chombo-discharge Documentation

Robert Marskar

May 28, 2022

INTRODUCTION

1 Introduction **3**

1.1 Using this documentation . . . 3
1.1.1 Doxygen documentation . . . 3
1.2 Overview . . . 3
1.2.1 History . . . 3
1.2.2 Key functionalities . . . 3
1.2.3 Design approach . . . 4
1.3 Installation . . . 5
1.3.1 Obtaining `chombo-discharge` . . . 5
1.3.2 Organization . . . 6
1.3.3 Cloning `chombo-discharge` . . . 6
1.3.4 Setting up the environment . . . 6
1.3.5 Test build . . . 7
1.3.6 Advanced configuration . . . 7
1.3.7 Running an example application . . . 10
1.3.8 Troubleshooting . . . 10
1.4 Visualization . . . 11
1.5 Controlling `chombo-discharge` . . . 11
1.5.1 Compiling and running . . . 11
1.5.2 Running applications . . . 12
1.5.3 Simulation inputs . . . 12
1.5.4 Simulation outputs . . . 13
1.5.5 Controlling parallel processor verbosity . . . 13
1.5.6 Restarting simulations . . . 14
1.5.7 Run-time configurations . . . 14

2 Discretization **15**

2.1 Spatial discretization . . . 15
2.1.1 Cartesian AMR . . . 15
2.1.2 Embedded boundaries . . . 16
2.1.3 Geometry representation . . . 17
2.1.4 Geometry generation . . . 19
2.1.5 Mesh generation . . . 20
2.1.6 Cell refinement philosophy . . . 21
2.2 `Chombo-3` basics . . . 22
2.2.1 Real . . . 22
2.2.2 RealVect . . . 22
2.2.3 IntVect . . . 22
2.2.4 Box . . . 22
2.2.5 EBCellFAB and FArrayBox . . . 22

2.2.6 Vector . . . 23
2.2.7 RefCountedPtr . . . 23
2.2.8 DisjointBoxLayout . . . 23
2.2.9 LevelData . . . 24
2.2.10 EBISLayout and EBISBox . . . 24
2.2.11 BaseIF . . . 25
2.3 Mesh data . . . 25
2.3.1 Allocating mesh data . . . 26
2.3.2 Iterating over patches . . . 26
2.3.3 Iterating over cells . . . 27
2.3.4 Coarsening data . . . 27
2.3.5 Filling ghost cells . . . 28
2.3.6 Computing gradients . . . 28
2.3.7 Copying data . . . 29
2.3.8 DataOps . . . 30
2.4 Particles . . . 30
2.4.1 ParticleContainer . . . 30
2.4.2 Data structures . . . 30
2.4.3 Basic use . . . 31
2.4.4 Sorting particles . . . 32
2.4.5 Allocating particles . . . 33
2.4.6 Mapping and remapping . . . 34
2.4.7 Regridding . . . 35
2.4.8 Masked particles . . . 35
2.4.9 Embedded boundaries . . . 36
2.4.10 Particle depositon . . . 37
2.4.11 Particle interpolation . . . 39
2.5 Realm . . . 40
2.5.1 Dual grid . . . 40
2.5.2 Realm registration . . . 41
2.5.3 Operator registration . . . 41
2.5.4 Interacting with realms . . . 42
2.6 Linear solvers . . . 42
2.6.1 Helmholtz equation . . . 42
2.6.2 Multiphase Helmholtz equation . . . 47
2.6.3 AMRMultiGrid . . . 48

3 Design **49**
3.1 Driver . . . 49
3.1.1 Simulation setup . . . 49
3.1.2 Simulation advancement . . . 50
3.1.3 Regridding . . . 50
3.1.4 Class options . . . 51
3.1.5 Runtime options . . . 52
3.2 ComputationalGeometry . . . 53
3.2.1 Electrode . . . 53
3.2.2 Dielectric . . . 53
3.2.3 Retrieving distance functions . . . 54
3.3 TimeStepper . . . 54
3.4 AmrMesh . . . 55
3.4.1 Main functionality . . . 55
3.4.2 Class options . . . 55
3.4.3 Runtime options . . . 56
3.5 CellTagger . . . 57

3.5.1 Design . . . 57
3.5.2 User interface . . . 57
3.5.3 Restrict tagging . . . 58
3.5.4 Adding a buffer . . . 58
3.6 GeoCoarsener . . . 58

4 Solvers **59**
4.1 Convection-Diffusion-Reaction . . . 59
4.1.1 Design . . . 59
4.1.2 Using CdrSolver . . . 61
4.1.3 Discretization details . . . 62
4.1.4 Example application . . . 66
4.2 Electrostatics . . . 67
4.2.1 FieldSolver . . . 67
4.2.2 Using FieldSolver . . . 67
4.2.3 Domain boundary conditions . . . 68
4.2.4 EB boundary conditions . . . 70
4.2.5 FieldSolverMultigrid . . . 71
4.2.6 Frequency dependent permittivity . . . 73
4.2.7 Limitations . . . 75
4.2.8 Example application . . . 75
4.3 Mesh ODE solver . . . 75
4.4 Radiative transfer . . . 75
4.4.1 RtSpecies . . . 75
4.4.2 RtSolver . . . 75
4.4.3 Diffusion approximation . . . 76
4.4.4 Monte Carlo methods . . . 80
4.4.5 Limitations . . . 82
4.4.6 Example application . . . 82
4.5 Surface charge solver . . . 82
4.6 Tracer particles . . . 82
4.7 Îto diffusion . . . 82
4.7.1 The Îto particle . . . 82
4.7.2 ito_species . . . 83
4.7.3 Computing time steps . . . 83
4.7.4 Remapping particles . . . 84
4.7.5 Limitations . . . 85
4.7.6 Example application . . . 85

5 Physics models **87**
5.1 Implemented models . . . 87
5.2 Advection diffusion . . . 87
5.3 Brownian walker . . . 87
5.4 CDR plasma . . . 87
5.4.1 Equations of motion . . . 88
5.4.2 CdrPlasmaPhysics . . . 88
5.4.3 Time discretizations . . . 93
5.4.4 JSON interface . . . 96
5.4.5 Simulation quick start . . . 114
5.5 Electrostatics . . . 115
5.6 Geometry . . . 115
5.7 Radiative transfer . . . 115

6 Tutorial **117**

6.1 Introduction . . . 117
6.2 Creating a geometry . . . 117
6.3 Setting up a `TimeStepper` . . . 117

7 Utilities **119**
7.1 Lookup tables . . . 119
7.1.1 Inserting data . . . 120
7.1.2 Restricting ranges . . . 120
7.1.3 Independent variable . . . 120
7.1.4 Swapping columns . . . 121
7.1.5 Regularize table . . . 121
7.1.6 Retrieving data . . . 122
7.1.7 Viewing tables . . . 122
7.2 Random numbers . . . 123
7.2.1 Drawing random numbers . . . 123
7.2.2 Setting the seed . . . 123
7.3 Least squares . . . 124
7.3.1 Polynomial expansion . . . 124
7.3.2 Neighborhood algorithm . . . 124
7.3.3 Weighted equations . . . 125
7.3.4 Pseudo-inverse . . . 125
7.3.5 Pruning equations . . . 126
7.3.6 Source code . . . 126

8 Contributing **127**
8.1 Contributions . . . 127
8.2 Convergence testing . . . 127
8.3 Continuous integration . . . 127
8.3.1 GitHub actions . . . 127
8.3.2 Running tests locally . . . 127
8.4 Code standard . . . 128
8.4.1 C++ standard . . . 128
8.4.2 Namespace . . . 128
8.4.3 File names . . . 128
8.4.4 File headers . . . 129
8.4.5 Code syntax . . . 130
8.4.6 Options files . . . 130
8.5 References . . . 130

Bibliography **131**

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

Important: `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` features include:

- 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.

This documentation is built as a part of the continuous integration (CI) at GitHub, and is automatically kept up-to-date with the latest version of `chombo-discharge`. It was built using *reStructuredText* with *Sphinx*, and is HTML browser-friendly.

1.1.1 Doxygen documentation

A separate Doxygen documentation of the `chombo-discharge` code is found at https://chombo-discharge.github.io/chombo-discharge/doxygen/html/index.html

1.2 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 simulating discharges in high-voltage engineering. Further development was started in 2021, where much of the code was redesigned for improved modularity and performance.

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. Key functionalities are provided in Table 1.2.1.

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*).

Table 1.2.1: Key capabilities.

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.
Computational particles?	**Yes.**
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 multiply parallel distributed grids (see *Realm*) for individually load balancing particle and fluid kernels. 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. See e.g. *Implemented models*.

1.2.3 Design approach

A fundamental design principle in `chombo-discharge` is the division between the AMR core, geometry, solvers, physics coupling, 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 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 deep 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 the solvers. In general, solvers may share common features (such as elliptic discretizations) or be completely disjoint. For this reason numerical solvers are asked to *register* AMR requirements. 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 make it available to solver, as illustrated in Fig. 1.2.1.

`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. Some 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

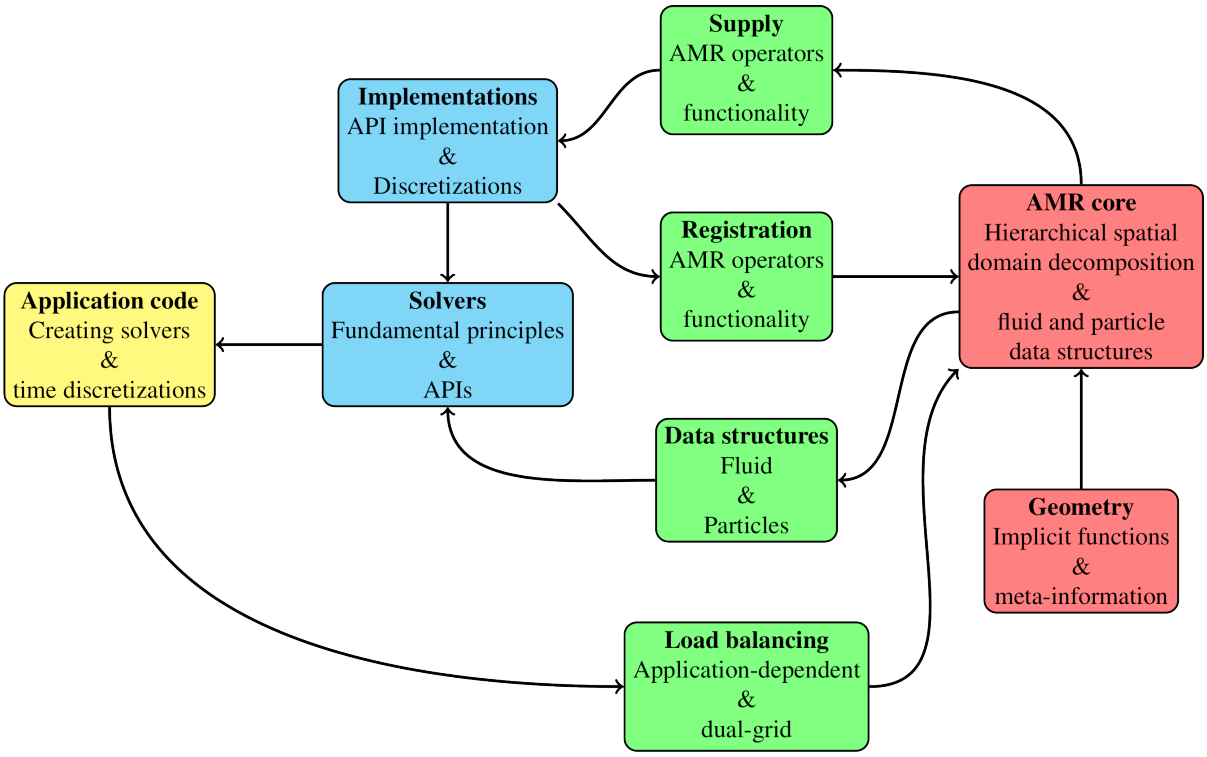

Fig. 1.2.1: Concept design sketch for `chombo-discharge`.

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 only interact with one another through these modules, and these modules usually advance the equations of motion using the method-of-lines.

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.
- The `EBGeometry` package, see https://github.com/rmrsk/EBGeometry.
- LAPACK and BLAS

The `Chombo`, `nlohmann/json`, and `EBGeometry` 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

- A serial or parallel version of HDF5, which is used for writing plot and checkpoint files.
- An MPI installation, which is used for parallelization.
- https://visit-dav.github.io/visit-website/ visualization, which used for visualization.

1.3.2 Organization

The chombo-discharge source files are organized as follows:

Table 1.3.1: Code organization.

Folder	Explanation
Source	Source files for the AMR core, solvers, and various utilities.
Physics	Various implementations that can run the chombo-discharge source code.
Geometries	Various geometries.
Submodules	Git submodule dependencies.
Exec	Various executable applications.

1.3.3 Cloning chombo-discharge

To obtain chombo-discharge, the simplest approach is to clone it and recursively fetch the submodules:

```
git clone --recursive git@github.com:chombo-discharge/chombo-discharge.git
```

1.3.4 Setting up the environment

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

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

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.

Note: To clone chombo-discharge directly to $DISCHARGE_HOME, 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}
```

chombo-discharge is built using a configuration file supplied to Chombo. This file must reside in $CHOMBO_HOME/mk. Some standard configuration files are supplied with chombo-discharge, and reside in $DISCHARGE_HOME/Lib/Local. These files may or may not work right off the bat.

1.3.5 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/Lib/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
```

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.

1.3.6 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/Lib/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 MPI or HDF5 installed. 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/Lib/Local/Make.defs.GNU $CHOMBO_HOME/mk/Make.defs.local
```

   - **Serial build with HDF5**:

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

   - **MPI build without HDF5**:

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

   - **MPI build with HDF5**:

```
cp $DISCHARGE_HOME/Lib/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/Lib/Local if it exists for your computer. Alternatively, copy $DISCHARGE_HOME/Lib/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.

1.3.7 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/Exec/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/Exec/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.

1.3.8 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 always written to HDF5. The plot files will reside in the `plt` subfolder where the application was run.

Currently, we have only used VisIt for visualizing the plot files. Learning how to use VisIt is not a part of this documentation; there are great tutorials on the VisIt website.

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 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 Chap:GettingStarted.

1.5.2 Running applications

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.

Parallel

If the executable was compiled with MPI, one executes with e.g. mpirun (or one of its aliases):

```
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.

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). 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
```

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.

1.5.4 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.

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 given in *Simulation output organization.*.

Table 1.5.1: Simulation output organization.

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

The reason for the output folder 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 generally speaking responsible for specifying what goes into the plot 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 CdrGodunov.plt_vars = phi. An empty entry like CdrGodunov.plt_vars = may lead to run-time errors, so if you do not want a class to provide plot data you may put CdrGodunov.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 that determines which MPI ranks write pout.n files. For example, if you only want the master MPI rank to write pout.0, you would do

```
export CH_OUTPUT_INTERVAL=999999999
```

Important: 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, see Fig. 2.1.1. 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.

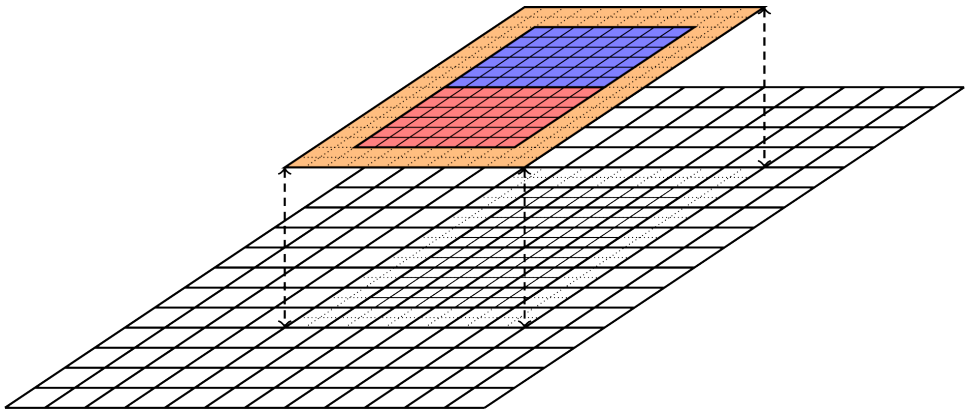

Fig. 2.1.1: Cartesian patch-based refinement showing two grid levels. The fine-grid level lives on top of the coarse level, and consists of two patches (red and blue colors) with two layers of ghost cells (dashed lines and orange shaded region).

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. 2.1.2 shows an example of patch-based grid refinement for a complex surface.

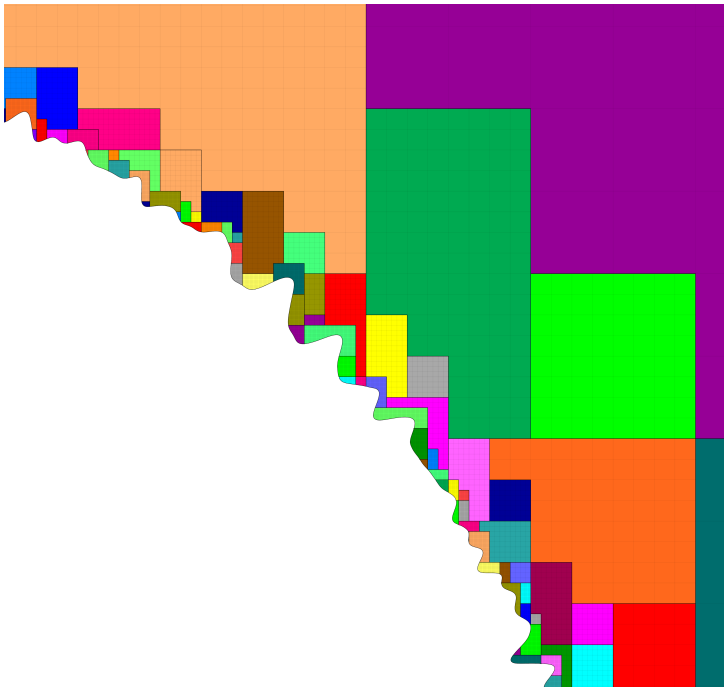

Fig. 2.1.2: Patch-based refinement (factor 4 between levels) of a complex surface. Each color shows a patch, which is a rectangular computational unit.

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. 2.1.3. 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 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).

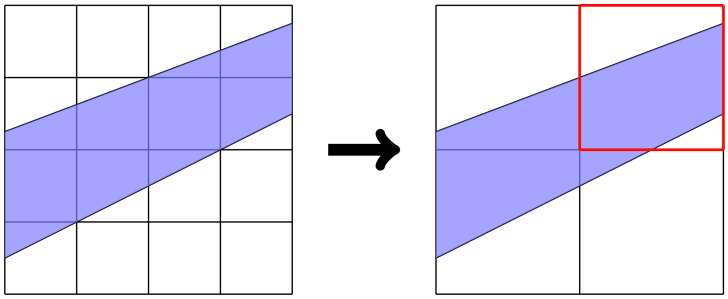

Fig. 2.1.3: Example of how multi-valued cells occur during grid coarsening. Left: Original grid. Right: Coarsened grid.

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*.

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 $d_2(\mathbf{x})$:

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

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.

EBGeometry

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 through the EBGeometry package.

> **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 `EBGeometry` 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 the EBGeometry documentation.

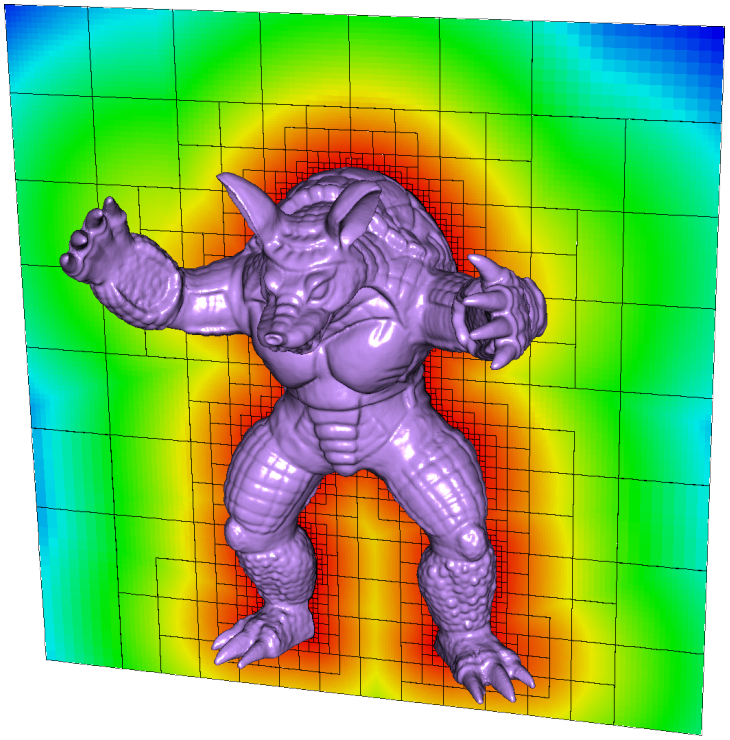

Fig. 2.1.4: Example of an SDF reconstruction and cut-cell grid from a surface tesselation in `chombo-discharge`.

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 and then generate the coarser levels cells through grid coarsening. This will look through all cells on the finest level, so for a domain which is effectively $N \times N \times N$ cells there are $\mathcal{O}\left(N^3\right)$ implicit function queries (in 2D, the complexity is $\mathcal{O}\left(N^2\right)$). Note that as N becomes large, say $N = 10^5$, geometric queries of this type become a bottleneck.

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 yields a reduction of the original $\mathcal{O}\left(N^3\right)$ scaling in Chombo grid generation to an $\mathcal{O}\left(N^2\right)$ scaling in chombo-discharge. Typically, we find that this makes geometry generation computationally trivial (in the sense that it is very fast compared to the simulation).

To understand this process, note that the SDF satisfies the Eikonal equation

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

and so it is well-behaved for all $\mathbf{x}$. The SDF can thus 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, as shown in Fig. 2.1.5. We know that cut cells do not exist in the grid patch if $|f\left(\mathbf{x_i}\right)| > \frac{1}{2}\Delta x$ 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. 2.1.5. Once all the grid patches that contain cut-cells have been found, these patches are distributed (i.e., load balanced) to the various MPI ranks for computing the discrete grid information.

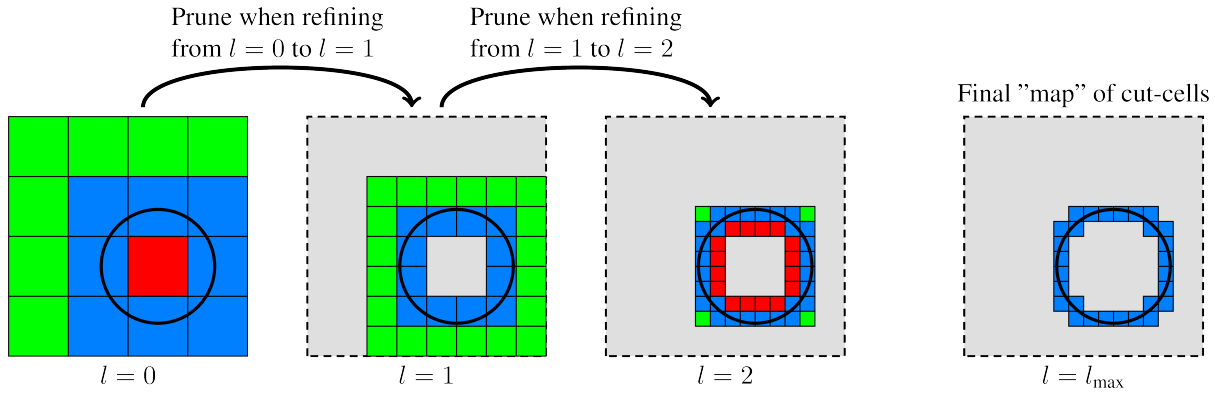

Fig. 2.1.5: Pruning cut-cells with the signed distance field. Red-colored grid patches are grid patches entirely contained inside the EB. Green-colored grid patches are entirely outside the EB, while blue-colored grid patches contain cut-cells.

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 $d\left(\mathbf{x}\right) = R^2 - \mathbf{x}\cdot\mathbf{x}$ must be used with caution.

2.1.5 Mesh generation

`chombo-discharge` supports two algorithm for AMR grid generation:

1. The classical Berger-Rigoutsos algorithm [1].
2. A *tiled* algorithm [5].

Both algorithms work by taking a set of flagged cells on each grid level and generating new boxes that cover the flags. Only *properly nested* grids are generated, in which case two grid levels $l-1$ and $l+1$ are separated by a non-zero number of grid cells on level l. This requirement is specific for patch-based grids; itd dieoes not fundamentally exist for quad- and oct-tree grids. For patch based AMR, the rationale for this requirement is that stencils on level $l+1$ should should only reach into grid cells on levels l and $l+1$. For example, ghost cells on level $l+1$ should be interpolated from data only on levels l and $l+1$.

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, see [1] for implementation details. Typically, it is not used at large scale in 3D due to its memory consumption.

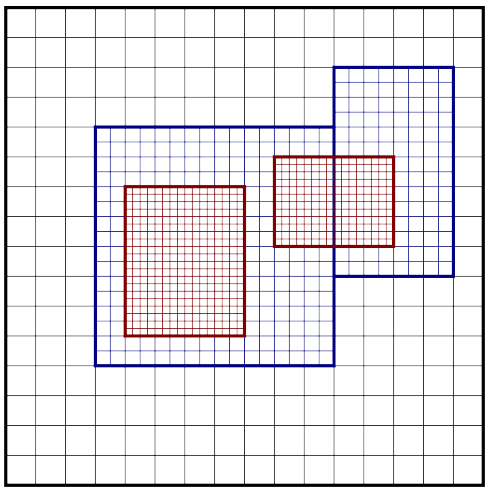

Fig. 2.1.6: Classical cartoon of patch-based refinement. Bold lines indicate entire grid blocks.

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 slightly 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.

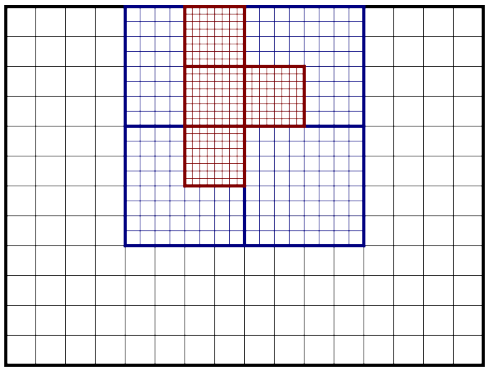

Fig. 2.1.7: Classical cartoon of tiled patch-based refinement. Bold lines indicate entire grid blocks.

