `TimeStepper` for setting up a static calculation that solves the Poisson equation in 2D/3D.
5. *Geometry* Implement `TimeStepper` with empty functionality. Often used when setting up new geometries/cases.
6. *Radiative transfer* Implements `TimeStepper` for solving radiative transfer problems. Used e.g. for regression testing.

3.2 Advection diffusion

3.3 Brownian walker

3.4 CDR plasma

The CDR plasma model resides in `/Physics/CdrPlasma` and describes plasmas in the drift-diffusion approximation. This physics model also includes the following subfolders:

- `/Physics/CdrPlasma/PlasmaModel` which contains various implementation of some plasma models that we have used.