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){
```

(continues on next page)

(continued from previous page)

```
   (...)
}
```

Important: The `FArrayBox` stores the data in column major order.

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.
```

(continues on next page)

(continued from previous page)

```
   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 of the type discussed in *Spatial discretization* 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, see e.g. Fig. 2.3.1.

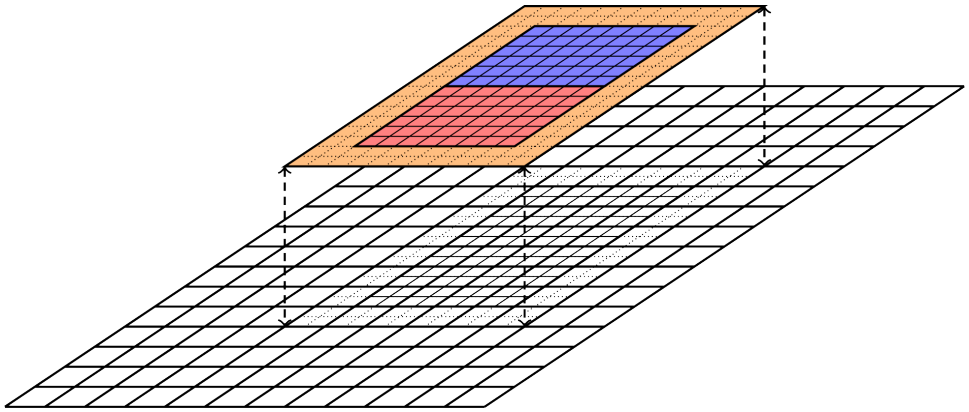

Fig. 2.3.1: Cartesian patch-based refinement showing two grid levels. This is encapsulated by EBAMRData where the levels are stored in a Vector and the grid patches in the LevelData object.

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 interact with EBAMRData<T> 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 on a Realm, and EBAMRData<T> itself it typically not used anywhere elsewhere within chombo-discharge.

A number of explicit template specifications exist and are frequently used. 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*:

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

Here, `nComps` determine the number of cell-centered data components. 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 patches

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()];
   }
}
```

2.3.3 Iterating over cells

For single-valued data, `chombo-discharge` uses standard loops (in column-major order) for iterating over data. For example, the standard loops for iterating over cell-centered data are

```
namespace BoxLoops {

   template <typename Functor>
   ALWAYS_INLINE void
   loop(const Box& a_computeBox, Functor&& kernel, const IntVect& a_stride = IntVect::Unit);

   template <typename Functor>
   ALWAYS_INLINE void
   loop(VoFIterator& a_iter, Functor&& a_kernel);
}
```

Here, the `Functor` argument is a C++ lambda or `std::function` which takes a grid cell as a single argument. For the first loop, we iterate over all grid cells in `a_computeBox`. In the second function we use a `VoFIterator`, which Iterating over the cells in a patch data holder (like the `EBCellFAB`) can be done with a `VoFIterator`, which can iterate through cells on an `EBCellFAB` that are not covered by the geometry 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){


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

      auto regularKernel = [&](const IntVect& iv) -> void {
         regularData(iv, component) = 1.0;
      };

      auto irregularKernel = [&](const VolIndex& vof) -> void {
         patchData(vof, component = 1.0;
      };

      // Kernel regions (defined by user)
      Box computeBox;
      VoFIterator vofit;

      BoxLoops::loop(computeBox, regularKernel);
      BoxLoops::loop(vofit, irregularKernel);
   }
}
```

There are loops available for other types of data (e.g., face-centered data), see the BoxLoop documentation.

2.3.4 Coarsening data

Conservative coarsening of data is done using the `averageDown(...)` functions in *AmrMesh*. When using these functions, coarse-grid data is replaced by a conservative average of fine grid data throughout the entire AMR hierarchy. The signatures for various types of data are 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
```

(continues on next page)

(continued from previous page)

```
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;
```

There are other types of coarsening available also. For example, the `averageFaces(...)` will use unweighted averaging, see the AmrMesh API for further details.

2.3.5 Filling ghost cells

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

```
void interpGhost(MFAMRCellData& a_data, const std::string a_realm) const;

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.

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* for details. The functions for filling ghost cells in this way are

```
void interpGhostMG(MFAMRCellData& a_data, const std::string a_realm) const;

void interpGhostMG(EBAMRCellData& a_data, const std::string a_realm, const phase::which_phase a_phase) const;
```

See the AmrMesh API for further details.

2.3.6 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. 2.3.2 which shows the typical 5-point stencil in regular grid regions, and also a much larger and more complex stencil.

In Fig. 2.3.2 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. 2.3.2 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*.

To compute gradients of a scalar, one can simply call `AmrMesh::computeGradient(...)` functions:

```
void computeGradient(EBAMRCellData&          a_gradient,
                     const EBAMRCellData&    a_phi,
                     const std::string       a_realm,
```

(continues on next page)

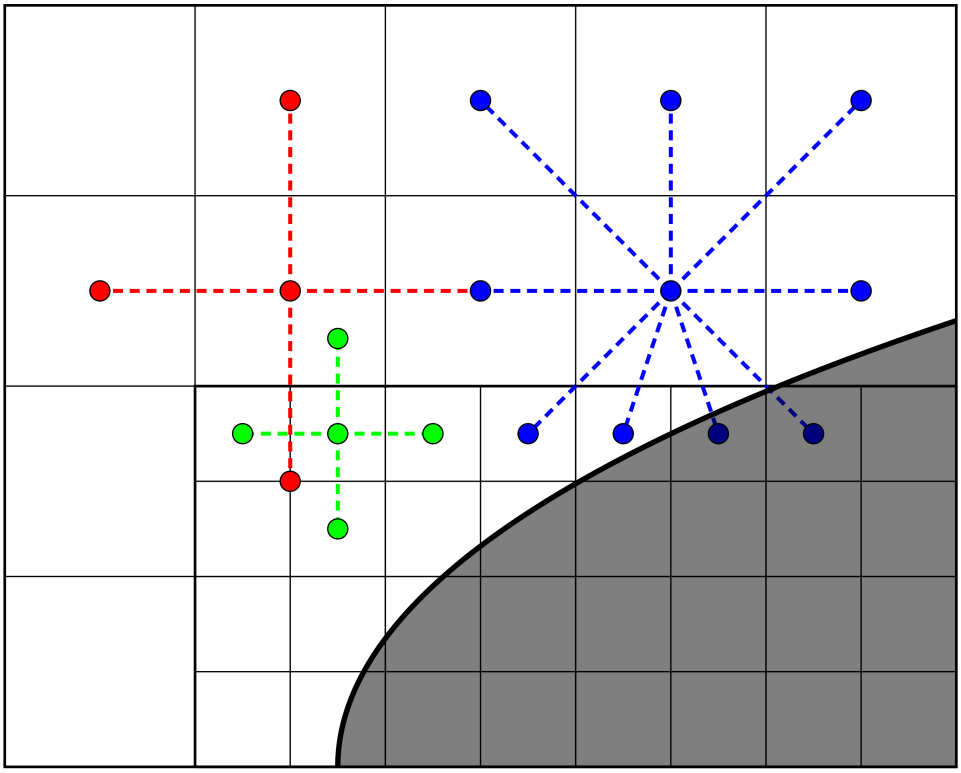

Fig. 2.3.2: Example of stencils for computing gradients near embedded boundaries. The red stencil shows a regular 5-point stencil for computing the gradient on the coarse side of the refinement boundary; it reaches into the coarsened data beneath the fine level. The green stencil shows a similar 5-point 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.

(continued from previous page)

```
                    const phase::which_phase a_phase) const;

void computeGradient(MFAMRCellData& a_gradient, const MFAMRCellData& a_phi, const std::string a_realm) const;
```

See *AmrMesh* or refer to the AmrMesh API for further details.

2.3.7 Copying data

To copy data, one may use the `EBAMRData<T>::copy(...)` function *or* `DataOps::copy` (see *DataOps*). These differ in the following way:

- `EBAMRData<T>::copy` works across realms, but will not copy ghost cells.
- `DataOps::copy` will always do a local copy, and thus the data that is copied *must* be defined on the same realm.

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, which will include ghost cells. Communication copies involving MPI are performed otherwise, in which case ghost cells are *not* copied into the new data holder.

2.3.8 DataOps

We have prototyped functions for many common data operations in a static class `DataOps`. For example, setting the value of various data holders 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);
```

For the full API, see the DataOps documentation.

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. There are many examples of `chombo-discharge` particles, see e.g. TracerParticle or Photon. 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` with non-random access.

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, 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<ListBox<P> >`, although it actually inherits from `LayoutData<ListBox<P> >`. Each grid patch 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

Here, we give some examples of basic use of `ParticleContainer`. For the full API, see the ParticleContainer doxygen documentation.

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 as follows:

```
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){
         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){
            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, which adds algorithmic complexity (`chombo-discharge` can handle this complexity).
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.

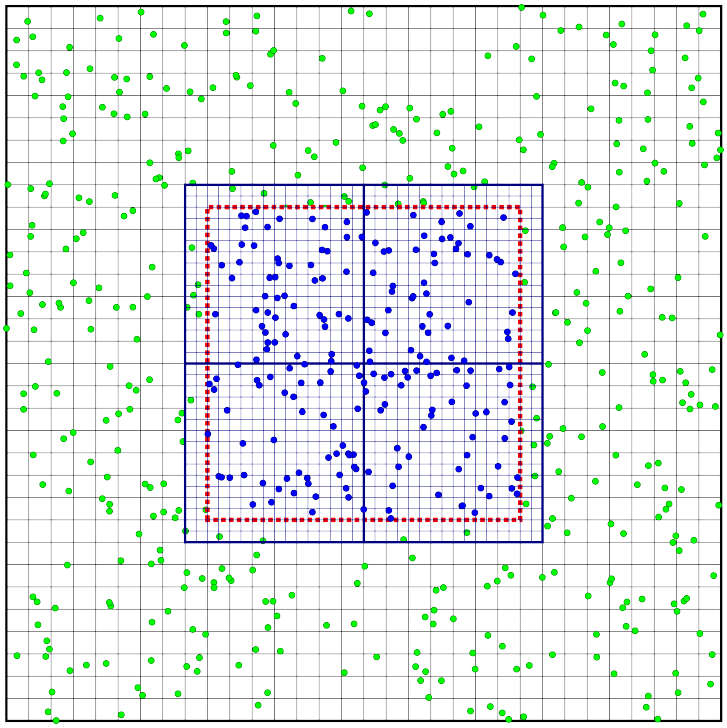

Fig. 2.4.1: The `ParticleValidRegion` allows particles whose position fall into a fine grid patch to be moved to a coarser level if they are within a specified distance from the refinement boundary. In this case, the green particles that overlap with the fine-level grid are remapped to the coarse level.

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.

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
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();
```

Creating particle halo 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` must be `"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, see Fig. 2.4.2.

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 or discrete information, as well as 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);
}
```

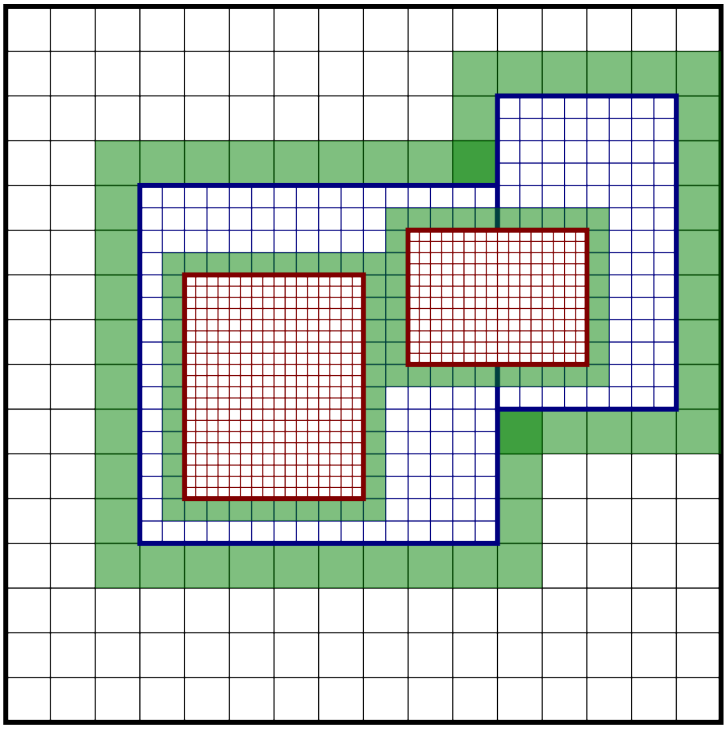

Fig. 2.4.2: Example of a particle halo mask (shaded green color) surrounding refined grid levels.

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 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);

template <class P, const RealVect&(P::*particleVectorField)() 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.

Note that when depositing vector-quantities (such as electric currents), one must call the version which takes `RealVect`

`P::particleVectorField() const` as a template parameter. The supplied function must return a `RealVect` and `a_meshData` must have `SpaceDim` components.

Next, the input arguments to `depositParticles` are the output mesh data holder (must have exactly one or `SpaceDim` components), 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. Strictly speaking, this argument only affects how the particle mass is deposited from the coarse level to the fine level. 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.
- `CoarseFineDeposition::HaloNGP` This uses halo particles, but the particles along the refinement boundary are deposited with an NGP scheme.

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 never deposited into covered grid cells.

As an 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.
   }

   RealVect momentum() const {
      return m_mass*m_velocity;
   }

protected:

   Real m_mass;

   Real m_velocity;
};
```

Here, we have included an extra member function `mass()` 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>(...);
```

When depositing momentum, use

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

2.4.11 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.

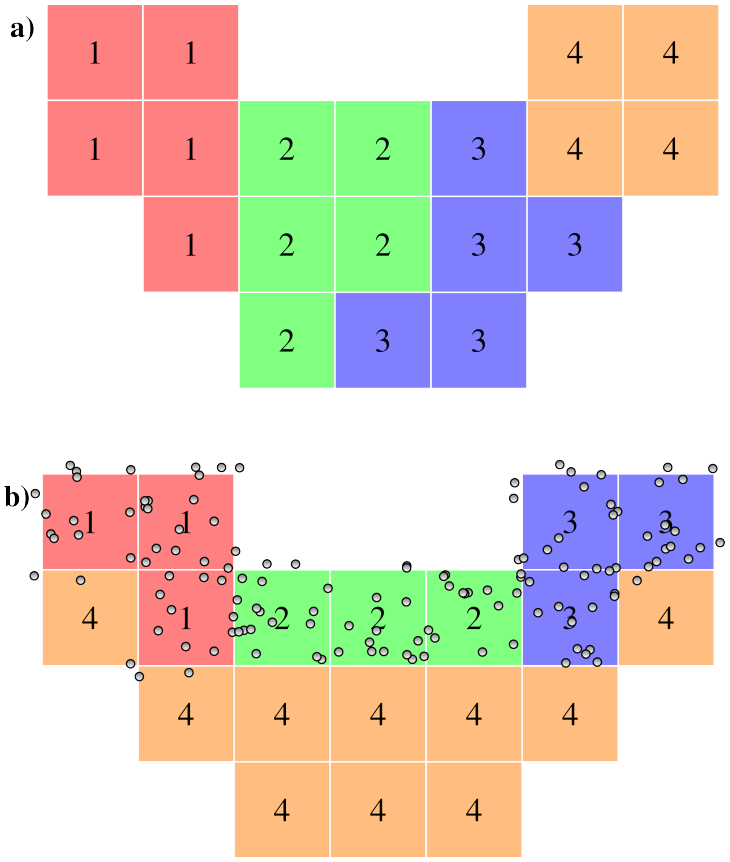

Fig. 2.5.1: Sketch of dual grid approach. Each rectangle represents a grid patch and the numbers show MPI ranks. a) Load balancing with the number of grid cells. b) Load balancing with the number of particles.

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.

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"`. Fundamentally, there is no limitation to the number of realms that can be allocated.

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 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()`.

Currently available operators are:

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_fine_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 probably be in use.

2.5.4 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 piecewise smooth.

To solve the Helmholtz equation, it is solved 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

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

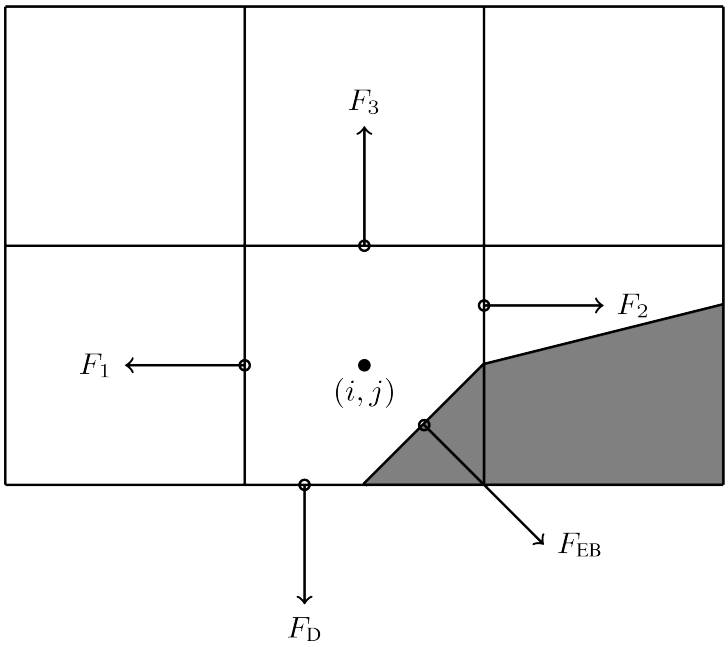

Fig. 2.6.1: Location of fluxes for finite volume discretization.

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

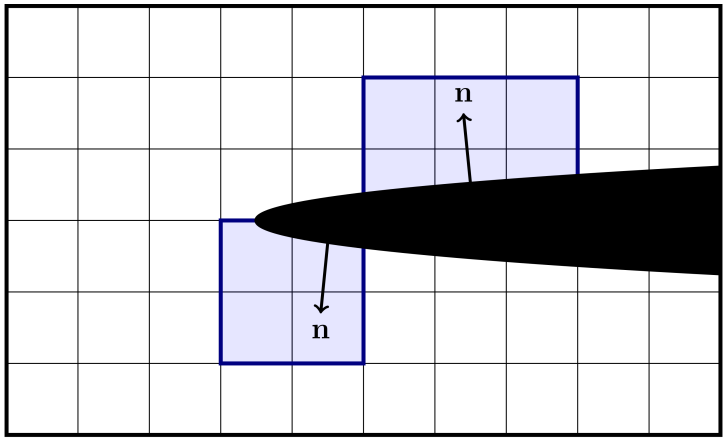

Fig. 2.6.2: Examples of neighborhoods (quadrant and half-space) used for gradient reconstruction on the EB.

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.

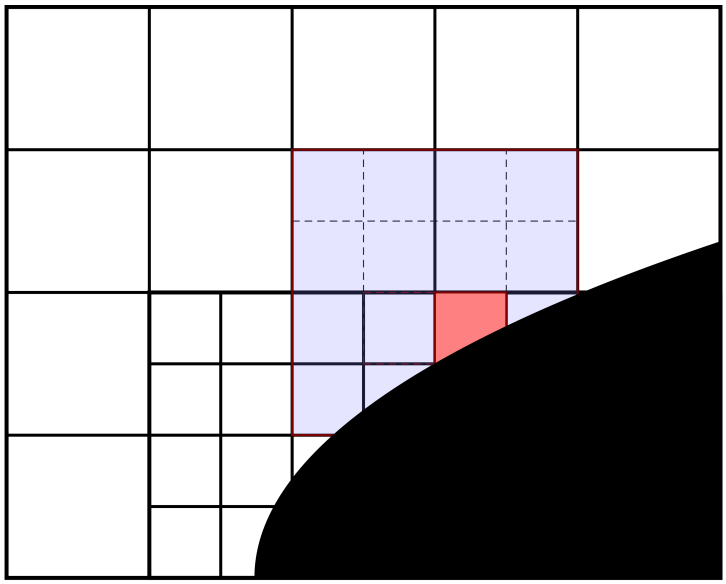

Fig. 2.6.3: Example of the region of a second order stencil for the Laplacian operator with second order gradient reconstruction on the embedded boundary.

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.

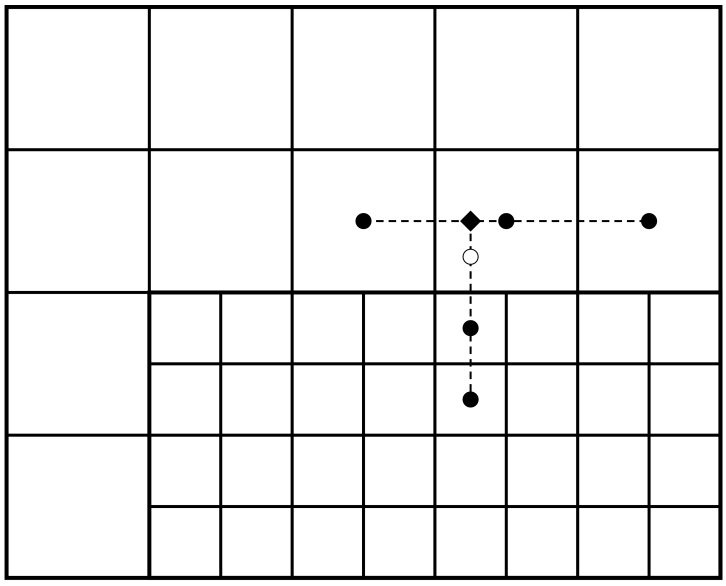

Fig. 2.6.4: Standard finite-difference stencil for ghost cell interpolation (open circle). We first interpolate the coarse-grid cells to the centerline (diamond). The coarse-grid interpolation is then used together with the fine-grid cells (filled circles) for interpolation to the ghost cell (open circle).

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).

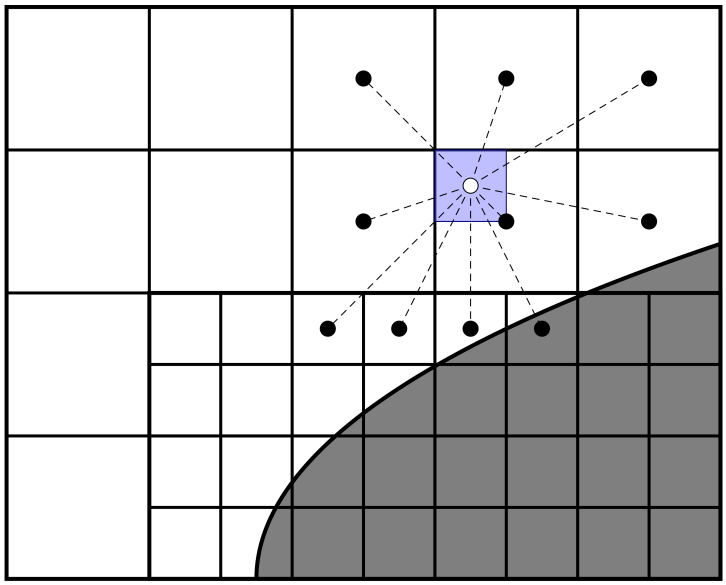

Fig. 2.6.5: Multigrid interpolation for refinement boundaries away from and close to an embedded boundary.

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.

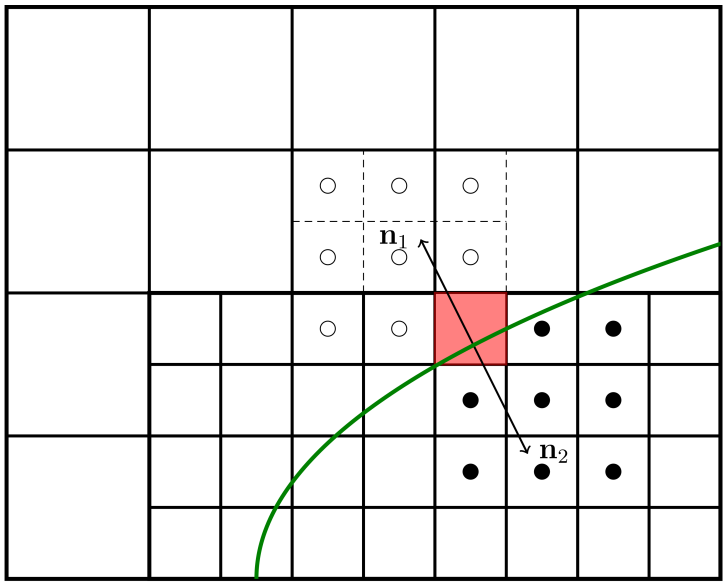

Fig. 2.6.6: Example of cells and stencils that are involved in discretizing the jump condition. Open and filled circles indicate cells in separate phases.

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

DESIGN

3.1 Driver

The `Driver` class is the class that runs `chombo-discharge` simulations and is defined in `/Source/Driver/CD_Driver.cpp(H)`. The constructor for this class is

```
Driver(const RefCountedPtr<ComputationalGeometry>& a_compgeom,
       const RefCountedPtr<TimeStepper>&           a_timestepper,
       const RefCountedPtr<AmrMesh>&               a_amr,
       const RefCountedPtr<CellTagger>&            a_celltagger = RefCountedPtr<CellTagger>(nullptr),
       const RefCountedPtr<GeoCoarsener>&          a_geocoarsen = RefCountedPtr<GeoCoarsener>(nullptr));
```

Observe that the `Driver` class does not *require* an instance of *CellTagger*. If users decide to omit a cell tagger, regridding functionality is completely turned off and only the initially generated grids will be used throughout the simulation.

The usage of the `Driver` class is primarily object construction with dependency injection of the geometry, the physics (i.e. `TimeStepper`), the `AmrMesh` instance, and possibly a cell tagger. The `Driver` class will retrieve configurations from the input script during construction.

3.1.1 Simulation setup

Usually, only a single routine is used:

```
void setupAndRun(const std::string a_input_file);
```

This routine will set up and run a simulation. Simulation setup depends on the way a simulation is run.

New simulations

If a simulation starts from the first time step, the `Driver` class will perform the following major steps within `setupAndRun()`.

1. Ask `ComputationalGeometry` to generate the cut-cell moments.
2. Collect all the cut-cells and ask `AmrMesh` to set up an initial grid. This initial grid is always generated by flagging cells for refinement along the boundary. Various options are available for configuring this initial grid, see *Cell refinement philosophy*. Also note that it is possible to restrict the maximum level that can be generated from the geometric tags, or remove some of the cut-cell refinement flags through the auxiliary class `GeoCoarsener`.
3. Ask the `TimeStepper` to set up relevant solvers and fill them with initial data.
4. Perform the number of initial regrids that the user asks for.

Step 3 and 4 will differ significantly depending on the physics that is solved for.

The full code is given in `Driver::setupFresh()`.

Restarting simulations

If a simulation *does not start* from the first time step, the `Driver` class will perform the following major steps within `setupAndRun(...)`.

1. Ask `ComputationalGeometry` to generate the cut-cell moments.
2. Read a checkpoint file that contains the grids and all the data that have been checkpointed by the solvers. `Driver` will issue an error and abort if the checkpoint file does not exist.
3. Ask the `TimeStepper` to perform a "post-checkpoint" step to initialize any remaining data so that a time step can be taken. This functionality has been included because not all data in every solver needs to be checkpointed. For example, an electric field solver only needs to write the electric potential to the checkpoint file because the electric field is simply obtained by taking the gradient.
4. Perform the number of initial regrids that the user asks for.

Again, step 3 will differ significantly depending on the physics that is solved for.

The full code is given in `Driver::setupForRestart()`.

3.1.2 Simulation advancement

The algorithm for running a simulation is conceptually simple; the `Driver` class simply calls `TimeStepper` for computing a reasonable time step for advancing the equations, and then it asks `TimeStepper` to actually perform the advance. Regrids, plot files, and checkpoint files are written at certain step intervals (or when the `TimeStepper` demands them). In essence, the algorithm looks like this:

```
Driver::run(...){

   while(KeepRunningTheSimulation){
      if(RegridEverything){
         Driver->regrid()
      }

      TimeStepper->computeTimeStep()
      TimeStepper->advanceAllEquationsOneStep()

      if(WriteAPlotFile || EndOfSimulation){
         Driver->writePlotFile();
      }
      if(TimeToWriteACheckpointFile || EndOfSimulation){
         Driver->writeCheckpointFile()
      }

      KeepRunningTheSimulation = true or false
   }
}
```

3.1.3 Regridding

Regrids are called by the `Driver` class and occur as follows in `Driver::regrid(...)`:

1. Ask `CellTagger` to generate tags for grid refinement and coarsening.
2. The `TimeStepper` class stores data that is subject to regrids. How this happens depends on the solver that is run. For grid-based solvers, e.g. CDR solvers, the scalar ϕ is copied into a scratch space. The reason for this backup is that during the regrid ϕ will be allocated on the *new* AMR grids, but we must still have access to the previously defined data in order to interpolate to the new grids.
3. If necessary, `TimeStepper` can deallocate unecessary storage. Implementing a deallocation function for `TimeStepper`-derived classes is not a requirement, but can in certain cases be useful, for example when using the Berger-Rigoutsous algorithm at large scale.

4. The `AmrMesh` class generates the new grids and defines new AMR operators.
5. The `TimeStepper` class regrids its solvers and internal data.
6. The `TimerStepper` performs a *post-regrid* operation (e.g. filling solvers with auxiliary data).

In C++ pseudo-code, this looks something like:

```
Driver::regrid(){

   // Tag cells
   CellTagger->tagCellsForRefinement()

   // Store old data and free up some memory
   TimeStepper->storeOldGridData()
   TimeStepper->deallocateUnneccesaryData()

   // Generate the new grids
   AmrMesh->regrid()

   // Regrid physics and all solvers
   TimeStepper->regrid()

   // Do a post-regrid step
   TimeStepper->postRegrid()
}
```

The full code is defined in `Driver::regrid()` in file `$DISCHARGE_HOME/Source/Driver/CD_Driver.cpp`.

3.1.4 Class options

Various class options are available for adjusting the behavior of the `Driver` class

- `Driver.verbosity` controls output will be given to `pout.n`. We use 2 or 3 - higher values are for debugging.
- `Driver.geometry_generation` controls the grid generation method (see *Geometry generation*). Valid options are *chombo-discharge* or *chombo*.
- `Driver.geometry_scan_level`. Which refinement level to initiate the `chombo-discharge` geometry generation method. This entry indicates the number of refinements of the coarsest AMR level used in the simulation. E.g. if the `Driver.geometry_scan_level=1` and the coarsest AMR level is 128^3 then the signed distance pruning (see *Geometry generation*) begins at the AMR level 256^3. Note that negative numbers are also permitted, in which case the pruning initiates at a coarsened level.
- `Driver.ebis_memory_load_balance`. If using Chombo geo-gen, use memory as loads for EBIS generation. Valid options are *true* or *false*.
- `Driver.plot_interval`. Time steps between each plot file.
- `Driver.checkpoint_interval`. Time steps between each checkpoint file.
- `Driver.regrid_interval`. Time steps between each regrid.
- `Driver.write_regrid_files`. Write plot files during regrids. Valid options are *true* or *false*.
- `Driver.write_restart_files`.Write plot files during restarts. Valid options are *true* or *false*.
- `Driver.initial_regrids`. Number of initial regrids to perform when starting (or restarting) a simulation.
- `Driver.start_time`. Simulation start time.
- `Driver.stop_time`.Simulation stop time.
- `Driver.max_steps`. Maximum number of simulation time steps.
- `Driver.geometry_only`. If *true*, do not run the simulation and only write the geometry to file.
- `Driver.write_memory`. Write MPI memory report. Valid options are *true* or *false*.

- `Driver.write_loads`. Write computational loads. Valid options are *true* or *false*.
- `Driver.output_directory`. Output directory.
- `Driver.output_names`. Simulation file names.
- `Driver.max_plot_depth`. Maximum plot depth. Values < 0 means all levels.
- `Driver.max_chk_depth`. Maximum checkpoint file depth. Values < 0 means all levels.
- `Driver.num_plot_ghost`. Number of ghost cells in plot files.
- `Driver.plt_vars`. Plot variables for `Driver`. Valid options are *tags*, *mpi_rank*, *levelset*.
- `Driver.restart`. Restart step (less or equal to 0 implies fresh simulation)
- `Driver.allow_coarsening`. Allows removal of grid levels if cell tags dont run deep enough.
- `Driver.grow_geo_tags`. How much to grow boundary cell tags.
- `Driver.refine_angles`. Refine cells if the angle between normal vector in neighboring cells exceed this threshold.
- `Driver.refine_electrodes`. Refine electrode surfaces. Values < 0 will refine all the way down.
- `Driver.refine_dielectrics`. Refine dielectric surfaces. Values < 0 will refine all the way down.

3.1.5 Runtime options

The following options can be adjusted during runtime:

- `Driver.verbosity`.
- `Driver.plot_interval`.
- `Driver.checkpoint_interval`.
- `Driver.regrid_interval`.
- `Driver.write_regrid_files`.
- `Driver.write_restart_files`.
- `Driver.stop_time`.
- `Driver.max_steps`.
- `Driver.write_memory`.
- `Driver.write_loads`.
- `Driver.num_plot_ghost`.
- `Driver.plt_vars`.
- `Driver.allow_coarsening`.

3.2 ComputationalGeometry

`ComputationalGeometry` is the class that implements geometries in `chombo-discharge`. The source for for this class is located in `$DISCHARGE_HOME/Source/Geometry/`.

Note: `ComputationalGeometry` is *not* an abstract class. The default implementation is an empty geometry – i.e. a geometry without any objects.

Making a non-empty `ComputationalGeometry` class requires that you inherit from `ComputationalGeometry` and set the following class members:

```
Real m_eps0;
Vector<Electrode> m_electrodes;
Vector<Dielectric> m_dielectrics;
```

Here, `m_eps0` is the relative gas permittivity, `m_electrodes` are the electrodes for the geometry and `m_dielectrics` are the dielectrics for the geometry. These are described in detail further down.

When geometries are created, the `ComputationalGeometry` class will first create the (approximations to the) signed distance functions that describe two possible material phases (gas and solid). Here, the *solid* phase is the part of the computational domain inside the dielectrics, while the *gas phase* is the part of the computational domain that is outside the electrodes and dielectrics.

3.2.1 Electrode

The `Electrode` class is responsible for describing an electrode and its boundary conditions. Internally, this class is lightweight and consists only of a tuple that holds a level-set function and an associated boolean value that tells whether or not the level-set function has a live potential or not. The constructor for the electrode class is:

```
Electrode(RefCountedPtr<BaseIF> a_baseif, bool a_live, Real a_fraction = 1.0);
```

where the first argument is the level-set function and the second argument is responsible for setting the potential. The third argument is an optional argument that allows the user to set the potential to a specified fraction of the applied potential.

3.2.2 Dielectric

The `Dielectric` class describes a dielectric. This class is lightweight and consists only of a tuple that holds a level-set function and the associated permittivity. The constructors are

```
Dielectric(RefCountedPtr<BaseIF> a_baseif, Real a_permittivity);
Dielectric(RefCountedPtr<BaseIF> a_baseif, Real (*a_permittivity)(const RealVect a_pos);
```

where the first argument is the level-set function and the second argument sets a constant permittivity (first constructor) or a variable permittivity (second constructor).

3.2.3 Retrieving distance functions

It is possible to retrieve the SDFs for each phase, as well as the the electrodes and dielectrics. This functionality is provided by:

```
const Vector<Dielectric>& getDielectrics() const;
const Vector<Electrode>& getElectrodes() const;

const RefCountedPtr<BaseIF>& getGasImplicitFunction() const;
const RefCountedPtr<BaseIF>& getSolidImplicitFunction() const;
```

3.3 TimeStepper

The `TimeStepper` class is the physics class in `chombo-discharge` - it has direct responsibility of setting up the solvers and performing time steps.

Since it is necessary to implement different solvers for different types of physics, `TimeStepper` is an abstract class with the following pure functions:

```
// Setup routines
virtual void setupSolvers() = 0;
virtual void allocate() = 0;
virtual void initialData() = 0;
virtual void postInitialize() = 0;
virtual void postCheckpointSetup() = 0;
virtual void registerRealms() = 0;
virtual void registerOperators() = 0;
virtual void parseRuntimeOptions();

// IO routines
virtual void writeCheckpointData(HDF5Handle& a_handle, const int a_lvl) const = 0;
virtual void readCheckpointData(HDF5Handle& a_handle, const int a_lvl) = 0;
virtual void writePlotData(EBAMRCellData& a_output, Vector<std::string>& a_plotVariableNames, int& a_icomp) const = 0;
virtual int  getNumberOfPlotVariables() const = 0;
virtual Vector<long int> getCheckpointLoads(const std::string a_realm, const int a_level) const;

// Advance routines
virtual void computeDt(Real& a_dt, TimeCode& a_timeCode) = 0;
virtual Real advance(const Real a_dt) = 0;
virtual void synchronizeSolverTimes(const int a_step, const Real a_time, const Real a_dt) = 0;
virtual void printStepReport() = 0;

// Regrid routines
virtual void preRegrid(const int a_lmin, const int a_oldFinestLevel) = 0;
virtual void postRegrid() = 0;
virtual void regrid(const int a_lmin, const int a_oldFinestLevel, const int a_newFinestLevel) = 0;
virtual bool needToRegrid();
virtual bool loadBalanceThisRealm(const std::string a_realm) const;
virtual void loadBalanceBoxes(Vector<Vector<int> >&           a_procs,
                              Vector<Vector<Box> >&           a_boxes,
                              const std::string               a_realm,
                              const Vector<DisjointBoxLayout>& a_grids,
                              const int                       a_lmin,
                              const int                       a_finestLevel);
```

These functions are all used in the `Driver` class at various stages. The three functions in the category *setup routines* are, for example using during simulation setup or after reading a checkpoint file for simulation restarts. The IO routines are there so that users can choose which solvers perform any output, and the advance routines are there such that the user can implement new algorithms for time integration. Finally, the *regrid routines* are there so that the solvers can back up their data before the old grids are destroyed (`preRegrid()`) and the data is interpolated data onto the new AMR grids `regrid(...)`.

To see how these functions can be implemented, see *Introduction*.

3.4 AmrMesh

AmrMesh handles (almost) all spatial operations in `chombo-discharge`. Internally, *AmrMesh* contains a bunch of operators that are useful across classes, such as ghost cell interpolation operators, coarsening operators, and stencils for interpolation and extrapolation near the embedded boundaries. *AmrMesh* also contains routines for generation and load-balancing of grids based and also contains simple routines for allocation and deallocation of memory.

Note: *AmrMesh* only handles spatial *operations*, it otherwise has limited knowledge of numerical discretizations.

AmrMesh is an integral part of `chombo-discharge`, and users will never have the need to modify it unless they are implementing something entirely new. The behavior of *AmrMesh* is modified through it's available input parameters, listed below:

3.4.1 Main functionality

There are two main functionalities in `AmrMesh`:

1. Building grid hierarchies, and providing geometric information
2. Providing AMR operators.

The grids in `AmrMesh` consist of a `DisjointBoxLayout` (see *Chombo-3 basics*) on each level, supported also by the EB information (`EBISLayout`). Recall that each grid patch in a `DisjointBoxLayout` is owned by a unique rank. However, since `chombo-discharge` supports multiple decompositions, we support the use of multiple `DisjointBoxLayout` describing the same grid level. Although these `DisjointBoxLayout` consist of the same boxes, the patch-to-rank mapping can be different (see *Realm*). To fetch a grid on a particular level, one can call `AmrMesh::getGrids(const std::string a_realm)`. E.g.

```
const std::string myRealm;
const int myLevel;

const DisjointBoxLayout& dbl = m_amr->getGrids("myRealm")[myLevel];
```

Likewise, to fetch the geometric (EB) information for a specified realm, phase, and level:

```
const std::string myRealm;
const int myLevel;
const phase::which_phase myPhase;

const EBISLayout& ebisl = m_amr->getEBISLayout(myRealm, myPhase)[myLevel];
```

In addition to the grids, the user can fetch AMR operators. These are, for example, coarsening operators, interpolation operators, ghost cell interpolators etc. To save some regrid time, we don't always build every AMR operator that we might ever need, but have solvers *register* the ones that they specifically need. See *Realm* for details.

3.4.2 Class options

The class options below control `AmrMesh`:

- `AmrMesh.lo_corner`. Low corner of problem domain (e.g. 0 0 0)
- `AmrMesh.hi_corner`. High corner of problem domain (e.g. 1 1 1).
- `AmrMesh.verbosity`. Class verbosity. Leave to -1 unless you are debugging.
- `AmrMesh.coarsest_domain`. Number of grid cells on coarsest domain
- `AmrMesh.max_amr_depth`. Maximum number of refinement levels.

- `AmrMesh.max_sim_depth`. Maximum simulation depth. Values < 0 means that grids can be generated with depths up to `AmrMesh.max_amr_depth`.
- `AmrMesh.fill_ratio`. Fill ratio for BR grid generation
- `AmrMesh.irreg_growth`. Buffer region around irregular tagged cells.
- `AmrMesh.buffer_size`. Buffer size for BR grid generation.
- `AmrMesh.grid_algorithm`. Grid generation algorithm. Valid options are *br* or *tiled*. See *Mesh generation* for details.
- `AmrMesh.box_sorting`. Box sorting algorithm. Valid options are *std*, *morton*, or *shuffle*.
- `AmrMesh.blocking_factor`. Blocking factor.
- `AmrMesh.max_box_size`. Maximum box size.
- `AmrMesh.max_ebis_box`. Maximum box size during EB geometry generation.
- `AmrMesh.ref_rat`. Refinement ratios.
- `AmrMesh.num_ghost`. Number of ghost cells for mesh data.
- `AmrMesh.lsf_ghost`. Number of ghost cells when allocating level-set function on the grid.
- `AmrMesh.eb_ghost`. Number of ghost cells for EB moments.
- `AmrMesh.centroid_sten`. Which centroid interpolation stencils to use. Valid options are *pwl*, *linear*, *taylor*, *lsq*. Only *linear* is guaranteed monotone.
- `AmrMesh.eb_sten`. EB interpolation stencils.
- `AmrMesh.redist_radius`. Redistribution radius.
- `AmrMesh.ghost_interp`. Default ghost cell interpolation type. Valid options are *pwl* or *quad*.
- `AmrMesh.ebcf`. Can be set to false if refinement boundaries do not cross the EB. Valid options are *true* and *false*.

3.4.3 Runtime options

The following options are runtime options for `AmrMesh`:

- `AmrMesh.verbosity`.
- `AmrMesh.fill_ratio`.
- `AmrMesh.irreg_growth`.
- `AmrMesh.buffer_size`.
- `AmrMesh.grid_algorithm`.
- `AmrMesh.box_sorting`.
- `AmrMesh.blocking_factor`.
- `AmrMesh.max_box_size`.

These options only affect the grid generation method and parameters, and are thus only effective after the next regrid.

Note: `chombo-discharge` only supports uniform resolution (i.e., cubic grid cells). I.e. the user must specify consist domain sizes and resolutions.

3.5 CellTagger

The `CellTagger` class is responsible for flagging grid cells for refinement or coarsening. If the user wants to implement a new refinement routine, he will do so by writing a new derived class from `CellTagger`. The `CellTagger` parent class is a stand-alone class - it does not have a view of `AmrMesh`, `Driver`, or `TimeStepper`. Since refinement is intended to be quite general, the user is responsible for providing `CellTagger` with the appropriate depedencies. For example, for streamer simulations we often use the electric field when tagging grid cells, in which case the user should supply either a reference to the electric field, the Poisson solver, or the time stepper.

3.5.1 Design

In `chombo-discharge`, refinement flags live in a data holder called `EBAMRTags` inside of `Driver`. This data is typedef'ed as

```
typedef Vector<RefCountedPtr<LayoutData<DenseIntVectSet> > > EBAMRTags;
```

The `LayoutData<T>` structure can be thought of as a `LevelData<T>` without possibilities for communication. `CellTagger` is an abstract class that the user *must* overwrite.

When the regrid routine enters, the `CellTagger` will be asked to generate the refinement flags through a function

```
bool tagCells(EBAMRTags& a_tags) = 0;
```

Typically, code for refinement looks something like:

```
bool myCelltagger::tagCells(EBAMRTags& a_tags){

   for (int l 0; l <= finestLevel; lvl++){
      const DisjointBoxLayout& dbl = m_amr->get_grids()[lvl];

      for (DataIterator dit(dbl); dit.ok(); ++dit){
          DenseIntVectSet& boxTags = (*a_tags[lvl])[dit()];
          const Box& box           = dbl[dit()];

          for (BoxIterator bit(box); bit.ok(); ++bit){
             const IntVect iv = bit();

             const bool refineThisCell = myRefinementFunction(iv, ...);

             if(refineThisCell){
                boxTags(iv) = true;
             }
             else{
                boxTags(iv) = false;
             }
          }
      }

   return true;
}
```

3.5.2 User interface

To implement a new `CellTagger`, the following functions must be implemented:

```
virtual void regrid() = 0;
virtual void parseOptions() = 0;
virtual void parseRuntimeOptions();
virtual bool tagCells(EBAMRTags& a_tags) = 0;
```

Users can also parse run-time options and have `CellTagger` write to file, by implementing

```
virtual int getNumberOfPlotVariables();
virtual void writePlotData(EBAMRCellData& a_output, Vector<std::string>& a_plotvar_names, int& a_icomp);
```

This is primarily useful for debugging the tracer fields that are used for flagging cells for refinement.

3.5.3 Restrict tagging

It is possible to prevent cell tags in certain regions. The default is simply to add a number of boxes where refinement and coarsening is allowed by specifying a number of boxes in the options file for the cell tagger:

```
MyCellTagger.num_boxes   = 0            # Number of allowed tag boxes (0 = tags allowe everywhere)
MyCellTagger.box1_lo     = 0.0 0.0 0.0  # Only allow tags that fall between
MyCellTagger.box1_hi     = 0.0 0.0 0.0  # these two corners
```

Here, *MyCellTagger* is a placeholder for the name of the class that is used. By adding restrictive boxes, tagging will only be allowed inside the specified box corners `box1_lo` and `box1_hi`. More boxes can be specified by following the same convention, e.g. `box2_lo` and `box2_hi` etc.

3.5.4 Adding a buffer

By default, each MPI rank can only tag grid cells where he owns data. This has been done for performance and communication reasons. Under the hood, the `DenseIntVectSet` is an array of boolean values on a patch which is very fast and simple to communicate with MPI. Adding a grid cell for refinement which lies outside the patch will lead to memory corruptions. It is nonetheless still possible to do this by growing the final generated tags like so:

```
MyCellTagger.buffer = 4 # Add a buffer region around the tagged cells
```

Just before passing the flags into `AmrMesh` grid generation routines, the tagged cells are put in a different data holder (`IntVectSet`) and this data holder *can* contain cells that are outside the patch boundaries.

3.6 GeoCoarsener

The `GeoCoarsener` class can remove refinement flags along a geometric surface. To remove these so-called "geometric tags", the user specifies boxes in space where geometric tags will be removed:

```
GeoCoarsener.num_boxes   = 1            # Number of coarsening boxes (0 = don't coarsen)
GeoCoarsener.box1_lo     = -1 -1 -1     # Lower-left corner
GeoCoarsener.box1_hi     =  1  1  1     # Upper-right corner
GeoCoarsener.box1_lvl    = 0            # Remove tags down to this level.
GeoCoarsener.box1_inv    = false        # Flip removal.
```

If users want more boxes, they can specify it using the same syntax, e.g.

```
GeoCoarsener.num_boxes   = 2            # Number of coarsening boxes (0 = don't coarsen)

GeoCoarsener.box1_lo     = -1 -1 -1     # Lower-left corner
GeoCoarsener.box1_hi     =  1  1  1     # Upper-right corner
GeoCoarsener.box1_lvl    = 0            # Remove tags down to this level.
GeoCoarsener.box1_inv    = false        # Flip removal.

GeoCoarsener.box2_lo     =  2  2  2     # Lower-left corner
GeoCoarsener.box2_hi     =  3  3  3     # Upper-right corner
GeoCoarsener.box2_lvl    = 0            # Remove tags down to this level.
GeoCoarsener.box2_inv    = false        # Flip removal.
```

CHAPTER

FOUR

SOLVERS

4.1 Convection-Diffusion-Reaction

Here, we discuss the discretization of the equation

$$\frac{\partial \phi}{\partial t} + \nabla \cdot \left(\mathbf{v}\phi - D\nabla\phi + \sqrt{2D\phi}\mathbf{Z} \right) = S.$$

We assume that ϕ is discretized by cell-centered averages (note that cell centers may lie inside solid boundaries), and use finite volume methods to construct fluxes in a cut-cells and regular cells. Here, $\mathbf{v}$ indicates a drift velocity, D is the diffusion coefficient, and the term $\sqrt{2D\phi}\mathbf{Z}$ is a stochastic diffusion flux. S is the source term.

Note: Using cell-centered versions ϕ might be problematic for some models since the state is extended outside the valid region. Models might have to recenter the state in order compute e.g. physically meaningful reaction terms in cut-cells.

Source code for the convection-diffusion-reaction solvers reside in `$DISCHARGE_HOME/Source/ConvectionDiffusionReaction`.

4.1.1 Design

CdrSpecies

The `CdrSpecies` class is a supporting class that passes information and initial conditions into `CdrSolver` instances. `CdrSpecies` specifies whether or not the advect-diffusion solver will use only advection, diffusion, both advection and diffusion, or neither. It also specifies initial data, and provides a string identifier to the class (e.g. for identifying output in plot files).

The below code block shows an example of how to instantiate a species. Here, diffusion code is turned off and the initial data is one everywhere.

```
class mySpecies : public CdrSpecies{
public:

   mySpecies(){
      m_mobile    = true;
      m_diffusive = false;
      m_name      = "mySpecies";
   }

   ~mySpecies() = default;

   Real initial_data(const RealVect a_pos, const Real a_time) const override {
      return 1.0;
   }
}
```

Note that you can also deposit computational particles as an initial condition. In this case you need to fill `m_initial_particles`. By default, these are deposited with a nearest-grid-point scheme.

CdrSolver

The `CdrSolver` class contains the interface for solving advection-diffusion-reaction problems. The class is abstract but there are currently two specific implementations of this class. By design `CdrSolver` does not contain any specific advective and diffusive discretization, and these are supposed to be added through inheritance. For example, `CdrTGA` inherits from `CdrSolver` and adds a second order diffusion discretization. It also add multigrid code for performing implicit diffusion. Below that, the classes `CdrGodunov` and `CdrMuscl` inherit everything from `CdrTGA` and also adds in the advective discretization. Thus, adding new advection code is done by inheriting from `CdrTGA` and implementing new advection schemes.

Currently, we mostly use the `CdrGodunov` class which contains a second order accurate discretization with slope limiters, and the advection code for this is distributed by the `Chombo` team. The alternative implementation in `/src/CdrMuscl.H(cpp)` contains a MUSCL implementation with van Leer slope limiting (i.e. much the same as the Chombo code), but it does not include extrapolation in time.

CdrTGA

`CdrTGA` adds second-order accurate implicit diffusion code to `CdrSolver`, but leaves the advection code unimplemented. The class can use either implicit or explicit diffusion using second-order cell-centered stencils. In addition, `CdrTGA` adds two implicit time-integrators, an implicit Euler method and the Twizel-Gumel-Arigu (TGA) method.

CdrGodunov

`CdrGodunov` inherits from `CdrTGA` and adds advection code for Godunov methods. This class borrows from `Chombo` internals (specifically, `EBLevelAdvectIntegrator`) and can do second-order advection with time-extrapolation. For example, when extrapolating cell-centered data to faces, the extrapolation can be done (with Van Leer limiters) in both space and time.

CdrMuscl

`CdrMuscl` adds MUSCL advection code to `CdrTGA`. It uses the same slope limiters as `CdrGodunov` but can not extrapolate in time.

Implementations

To use a `CdrSolver`, one must instantiate either `CdrGodunov` or `CdrMuscl` (which differ only in their treatment of advection). For example:

```
CdrSpecies* spec   = (CdrSpecies*) mySpecies();
CdrSolver* solver = (CdrSolver*)  new CdrGodunov();

solver->set_species(spec);
```

Instantiating `CdrSolver` or `CdrTGA` directly will cause compile-time errors.

Note that if you want to add new advection code to `CdrSolver`, you may inherit from `CdrTGA` and implement new advection routines.

4.1.2 Using CdrSolver

The `CdrSolver` is intended to be used in a method-of-lines context where the user will

1. Fill the solver with relevant data (e.g. velocities, diffusion coefficients, source terms etc.).
2. Call public member functions for explicit advection or diffusion, or for performing implicit diffusion advances.

It is up to the developer to ensure that the solver is filled with appropriate data before calling the public member functions. This would typically look something like this:

```
EBAMRCellData& vel = m_solver->getCellCenteredVelocity();
for (int lvl = 0; lvl <= m_amr->getFinestLevel(); lvl++){
   const DisjointBoxLayout& dbl = m_amr->getGrids()[lvl];

   for (DataIterator dit = dbl.dataIterator(); dit.ok(); ++dit){
      EBCellFAB& patchVel = (*vel[lvl])[dit()];

      // Set velocity of some patch
      callSomeFunction(patchVel);
   }
}

// Compute div(v*phi)
computeDivF(....)
```

There are no time integration algorithms built into the `CdrSolver`, and the user will have to supply these through `TimeStepper`. More complete code is given in the physics module for advection-diffusion problems in `$DISCHARGE_HOME/Physics/AdvectionDiffusion/`. This code is also part of a regression test found in `$DISCHARGE_HOME/Regression/AdvectionDiffusion`.

Setting up the solver

To set up the `CdrSolver`, the following commands are usually included in `time_stepper::setup_solvers()`:

```
// Assume m_solver and m_species are pointers to a CdrSolver and CdrSpecies
m_solver  = RefCountedPtr<CdrSolver>  (new MyCdrSolver());
m_species = RefCountedPtr<CdrSpecies> (new MyCdrSpecies());

// Solver setup
m_solver->setVerbosity(10);
m_solver->setSpecies(m_species);
m_solver->parseOptions();
m_solver->setPhase(phase::gas);
m_solver->setAmr(m_amr);
m_solver->setComputational_geometry(m_compgeom);
m_solver->sanityCheck();
m_solver->allocateInternals();
```

To see an example, the advection-diffusion code in `/physics/AdvectionDiffusion/AdvectionDiffusion_stepper` shows how to set up this particular solver.

Filling the solver

In order to obtain mesh data from the `CdrSolver`, the user should use the following public member functions:

```
EBAMRCellData& getPhi();                            // Return  phi
EBAMRCellData& getSource();                         // Returns S
EBAMRCellData& getCellCenteredVelocity();           // Get cell-centered velocity
EBAMRFluxData& getFaceCenteredDiffusionCoefficient(); // Returns D
EBAMRIVData& getEbFlux();                           // Returns flux at EB
EBAMRIFData& getDomainFlux();                       // Returns flux at domain boundaries
```

To set the drift velocities, the user will fill the *cell-centered* velocities. Interpolation to face-centered transport fluxes are done by `CdrSolver` during the discretization step.

The general way of setting the velocity is to get a direct handle to the velocity data:

```
CdrSolver solver(...);

EBAMRCellData& veloCell = solver.getCellCenteredVelocity();
```

Then, `veloCell` can be filled with the cell-centered velocity. The same procedure goes for the source terms, diffusion coefficients, boundary conditions and so on.

For example, an explicit Euler discretization for the problem $\partial_t \phi = S$ is:

```
CdrSolver* solver;

const Real dt = 1.0;

EBAMRCellData& phi = solver->getPhi();
EBAMRCellData& src = solver->getSource();

DataOps::incr(phi, src, dt);
```

Adjusting output

It is possible to adjust solver output when plotting data. This is done through the input file for the class that you're using (e.g. `/src/CdrSolver/CdrGodunov.options`):

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

Here, you adjust the plotted variables by adding or omitting them from your input script. E.g. if you only want to plot the cell-centered states you would do:

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

4.1.3 Discretization details

Computing explicit divergences

Computing explicit divergences for equations like

$$\frac{\partial \phi}{\partial t} + \nabla \cdot \mathbf{G} = 0$$

is problematic because of the arbitarily small volume fractions of cut cells. In general, we seek a method-of-lines update $\phi^{k+1} = \phi^k - \Delta t \left[\nabla \cdot \mathbf{G}^k\right]$ where $\left[\nabla \cdot \mathbf{G}\right]$ is a stable numerical approximation based on some finite volume approximation.

Pure finite volume methods use

$$\phi^{k+1} = \phi^k - \frac{\Delta t}{\kappa \Delta x^{\mathrm{DIM}}} \int_V \nabla \cdot \mathbf{G} dV, \tag{4.1.1}$$

where κ is the volume fraction of a grid cell, DIM is the spatial dimension and the volume integral is written as discretized surface integral

$$\int_V \nabla \cdot \mathbf{G} dV = \sum_{f \in f(V)} \left(\mathbf{G}_f \cdot \mathbf{n}_f\right) \alpha_f \Delta x^{\mathrm{DIM}-1}.$$

The sum runs over all cell edges (faces in 3D) of the cell where G_f is the flux on the edge centroid and α_f is the edge (face) aperture.

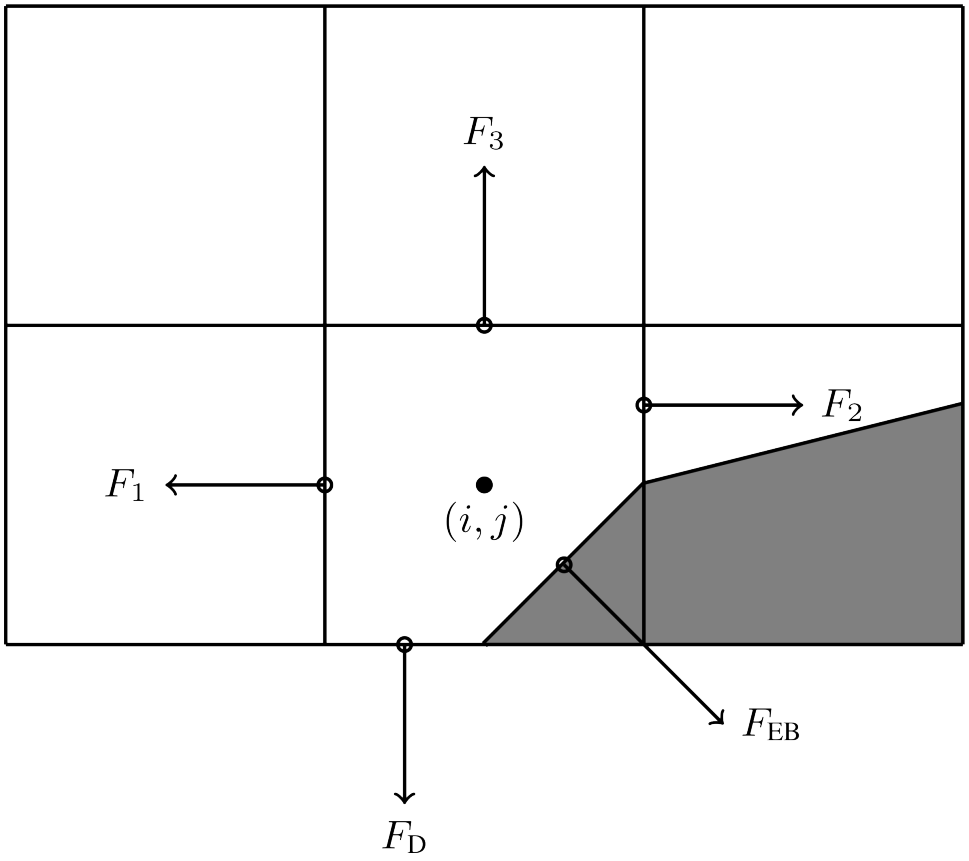

Fig. 4.1.1: Location of centroid fluxes for cut cells.

However, taking $[\nabla \cdot \mathbf{G}^k]$ to be this sum leads to a time step constraint proportional to κ, which can be arbitrarily small. This leads to an unacceptable time step constraint for Eq. 4.1.1. We use the Chombo approach and expand the range of influence of the cut cells in order to stabilize the discretization and allow the use of a normal time step constraint. First, we compute the conservative divergence

$$\kappa_{\mathbf{i}} D_{\mathbf{i}}^c = \sum_f G_f \alpha_f \Delta x^{\mathrm{DIM}-1},$$

where $G_f = \mathbf{G}_f \cdot \mathbf{n}_f$. Next, we compute a non-conservative divergence $D_{\mathbf{i}}^{nc}$

$$D_{\mathbf{i}}^{nc} = \frac{\sum_{\mathbf{j} \in N(\mathbf{i})} \kappa_{\mathbf{j}} D_{\mathbf{i}}^c}{\sum_{\mathbf{j} \in N(\mathbf{i})} \kappa_{\mathbf{j}}}$$

where $N(\mathbf{i})$ indicates some neighborhood of cells around cell $\mathbf{i}$. Next, we compute a hybridization of the divergences,

$$D_{\mathbf{i}}^H = \kappa_{\mathbf{i}} D_{\mathbf{i}}^c + (1 - \kappa_{\mathbf{i}}) D_{\mathbf{i}}^{nc},$$

and perform an intermediate update

$$\phi_{\mathbf{i}}^{k+1} = \phi_{\mathbf{i}}^k - \Delta t D_{\mathbf{i}}^H.$$

The hybrid divergence update fails to conserve mass by an amount $\delta M_{\mathbf{i}} = \kappa_{\mathbf{i}} (1 - \kappa_{\mathbf{i}}) (D_{\mathbf{i}}^c - D_{\mathbf{i}}^{nc})$. In order to main overall conservation, the excess mass is redistributed into neighboring grid cells. Let $\delta M_{\mathbf{i},\mathbf{j}}$ be the redistributed mass from $\mathbf{j}$ to $\mathbf{i}$ where

$$\delta M_{\mathbf{i}} = \sum_{\mathbf{j} \in N(\mathbf{i})} \delta M_{\mathbf{i},\mathbf{i}}.$$

This mass is used as a local correction in the vicinity of the cut cells, i.e.

$$\phi_{\mathbf{i}}^{k+1} \rightarrow \phi_{\mathbf{i}}^{k+1} + \delta M_{\mathbf{j}\in N(\mathbf{i}),\mathbf{i}},$$

where $\delta M_{\mathbf{j}\in N(\mathbf{i}),\mathbf{i}}$ is the total mass redistributed to cell $\mathbf{i}$ from the other cells. After these steps, we define

$$\left[\nabla\cdot\mathbf{G}^{k}\right]_{\mathbf{i}} \equiv \frac{1}{\Delta t}\left(\phi_{\mathbf{i}}^{k+1} - \phi_{\mathbf{i}}^{k}\right)$$

Numerically, the above steps for computing a conservative divergence of a one-component flux $\mathbf{G}$ are implemented in the convection-diffusion-reaction solvers, which also respects boundary conditions (e.g. charge injection). The user will need to call the function

```
virtual void CdrSolver::computeDivG(EBAMRCellData& a_divG, EBAMRFluxData& a_G, const EBAMRIVData& a_ebG)
```

where `a_G` is the numerical representation of $\mathbf{G}$ over the cut-cell AMR hierarchy and must be stored on cell-centered faces, and `a_ebG` is the flux on the embedded boundary. The above steps are performed by interpolating `a_G` to face centroids in the cut cells for computing the conservative divergence, and the remaining steps are then performed successively. The result is put in `a_divG`.

Note that when refinement boundaries intersect with embedded boundaries, the redistribution process is far more complicated since it needs to account for mass that moves over refinement boundaries. These additional complicated are taken care of inside `a_divG`, but are not discussed in detail here.

Note: Mass redistribution has the effect of not being monotone and thus not TVD, and the discretization order is formally $\mathcal{O}(\Delta x)$. If negative densities are a problem, the `CdrSolver` has an option to use mass-weighted redistribution in order to redistribute mass in the neighborhood of the cut cells. The default is false, in which case the redistribution uses volume-weighted redistribution.

Explicit advection

Scalar advection updates follows the computation of the explicit divergence discussed in *Computing explicit divergences*. The face-centered fluxes $\mathbf{G} = \phi\mathbf{v}$ are computed by instantiation classes for the convection-diffusion-reaction solvers. The function signature for explicit advection is

```
void computeDivF(EBAMRCellData& a_divF, const EBAMRCellData& a_state, const Real a_extrap_dt)
```

where the face-centered fluxes are computed by using the velocities and boundary conditions that reside in the solver, and result is put in `a_divF` using the procedure outlined above. For example, in order to perform an advective advance over a time step Δt, one would perform the following:

```
CdrSolver* solver;

const Real dt = 1.0;

EBAMRCellData& phi  = solver->getPhi();     // Cell-centered state
EBAMRCellData& divF = solver->getScratch(); // Scratch storage in solver
solver->computeDivF(divF, phi, 0.0);        // Computes divF
DataOps:incr(phi, divF, -dt);               // makes phi -> phi - dt*divF
```

Explicit diffusion

Explicit diffusion is performed in much the same way as implicit advection, with the exception that the general flux $\mathbf{G} = D\nabla\phi$ is computed by using centered differences on face centers. The function signature for explicit diffusion is

```
void computeDivD(EBAMRCellData& a_divF, const EBAMRCellData& a_state)
```

and we increment in the same way as for explicit advection:

```
CdrSolver* solver;

const Real dt = 1.0;

EBAMRCellData& phi  = solver->getPhi();     // Cell-centered state
EBAMRCellData& divD = solver->getScratch(); // Scratch storage in solver
solver->computeDivF(divD, phi, 0.0);        // Computes divD
DataOps:incr(phi, divD, dt);                // makes phi -> phi + dt*divD
```

Explicit advection-diffusion

There is also functionality for aggregating explicit advection and diffusion advances. The reason for this is that the cut-cell overhead is only applied once on the combined flux $\phi\mathbf{v} - D\nabla\phi$ rather than on the individual fluxes. For non-split methods this leads to some performance improvement since the interpolation of fluxes on cut-cell faces only needs to be performed once. The signature for this is precisely the same as for explicit advection only:

```
void computeDivJ(EBAMRCellData& a_divJ, const EBAMRCellData& a_state, const Real a_extrapDt)
```

where the face-centered fluxes are computed by using the velocities and boundary conditions that reside in the solver, and result is put in `a_divF`. For example, in order to perform an advective advance over a time step Δt, one would perform the following:

```
const Real dt = 1.0;

EBAMRCellData& phi  = solver->getPhi();     // Cell-centered state
EBAMRCellData& divJ = solver->getScratch(); // Scratch storage in solver
solver->computeDivJ(divJ, phi, 0.0);        // Computes divD
DataOps:incr(phi, divJ, -dt);               // makes phi -> phi - dt*divJ
```

Often, time integrators have the option of using implicit or explicit diffusion. If the time-evolution is not split (i.e. not using a Strang or Godunov splitting), the integrators will often call `computeDivJ` rather separately calling `computeDivF` and `computeDivD`. If you had a split-step Godunov method, the above procedure for a forward Euler method for both parts would be:

```
CdrSolver* solver;

const Real dt = 1.0;

solver->computeDivF(divF, phi, 0.0); // Computes divF = div(n*phi)
DataOps:incr(phi, divF, -dt);        // makes phi -> phi - dt*divF

solver->computeDivD(divD, phi);      // Computes divD = div(D*nabla(phi))
DataOps:incr(phi, divD, dt);         // makes phi -> phi + dt*divD
```

However, the cut-cell redistribution dance (flux interpolation, hybrid divergence, and redistribution) would be performed twice.

Implicit diffusion

Occasionally, the use of implicit diffusion is necessary. The convection-diffusion-reaction solvers support two basic diffusion solves: Backward Euler and the Twizel-Gumel-Arigu (TGA) methods. The function signatures for these are

```
void advanceEuler(EBAMRCellData& phiNew, const EBAMRCellData& phiOld, const EBAMRCellData& src, const Real dt)
void advanceTGA(  EBAMRCellData& phiNew, const EBAMRCellData& phiOld, const EBAMRCellData& src, const Real dt)

void advanceEuler(EBAMRCellData& phiNew, const EBAMRCellData& phiOld, const Real dt)
void advanceTGA(  EBAMRCellData& phiNew, const EBAMRCellData& phiOld, const Real dt)
```

where `phiNew` is the state at the new time $t + \Delta t$, `phiOld` is the state at time t and `src` is the source term which strictly speaking should be centered at time $t + \Delta t$ for the Euler update and at time $t + \Delta t/2$ for the TGA update. This may or may not be possible for your particular problem.

For example, performing a split step Godunov method for advection-diffusion is as simple as:

```
solver->computeDivF(divF, phi, 0.0); // Computes divF = div(n*phi)
DataOps:incr(phi, divF, -dt);        // makes phi -> phi - dt*divF
solver->redistribute_negative(phi);   // Redist negative mass in cut cells

DataOps::copy(phiOld, phi);            // Copy state
solver->advanceEuler(phi, phiOld, dt); // Backward Euler diffusion solve
```

Note: The backward Euler method can easily by turned into a Crank-Nicholson method by modifying the source term and time step.

Adding a stochastic flux

It is possible to add a stochastic flux through the public member functions of `CdrSolver` in the odd case that one wants to use fluctuating hydrodynamics (FHD). This is done by calling a function that computes the term $\sqrt{2D\phi}\mathbf{Z}$:

```
void gwnDiffusionSource(EBAMRCellData& a_noiseSource, const EBAMRCellData& a_cellPhi);
```

When FHD is used, there is no guarantee that the evolution leads to non-negative values. We do our best to ensure that the stochastic flux is turned off when $\phi\Delta V$ approaches 0 by computing the face-centered states for the stochastic term using an arithmetic mean that goes to zero as ϕ approaches 0.

In the above function, `a_ransource` can be used directly in a MOL context, e.g.

```
solver->computeDivF(divF, phi, 0.0); // Computes divF = div(n*phi)
DataOps:incr(phi, divF, -dt);        // makes phi -> phi - dt*divF

solver->gwnDiffusionSource(ransource, phi);  // Compute stochastic flux
DataOps::copy(phiOld, phi);                  // phiOld = phi - dt*divF
DataOps::incr(phiOld, ransource, a_dt);      // phiOld = phi - dt*divF + dt*sqrt(2D*phi)Z
solver->advanceEuler(phi, phiOld, dt);       // Backward Euler diffusion solve.
```

4.1.4 Example application

An example application of usage of the `CdrSolver` is found in *Advection diffusion*.

4.2 Electrostatics

Here, we discuss the discretization of the equation

$$\nabla \cdot (\epsilon_r \nabla \Phi) = -\frac{\rho}{\epsilon_0} \tag{4.2.1}$$

where Φ is the electric potential, ρ is the space charge density and ϵ_0 is the vacuum permittivity. The relative permittivity is $\epsilon_r = \epsilon_r(\mathbf{x})$ and can additionally be discontinuous at gas-dielectric interfaces.

Note: All current electrostatic field solvers solve for the potential at the cell center (not the cell centroid). The code for the electrostatics solver is given in `/Source/Electrostatics` and `/Source/Elliptic`.

4.2.1 FieldSolver

`FieldSolver` is an abstract class for electrostatic solves in an EB context and contains most routines required for setting up and solving electrostatic problems. `FieldSolver` can solve over three phases, gas, dielectric, and electrode, and thus it is uses `MFAMRCellData` functionality where data is defined over multiple phases (see *Mesh data*).

Note that in order to separate the electrostatic solver interface from the implementation, `FieldSolver` is a pure class without knowledge of numerical discretizations. Currently, our only supported implementation is `FieldSolverMultigrid` (see *FieldSolverMultigrid*).

On gas-dielectric interfaces we enforce an extra equation

$$\epsilon_1 \partial_{n_1} \Phi + \epsilon_2 \partial_{n_2} \Phi = \sigma/\epsilon_0 \tag{4.2.2}$$

where $\mathbf{n}_1 = -\mathbf{n}_2$ are the normal vectors pointing out of each side of the interface, and σ is the surface charge density.

We point out that this equation can be enforced in various formats. The most common case is that $\partial_n \Phi$ are free parameters and σ is a fixed parameter. However, we *can* also fix $\partial_n \Phi$ on either side of the boundary and have σ as a free parameter. When using `FieldSolverMultigrid` (see *FieldSolverMultigrid*), users can choose between these two natural boundary conditions, see *EB boundary conditions*.

4.2.2 Using FieldSolver

Using the `FieldSolver` is usually straightforward by first constructing the solver and then parsing the class options. Creating a solver is usually done by means of a pointer cast:

```
RefCountedPtr<FieldSolver> fieldSolver = RefCountedPtr<FieldSolver> (new FieldSolverMultigrid());
```

In addition, one must parse run-time options to the class, provide the `AmrMesh` and `ComputationalGeometry` instances, and set the initial conditions. This is done as follows:

```
RefCountedPtr<AmrMesh> amr;
RefCountedPtr<ComputationalGeometry> geo;

std::function<Real(const Real)> voltage;

fieldSolver->parseOptions();                // Parse class options
fieldSolver->setAmr(amr);                   // Set amr - we assume that `amr` is an object
fieldSolver->setComputationalGeometry(geo); // Set the computational geometry
fieldSolver->allocateInternals();           // Allocate storage for potential etc.
fieldSolver->setVoltage(voltage);           // Set the voltage
```

The argument in the function `setVoltage(...)` is a function pointer of the type:

```
Real voltage(const Real a_time)
```

This allows setting a time-dependent voltage on electrodes and domain boundaries. As shown above, one can also use `std::function<Real(const Real)>` or lambdas to set the voltage. E.g.,

```
FieldSolver* fieldSolver;

Real myVoltage = [] (const Real a_time) -> Real {
   return 1.0*a_time;
};

fieldSolver->setVoltage(myVoltage);
```

The electrostatic solver `chombo-discharge` has a lot of supporting functionality, but essentially relies on only one critical function: Solving for the potential. This is done by the member function

```
bool FieldSolver::solve(MFAMRCellData& phi, const MFAMRCellData& rho, const EBAMRIVData& sigma);
```

where `phi` is the resulting potential that was computing with the space charge density `rho` and surface charge density `sigma`. This function is implemented by *FieldSolverMultigrid*.

4.2.3 Domain boundary conditions

Domain boundary conditions for the solver must be set by the user through an input script, whereas the boundary conditions on internal surfaces are Dirichlet by default. Note that on multifluid-boundaries the boundary condition is enforced by the conventional matching boundary condition that follows from Gauss` law.

General format

The most general form of setting domain boundary conditions for `FieldSolver` is to specify a boundary condition *type* (e.g., Dirichlet) together with a function specifying the value. Domain boundary condition *types* are parsed through a member function `FieldSolver::parseDomainBc`. This function will read string identifiers from the input script, and these identifiers are either in the format `<string> <float>` (simplified format) or in the format `<string>` (general format). For setting general types of Neumann or Dirichlet BCs on the domain sides, one will specify

```
FieldSolverMultigrid.bc_x_low  = dirichlet_custom
FieldSolverMultigrid.bc_x_high = dirichlet_neumann
```

Unfortunately, due to the many degrees of freedom in setting domain boundary conditions, the procedure is a bit convoluted. We first explain the general procedure.

`FieldSolver` will always set individual space-time functions on each domain side, and these functions are always in the form

```
std::function<Real(const RealVect a_position, const Real a_time)> bcFunction;
```

To set a domain boundary condition function on a side, one can use the following member function:

```
void FieldSolver::setDomainSideBcFunction(const int a_dir,
                                          const Side::LoHiSide a_side,
                                          const std::function<Real(const RealVect a_position, const Real a_time)>);
```

For a general way of setting the function value on the domain side, one will use the above function together with an identifier `dirichlet_custom` or `neumann_custom` in the input script. This identifier simply tells `FieldSolver` to use that function to either specifiy Φ or $\partial_n \Phi$ on the boundary. These functions are then directly processed by the numerical discretizations.

Note: On construction, `FieldSolver` will set all the domain boundary condition functions to a constant of one (because the functions need to be populated).

Simplified format

`FieldSolver` also supports a simplified method of setting the domain boundary conditions, in which case the user will specify Neumann or Dirichlet values (rather than functions) for each domain side. These values are usually, but not necessarily, constant values.

In this case one will use an identifier `<string> <float>` in the input script, like so:

```
FieldSolverMultigrid.bc_x_low  = neumann   0.0
FieldSolverMultigrid.bc_x_high = dirichlet 1.0
```

The floating point number has a slightly different interpretation for the two types of BCs. Moreover, when using the simplified format the function specified through `setDomainSideBcFunction` will be used as a multiplier rather than being parsed directly into the numerical discretization. Although this may *seem* more involved, this procedure is usually easier to use when setting constant Neumann/Dirichlet values on the domain boundaries. It also automatically provides a link between a specified voltage wave form and the boundary conditions (unlike the general format, where the user must supply that link themselves).

Dirichlet

When using simplified parsing of Dirichlet domain BCs, `FieldSolver` will generate and parse a different function into the discretizations. This function is *not* the same function as that which is parsed through `setDomainSideBcFunction`. In C++ pseudo-code, this function is in the format

```
Real dirichletFraction;

auto func = [&func, ...](const RealVect a_pos, const Real a_time) -> Real {
   return func(a_pos, a_time) * voltage(a_time) * dirichletFraction;
};
```

where `voltage` is the voltage wave form specified through `FieldSolver::setVoltage`, and `dirichletFraction` is a placeholder for the floating point number specified in the input script, i.e. the floating point number in the input option. The function `func(a_pos, a_time)` *is* the space-time function set through `setDomainSideBcFunction`. Recall that, by default, this function is set to one so that the default voltage that is parsed into the numerical discretization is simply the specified voltage multiplied by the specified fraction in the input script. For example, using

```
FieldSolverMultigrid.bc_y_low  = dirichlet 0.0
FieldSolverMultigrid.bc_y_high = dirichlet 1.0
```

will the set voltage on the lower y-plane to ground and the voltage on the upper y-plane to a live voltage.

In order to set the voltage on the domain side to also be spatially dependent, one can either use `dirichlet_custom` as an input option, or still `dirichlet <float>` and set a different multiplier on the domain edge (face). As an example. one can specify `bc_y_high = dirichlet 1.234` in the input script AND set the multiplier on the wall as follows:

```
auto wallFunc = [](const RealVect a_pos, const Real a_time) -> Real {
   const Real y = a_pos[1];
   return 1.0 - y;
};

fieldSolver->setDomainSideBcFunction(1, Side::Hi, wallFunc);
```

Note that this will essentially parse a voltage of

$$V(\mathbf{x}, t) = 1.234(1 - y)V(t)$$

on the upper y-plane.

Neumann

When using simplified parsing of Neumann boundary conditions, the procedure is precisely like that for Dirichlet boundary conditions *except* that multiplication by the voltage wave form is not made. I.e. the boundary condition function that is passed into the numerical discretization is

```
Real neumannFraction;

auto func = [&func, ...](const RealVect a_pos, const Real a_time) -> Real {
   return func(a_pos, a_time) * neumannFraction;
};
```

Note that since `func` is initialized to one, the floating point number in the input option directly specifies the value of $\partial_n \Phi$.

4.2.4 EB boundary conditions

Electrodes

For the current `FieldSolver` the natural BC at the EB is Dirichlet with a specified voltage, whereas on dielectrics we enforce Eq. 4.2.2. The voltage on the electrodes are automatically retrieved from the specified voltages on the electrodes in the geometry being used (see `ComputationalGeometry`). The exception to this is that while `ComputationalGeometry` specifies that an electrode will be at some fraction of a specified voltage, `FieldSolverMultigrid` uses this fraction *and* the specified voltage wave form in `setVoltage`.

To understand how the voltage on the electrode is being set, we first remark that our implementation uses a completely general specification of the voltage on each electrode in both space and time. This voltage has the form

$$V_i = V_i\left(\mathbf{x}, t\right).$$

where V_i is the voltage on electrode i. It is possible to interact with this function directly, doing through all electrodes and setting the electrode to be spatially and temporally varying. The member function that does this is

```
void FieldSolver::setElectrodeDirichletFunction(const int a_electrode,
                                                const ElectrostaticEbBc::BcFunction& a_function);
```

Here, the type `ElectrostaticEbBc::BcFunction` is just an alias:

```
using ElectrodestaticEbBc::BcFunction = std::function<Real(const RealVect a_position, const Real a_time)>;
```

The voltage on an electrode i could thus be set as

```
int electrode;

auto myElectrodeVoltage = [](const RealVect a_position, const Real a_time) -> Real{
    return 1.0;
};

fieldSolver->setElectrodeDirichletFunction(electrode, myElectrodeVoltage);
```

where the return value can be replaced by the user' function.

In the majority of cases the voltage on electrodes is either a live voltage or ground. Thus, although the above format is a general way of setting the voltage individually on each electrode (in both space and time) `FieldSolver` supports a simpler way of generating these voltage waveforms. When `FieldSolver` is instantiated, it will interally generate these functions through simplified expression such that the user only needs to set a single wave form that applies to all electrodes. The voltages that are set on the various electrodes are thus in the form:

```
int electrode;
Real voltageFraction;
std::function<Real(const Real a_time)> voltageWaveForm;

auto defaultElectrodeVoltage = [...](const RealVect a_position, const Real a_time) -> Real{
   return voltageFraction * voltageWaveForm(a_time);
};

fieldSolver->setElectrodeDirichletFunction(electrode, defaultElectrodeVoltage);
```

Thus, the default voltage which is set on an electrode is the voltage *fraction* specified on the electrodes (in `ComputationalGeometry`) multiplied by a voltage wave form (specified by `FieldSolver::setVoltage`).

Dielectrics

As mentioned above, on dielectric interfaces the user can choose to specify which "form" of Eq. 4.2.2 to solve. If the user wants the natural form in which the surface charge is the free parameter, he can specify

```
FieldSolverMultigrid.which_jump = natural
```

To use the other format (in which one of the fluxes is specified), use

```
FieldSolverMultigrid.which_jump = saturation_charge
```

Note: The `saturation_charge` option will set the derivative of $\partial_n \Phi$ to zero on the gas side. Support for setting $\partial_n \Phi$ to a specified (e.g., non-zero) value on either side is missing, but is straightforward to implement.

4.2.5 FieldSolverMultigrid

`FieldSolverMultigrid` implements a multigrid routine for solving Eq. 4.2.1, and is currently the only implementation of `FieldSolver`.

The discretization used by `FieldSolverMultigrid` is described in *Linear solvers*. The underlying solver type is a Helmholtz solver, but `FieldSolverMultigrid` considers only the Laplacian term.

Solver configuration

`FieldSolverMultigrid` has a number of switches for determining how it operates. Some of these switches are intended for parsing boundary conditions, whereas others are settings for operating multigrid or for I/O. The current list of configuration options are indicated below

```
# ================================================================================================
# FieldSolverMultigrid class options
# ================================================================================================
FieldSolverMultigrid.verbosity         = -1               # Class verbosity
FieldSolverMultigrid.jump_bc           = natural          # Jump BC type ('natural' or 'saturation_charge')
FieldSolverMultigrid.bc.x.lo           = dirichlet 0.0    # Bc type (see docs)
FieldSolverMultigrid.bc.x.hi           = dirichlet 0.0    # Bc type (see docs)
FieldSolverMultigrid.bc.y.lo           = dirichlet 0.0    # Bc type (see docs)
FieldSolverMultigrid.bc.y.hi           = dirichlet 0.0    # Bc type (see docs)
FieldSolverMultigrid.bc.z.lo           = dirichlet 0.0    # Bc type (see docs)
FieldSolverMultigrid.bc.z.hi           = dirichlet 0.0    # Bc type (see docs)
FieldSolverMultigrid.plt_vars          = phi rho E        # Plot variables. Possible vars are 'phi', 'rho', 'E', 'res', 'sigma'
FieldSolverMultigrid.kappa_source      = true             # Volume weighted space charge density or not (depends on algorithm)

FieldSolverMultigrid.gmg_verbosity     = -1               # GMG verbosity
FieldSolverMultigrid.gmg_pre_smooth    = 12               # Number of relaxations in downsweep
FieldSolverMultigrid.gmg_post_smooth   = 12               # Number of relaxations in upsweep
FieldSolverMultigrid.gmg_bott_smooth   = 12               # Number of at bottom level (before dropping to bottom solver)
```

(continues on next page)

(continued from previous page)

```
FieldSolverMultigrid.gmg_min_iter      = 5              # Minimum number of iterations
FieldSolverMultigrid.gmg_max_iter      = 32             # Maximum number of iterations
FieldSolverMultigrid.gmg_exit_tol      = 1.E-10         # Residue tolerance
FieldSolverMultigrid.gmg_exit_hang     = 0.2            # Solver hang
FieldSolverMultigrid.gmg_min_cells     = 16             # Bottom drop
FieldSolverMultigrid.gmg_bc_order      = 2              # Boundary condition order for multigrid
FieldSolverMultigrid.gmg_bc_weight     = 2              # Boundary condition weights (for least squares)
FieldSolverMultigrid.gmg_jump_order    = 2              # Boundary condition order for jump conditions
FieldSolverMultigrid.gmg_jump_weight   = 2              # Boundary condition weight for jump conditions (for least squares)
FieldSolverMultigrid.gmg_bottom_solver = bicgstab       # Bottom solver type. 'simple', 'bicgstab', or 'gmres'
FieldSolverMultigrid.gmg_cycle         = vcycle         # Cycle type. Only 'vcycle' supported for now.
FieldSolverMultigrid.gmg_smoother      = red_black      # Relaxation type. 'jacobi', 'multi_color', or 'red_black'
```

Note that *all* options pertaining to IO or multigrid are run-time configurable (see *Run-time configurations*).

Setting boundary conditions

The flags that are in the format `bc.coord.side` (e.g., `bc.x.low`) parse the domain boundary condition type to the solver. See *Domain boundary conditions* for details.

The flag `jump_bc` indicates how the dielectric jump condition is enforced. See *Dielectrics* for additional details.

Note: Currently, we only solve the dielectric jump condition on gas-dielectric interfaces and dielectric-dielectric interfaces are not supported. If you want to use numerical mock-ups of dielectric-dielectric interfaces, you can change ϵ_r inside a dielectric, but note that the dielectric boundary condition $\partial_{n_1}\Phi + \partial_{n_2}\Phi = \sigma/\epsilon_0$ is *not* solved in this case.

Algorithmic adjustments

By default, the Helmholtz operator uses a diagonally weighting of the operator using the volume fraction as weight. This means that the quantity that is passed into `AMRMultiGrid` should be weighted by the volume fraction to avoid the small-cell problem of EB grids. The flag `kappa_source` indicates whether or not we should multiply the right-hand side by the volume fraction before passing it into the solver routine. If this flag is set to `false`, it is an indication that the user has taken responsibility to perform this weighting prior to calling `FieldSolver::solve(...)`. If this flag is set to `true`, `FieldSolverMultigrid` will perform the multiplication before the multigrid solve.

Tuning multigrid performance

Multigrid operates by coarsening the solution (and the geometry with it) on a hierarchy of grid levels, and smoothing the solution on each level. There are a number of factors that influence the multigrid performance. Often the most critical factors are the radius of the cut-cell stencils and how far multigrid is allowed to coarsen. In addition, the multigrid convergence is improved by increasing the number of smoothings per grid level (up to a certain point), as well as the type of smoother and bottom solver being used. We explain these options below:

- `FieldSolverMultigrid.gmg_verbosity`. Controls the multigrid verbosity. Setting it to a number > 0 will print multigrid convergence information.
- `FieldSolverMultigrid.gmg_pre_smooth`. Controls the number of relaxations on each level during multigrid downsweeps.
- `FieldSolverMultigrid.gmg_post_smooth`. Controls the number of relaxations on each level during multigrid upsweeps.
- `FieldSolverMultigrid.gmg_bott_smooth`. Controls the number of relaxations before entering the bottom solve.
- `FieldSolverMultigrid.gmg_min_iter`. Sets the minimum number of iterations that multigrid will perform.

- `FieldSolverMultigrid.gmg_max_iter`. Sets the maximum number of iterations that multigrid will perform.
- `FieldSolverMultigrid.gmg_exit_tol`. Sets the exit tolerance for multigrid. Multigrid will exit the iterations if $r < \lambda r_0$ where λ is the specified tolerance, $r = |L\Phi - \rho|$ is the residual and r_0 is the residual for $\Phi = 0$.
- `FieldSolverMultigrid.gmg_exit_hang`. Sets the minimum permitted reduction in the convergence rate before exiting multigrid. Letting r^k be the residual after k multigrid cycles, multigrid will abort if the residual between levels is not reduce by at least a factor of $r^{k+1} < (1-h)r^k$, where h is the "hang" factor.
- `FieldSolverMultigrid.gmg_min_cells`. Sets the minimum amount of cells along any coordinate direction for coarsened levels. Note that this will control how far multigrid will coarsen. Setting a number `gmg_min_cells = 16` will terminate multigrid coarsening when the domain has 16 cells in any of the coordinate direction.
- `FieldSolverMultigrid.gmg_bc_order`. Sets the stencil order for Dirichlet boundary conditions (on electrodes). Note that this is also the stencil radius.
- `FieldSolverMultigrid.gmg_bc_weight`. Sets the least squares stencil weighting factor for least squares gradient reconstruction on EBs. See *Least squares* for details.
- `FieldSolverMultigrid.gmg_jump_order`. Sets the stencil order when performing least squares gradient reconstruction on dielectric interfaces. Note that this is also the stencil radius.
- `FieldSolverMultigrid.gmg_jump_weight`. Sets the least squares stencil weighting factor for least squares gradient reconstruction on dielectric interfaces. See *Least squares* for details.
- `FieldSolverMultigrid.gmg_bottom_solver`. Sets the bottom solver type.
- `FieldSolverMultigrid.gmg_cycle`. Sets the multigrid method. Currently, only V-cycles are supported.
- `FieldSolverMultigrid.gmg_smoother`. Sets the multigrid smoother.

Note: When setting the bottom solver (which by default is a biconjugate gradient stabilized method) to a regular smoother, one must also specify the number of smoothings to perform. E.g., `FieldSolverMultigrid.gmg_bottom_solver = simple 64`. Setting the bottom solver to `simple` without specifying the number of smoothings that will be performed will issue a run-time error.

Adjusting output

The user may plot the potential, the space charge, the electric, and the GMG residue as follows:

```
FieldSolverMultigrid.plt_vars  = phi rho E res     # Plot variables. Possible vars are 'phi', 'rho', 'E', 'res'
```

4.2.6 Frequency dependent permittivity

Frequency-dependent permittivities are fundamentally supported by the `chombo-discharge` elliptic discretization but none of the solvers implement it. Recall that the polarization (in frequency space) is

$$\mathbf{P}(\omega) = \epsilon_0 \chi(\omega) \mathbf{E}(\omega),$$

where $\chi(\omega)$ is the dielectric susceptibility.

There are two forms that `chombo-discharge` can support frequency dependent permittivities; through convolution or through auxiliary differential equations (ADEs).

Convolution approach

In the time domain, the displacement field is.

$$\mathbf{D}(t_k) = \epsilon_0 \mathbf{E}(t_k) + \epsilon_0 \int_0^{t_k} \chi(t)\mathbf{E}(t_k - t)\mathrm{d}t.$$

There are various forms of discretizing the integral. E.g. with the trapezoidal rule then

$$\begin{aligned}
\int_0^{t_k} \chi(t)\mathbf{E}(t-t)\mathrm{d}t &= \sum_{n=0}^{k-1} \int_{t_n}^{t_{n+1}} \chi(t)\mathbf{E}(t_k - t)\mathrm{d}t \\
&\approx \frac{1}{2}\sum_{n=0}^{k-1} \Delta t_n \left[\chi(t_n)\mathbf{E}(t_k - t_n) + \chi(t_{n+1})\mathbf{E}(t_k - t_{n+1})\right] \\
&= \frac{\Delta t_0}{2}\chi_0 \mathbf{E}(t_k) + \frac{1}{2}\sum_{n=1}^{k-1} \Delta t_n \chi_n \mathbf{E}(t_k - t_n) + \frac{1}{2}\sum_{n=0}^{k-1} \Delta t_n \chi_{n+1}\mathbf{E}(t_k - t_{n+1})
\end{aligned}$$

The Gauss law becomes

$$\begin{aligned}
\nabla \cdot \left[\left(1 + \frac{\chi_0 \Delta t_0}{2}\right)\mathbf{E}(t_k)\right] &= \frac{\rho(t_k)}{\epsilon_0} \\
&\quad - \nabla \cdot \left[\frac{1}{2}\sum_{n=1}^{k-1} \Delta t_n \chi_n \mathbf{E}(t_k - t_n) + \frac{1}{2}\sum_{n=0}^{k-1} \Delta t_n \chi_{n+1}\mathbf{E}(t_k - t_{n+1})\right].
\end{aligned}$$

Note that the dispersion enters as an extra term on the right-hand side, emulating a space charge. Unfortunately, inclusion of dispersion means that we must store $\mathbf{E}(t_n)$ for all previous time steps.

Auxiliary differential equation

With the ADE approach we seek a solution to $\mathbf{P}(\omega) = \epsilon_0 \chi(\omega)\mathbf{E}(\omega)$ in the form

$$\sum_k a_k (i\omega)^k \mathbf{P}(\omega) = \epsilon_0 \mathbf{E}(\omega),$$

where $\sum a_k (i\omega)^k$ is the Taylor series for $1/\chi(\omega)$. This can be written as a partial differential equation

$$\sum_k a_k \partial_t^k \mathbf{P}(t) = \epsilon_0 \mathbf{E}(t).$$

This equation can be discretized using finite differences, and centering the solution on t_k with backward differences yields an expression

$$\mathbf{P}^k = \epsilon_0 C_0^k \mathbf{E}^k - \sum_{m>0} C_m^k \mathbf{P}^{k-m}.$$

where C_k are stencil coefficients to be worked out for each case. The displacement field $\mathbf{D}^k = \epsilon_0 \mathbf{E}^k + \mathbf{P}^k$ is then

$$\mathbf{D} = \epsilon_0 (1 + C_0^k)\mathbf{E} - \sum_{m>0} C_m^k \mathbf{P}^{k-m}.$$

The Gauss law yields

$$\nabla \cdot \left[\left(1 + C_0^k\right)\mathbf{E}^k\right] = \frac{\rho}{\epsilon_0} - \frac{1}{\epsilon_0}\nabla \cdot \sum_{m>0} C_m^k \mathbf{P}^{k-m}.$$

Unlike the convolution approach, this only requires storing terms required for the ADE description. This depends both on the order of the ADE, as well as it's discretization. Normally, the ADE is a low-order PDE and a few terms are sufficient.

4.2.7 Limitations

Warning: There is currently a bug where having a dielectric interface align *completely* with a grid face will cause the cell to be identified as an electrode EB. This bug is due to the way `Chombo` handles cut-cells that align completely with a grid face. In this case the cell with volume fraction $\kappa = 1$ will be identified as an irregular cell. For the opposite phase (i.e., viewing the grids from inside the boundary) the situation is opposite and thus the two "matching cells" can appear in different grid patches. A fix for this is underway. In the meantime, a sufficient workaround is simply to displace the dielectric slightly away from the interface (any non-zero displacement will do).

4.2.8 Example application

An example application of usage of the `FieldSolver` is found in *Electrostatics*.

4.3 Mesh ODE solver

4.4 Radiative transfer

Radiative transfer is supported in the diffusion (i.e. Eddington or Helmholtz) approximation and with Monte Carlo sampling of discrete photons. The solvers share a common interface (a parent class), but note that the radiative transfer equation is inherently deterministic while Monte Carlo photon transport is inherently stochastic. The diffusion approximation relies on solving an elliptic equation in the stationary case and a parabolic equation in the time-dependent case, while the Monte-Carlo approach solves solves for fully transient or ''stationary" transport.

4.4.1 RtSpecies

The class `RtSpecies` is an abstract base class for parsing necessary information into radiative transfer solvers. When creating a radiative transfer solver one will need to pass in a pointer to `RtSpecies` such that the solvers can look up the required infromation. Currently, `RtSpecies` is a lightweight class where the user needs to implement the function

```
virtual Real RtSpecies::getAbsorptionCoefficient(const RealVect a_pos) const = 0;
```

The absorption coefficient is used in the diffusion (see *Diffusion approximation*) and Monte Carlo (see *Monte Carlo methods*) solvers.

One can also assign a name to the species through the member variable `RtSpecies::m_name`.

4.4.2 RtSolver

`RtSolver` is the base class for encapsulating a radiative transfer solver. The source code for the solver is located in `$DISCHARGE_HOME/Source/RadiativeTransfer` and it is a fairly lightweight abstract class. As with other solvers, `RtSolver` can use a specified *Realm*.

To use the `RtSolver` interface the user must cast from one of the inherited classes (see *Diffusion approximation* or *Monte Carlo methods*). Since most of the `RtSolver` is an interface which is implemented by other radiative transfer solvers, documentation of boundary conditions, kernels and so on are found in the implementation classes.

4.4.3 Diffusion approximation

EddingtonSP1

The first-order diffusion approximation to the radiative transfer equation is encapsulated by the `EddingtonSP1` class which implements a first order Eddington approximation of the radiative transfer equation. `EddingtonSP1` implements `RtSolver` using both stationary and transient advance methods (e.g. for stationary or time-dependent radiative transport). The source code is located in `$DISCHARGE_HOME/RadiativeTransfer`.

Equation of motion

In the diffusion approximation, the radiative transport equation is

$$\partial_t \Psi + \kappa \Psi - \nabla \cdot \left(\frac{1}{3\kappa} \nabla \Psi \right) = \frac{\eta}{c}, \tag{4.4.1}$$

where κ is the absorption coefficient (i.e., inverse absorption length). Note that in the context below, κ is *not* the volume fraction of a grid cell. This is called the Eddington approximation, and the radiative flux is $F = -\frac{c}{3\kappa}\nabla\Psi$.

In the stationary case this yields a Helmholtz equation

$$\kappa \Psi - \nabla \cdot \left(\frac{1}{3\kappa} \nabla \Psi \right) = \frac{\eta}{c}, \tag{4.4.2}$$

Implementation

`EddingtonSP1` uses multigrid methods for solving Eq. 4.4.1 and Eq. 4.4.2, see *Linear solvers*. The class implements `RtSolver::advance()`, which can switch between Eq. 4.4.1 and Eq. 4.4.2. Note that for both the stationary and time-dependent cases the absorption coefficient κ in Eq. 4.4.1 and Eq. 4.4.2 are filled using the `RtSpecies` implementation provided to the solver. Also note that the absorption coefficient does not need to be constant in space.

Stationary kernel

For the stationary kernel we solve Eq. 4.4.2 directly, using a single multigrid solve. See *Linear solvers* for discretization details.

Transient kernel

For solving Eq. 4.4.1, `EddingtonSP1` implements both the backward Euler method and the Twizell-Gumel-Arigu (TGA) scheme. Explicit discretizations are not available. The Euler discretization is

$$\left(1 + \kappa \Delta t\right) \Psi^{k+1} - \Delta t \nabla \cdot \left(\frac{1}{3\kappa} \nabla \Psi^{k+1} \right) = \Psi^k + \frac{\Delta t \eta^{k+1}}{c},$$

Again, this is a Helmholtz equation for *Psi^{k+1}* which is solved using geometric multigrid. Expressions for the TGA scheme are found in [9], but note that the TGA scheme requires a solution to two elliptic equations (thus it has approximately twice the cost).

Boundary conditions

Simplified domain boundary conditions

The `EddingtonSP1` solver supports the following boundary conditions on domain faces and EBs. The domain boundary condition *type*, which is either Dirichlet, Neumann, or Larsen (a special type of Robin boundary condition) is always passed in through the input file. If the user passes in a value, say `neumann 0.0`, for a particular domain side/face, then the class will use a homogeneous Neumann boundary for the entire domain edge/face.

Custom domain boundary conditions

It is possible to use more complex boundary conditions by passing in `dirichlet_custom`, `neumann_custom`, or `larsen_custom` options. In this case the `EddingtonSP1` solver will use a specified function at the domain edge/face. To specify that function, `EddingtonSP1` has a member function

```
void setDomainSideBcFunction(const int a_dir,
                             const Side::LoHiSide a_side,
                             const std::function<Real(const RealVect a_pos, const Real a_time)> a_function);
```

which species a boundary condition value for one of the edges (faces in 3D). Note that the boundary condition *type* is still Dirichlet, Neumann, or Larsen (depending on whether or not `dirichlet_custom`, `neumann_custom`, or `larsen_custom` was passed in). For example, to set the boundary condition on the left x face in the domain, one can create a `EddingtonSP1DomainBc::BcFunction` object as follows:

```
// Assume this has been instantiated.
RefCountedPtr<EddingtonSP1> eddingtonSolver;

// Make a lambda which we can bind to std::function.
auto myValue = [](const RealVect a_pos, const Real a_time) -> Real {
   return a_pos[0] * a_time;
}

// Set the domain bc function in the solver.
eddingtonSolver.setDomainSideBcFunction(0, Side::Lo, myValue);
```

Note: If the user specifies one of the custom boundary conditions but does not set the function, it will issue a run-time error.

Embedded boundaries

On the EB, we currently only support constant-value boundary conditions. In the input script, the user can specify

- `dirichlet <value>` For setting a constant Dirichlet boundary condition everywhere.
- `neumann <value>` For setting a constant Neumann boundary condition everywhere.
- `larsen <value>` For setting a constant Larsen boundary condition everywhere.

Boundary condition types

1. **Dirichlet**. For Dirichlet boundary conditions we specify the value of Ψ on the boundary. Note that this involves reconstructing the gradient $\partial_n \Psi$ on domain faces and edges, see Chap:LinearSolversDirichletBC.
2. **Neumann**. For Neumann boundary conditions we specify the value of $\partial_n \Psi$ on the boundary. Note that the linear solver interface also supports setting $B\partial_n \Psi$ on the boundary (where B is the Helmholtz equation B coefficient). However, the `EddingtonSP1` solver does not use this functionality.
3. **Larsen**. The Larsen boundary condition is an absorbing boundary condition, taking the form of a Robin boundary as follows:

   $$\kappa \partial_n \Psi + \frac{3\kappa^2}{2}\frac{1-3r_2}{1-2r_1}\Psi = g,$$

   where r_1 and r_2 are reflection coefficients and g is a surface source, see [6] for details. Note that when the user specifies the boundary condition value (e.g. by setting the BC function), he is setting the surface sourge g. In the majority of cases, however, we will have $r_1 = r_2 = g = 0$ and the BC becomes

   $$\partial_n \Psi + \frac{3\kappa}{2}\Psi = 0.$$

Solver configuration

The `EddingtonSP1` implementation has a number of configurable options for running the solver, and these are given below:

```
# ================================================================================================
# EddingtonSP1 class options
# ================================================================================================
EddingtonSP1.stationary          = true         # Stationary solver
EddingtonSP1.reflectivity        = 0.           # Reflectivity
EddingtonSP1.use_tga             = false        # Use TGA for integration
EddingtonSP1.kappa_scale         = true         # Kappa scale source or not (depends on algorithm)
EddingtonSP1.plt_vars            = phi src      # Plot variables. Available are 'phi' and 'src'

EddingtonSP1.ebbc                = larsen 0.0   # Bc on embedded boundaries
EddingtonSP1.bc.x.lo             = larsen 0.0   # Bc on domain side. 'dirichlet', 'neuman', or 'larsen'
EddingtonSP1.bc.x.hi             = larsen 0.0   # Bc on domain side. 'dirichlet', 'neuman', or 'larsen'
EddingtonSP1.bc.y.lo             = larsen 0.0   # Bc on domain side. 'dirichlet', 'neuman', or 'larsen'
EddingtonSP1.bc.y.hi             = larsen 0.0   # Bc on domain side. 'dirichlet', 'neuman', or 'larsen'
EddingtonSP1.bc.z.lo             = larsen 0.0   # Bc on domain side. 'dirichlet', 'neuman', or 'larsen'
EddingtonSP1.bc.z.hi             = larsen 0.0   # Bc on domain side. 'dirichlet', 'neuman', or 'larsen'
EddingtonSP1.bc.z.hi             = larsen 0.0   # Boundary on domain. 'neumann' or 'larsen'

EddingtonSP1.gmg_verbosity       = -1           # GMG verbosity
EddingtonSP1.gmg_pre_smooth      = 8            # Number of relaxations in downsweep
EddingtonSP1.gmg_post_smooth     = 8            # Number of relaxations in upsweep
EddingtonSP1.gmg_bott_smooth     = 8            # NUmber of relaxations before dropping to bottom solver
EddingtonSP1.gmg_min_iter        = 5            # Minimum number of iterations
EddingtonSP1.gmg_max_iter        = 32           # Maximum number of iterations
EddingtonSP1.gmg_exit_tol        = 1.E-6        # Residue tolerance
EddingtonSP1.gmg_exit_hang       = 0.2          # Solver hang
EddingtonSP1.gmg_min_cells       = 16           # Bottom drop
EddingtonSP1.gmg_bottom_solver   = bicgstab     # Bottom solver type. Valid options are 'simple <number>' and 'bicgstab'
EddingtonSP1.gmg_cycle           = vcycle       # Cycle type. Only 'vcycle' supported for now
EddingtonSP1.gmg_ebbc_weight     = 2            # EBBC weight (only for Dirichlet)
EddingtonSP1.gmg_ebbc_order      = 2            # EBBC order (only for Dirichlet)
EddingtonSP1.gmg_smoother        = red_black    # Relaxation type. 'jacobi', 'red_black', or 'multi_color'
```

Basic options

Basic input options to `EddingtonSP1` are as follows:

- `EddingtonSP1.stationary` for setting whether or not the solver is stationary.
- `EddingtonSP1.reflectivity` for controlling the reflectivity in the Larsen boundary conditions.
- `EddingtonSP1.use_tga` for switching between backward Euler and TGA time discretizations. Only relevant if `EddingtonSP1.stationary = false`.
- `EddingtonSP1.kappa_scale` Switch for multiplying the source with with the volume fraction or not. Note that the multigrid Helmholtz solvers require a diagonal weighting of the operator. If `EddingtonSP1.kappa_scale = false` then the solver will assume that this weighting of the source term has already been made.
- `EddingtonSP1.plt_vars` For setting which solver plot variables are included in plot files.

Setting boundary conditions

Boundary conditions are parsed through the flags

- `EddingtonSP1.ebbc` Which sets the boundary conditions on the EBs.
- `EddingtonSP1.bc.dim.side` Which sets the boundary conditions on the domain sides, see *Boundary conditions* for details.

Tuning multigrid performance

All parameters that begin with the form `EddingtonSP1.gmg_` indicate a tuning parameter for geometric multigrid.

- `EddingtonSP1.gmg_verbosity`. Controls the multigrid verbosity. Setting it to a number > 0 will print multigrid convergence information.
- `EddingtonSP1.gmg_pre_smooth`. Controls the number of relaxations on each level during multigrid downsweeps.
- `EddingtonSP1.gmg_post_smooth`. Controls the number of relaxations on each level during multigrid upsweeps.
- `EddingtonSP1.gmg_bott_smooth`. Controls the number of relaxations before entering the bottom solve.
- `EddingtonSP1.gmg_min_iter`. Sets the minimum number of iterations that multigrid will perform.
- `EddingtonSP1.gmg_max_iter`. Sets the maximum number of iterations that multigrid will perform.
- `EddingtonSP1.gmg_exit_tol`. Sets the exit tolerance for multigrid. Multigrid will exit the iterations if $r < \lambda r_0$ where λ is the specified tolerance, $r = |L\Phi - \rho|$ is the residual and r_0 is the residual for $\Phi = 0$.
- `EddingtonSP1.gmg_exit_hang`. Sets the minimum permitted reduction in the convergence rate before exiting multigrid. Letting r^k be the residual after k multigrid cycles, multigrid will abort if the residual between levels is not reduce by at least a factor of $r^{k+1} < (1 - h)r^k$, where h is the "hang" factor.
- `EddingtonSP1.gmg_min_cells`. Sets the minimum amount of cells along any coordinate direction for coarsened levels. Note that this will control how far multigrid will coarsen. Setting a number `gmg_min_cells = 16` will terminate multigrid coarsening when the domain has 16 cells in any of the coordinate direction.
- `EddingtonSP1.gmg_bottom_solver`. Sets the bottom solver type.
- `EddingtonSP1.gmg_cycle`. Sets the multigrid method. Currently, only V-cycles are supported.

- `EddingtonSP1.gmg_ebbc_order`. Sets the stencil order on EBs when using Dirichlet boundary conditions. Note that this is also the stencil radius. See *Linear solvers* for details.
- `EddingtonSP1.gmg_ebbc_weight`. Sets the least squares stencil weighting factor for least squares gradient reconstruction on EBs when using Dirichlet boundary conditions. See *Least squares* for details.
- `EddingtonSP1.gmg_smoother`. Sets the multigrid smoother.

Runtime parameters

The following parameters for `EddingtonSP1` are run-time configurable:

- All multigrid tuning parameters, i.e. parameters starting with `EddingtonSP1.gmg_`.
- Plot variables, i.e. `EddingtonSP1.plt_vars`.
- Kappa scaling (for algorithmic adjustments), i.e. `EddingtonSP1.kappa_scale`.

4.4.4 Monte Carlo methods

All types of moment-closed radiative transfer equations contain nonphysical artifacts (which may or may not be acceptable). For example, in the diffusion approximation the radiative flux is $F = -\frac{c}{3\kappa}\nabla\Psi$, implying that photons can leak around boundaries. I.e. the diffusion approximation does not correctly describe shadows. It is possible to go beyond the diffusion approximation by also solving for higher-order moments like the radiative flux. While such methods can describe shadows, they do, contain other nonphysical features.

Both ‘’stationary” and transient Monte Carlo methods are offered as an alternative to the diffusion approximation.

photon particle

The Îto particle is a computational particle class in *chombo-discharge* which can be used together with the particle tools in *Chombo*. The following data fields are implemented in the particle:

```
RealVect m_position;
RealVect m_velocity;
Real m_mass;
Real m_kappa;
```

To obtain the fields, the user will call

```
RealVect& position();
RealVect& velocity();
Real& mass();
Real& diffusion();
```

All functions also have `const` versions. Note that the field `m_mass` is the same as the *weight* of the computational particle. The following functions are used to set the various properties:

```
setPosition(const RealVect a_pos);
setVelocity(const RealVect a_vel);
setMass(const Real a_mass);
setDiffusion(const Real a_diffusion;
```

Interaction with boundaries

Stationary Monte Carlo

The stationary Monte Carlo method proceeds as follows.

1. For each cell in the mesh, draw a discrete number of photons $\mathcal{P}\left(\eta\Delta V\Delta t\right)$ where $\mathcal{P}$ is a Poisson distribution. The user may also choose to use pseudophotons rather than physical photons by modifying photon weights. Each photon is generated in the cell centroid $\mathbf{x}_0$ and given a random propagation direction $\mathbf{n}$.
2. Draw a propagation distance r by drawing random numbers from an exponential distribution $p(r) = \kappa\exp\left(-\kappa r\right)$. The absorbed position of the photon is $\mathbf{x} = \mathbf{x}_0 + r\mathbf{n}$.
3. Check if the path from $\mathbf{x}_0$ to $\mathbf{x}$ intersects an internal or domain boundary. If it does, absorb the photon on the boundary. If not, move the photon to $\mathbf{x}$ or reflect it off symmetry boundaries.
4. Rebin the absorbed photons onto the AMR grid. This involves parallel communication.
5. Compute the resulting photoionization profile. The user may choose between several different deposition schemes (like e.g. cloud-in-cell).

The Monte Carlo methods use computational particles for advancing the photons in exactly the same way a Particle-In-Cell method would use them for advancing electrons. Although a computational photon would normally live on the finest grid level that overlaps its position, this is not practical for all particle deposition kernels. For example, for cloud-in-cell deposition schemes it is useful to have the restrict the interpolation kernels to the grid level where the particle lives. In Chombo-speak, we therefore use a buffer region that extends some cells from a refinement boundary where the photons are not allowed to live. Instead, photons in that buffer region are transferred to a coarser level, and their deposition clouds are first interpolated to the fine level before deposition on the fine level happens. Selecting a deposition scheme and adjusting the buffer region is done through an input script associated with the solver.

Transient Monte Carlo

The transient Monte Carlo method is almost identical to the stationary method, except that it does not deposit all generated photons on the mesh but tracks them through time. The transient method is implemented as follows:

1. For each cell in the mesh, draw a discrete number of photons $\mathcal{P}\left(\eta\Delta V\Delta t\right)$ as above, and append these to the already existing photons. Each photon is given a uniformly distributed random creation time within Δt.
2. Each photon is advanced over the time step Δt by a sequence of N substeps (N may be different for each photon).
    a. We compute N such that we sample $N\Delta\tau = \Delta t$ with $c\kappa\Delta\tau < 1$.
    b. A photon at position $\mathbf{x}_0$ is moved a distance $\Delta\mathbf{x} = c\mathbf{n}\Delta\tau$. For each step we compute the absorption probability $p = \kappa\left|\Delta\mathbf{x}\right|$ where $p \in [0,1]$ is a uniform random number. If the photon is absorbed on this interval, draw a new uniform random number $r \in [0,1]$ and absorb the photon at the position $\mathbf{x}_0 + r\Delta\mathbf{x}$. If the photon is not absorbed, it is moved to position $\mathbf{x}_0 + r\Delta\mathbf{x}$.
3. Check if the path from $\mathbf{x}_0$ to $\mathbf{x}$ intersects an internal or domain boundary. If it does, absorb the photon on the boundary. If not, move the photon to $\mathbf{x}$.
4. Rebin the absorbed photons onto the AMR grid. This involves parallel communication.
5. Compute the resulting photoionization profile. The user may choose between several different deposition schemes (like e.g. cloud-in-cell).

4.4.5 Limitations

4.4.6 Example application

An example application of usage of the `RtSolver` is found in *Radiative transfer*.

4.5 Surface charge solver

In order to conserve charge on solid insulators, *chombo-discharge* has a solver that is defined on the gas-dielectric interface where the surface charge is updated with the incoming flux

$$F_\sigma(\phi) = \sum_\phi q_\phi F_{\mathrm{EB}}(\phi),$$

where q_ϕ is the charge of a species ϕ. This ensures strong conservation on insulating surfaces.

4.6 Tracer particles

4.7 Îto diffusion

The Îto diffusion model advances computational particles as Brownian walkers with drift:

$$d\mathbf{X}_i = \mathbf{v}_i dt + \sqrt{2D_i}\mathbf{W}_i dt,$$

where $\mathbf{X}_i$ is the spatial position of a particle i, $\mathbf{v}_i$ is the drift coefficient and D_i is the diffusion coefficient *in the continuum limit*. That is, both $\mathbf{v}_i$ and D_i are the quantities that appear in *Convection-Diffusion-Reaction*. The vector term $\mathbf{W}_i$ is a Gaussian random field with a mean value of 0 and standard deviation of 1.

The code for Îto diffusion is given in `/src/ito_solver` and only a brief explanation is given here. The source code is used by a physics module in `/physics/brownian_walker` and in the regression test `/regression/brownian_walker`.

4.7.1 The Îto particle

The Îto particle is a computational particle class in *chombo-discharge* which can be used together with the particle tools in *Chombo*. The following data fields are implemented in the particle:

```
RealVect m_position;
RealVect m_velocity;
Real m_mass;
Real m_diffusion;
```

To obtain the fields, the user will call

```
RealVect& position();
RealVect& velocity();
Real& mass();
Real& diffusion();
```

All functions also have `const` versions. Note that the field `m_mass` is the same as the *weight* of the computational particle. The following functions are used to set the various properties:

```
setPosition(const RealVect a_pos);
setVelocity(const RealVect a_vel);
setMass(const Real a_mass);
setDiffusion(const Real a_diffusion;
```

4.7.2 ito_species

`ito_species` is a class for parsing information into the solver class. The constructor for the `ito_species` class is

```
ito_species(const std::string a_name, const int a_charge, const bool a_mobile, const bool a_diffusive);
```

and this will set the name of the class, the charge, and whether or not the transport kernels account for drift and diffusion.

Setting initial conditions

In order to set the initial conditions the user must fill the list `List<ito_particle> m_initial_particles` in `ito_species`. When `initial_data()` is called from `ito_solver`, the initial particles are transferred from the instance of `ito_species` and into the instance of `ito_solver`.

We remark that it is a bad idea to replicate the initial particle list over all MPI ranks in a simulation. If one has a list of initial particles, or wants to draw a specified number of particles from a distribution, the initial particles *must* be distributed over the available MPI ranks. For example, the code in `/physics/brownian_walker/brownian_walker_species.cpp` draws a specified number of particles distributed over all MPI ranks as (the code is called in `brownian_walker_species::draw_initial_particles`)

```
// To avoid that MPI ranks draw the same particle positions, increment the seed for each rank
m_seed += procID();

// Set up the RNG
m_rng = std::mt19937_64(m_seed);
m_gauss = std::normal_distribution<Real>(0.0, m_blob_radius);
m_udist11 = std::uniform_real_distribution<Real>(-1., 1.);

// Each MPI process draws the desired number of particles from a distribution
const int quotient  = m_num_particles/numProc();
const int remainder = m_num_particles % numProc();

Vector<int> particlesPerRank(numProc(), quotient);

for (int i = 0; i < remainder; i++){
  particlesPerRank[i] += 1;
}

// Now make the particles
m_initial_particles.clear();
for (int i = 0; i < particlesPerRank[procID()]; i++){
  const Real weight  = 1.0;
  const RealVect pos = m_blob_center + random_gaussian();
  m_initial_particles.add(ito_particle(weight, pos));
}
```

4.7.3 Computing time steps

The signatures for computing a time step for the `ito_solver` are given separately for the drift part and the diffusion part.

Drift

The drift time step routines are implemented such that one restricts the time step such that the fastest particle does not move more than a specified number of grid cells.

For the drift, the signatures are

```
Real compute_min_drift_dt(const Real a_maxCellsToMove) const;
Vector<Real> compute_drift_dt(const Real a_maxCellsToMove) const;

Vector<Real> compute_drift_dt() const; // Compute dt on all AMR levels, return vector of time step
Real compute_drift_dt(const int a_lvl) const;
Real compute_drift_dt(const int a_lvl, const DataIndex& a_dit, const RealVect a_dx) const;
```

These last three functions all compute $\Delta t = \Delta x/Max(v_x, v_y, v_z)$ on the the various AMR levels and patches. The routine

```
Vector<Real> compute_drift_dt(const Real a_maxCellsToMove) const;
```

simply scales Δt by `a_maxCellsToMove` on every level. Finally, the function `compute_min_drift_dt(...)` computes the smallest time step across every AMR level.

Diffusion

The signatures for the diffusion time step are similar to the ones for drift:

```
Real compute_min_diffusion_dt(const Real a_maxCellsToMove) const;
Vector<Real> compute_diffusion_dt(const Real a_maxCellsToMove) const;

Vector<Real> compute_diffusion_dt() const;
Real compute_diffusion_dt(const int a_lvl) const;
Real compute_diffusion_dt(const int a_lvl, const DataIndex& a_dit, const RealVect a_dx) const;
```

In these routines, the time step is computed as $\Delta t = \frac{\Delta x}{\sqrt{2D}}$. Note that there is still a chance that a particle jumps further than specified by `a_maxCellsToMove` since the diffusion hop is

$$\mathbf{d} = \sqrt{2D}\mathbf{Z}\Delta t,$$

where $\mathbf{Z}$ is a random Gaussian. The probability that a diffusion hop leads to a jump larger than N cells can be evaluated and is $P = \text{erf}\left(\sqrt{2}N\right)$. It is useful to keep this probability in mind when deciding on the PVR.

4.7.4 Remapping particles

Particle remapping has been implemented for the whole AMR hierarchy as a two step process.

1. Perform two-level remapping where particles are transferred up or down one grid level if they move out the level PVR.
2. Gather all particles that are remnant in the outcast list on the coarsest level, and then distribute them back to their appropriate levels. For example, particles that hopped over more than one refinement boundary cannot be transferred with a (clean) two-level transfer.

4.7.5 Limitations

4.7.6 Example application

An example application of usage of the `ItoSolver` is found in *Brownian walker*.

CHAPTER

FIVE

PHYSICS MODELS

5.1 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.

5.2 Advection diffusion

5.3 Brownian walker

5.4 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.
- `/Physics/CdrPlasma/TimeSteppers` contains various algorithms for advancing the equations of motion.
- `/Physics/CdrPlasma/CellTaggers` contains various algorithms for flagging cells for refinement and coarsening.

- `/Physics/CdrPlasma/python` contains Python source files for quickly setting up new applications.

5.4.1 Equations of motion

In the CDR plasma model we are solving

$$\begin{aligned}\nabla\cdot(\epsilon_r\nabla\Phi) &= -\frac{\rho}{\epsilon_0},\\ \frac{\partial\sigma}{\partial t} &= F_\sigma,\\ \frac{\partial n}{\partial t} + \nabla\cdot(\mathbf{v}n - D\nabla n) &= S,\end{aligned}$$

The above equations must be supported by additional boundary conditions on electrodes and insulating surfaces.

Radiative transport can be done either in the diffusive approximation or by means of Monte Carlo methods. Diffusive RTE methods involve solving

$$\partial_t\Psi + \kappa\Psi - \nabla\cdot\left(\frac{1}{3\kappa}\nabla\Psi\right) = \frac{\eta}{c},$$

where Ψ is the isotropic photon density, κ is an absorption length and η is an isotropic source term. I.e., η is the number of photons produced per unit time and volume. The time dependent term can be turned off and the equation can be solved stationary.

The module also supports discrete photons where photon transport and absorption is done by sampling discrete photons. In general, discrete photon methods incorporate better physics (like shadows) They can easily be adapted to e.g. scattering media. They are, on the other hand, inherently stochastic which implies that some extra caution must be exercised when integrating the equations of motion.

The coupling that is (currently) available in `chombo-discharge` is

$$\begin{aligned}\epsilon_r &= \epsilon_r(\mathbf{x}),\,(\text{can additionally be discontinuous}),\\ \mathbf{v} &= \mathbf{v}\left(t,\mathbf{x},\mathbf{E},n\right),\\ D &= \mathbf{v}\left(t,\mathbf{x},\mathbf{E},n\right),\\ S &= S\left(t,\mathbf{x},\mathbf{E},\nabla\mathbf{E},n,\nabla n,\Psi\right),\\ \eta &= \eta\left(t,\mathbf{x},\mathbf{E},n\right),\\ F &= F(t,\mathbf{x},\mathbf{E},n),\end{aligned} \tag{5.4.1}$$

where F is the boundary flux on insulators or electrodes (which must be separately implemented).

`chombo-discharge` works by embedding the equations above into an abstract C++ framework (see *CdrPlasmaPhysics*) that the user must implement or reuse existing pieces of, and then compile into an executable.

5.4.2 CdrPlasmaPhysics

CdrPlasmaPhysics is an abstract class which represents the plasma physics for the CDR plasma module, i.e. it provides the coupling functions in Eq. 5.4.1. The source code for the class resides in `/Physics/CdrPlasma/CD_CdrPlasmaPhysics.H`. Note that the entire class is an interface, whose implementations are used by the time integrators that advance the equations.

There are no default input parameters for *CdrPlasmaPhysics*, as users must generally implement their own kinetics. The class exists solely for providing the integrators with the necessary fundamentals for filling solvers with the correct quantities at the same time, for example filling source terms and drift velocities.

A successful implementation of *CdrPlasmaPhysics* has the following:

1. Instantiated a list of *CdrSpecies*. These become *Convection-Diffusion-Reaction* solvers and contain initial conditions and basic transport settings for the convection-diffusion-reaction solvers.
2. Instantiated a list *RtSpecies*. These become *Radiative transfer* solvers and contain metadata for the radiative transport solvers.
3. Implemented the core functionality that couple the solvers together.

`chombo-discharge` automatically allocates the specified number of convection-diffusion-reaction and radiative transport solvers from the list of species the is intantiated. For information on how to interface into the CDR solvers, see *CdrSpecies*. Likewise, see *RtSpecies* for how to interface into the RTE solvers.

Implementation of the core functionality is comparatively straightforward, but can lead to boilerplate code. For this reason we also provide an implementation layer *JSON interface* that provides a plug-and-play interface for specifying the plasma physics.

API

The API for `CdrPlasmaPhysics` is as follows:

```
virtual Real computeAlpha(const RealVect a_E) const  = 0;

virtual void advanceReactionNetwork(Vector<Real>&          a_cdrSources,
                                    Vector<Real>&          a_rteSources,
                                    const Vector<Real>     a_cdrDensities,
                                    const Vector<RealVect> a_cdrGradients,
                                    const Vector<Real>     a_rteDensities,
                                    const RealVect         a_E,
                                    const RealVect         a_pos,
                                    const Real             a_dx,
                                    const Real             a_dt,
                                    const Real             a_time,
                                    const Real             a_kappa) const  = 0;

virtual Vector<RealVect> computeCdrDriftVelocities(const Real         a_time,
                                                   const RealVect     a_pos,
                                                   const RealVect     a_E,
                                                   const Vector<Real> a_cdrDensities) const  = 0;

virtual Vector<Real> computeCdrDiffusionCoefficients(const Real         a_time,
                                                     const RealVect     a_pos,
                                                     const RealVect     a_E,
                                                     const Vector<Real> a_cdrDensities) const  = 0;

virtual Vector<Real> computeCdrElectrodeFluxes(const Real         a_time,
                                               const RealVect     a_pos,
                                               const RealVect     a_normal,
                                               const RealVect     a_E,
                                               const Vector<Real> a_cdrDensities,
                                               const Vector<Real> a_cdrVelocities,
                                               const Vector<Real> a_cdrGradients,
                                               const Vector<Real> a_rteFluxes,
                                               const Vector<Real> a_extrapCdrFluxes) const  = 0;

virtual Vector<Real> computeCdrDielectricFluxes(const Real         a_time,
                                                const RealVect     a_pos,
                                                const RealVect     a_normal,
                                                const RealVect     a_E,
                                                const Vector<Real> a_cdrDensities,
                                                const Vector<Real> a_cdrVelocities,
                                                const Vector<Real> a_cdrGradients,
                                                const Vector<Real> a_rteFluxes,
                                                const Vector<Real> a_extrapCdrFluxes) const  = 0;

virtual Vector<Real> computeCdrDomainFluxes(const Real           a_time,
                                            const RealVect       a_pos,
                                            const int            a_dir,
                                            const Side::LoHiSide a_side,
                                            const RealVect       a_E,
                                            const Vector<Real>   a_cdrDensities,
                                            const Vector<Real>   a_cdrVelocities,
                                            const Vector<Real>   a_cdrGradients,
```

(continues on next page)

(continued from previous page)

```
                                        const Vector<Real>   a_rteFluxes,
                                        const Vector<Real>   a_extrapCdrFluxes) const  = 0;

virtual Real initialSigma(const Real a_time, const RealVect a_pos) const  = 0;
```

The above code blocks do the following:

- `computeAlpha` computes the Townsend ionization coefficient. This is used by the cell tagger.
- `advanceReactionNetwork` provides the coupling $S = S(t, \mathbf{x}, \mathbf{E}, \nabla\mathbf{E}, n, \nabla n, \Psi)$.
- `computeCdrDriftVelocities` provides the coupling $\mathbf{v} = \mathbf{v}\left(t, \mathbf{x}, \mathbf{E}, n\right)$.
- `computeCdrDiffusionCoefficients` provides the coupling $D = \mathbf{v}\left(t, \mathbf{x}, \mathbf{E}, n\right)$.
- `computeCdrElectrodeFluxes` provides the coupling $F = F(t, \mathbf{x}, \mathbf{E}, n)$ on electrode EBs.
- `computeCdrDielectricFluxes` provides the coupling $F = F(t, \mathbf{x}, \mathbf{E}, n)$ on dielectric EBs.
- `computeCdrDomainFluxes` provides the coupling $F = F(t, \mathbf{x}, \mathbf{E}, n)$ on domain sides.

For a fully documented API, see the doxygen API.

Below, we include a brief overview of how `CdrPlasmaPhysics` can be directly implemented. Note that direct implements like these tend to become boilerplate, we also include an interface which implements these functions with pre-defined rules, see *JSON interface*.

Initializing species

In the constructor, the user should define the advected/diffused species and the radiative transfer species. These are stored in vectors `Vector<RefCountedPtr<CdrSpecies> > m_CdrSpecies` and `Vector<RefCountedPtr<RtSpecies> > m_RtSpecies`. Each species in these vectors become a convection-diffusion-reaction solver or a radiative transfer solver. See *CdrSpecies* and *RtSpecies* for details on how to implement these.

Defining drift velocities

To set the drift velocities, implement `computeCdrDriftVelocities` – this will set the drift velocity $\mathbf{v}$ in the CDR equations:

```
Vector<RealVect> computeCdrDriftVelocities(const Real         a_time,
                                           const RealVect     a_pos,
                                           const RealVect     a_E,
                                           const Vector<Real> a_cdrDensities) const  {
   return Vector<RealVect>(m_numCdrSpecies, a_E);
}
```

This implementation is set the advection velocity equal to $\mathbf{E}$. For a full plasma simulation, there will also be mobilities involved, which the user is reponsible for obtaining.

Defining diffusion coefficients

To set the diffusion coefficients, implement `computeCdrDiffusionCoefficients` – this will set the diffusion coefficient D in the CDR equations:

```
Vector<Real> computeCdrDiffusionCoefficients(const Real         a_time,
                                             const RealVect     a_pos,
                                             const RealVect     a_E,
                                             const Vector<Real> a_cdrDensities) const {
   return Vector<Real>(m_numCdrSpecies, 1.0);
}
```

This sets $D = 1$ for all species involved.

Defining chemistry terms

To set the source terms S, implement `advanceReactionNetwork`. This routine should set the reaction terms for both the CDR equations *and* the radiative transfer equations.

Note: For the radiative transfer equations we set the isotropic source term η which is the number of ionizing photons produced per unit volume and time.

```
virtual void advanceReactionNetwork(Vector<Real>&          a_cdrSources,
                                    Vector<Real>&          a_rteSources,
                                    const Vector<Real>     a_cdrDensities,
                                    const Vector<RealVect> a_cdrGradients,
                                    const Vector<Real>     a_rteDensities,
                                    const RealVect         a_E,
                                    const RealVect         a_pos,
                                    const Real             a_dx,
                                    const Real             a_dt,
                                    const Real             a_time,
                                    const Real             a_kappa) const {
   a_cdrSources = Vector<Real>(m_numCdrSpecies, 1.0);
   a_rteSources = Vector<Real>(m_numRteSpecies, 1.0);
}
```

The above code will set $S = \eta = 1$ for all species.

We point out that in the plasma module the source terms are *always* used in the form

$$n^{k+1} = n^k + \Delta t S,$$

where S is the source term obtained from `advanceReactionNetwork`. This implies that it *is* possible to define fully implicit integrators directly in `advanceReactionNetwork`. For example, if the reactive problem consisted only of $\partial_t n = -\frac{n}{\tau}$, one could form a reactive integrator with the implicit Euler rule by first computing $n^{k+1} = \frac{n^k}{1+\Delta t/\tau}$ and then linearizing $S = \frac{n^{k+1}-n^k}{\Delta t}$.

Fluxes at electrode boundaries

To set the fluxes F on electrode EBs, implement `computeCdrElectrodeFluxes`. Note that the fluxes F are those occuring in a finite-volume context; i.e. the total injected or extracted mass.

```
Vector<Real> computeCdrElectrodeFluxes(const Real         a_time,
                                       const RealVect     a_pos,
                                       const RealVect     a_normal,
                                       const RealVect     a_E,
                                       const Vector<Real> a_cdrDensities,
                                       const Vector<Real> a_cdrVelocities,
                                       const Vector<Real> a_cdrGradients,
```

(continues on next page)

(continued from previous page)

```
                                    const Vector<Real> a_rteFluxes,
                                    const Vector<Real> a_extrapCdrFluxes) const {
   return Vector<Real>(m_numCdrSpecies, 0.0);
}
```

The input variable `a_extrapCdrFluxes` are cell-centered fluxes extrapolated to the EBs.

Fluxes at dielectric boundaries

To set the fluxes F on dielectric EBs, implement `computeCdrDielectricFluxes`. Note that the fluxes F are those occuring in a finite-volume context; i.e. the total injected or extracted mass.

```
Vector<Real> computeCdrDielectricFluxes(const Real         a_time,
                                        const RealVect     a_pos,
                                        const RealVect     a_normal,
                                        const RealVect     a_E,
                                        const Vector<Real> a_cdrDensities,
                                        const Vector<Real> a_cdrVelocities,
                                        const Vector<Real> a_cdrGradients,
                                        const Vector<Real> a_rteFluxes,
                                        const Vector<Real> a_extrapCdrFluxes) const {
   return Vector<Real>(m_numCdrSpecies, 0.0);
}
```

The input variable `a_extrapCdrFluxes` are cell-centered fluxes extrapolated to the EBs.

Fluxes at domain boundaries

To set the fluxes F on dielectric EBs, implement `computeCdrDielectricFluxes`. Note that the fluxes F are those occuring in a finite-volume context; i.e. the total injected or extracted mass.

```
Vector<Real> computeCdrDomainFluxes(const Real           a_time,
                                    const RealVect       a_pos,
                                    const int            a_dir,
                                    const Side::LoHiSide a_side,
                                    const RealVect       a_E,
                                    const Vector<Real>   a_cdrDensities,
                                    const Vector<Real>   a_cdrVelocities,
                                    const Vector<Real>   a_cdrGradients,
                                    const Vector<Real>   a_rteFluxes,
                                    const Vector<Real>   a_extrapCdrFluxes) const {
   return Vector<Real>(m_numCdrSpecies, 0.0);
}
```

The input variable `a_extrapCdrFluxes` are cell-centered fluxes extrapolated to the domain sides.

Setting initial surface charge

To set the initial surface charge on dielectric boundaries, implement

```
Real initialSigma(const Real a_time, const RealVect a_pos) const{
   return 0.0;
}
```

5.4.3 Time discretizations

Here, we discuss two discretizations of `CdrPlasmaEquations`. Firstly, note that there are two layers to the time integrators:

1. A pure class `CdrPlasmaStepper` which inherits from `TimeSteppers` but does not implement an `advance` method. This class simply provides the base functionality for more easily developing time integrators. `CdrPlasmaStepper` contains methods that are necessary for coupling the solvers, e.g. calling the CdrPlasmaPhysics methods at the correct time.
2. Implementations of `CdrPlasmaPhysics`, which implement the `advance` method and can thus be used for advancing models.

The supported time integrators are located in `$DISCHARGE_HOME/CdrPlasma/TimeSteppers`. There are two integrators that are commonly used.

- A Godunov operator splitting with either explicit or implicit diffusion. This integrator also supports semi-implicit formulations.
- A spectral deferred correction (SDC) integrator with implicit diffusion. This integrator is an implicit-explicit.

Briefly put, the Godunov operator is our most stable integrator, while the SDC integrator is our most accurate integrator.

Godunov operator splitting

The `CdrPlasmaGodunovStepper` implements `CdrPlasmaStepper` and defines an operator splitting method between charge transport and plasma chemistry. It has a formal order of convergence of one. The source code is located in `$DISCHARGE_HOME/Physics/CdrPlasma/TimeSteppers/CdrPlasmaGodunovStepper`.

> **Warning:** Splitting the terms yields *splitting errors* which can dominate for large time steps. Typically, the operator splitting discretization is not suitable for large time steps.

The basic advancement routine for `CdrPlasmaGodunovStepper` is as follows:

1. Advance the charge transport $\phi^k \rightarrow \phi^{k+1}$ with the source terms set to zero.
2. Compute the electric field.
3. Advance the plasma chemistry over the same time step using the field computed above I.e., advance $\partial_t \phi = S$ over a time step Δt.
4. Advance the radiative transport part. This can also involve discrete photons.

The transport/field steps can be done in various ways: The following transport algorithms are available:

- **Euler**, where everything is advanced with the Euler rule.
- **Semi-implicit**, where the Euler field/transport step is performed with an implicit coupling to the electric field.

In addition, diffusion can be treated

- **Explicitly**, where all diffusion advances are performed with an *explicit* rule.
- **Implicitly**, where all diffusion advances are performed with an *implicit* rule.
- **Automatically**, where diffusion advances are performed with an implicit rule only if time steps dictate it, and explicitly otherwise.

Note: When setting up a new problem with the Godunov time integrator, the default setting is to use automatic diffusion and a semi-implicit coupling. These settings tend to work for most problems.

Specifying transport algorithm

To specify the transport algorithm, modify the flag `CdrPlasmaGodunovStepper.transport`, and set it to `semi_implicit` or `euler`. Everything else is an error.

Note that for the Godunov integrator, it is possible to center the advective discretization at the half time step. That is, the advancement algorithm is

$$n^{k+1} = n^k - \nabla \cdot \left(n^{k+1/2}\mathbf{v}\right) + \nabla \cdot \left(D\nabla\phi^k\right),$$

where $n^{k+1/2}$ is obtained by also including transverse slopes (i.e., extrapolation in time). See Trebotich and Graves [8] for details. Note that the formal order of accuracy is still one, but the accuracy of the advective discretization is increased substantially.

Specifying diffusion

To specify how diffusion is treated, modify the flag `CdrPlasmaGodunovStepper.diffusion`, and set it to `auto`, `explicit`, or `implicit`. In addition, the flag `CdrPlasmaGodunovStepper.diffusion_thresh` must be set to a number.

When diffusion is set to `auto`, the integrator switches to implicit diffusion when

$$\frac{\Delta t_{\mathrm{A}}}{\Delta t_{\mathrm{AD}}} > \epsilon,$$

where Δt_{A} is the advection-only limited time step and Δt_{AD} is the advection-diffusion limited time step.

Note: When there are multiple species being advected and diffused, the integrator will perform extra checks in order to maximize the time steps for the other species.

Time step limitations

The basic time step limitations for the Godunov integrator are:

- Manually set maximum and minimum time steps
- Courant-Friedrichs-Lewy conditions, either on advection, diffusion, or both.
- The dielectric relaxation time.

The user is responsible for setting these when running the simulation. Note when the the semi-implicit scheme is used, it is not necessary to restrict the time step by the dielectric relaxation time.

Spectral deferred corrections

The `CdrPlasmaImExSdcStepper` uses implicit-explicit (ImEx) spectral deferred corrections (SDCs) to advance the equations. This integrator implements the `advance` method for `CdrPlasmStepper`, and is a high-order method with implicit diffusion.

SDC basics

First, we provide a quick introduction to the SDC procedure. Given an ordinary differential equation (ODE) as

$$\frac{\partial u}{\partial t} = F(u,t), \quad u(t_0) = u_0,$$

the exact solution is

$$u(t) = u_0 + \int_{t_0}^{t} F\left(u,\tau\right) d\tau.$$

Denote an approximation to this solution by $\widetilde{u}(t)$ and the correction by $\delta(t) = u(t) - \widetilde{u}(t)$. The measure of error in $\widetilde{u}(t)$ is then

$$R(\widetilde{u},t) = u_0 + \int_{t_0}^{t} F(\widetilde{u},\tau)d\tau - \widetilde{u}(t).$$

Equivalently, since $u = \widetilde{u} + \delta$, we can write

$$\widetilde{u} + \delta = u_0 + \int_{t_0}^{t} F\left(\widetilde{u} + \delta,\tau\right) d\tau.$$

This yields

$$\delta = \int_{t_0}^{t} \left[F\left(\widetilde{u} + \delta,\tau\right) - F\left(\widetilde{u},\tau\right)\right] d\tau + R\left(\widetilde{u},t\right).$$

This is called the correction equation. The goal of SDC is to iteratively solve this equation in order to provide a high-order discretization.

The ImEx SDC method in `chombo-discharge` uses implicit diffusion in the SDC scheme. Coupling to the electric field is always explicit. The user is responsible for specifying the quadrature nodes, as well as setting the number of sub-intervals in the SDC integration and the number of corrections. In general, each correction raises the discretization order by one.

Time step limitations

The ImEx SDC integrator is limited by

- The dielectric relaxation time.
- An advective CFL conditions.

In addition to this, the user can specify maximum/minimum allowed time steps.

5.4.4 JSON interface

Since implementations of *CdrPlasmaPhysics* are usually boilerplate, we provide a class `CdrPlasmaJSON` which can initialize and parse various types of initial conditions and reactions from a JSON input file. This class is defined in `$DISCHARGE_HOME/Physics/PlasmaModels/CdrPlasmaJSON`.

`CdrPlasmaJSON` is a full implementation of `CdrPlasmaPhysics` which supports the definition of various species (neutral, plasma species, and photons) and methods of coupling them. We expect that `CdrPlasmaJSON` provides the simplest method of setting up a new plasma model. It is also comparatively straightforward to extend the class with further required functionality.

In the JSON interface, the radiative transfer solvers always solve for the number of photons that lead to photoionization events. This means that the interpretation of Ψ is the number of photoionization events during the previous time step. This is true for both continuum and discrete radiative transfer models.

Usage

To use this plasma model, use `-physics CdrPlasmaJSON` when setting up a new plasma problem (see *Simulation quick start*). When `CdrPlasmaJSON` is instantiated, the constructor will parse species, reactions, initial conditions, and boundary conditions from a JSON file that the user provides. In addition, users can parse transport data or reaction rates from tabulated ASCII files that they provide.

To specify the input plasma kinetics file, include

Specifying input file

`CdrPlasmaJSON` will read a JSON file specified by the input variable `CdrPlasmaJSON.chemistry_file`.

Discrete photons

There are two approaches when using discrete photons, and both rely on the user setting up the application with the Monte Carlo photon solver (rather than continuum solvers). For an introduction to the particle radiative transfer solver, see *Monte Carlo methods*.

The user must use one of the following:

1. Set the following class options:

```
CdrPlasmaJSON.discrete_photons = true

McPhoto.photon_generation = deterministic
McPhoto.source_type       = number
```

   When specifying `CdrPlasmaJSON.discrete_photons=true`, `CdrPlasmaJSON` will do a Poisson sampling of the number of photons that are generated in each cell and put this in the radiative transfer solvers' source terms. This means that the radiative transfer solver source terms *contain the physical number of photons generated in one time step*. To turn off sampling inside the radiative transfer solver, we specify `McPhoto.photon_generation = stochastic` and set `McPhoto.source_type = number` to let the solver know that the source contains the number of physical photons.

2. Set the following class options:

```
CdrPlasmaJSON.discrete_photons = false

McPhoto.photon_generation = stochastic
McPhoto.source_type       = volume_rate
```

In this case the `CdrPlasmaJSON` class will fill the solver source terms with the volumetric rate, i.e. the number of photons produced per unit volume and time. When `McPhoto` generates the photons it will compute the number of photons generated in a cell through Poisson sampling $n = P\left(S_\gamma \Delta V \Delta t\right)$ where P indicates a Poisson sampling operator.

Fundamentally, the two approaches differ only in where the the Poisson sampling is performed. With the first approach, plotting the radiative transfer solver source terms will show the number of physical photons generated. In the second approach, the source terms will show the volume photo-generation rate.

Gas law and neutral background

General functionality

To include the gas law and neutral species, include a JSON object `gas` with the the field `law` specified. Currently, `law` can be either `ideal`, `troposphere`, or `table`.

The purpose of the gas law is to set the temperature, pressure, and neutral density of the background gas. In addition, we specify the neutral species that are used through the simulation. These species are *not* stored on the mesh; we only store function pointers to their temperature, density, and pressure.

It is also possible to include a field `plot` which will then include the temperature, pressure, and density in plot files.

Ideal gas

To specify an ideal gas law, specify ideal gas law as follows:

```
{"gas":
  {
    "law": "ideal",
    "temperature": 300,
    "pressure": 1
  }
 }
```

In this case the gas pressure and temperatures will be as indicated, and the gas number density will be computed as

$$\rho = \frac{p_0' N_\mathrm{A}}{R T_0},$$

where p' is the pressure converted to Pascals.

Note that the input temperature should be specified in Kelvin, and the input pressure in atmospheres.

Troposphere

It is also possible to specify the pressure, temperature, and density to be functions of tropospheric altitude. In this case one must specify the extra fields

- `molar mass` For specifying the molar mass (in $\mathrm{g} \cdot \mathrm{mol}^{-1}$) of the gas.
- `gravity` Gravitational acceleration g.
- `lapse rate` Temperature lapse rate L in units of K/m.

In this case the gas temperature pressure, and number density are computed as

$$T(h) = T_0 - Lh$$

$$p(h) = p_0 \left((1 - \frac{Lh}{T_0} \right)^{\frac{gM}{RL}}$$

$$\rho(h) = \frac{p'(h) N_{\mathrm{A}}}{RT(h)}$$

For example, specification of tropospheric conditions can be included by

```
{"gas":
  {
    "law": "troposphere",
    "temperature": 300,
    "pressure": 1,
    "molar_mass": 28.97,
    "gravity": 9.81,
    "lapse_rate": 0.0065,
    "plot": true
  }
}
```

Tabulated

To specify temperature, density, and pressure as function of altitude, set `law` to `table` and incldue the following fields:

- `file` For specifying which file we read the data from.
- `height` For specifying the column where the height is stored (in meters).
- `temperature` For specifying the column where the temperature (in Kelvin) is stored.
- `pressure` For specifying the column where the pressure (in Pascals) is stored.
- `density` For specifying the column where the density (in $\mathrm{kg} \cdot \mathrm{m}^{-3}$) is stored.
- `molar mass` For specifying the molar mass (in $\mathrm{g} \cdot \mathrm{mol}^{-1}$) of the gas.
- `min height` For setting the minimum altitude in the `chombo-discharge` internal table.
- `max height` For setting the minimum altitude in the `chombo-discharge` internal table.
- `res height` For setting the height resolution in the `chombo-discharge` internal table.

For example, assume that our file `MyAtmosphere.dat` contains the following data:

```
# z [m]              rho [kg/m^3]    T [K]           p [Pa]
0.0000000E+00        1.2900000E+00   2.7210000E+02   1.0074046E+05
1.0000000E+03        1.1500000E+00   2.6890000E+02   8.8751220E+04
2.0000000E+03        1.0320000E+00   2.6360000E+02   7.8074784E+04
3.0000000E+03        9.2860000E-01   2.5690000E+02   6.8466555E+04
4.0000000E+03        8.3540000E-01   2.4960000E+02   5.9844569E+04
```

If we want to truncate this data to altitude $z \; in [1000\,\mathrm{m}, 3000\,\mathrm{m}]$ we specify:

```
{"gas":
  {
    "law": "table",
    "file": "ENMSIS_Atmosphere.dat",
    "molar mass": 28.97,
    "height": 0,
    "temperature": 2,
    "pressure": 3,
    "density": 1,
    "min height": 1000,
    "max height": 3000,
    "res height": 10
  }
}
```

Neutral species background

Neutral species are included by an array `neutral species` in the `gas` object. Each neutral species must have the fields

- `name` Species name
- `molar fraction` Molar fraction of the species.

If the molar fractions do not add up to one, they will be normalized.

Warning: Neutral species are *not* tracked on the mesh. They are simply stored as functions that allow us to obtain the (spatially varying) density, temperature, and pressure for each neutral species. If a neutral species needs to be tracked on the mesh (through e.g. a convection-diffusion-reaction solver) it must be defined as a plasma species. See *Plasma species*.

For example, a standard nitrogen-oxygen atmosphere will look like:

```
  {"gas":
    {
      "law": "ideal",
      "temperature": 300,
      "pressure": 1,
      "plot": true,
      "neutral species":
      [
        {
          "name": "O2",
          "molar_fraction": 0.2
        },
        {
          "name": "N2",
          "molar_fraction": 0.8
        }
    ]
}
```

Plasma species

The list of plasma species is included by an array `plasma species`. Each entry *must* have the entries

- `name` (string) For identifying the species name.
- `Z` (integer) Species charge number.
- `mobile` (true/false) Mobile species or not.
- `diffusive` (true/false) Diffusive species or not.

Optionally, the field `initial data`, can be included for providing initial data to the species Details are discussed further below.

For example, a minimum version would look like

```
{"plasma species":
  [
    {"name": "N2+", "Z":  1, "mobile": false, "diffusive": false},
    {"name": "O2+", "Z":  1, "mobile": false, "diffusive": false},
    {"name": "O2-", "Z": -1, "mobile": false, "diffusive": false}
  ]
}
```

Initial data

To provide initial data one include `initial data` for each species. Currently, the following fields are supported:

- `uniform` For specifying a uniform background density. Simply the field `uniform` and a density (in units of m^{-3})
- `gauss2` for specifying Gaussian seeds $n = n_0 \exp\left(-\frac{(\mathbf{x}-\mathbf{x}_0)^2}{2R^2}\right)$. `gauss2` is an array where each array entry must contain
  - `radius`, for specifying the radius R:
  - `amplitude`, for specifying the amplitude n_0.
  - `position`, for specifying the seed position $\mathbf{x}$.

  The position must be a 2D/3D array.
- `gauss2` for specifying Gaussian seeds $n = n_0 \exp\left(-\frac{(\mathbf{x}-\mathbf{x}_0)^4}{2R^4}\right)$. `gauss4` is an array where each array entry must contain
  - `radius`, for specifying the radius R:
  - `amplitude`, for specifying the amplitude n_0.
  - `position`, for specifying the seed position $\mathbf{x}$.

  The position must be a 2D/3D array.
- `height profile` For specifying a height profile along y in 2D, and z in 3D. To include it, prepare an ASCII files with at least two columns. The height (in meters) must be specified in one column and the density (in units of m^{-3}) in another. Internally, this data is stored in a lookup table (see *Lookup tables*). Required fields are
  - `file` , for specifying the file.
  - `height`, for specifying the column that stores the height.
  - `density`, for specifying the column that stores the density.
  - `min height`, for trimming data to a minimum height.
  - `max height`, for trimming data to a maximum height.
  - `res height`, for specifying the resolution height in the `chombo-discharge` lookup tables.

  In addition, height and density columns can be scaled in the internal tables by including
  - `scale height` for scaling the height data.
  - `scale density` for scaling the density data.

Note: When multiple initial data fields are specified, `chombo-discharge` takes the superposition of all of them.

For example, a species with complex initial data can look like:

```
{"plasma species":
  [
    {
      "name": "N2+",
      "Z":  1,
      "mobile": false,
      "diffusive": false,
      "initial data": {
        "uniform": 1E10,
        "gauss2" :
```

(continues on next page)

(continued from previous page)

```
          [
            {
               "radius": 100E-6,
               "amplitude": 1E18,
               "position": [0,0,0]
            },
            {
               "radius": 200E-6,
               "amplitude": 2E18,
               "position": [1,0,0]
            }
          ],
        "gauss4":
          [
            {
               "radius": 300E-6,
               "amplitude": 3E18,
               "position": [0,1,0]
            },
            {
               "radius": 400E-6,
               "amplitude": 4E18,
               "position": [0,0,1]
            }
          ],
        "height profile": {
          "file": "MyHeightProfile.dat",
          "height": 0,
          "density": 1,
          "min height": 0,
          "max height": 100000,
          "res height": 10,
          "scale height": 100,
          "scale density": 1E6
        }
      }
    }
  ]
}
```

Mobilities

If a species is specified as mobile, the mobility is set from a field `mobility`, and the field `lookup` is used to specify the method for computing it. Currently supported are:

- Constant mobility.
- Function-based mobility, i.e. $\mu = \mu(E, N)$.
- Tabulated mobility, i.e. $\mu = \mu(E, N)$.

The cases are discussed below.

Constant mobility

Setting `lookup` to `constant` lets the user set a constant mobility. If setting a constant mobility, the field `value` is also required. For example:

```
{"plasma species":
  [
    {"name": "e", "Z":  -1, "mobile": true, "diffusive": false,
     "mobility": {
       "lookup" : "constant",
       "value": 0.05,
     }
    }
  ]
}
```

Function-based mobility

Setting `lookup` to `function E/N` lets the user set the mobility as a function of the reduced electric field. When setting a function-based mobility, the field `function` is also required.

Supported functions are:

- `ABC`, in which case the mobility is computed as

$$\mu(E) = A\frac{E^B}{N^C}.$$

The fields `A`, `B`, and `C` must also be specified. For example:

```
{"plasma species":
  [
    {"name": "e", "Z":  -1, "mobile": true, "diffusive": false,
     "mobility": {
       "lookup" : "function E/N",
       "function": "ABC",
       "A": 1,
       "B": 1,
       "C": 1
      }
    }
  ]
}
```

Tabulated mobility

Specifying `lookup` to `table E/N` lets the user set the mobility from a tabulated value of the reduced electric field. BOLSIG-like files can be parsed by specifying the header which contains the tabulated data, and the columns that identify the reduced electric field and mobilities. This data is then stored in a lookup table, see *Lookup tables*.

For example:

```
{"plasma species":
  [
    {"name": "e", "Z":  -1, "mobile": true, "diffusive": false,
     "mobility": {
       "lookup" : "table E/N",
       "file": "transport_file.txt",
       "header": "# Electron mobility (E/N, mu*N)",
       "E/N ": 0,
       "mu*N": 1,
       "min E/N": 10,
       "max E/N": 1000,
       "points": 100,
       "spacing": "exponential",
       "dump": "MyMobilityTable.dat"
      }
    }
  ]
}
```

In the above, the fields have the following meaning:

- `file` The file where the data is found. The data must be stored in rows and columns.
- `header`, the contents of the line preceding the table data.
- `E/N`, the column that contains E/N.
- `mu*N`, the column that contains $\mu \cdot E$.
- `min E/N`, for trimming the data range.
- `max E/N`, for trimming the data range.
- `points`, for specifying the number of points in the lookup table.
- `spacing`, for specifying how to regularize the table.

- `dump`, an optional argument (useful for debugging) which will write the table to file.

Note that the input file does *not* need regularly spaced or sorted data. For performance reasons, the tables are always resampled, see *Lookup tables*.

Diffusion coefficients

Setting the diffusion coefficient is done *exactly* in the same was as the mobility. If a species is diffusive, one must include the field `diffusion` as well as `lookup`. For example, the JSON input for specifying a tabulated diffusion coefficient is done by

```
{"plasma species":
  [
    {"name": "e", "Z":  -1, "mobile": false, "true": false,
     "diffusion": {
       "lookup" : "table E/N",
       "file": "transport_file.txt",
       "header": "# Electron diffusion coefficient (E/N, D*N)",
       "E/N ": 0,
       "D*N": 1,
       "min E/N": 10,
       "max E/N": 1000,
       "points": 1000,
       "spacing": "exponential"
      }
    }
  ]
}
```

Temperatures

Plasma species temperatures can set by including a field `temperature` for the plasma species.

Warning: If the `temperature` field is omitted, the species temperature will be set to the gas temperature.

Constant temperature

To set a constant temperature, include the field `temperature` and set `lookup` to constant and specify the temperature through the field `value` as follows:

```
{"plasma species":
  [
    {
      "name": "O2",
      "Z":  0,
      "mobile": false,
      "true": false,
      "temperature": {
        "lookup": "constant",
        "value": 300
      }
     }
  ]
}
```

Tabulated temperature

To include a tabulated temperature $T = T(E, N)$, set `lookup` to `table E/N`. The temperature is then computed as

$$T = \frac{2\epsilon}{3k_{\mathrm{B}}},$$

where ϵ is the energy and k_{B} is the Boltzmann constant.

The following fields are required:

- `file` for specifying which file the temperature is stored.
- `header` for specifying where in the file the temperature is stored.
- `E/N` for specifying in which column we find E/N.
- `eV` for specifying in which column we find the species energy (in units of electron volts).
- `min E/N` for trimming the data range.
- `max E/N` for trimming the data range.
- `points` for setting the number of points in the lookup table.
- `spacing` for setting the grid point spacing type.
- `dump` for writing the final table to file.

For a further explanation to these fields, see *Mobilities*.

A complete example is:

```
{"plasma species":
  [
    {
      "name": "e",
      "Z":  -1,
      "mobile": true,
      "true": true,
      "temperature": {
        "lookup": "table E/N",
        "file": "transport_data.txt",
        "header": "# Electron mean energy (E/N, eV)",
        "E/N": 0,
        "eV": 1,
        "min E/N": 10,
        "max E/N": 1000,
        "points": 1000,
        "spacing": "exponential",
        "dump": "MyTemperatureTable.dat"
      }
     }
  ]
}
```

Photon species

As for the plasma species, photon species (for including radiative transfer) are included by an array `photon species`. For each species, the required fields are

- `name` For setting the species name.
- `kappa` For specifying the absorption coefficient.

Currently, `kappa` can be either

- `constant` Which lets the user set a constant absorption coefficient.
- `helmholtz` Computes the absorption coefficient as

$$\kappa = \frac{p_X \lambda}{\sqrt{3}}$$

where λ is a specified input parameter and p_X is the partial pressure of some species X.

- `stochastic A` which samples a random absorption coefficient as

$$\kappa = K_1 \left(\frac{K_2}{K_1} \right)^{\frac{f-f1}{f2-f1}}.$$

  Here, f_1 and f_2 are frequency ranges, K_1 and K_2 are absorption coefficients, and f is a stochastically sampled frequency. Note that this method is only sensible when using discrete photons.

Constant absorption coefficients

When specifying a constant absorption coefficient, one must include a field `value` as well. For example:

```
{"photon species":
   [
     {
       "name": "UVPhoton",
       "kappa": "constant",
       "value": 1E4
     }
   ]
}
```

Helmholtz absorption coefficients

The interface for the Helmholtz-based absorption coefficients are inspired by Bourdon *et al.* [2] approach for computing photoionization. This method only makes sense if doing a Helmholtz-based reconstruction of the photoionization profile as a relation:

$$\left[\nabla^2 - \left(p_{\mathrm{O}_2}\lambda\right)^2\right] S_\gamma = -\left(A p_{\mathrm{O}_2}^2 \frac{p_q}{p+p_q} \xi\nu \right) S_i,$$

where

- S_γ is the number of photoionization events per unit volume and time.
- A is a model coefficient.
- $\frac{p_q}{p+p_q}$ is a quenching factor.
- ξ is a photoionization efficiency.
- ν is a relative excitation efficiency.
- S_i is the electron impact ionization source term.

Since the radiative transfer solver is based on the Eddington approximation, the Helmholtz reconstruction can be written as

$$\kappa\Psi - \nabla \cdot \left(\frac{1}{3\kappa} \nabla \Psi \right) = \frac{\eta}{c}$$

where the absorption coefficient is set as

$$\kappa(\mathbf{x}) = \frac{p_{\mathrm{O}_2}\lambda}{\sqrt{3}}.$$

The photogeneration source term is still

$$\eta = \frac{p_q}{p+p_q} \xi\nu S_i,$$

but the photoionization term is

$$S_\gamma = \frac{cAp_{\mathrm{O}_2}}{\sqrt{3}\lambda}\Psi.$$

Note that the photoionization term is, in principle, *not* an Eddington approximation. Rather, the Eddington-like equations occur here through an approximation of the exact integral solution to the radiative transfer problem. In the pure Eddington approximation, on the other hand, Ψ represents the total number of ionizing photons per unit volume, and we would have $S_\gamma = \frac{\Psi}{\Delta t}$ where Δt is the time step.

When specifying the `kappa` field as `helmholtz`, the absorption coefficient is computed as

$$\kappa(\mathbf{x}) = \frac{p_X\left(\mathbf{x}\right)\lambda}{\sqrt{3}}$$

where p_X is the partial pressure of a species X and λ is the same input parameter as in the Helmholtz reconstruction. These are specified through fields `neutral` and `lambda` as follows:

```
{"photon species":
   [
     {
       "name": "UVPhoton",
       "kappa": "helmholtz",
       "lambda": 0.0415,
       "neutral": "O2"
     }
   ]
}
```

This input will set $\kappa\left(\mathbf{x}\right) = \frac{p_{\mathrm{O}_2}(\mathbf{x})\lambda}{\sqrt{3}}$.

Note: The source term η is specified when specifying the plasma reactions, see *Plasma reactions*.

Stochastic sampling

Setting the `kappa` field to `stochastic A` will stochastically sample the absorption length from

$$\kappa = K_1\left(\frac{K_2}{K_1}\right)^{\frac{f-f1}{f2-f1}}.$$

where $K_1 = p_X\chi_{\mathrm{min}}$, $K_1 = p_X\chi_{\mathrm{max}}$, and f_1 and f_2 are frequency ranges. Like above, p_X is the partial pressure of some species X. Note that all input parameters are given in SI units.

Stochastic sampling of the absorption length only makes sense when using discrete photons – this particular method is inspired by the method in Chanrion and Neubert [3]. For example:

```
{"photon species":
   [
     {
       "name": "UVPhoton",
       "kappa": "stochastic A",
       "neutral": "O2",
       "f1":   2.925E15,
       "f2":   3.059E15,
       "chi min": 2.625E-2,
       "chi max": 1.5
     }
   ]
}
```

Plasma reactions

Plasma reactions are reactions between charged and neutral species and are written in the form

$$A + B + \ldots \rightarrow C + D + \ldots.$$

Importantly, the left hand side of the reaction can only consist of charged or neutral species. It is not permitted to put a photon species on the left hand side of these reactions; photo-ionization is handled separately by another set of reaction types (see *Photo-reactions*). However, photon species *can* appear on the left hand side of the equation.

When specifying reactions in this form, the reaction rate is computed as

$$R = k n_A n_B \ldots$$

When computing the source term for some species X, we subtract R for each time X appears on the left hand side of the reaction and add R for each time X appears on the right-hand side of the reaction.

Specifying reactions

Reactions of the above type are handled by a JSON array `plasma reactions`, with required fields:

- `reaction` (string) containing the reaction process.
- `lookup` (string) for determining how to compute the reaction rate.

```
{"plasma reactions":
  [
    {
      "reaction": "e + O2 -> e + e + O2+",
      "lookup": "constant",
      "rate": 1E-30
    }
  ]
}
```

This adds a reaction $\mathrm{e} + \mathrm{O}_2 \rightarrow \mathrm{e} + \mathrm{e} + \mathrm{O}_2^+$ to the reaction set. We compute

$$R = k n_{\mathrm{e}} n_{\mathrm{O}_2^+}$$

and set

$$S_{\mathrm{e}} = S_{\mathrm{O}_2^+} = R.$$

Some caveats when setting the reaction string are:

- Whitespace are separators. For example, `O2+e` will be interpreted as a species with string identifier `O2+e`, but `O2 + e` will interpreted as a reaction between `O2` and `e`.
- The reaction string *must* contain a left and right hand side separated by `->`. An error will be thrown if this symbol can not be found.
- The left-hand must consist *only* of neutral or plasma species. If the left-hand side consists of species that are not neutral or plasma species, an error will be thrown.
- The right-hand side can consist of either neutral, plasma species, or photon species. Otherwise, an error will be thrown.
- The reaction string will be checked for charge conservation.

Note that if a reaction involves a right-hand side that is not otherwise tracked, the user should omit the species from the right-hand side altogether. For example, if we have a model which tracks the species e and O_2^+ but we want to include the dissociative recombination reaction $e + \mathrm{O}_2^+ \rightarrow O + O$, this reaction should be added to the reaction with an empty right-hand side:

```
{"plasma reactions":
  [
    {
      "reaction": "e + O2 -> e + e + O2+",
      "lookup": "constant",
      "rate": 1E-30
    },
    {
      "reaction": "e + O2+ -> ",
      "lookup": "constant",
      "rate": 1E-30
    }
  ]
}
```

Wildcards

Reaction specifiers may include the wildcard `@` which is a placeholder for another species. The wildcards must be specified by including a JSON array `@` of the species that the wildcard is replaced by. For example:

```
{"plasma reactions":
  [
    {
      "reaction": "N2+ + N2 + @ -> N4+ + @",
      "@": ["N2", "O2"],
      "lookup": "constant",
      "rate": 1E-30
    }
  ]
}
```

The above code will add two reactions to the reaction set: $N_2 + N_2 + N_2 \rightarrow N_4^+ + N_2$ and $N_2 + N_2 + \mathrm{O}_2 \rightarrow N_4^+ + \mathrm{O}_2$. It is not possible to set different reaction rates for the two reactions.

Plotting reactions

It is possible to have `CdrPlasmaJSON` include the reaction rates in the HDF5 output files by including a field `plot` as follows:

```
{"plasma reactions":
  [
    {
      "reaction": "e + O2 -> e + e + O2+",
      "plot": true,
      "lookup": "constant",
      "rate": 1E-30,
    }
  ]
}
```

Plotting the reaction rate can be useful for debugging or analysis. Note that it is, by extension, also possible to add useful data to the I/O files from reactions that otherwise do not contribute to the discharge evolution. For example, if we know the rate k for excitation of nitrogen to a specific excited state, but do not otherwise care about tracking the excited state, we can add the reaction as follows:

```
{"plasma reactions":
  [
    {
      "reaction": "e + N2 -> e + N2",
      "plot": true,
      "lookup": "constant",
      "rate": 1E-30,
    }
  ]
}
```

This reaction is a dud in terms of the discharge evolution (the left and right hand sides are the same), but it can be useful for plotting the excitation rate.

Note: This functionality should be used with care because each reaction increases the I/O load.

Constant reaction rates

To set a constant reaction rate for a reaction, set the field `lookup` to `"constant"` and specify the rate. For example:

```
{"plasma reactions":
  [
    {
      "reaction": "e + O2 -> e + e + O2+",
      "lookup": "constant",
      "rate": 1E-30
    }
  ]
}
```

Function based rates

- To set a rate dependent on two species temperature in the form $k(T_1, T_2) = c_1 \left(T_1/T_2\right)^{c_2}$, set `lookup` to `functionT1T2 A`. The user must specify which temperatures are involved by specifying the fields `T1`, `T2`, as well as the constants through fields `c1` and `c2`. For example, to include the reaction $e + \mathrm{O}_2 + \mathrm{O}_2 \rightarrow \mathrm{O}_2^- + O2$ in the set, with this reaction having a rate

$$k = 2.4 \times 10^{-41} \left(\frac{T_{\mathrm{O}_2}}{T_e}\right),$$

  we add the following:

```
{"plasma reactions":
  [
    {
      "reaction": "e + O2 + O2 -> O2- + O2",
      "lookup": "functionT1T2 A",
      "T1": "O2",
      "T2": "e",
      "c1": 2.41E-41,
      "c2": 1
    }
  ]
}
```

Tabulated rates

To set a tabulated rate with $k = k(E, N)$, set the field `lookup` to `table E/N` and specify the file, header, and data format to be used. For example:

```
{"plasma reactions":
  [
    {
      "reaction": "e + O2 -> e + e + O2+",
      "lookup": "table E/N",
      "file": "transport_file.txt",
      "header": "# O2 ionization (E/N, rate/N)",
      "E/N ": 0,
      "rate/N": 1,
      "min E/N": 10,
```

(continues on next page)

(continued from previous page)

```
      "max E/N": 1000,
      "spacing": "exponential",
      "points": 1000,
      "plot": true,
      "dump": "O2_ionization.dat"
    }
  ]
}
```

The `file` field specifies which field to read the reaction rate from, while `header` indicates where in the file the reaction rate is found. The file parser will read the files below the header line until it reaches an empty line. The fields `E/N` and `rate/N` indicate the columns where the reduced electric field and reaction rates are stored.

The final fields `min E/N`, `max E/N`, and `points` are formatting fields that trim the range of the data input and organizes the data along a table with `points` entries. As with the mobilities (see Chap:CdrPlasmaJSONMobilities), the `spacing` argument determines whether or not the internal interpolation table uses uniform or exponential grid point spacing. Finally, the `dump` argument will tell `chombo-discharge` to dump the table to file, which is useful for debugging or quality assurance of the tabulated data.

Collisional quenching

To quench a reaction, include a field `qenching_pressure` and specify the *quenching pressure* (in atmospheres). When computing reaction rates, the rate for the reaction will be modified as

$$k \rightarrow k\frac{p_q}{p_q + p}$$

where p^q is the quenching pressure and $p = p(\mathbf{x})$ is the gas pressure.

Important: The quenching pressure should be specified in Pascal.

For example:

```
{"plasma reactions":
  [
    {
      "reaction": "e + N2 -> e + N2 + Y",
      "lookup": "table E/N",
      "file": "transport_file.txt",
      "header": "# N2 ionization (E/N, rate/N)",
      "E/N ": 0,
      "rate/N": 1,
      "min E/N": 10,
      "max E/N": 1000,
      "points": 1000,
      "spacing": "exponential",
      "quenching pressure": 4000
    }
  ]
}
```

Reaction efficiencies

To modify a reaction efficiency, include a field `efficiency` and specify it. This will modify the reaction rate as

$$k \rightarrow \nu k$$

where ν is the reaction efficiency. For example:

```
{"plasma reactions":
  [
    {
      "reaction": "e + N2 -> e + N2 + Y",
      "lookup": "table E/N",
      "file": "transport_file.txt",
      "header": "# N2 ionization (E/N, rate/N)",
      "E/N ": 0,
      "rate/N": 1,
      "min E/N": 10,
      "max E/N": 1000,
      "points": 1000,
      "spacing": "exponential",
      "efficiency": 0.6
    }
  ]
}
```

Scaling reactions

Reactions can be scaled by including a `scale` argument to the reaction. This works exactly like the `efficiency` field outlined above.

Energy correction

Occasionally, it can be necessary to incorporate an energy correction to models, accounting e.g. for electron energy loss near strong gradients. The JSON interface supports the correction in Soloviev and Krivtsov [7]. To use it, include an (optional) field `soloviev` and specify `correction` and `species`. For example:

```
{"plasma reactions":
  [
    {
      "reaction": "e + N2 -> e + N2 + Y",
      "lookup": "table E/N",
      "file": "transport_file.txt",
      "header": "# N2 ionization (E/N, rate/N)",
      "E/N ": 0,
      "rate/N": 1,
      "min E/N": 10,
      "max E/N": 1000,
      "points": 1000,
      "spacing": "exponential",
      "efficiency": 0.6,
      "soloviev": {
        "correction": true,
        "species": "e"
      }
    }
  ]
}
```

When this energy correction is enabled, the rate coefficient is modified as

$$k \rightarrow k\left(1 + \frac{\mathbf{E}\cdot D_s \nabla n_s}{\mu_s n_s E^2}\right),$$

where s is the species specified in the `soloviev` field, n_s is the density and D_s and μ_s are diffusion and mobility coefficients. We point out that the correction factor is restricted such that the reaction rate is always non-negative. Note that this correction makes sense when rates are dependent only on the electric field, see Soloviev and Krivtsov [7].

Note: When using the energy correction, the specifies species must be both mobile and diffusive.

Photo-reactions

Photo-reactions are reactions between charged/neutral and photons in the form

$$A + B + \gamma + \ldots \rightarrow C + D + \ldots.$$

where species $A, B, \ldots$ are charged and neutral species and γ is a photon. The left hand side can contain only *one* photon species, and the right-hand side can not contain a photon species. In other words, two-photon absorption is not supported, and photons that are absorbed on the mesh cannot become new photons. This is not a fundamental limitation, but a restriction imposed by the JSON interface.

Specifying reactions

Reactions of the above type are handled by a JSON array `photo reactions`, with required fields:

- `reaction` (string) containing the reaction process.
- `lookup` (string) for determining how to compute the reaction rate.

For example:

```
{"photo reactions":
  [
    {"reaction": "Y + O2 -> e + O2+"},
  ]
}
```

The rules for specifying reaction strings are the same as for the plasma reactions, see CdrPlasmaReactionsJSON. Wildcards also apply, see *Wildcards*.

Default behavior

Since the radiative transfer solvers solve for the number of ionizing photons, the CDR solver source terms are incremented by

$$S \rightarrow S + \frac{\Psi}{\Delta t}.$$

where Ψ is the number of ionizing photons per unit volume (i.e., the solution Ψ).

Helmholtz reconstruction

When performing a Helmholtz reconstruction the photoionization source term is

$$S = \frac{cAp_{\mathrm{O}_2}}{\sqrt{3}\lambda}\Psi.$$

To modify the source term for consistency with Helmholtz reconstruction specify the field `helmholtz` with variables

- `A`. the A coefficient.
- `lambda`. the λ coefficient. This value will also be specified in the photon species, but it is not retrieved automatically.
- `neutral`. The neutral species for which we obtain the partial pressure.

For example:

```
{"photo reactions":
  [
    {
      "reaction": "Y + O2 -> e + O2+",
      "helmholtz": {
        "A": 1.1E-4,
        "lambda": 0.0415,
        "neutral": "O2"
      }
    }
  ]
}
```

Scaling reactions

Photo-reactions can be scaled by including a `scale` argument. For example, to completely turn off the photoreaction above:

```
{"photo reactions":
  [
    {
      "reaction": "Y + O2 -> e + O2+",
      "helmholtz": {
        "A": 1.1E-4,
        "lambda": 0.0415,
        "neutral": "O2"
      },
      "scale": 0.0
    }
  ]
}
```

EB boundary conditions

Boundary conditions on the embedded boundary are included by the fields

- `electrode reactions`, for specifying secondary emission on electrodes.
- `dielectric reactions`, for specifying secondary emission on dielectrics.

To include secondary emission, the user must specify a reaction string in the form $A \rightarrow B$, and also include an emission rate. Currently, we only support constant emission rates (i.e., secondary emission coefficients). This is likely to change in the future.

The following points furthermore apply:

- By default, standard outflow boundary conditions. When `electrode reactions` or `dielectric reactions` are specified, the user only controls the *inflow* back into the domain.
- Wildcards can appear on the left hand side of the reaction.
- If one specifies $A + B \rightarrow C$ for a surface reaction, this is the same as specifying two reactions $A \rightarrow C$ and $B \rightarrow C$. The same emission coefficient will be used for both reactions.
- Both photon species and plasma species can appear on the left hand side of the reaction.
- Photon species can not appear on the right-hand side of the reaction; we do not include surface sources for photoionization.
- To scale reactions, include a modifier `scale`.

For example, the following specification will set secondary emission efficiencies to 10^{-3}:

```
{"electrode reactions":
  [
    { "reaction": "@ -> e",
      "@": ["N2+", "O2+", "N4+", "O4+", "O2+N2"],
      "lookup": "constant",
      "value": 1E-4
    }
  ],
 "dielectric reactions":
  [
    { "reaction": "@ -> e",
      "@": ["N2+", "O2+", "N4+", "O4+", "O2+N2"],
      "lookup": "constant",
      "value": 1E-3
    }
  ]
```

Domain boundary conditions

TODO.

5.4.5 Simulation quick start

New problems that use the `CdrPlasma` physics model are best set up by using the Python tools provided with the module. Navigate to `$DISCHARGE_HOME/Physics/CdrPlasma` and set up the problem with. To see the list of available options type

```
cd $DISCHARGE_HOME/Physics/CdrPlasma
./setup.py --help
```

The following options are helpful for setting up the problem:

- `base_dir` The base directory where the application will be placed. Defaults to `$DISCHARGE_HOME/MyApplications`.
- `app_name` The application name. The application will be put in `base_dir/app_name`.
- `geometry` The geometry to be used. The geometry must be one of the ones provided in `$DISCHARGE_HOME/Geometries` (users can also provide their own models).
- `physics` The plasma physics model. This must be one of the folders/class in `$DISCHARGE_HOME/Physics/CdrPlasma/PlasmaModel` (users can also provide their own models). Defaults to `CdrPlasmaJSON` (see *JSON interface*).
- `time_stepper` Time integrator. This must derive from `CdrPlasmaStepper` and must be one of the time steppers in `$DISCHARGE_HOME/Physics/CdrPlasma/TimeSteppers`. The default integrator is `CdrPlasmaGodunovStepper`.

- `cell_tagger` Cell tagger This must derive from `CdrPlasmaTagger` and must be one of the cell taggers in `$DISCHARGE_HOME/Physics/CdrPlasma/CellTaggers`.

For example, to set up a geometry-less that does not use AMR, do

```
cd $DISCHARGE_HOME
./setup.py -app_name=MyApplication
```

5.5 Electrostatics

5.6 Geometry

5.7 Radiative transfer

CHAPTER

SIX

TUTORIAL

6.1 Introduction

In this tutorial we will set up and examine the code for simulating advection-diffusion problems with AMR and regridding functionality. If you want to examine the final code, it is given in `$DISCHARGE_HOME/Physics/AdvectionDiffusion`.

6.2 Creating a geometry

6.3 Setting up a `TimeStepper`

CHAPTER

SEVEN

UTILITIES

7.1 Lookup tables

`LookupTable` is a class for looking up and interpolation data stored in a row-column format. It is used in order to easily retrieve input data that can be stored in table formats.

Important: LookupTable is used for data lookup *in one independent variable*. It does not support higher-dimensional data interpolation.

The class is templated as

```
template <int N>
class LookupTable
```

where the template parameter `N` indicates the number of columns in the data holders. Internally, the data is stored as an `std::vector<std::array<Real, N> >` where the vector entries are rows and the `std::array<Real, N>` are data in each row. Thus, a table `LookupTable<2>` always has two columns.

The `LookupTable` is used on regularly spaced data (for performance reasons). Although the user can fill irregularly spaced data into `LookupTable`, the class has routines for making that data regularly spaced and sorted along of its columns. Usage of `LookupTable` will therefore consist of the following:

1. Add data rows into the table.
2. Swap columns if necessary.
3. Restrict data ranges if necessary.
4. Sort the table along of it's columns, smallest to largest.
5. Regularize the table with a specified number of grid points.
6. *Retrieve data*.

The steps for these processes are explained in detail below.

7.1.1 Inserting data

To add data to the table, one will use the member function

```
template <typename... Ts>
inline
void addEntry(const Ts&... x);
```

where the parameter pack must have `N` entries. For example, to add two rows of data to a table:

```
LookupTable<3> myTable;

myTable.add(4.0, 5.0, 6.0);
myTable.add(1.0, 2.0, 3.0);
```

This will insert two new rows at the end up the table.

Important: Input data points do not need to be uniformly spaced, or even sorted. Users will insert rows one by one; `LookupTable` has functions for sorting and regularizing the table.

7.1.2 Restricting ranges

To restrict the data range, call

```
void setRange(const Real a_min, const Real a_max, const int a_independentVariable)
```

where `a_min` and `a_max` are the permissible ranges for data in the input column (`a_independentVariable`). Data outside these ranges are discarded from the table.

7.1.3 Independent variable

To select an independent variable, the user will select one of the columns and sort the data along that column. The C++ code for this is

```
template <int N>
void LookupTable<N>::sort(const int a_independentVariable);
```

where the input integer indicates the column (i.e., independent variable) used for sorting. Note that the sorting *is always from smallest to largest value*. Thus, if one has two rows

```
1.0  5.0  6.0
2.0  2.0  3.0
```

and one calls `LookupTable<N>::sort(1)` the final table becomes

```
2.0  2.0  3.0
1.0  5.0  6.0
```

Note that the second column now becomes the independent variable.

7.1.4 Swapping columns

Columns can be swapped by calling `LookupTable<N>::swap(int, int)`, which will swap two of the columns. For example if the original data is

```
2.0  2.0  3.0
1.0  5.0  6.0
```

and one calls `swap(1,2)` the final table becomes

```
2.0  3.0  2.0
1.0  6.0  5.0
```

Note that swapping two columns destroys the sorting and one will need to set the independent variable again afterwards.

7.1.5 Regularize table

To regularize the table the user must first determine if the grid points should be uniformly spaced or exponentially spaced in the independent variable.

Setting grid point spacing

The user can set the spacing by calling

```
void setTableSpacing(const TableSpacing a_spacing);
```

where `TableSpacing::Uniform` and `TableSpacing::Exponential` are supported. For example, to use uniformly or exponentially spaced grid points:

```
LookupTable<2> myTable;

myTable.setTableSpacing(TableSpacing::Uniform);     // For uniformly spaced points
myTable.setTableSpacing(TableSpacing::Exponential); // For exponentially spaced points
```

Uniform spacing

With uniform spacing, grid points in the table are spaced as

$$x_i = x_{\mathrm{min}} + \frac{i}{N-1}\left(x_{\mathrm{max}} - x_{\mathrm{min}}\right), \quad i \in [0, N-1]$$

where x_{min} and x_{max} is the minimum and maximum data range for the independent variable (i.e., column).

Exponential spacing

If grid points are exponentially spaced then

$$x_i = x_{\mathrm{min}} \left(\frac{x_{\mathrm{max}}}{x_{\mathrm{min}}}\right)^{\frac{i-1}{N}}, \quad i \in [0, N-1].$$

Regularizing table

```
void regularize(const int a_numRows)
```

which will make the table into a regularly spaced table with `a_numRows` rows. `LookupTable` will always use piecewise linear interpolation when regularizing the table. Specifying a number of rows that is smaller/larger than the original number of rows will downsample/upsample the table.

Important: When regularizing a table through `regularize`, the original data table is destroyed.

7.1.6 Retrieving data

To retrieve data from one of the columns, one can fetch either a specific value in a row, or the entire row.

```
// For fetching column K
template<int K>
Real getEntry(const Real a_x);

// For fetching the entire row
std::array<Real, N> getData(const Real a_x);
```

In the above, the template parameter `K` is the column to retrieve and `a_x` is the value of the independent variable.

Important: `LookupTable` will *always* use piecewise linear interpolation between two grid points.

For example, consider table regularized and sorted along the middle column:

```
2.0  1.0  3.0
1.0  3.0  6.0
1.0  5.0  4.0
```

To retrieve an interpolated value for `x=2.0` in the third column we call

```
LookupTable<3> myTable,

const Real val = myTable.getEntry<2>(2.0);
```

which will return a value of 4.5 (linearly interpolated).

7.1.7 Viewing tables

For debugging purposes, `LookupTable` can write the internal data to an output stream or a file through the two member functions:

```
void dumpTable(std::ostream& a_outputStream = std::cout) const;

void dumpTable(const std::string a_fileName) const;
```

For example:

```
LookupTable<10> myTable;

// Dump table to terminal window
myTable.dumpTable();

// Dump table to file.
myTable.dumpTable("myTable.dat");
```

7.2 Random numbers

`Random` is a static class for generating pseudo-random numbers, and exist so that all random number operations can be aggregated into a single class. Internally, `Random` use a Mersenne-Twister random number generation.

To use the `Random` class, simply include `<CD_Random.H>`, e.g.

```
#include <CD_Random.H>
```

See the Random API for further details.

7.2.1 Drawing random numbers

The general routine for drawing a random number is

```
template<typename T>
Real get(T& a_distribution);
```

which is for example used as follows:

```
std::uniform_real_distribution<Real> dist(0.0, 100.0);

const Real randomNumber = Random::get(dist);
```

Pre-defined distributions exist for performing the following operations:

1. For drawing a real number from a uniform distribution between 0 and 1, use `Real Random::getUniformReal01()`.
2. For drawing a real number from a uniform distribution between -1 and 1, use `Real Random::getUniformReal11()`.
3. For drawing a real number from a normal distribution centered at 0 and with a variance of 1, use `Real Random::getNormal01()`.
4. For drawing an integer from a Poisson distribution with a specified mean, use `T Random::getPoisson<T>(const Real a_mean)` where `T` is an integer type.
5. For drawing a random direction in space, use `RealVect Random::getDirection()`. The implementation uses the Marsaglia algorithm for drawing coordinates uniformly distributed over the unit sphere.

7.2.2 Setting the seed

By default, the random number generator is seeded with the MPI rank, which we do to avoid having the MPI ranks producing the same number sequences. `Driver` (see *Driver*) will seed the random number generator, and user can override the seed by setting `Random.seed = <number>` in the input script. If the user sets `<number> < 0` then a random seed will be produced based on the elapsed CPU clock time. If running with MPI, this seed is obtained by only one of the MPI ranks, and this seed is then broadcast to all the other ranks. The other ranks will then increment the seed by their own MPI rank number so that each MPI rank gets a unique seed.

7.3 Least squares

Least squares routines are useful for reconstructing a local polynomial in the vicinity of the embedded boundary. `chombo-discharge` supports the expansion of such solutions in a fairly general way. The need for such routines are motivated e.g. by the fact that the embedded boundary introduces grid pathologies which are difficult to meet with pure finite differencing. An example of this is filling ghost cells near the embedded boundary.

7.3.1 Polynomial expansion

Given some position $\mathbf{x}$, we expand the solution around a grid point $\mathbf{x_i}$ to some order Q:

$$f\left(\mathbf{x_i}\right) = f\left(\mathbf{x_i}\right) + \nabla f\left(\mathbf{x}\right) \cdot \left(\mathbf{x_i} - \mathbf{x}\right) + \ldots + \mathcal{O}\left(\Delta x^{Q+1}\right).$$

Using multi-index notation this is written as

$$f\left(\mathbf{x_i}\right) = \sum_{|\alpha| \leq Q} \frac{\left(\mathbf{x_i} - \mathbf{x}\right)^{\alpha}}{\alpha!} \left(\partial^{\alpha} f\right)\left(\mathbf{x}\right) + \mathcal{O}\left(\Delta x^{Q+1}\right),$$

where α is a multi-index. For a specified order Q there is also a specified number of unknowns. E.g. in two dimensions with $Q = 1$ the unknowns are $f\left(\mathbf{x}\right)$, $\partial_x f\left(\mathbf{x}\right)$, and $\partial_y f\left(\mathbf{x}\right)$.

By expanding the solution around more grid points, we can formulate an over-determined system of equations $\mathbf{i} = 1, 2, 3, \ldots, N$ that allows us to compute the coefficients (i.e., unknowns) in the Taylor expansion. By using lexicographical ordering of the multi-indices, it is straightforward to write the system out explicitly. E.g., for $Q = 1$ in two dimensions:

$$\begin{pmatrix} 1 & (x_1 - x) & (y - y_1) \\ 1 & (x_2 - x) & (y - y_2) \\ \vdots & \ddots & \vdots \\ 1 & (x_N - x) & (y - y_N) \end{pmatrix} \begin{pmatrix} f \\ \partial_x f \\ \partial_y f \end{pmatrix} \left(\mathbf{x}\right) = \begin{pmatrix} f(\mathbf{x}_1) \\ f(\mathbf{x}_2) \\ \vdots \\ f(\mathbf{x}_N) \end{pmatrix}$$

In general, we represent this system as

$$\mathbf{A}\mathbf{u} = \mathbf{b},$$

where unknowns in $\mathbf{u}$ are the coefficients in the Taylor series, ordered lexicographically (encoded with a Chombo `IntVect`). $\mathbf{b}$ is a column vector of grid point values representing the local expansion around each grid point, and $\mathbf{A}$ is the expansion matrix.

Note: `chombo-discharge` is not restricted to second order – it implements the above expansion to any order.

7.3.2 Neighborhood algorithm

To avoid reaching over or around embedded boundaries, the neighborhood algorithms only includes grid cells which can be reached by a *monotone* path. This path is defined by walking through neighboring grid cells without changing direction, see e.g. Fig. 7.3.1.

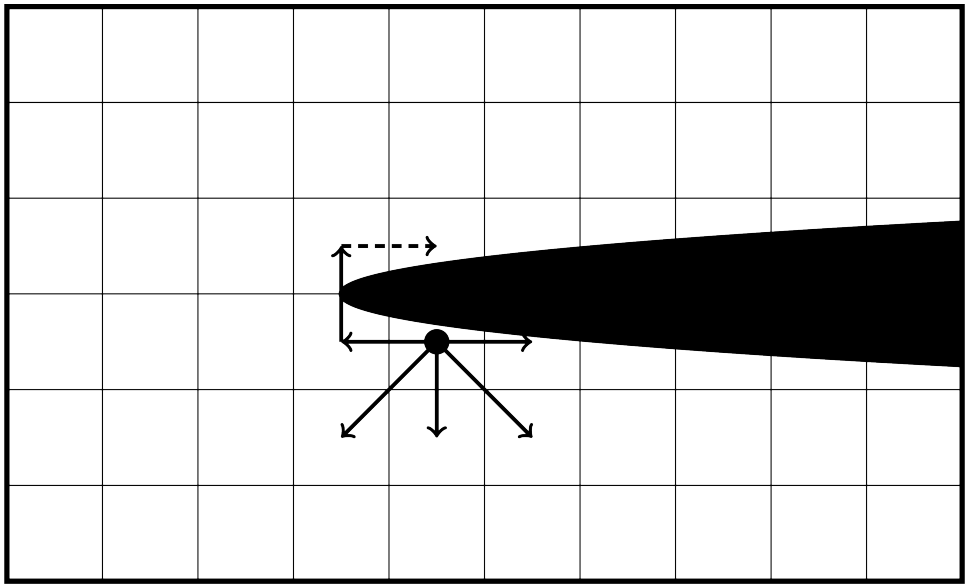

Fig. 7.3.1: Neighborhood algorithm, only reaching into grid cells that can be reached by a monotone path. The grid cell at the end of the dashed line is excluded (even though it is a neighbor to the starting grid cell) since the path circulates the embedded boundary.

7.3.3 Weighted equations

Weights can also be added to each equation, e.g. to ensure that close grid points are more important than remote ones:

$$\begin{pmatrix} w_1 & w_1(x_1-x) & w_N(y-y_1) \\ w_2 & w_2(x_2-x) & w_N(y-y_2) \\ \vdots & \ddots & \vdots \\ w_N & w_N(x_N-x) & w_N(y-y_N) \end{pmatrix} \begin{pmatrix} f \\ \partial_x f \\ \partial_y f \end{pmatrix} (\mathbf{x}) = \begin{pmatrix} w_1 f(\mathbf{x}_1) \\ w_2 f(\mathbf{x}_2) \\ \vdots \\ w_N f(\mathbf{x}_N) \end{pmatrix}$$

For weighted least squares the system is represented as

$$\mathbf{WAu} = \mathbf{Wb},$$

where $\mathbf{W}$ are the weights. Typically, the weights are some power of the Euclidean distance

$$w_{\mathbf{i}} = \frac{1}{|\mathbf{x_i} - \mathbf{x}|^p}.$$

7.3.4 Pseudo-inverse

An over-determined system does not have a unique solution, and so to obtain the solution to $\mathbf{u}$ for the system $\mathbf{WAu} = \mathbf{Wb}$ we use ordinary least squres. The solution is then

$$\mathbf{u} = \left[(\mathbf{WA})^+ \mathbf{W} \right] \mathbf{b},$$

where $(\mathbf{WA})^+$ is the Moore-Penrose inverse of $\mathbf{WA}$. The pseudo-inverse is computed using the singular value decomposition (SVD) routines in LAPACK.

Note that the column vector $\mathbf{b}$ consist of known values (grid points), and the result $\left[(\mathbf{WA})^{+}\mathbf{W}\right]$ can therefore be represented as a stencil. For example, in two dimensions with $Q = 1$ we find

$$\begin{pmatrix} f \\ \partial_x f \\ \partial_y f \end{pmatrix}(\mathbf{x}) = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1N} \\ C_{21} & C_{22} & \ddots & C_{2N} \\ C_{31} & C_{32} & \dots & C_{3N} \end{pmatrix} \begin{pmatrix} f(\mathbf{x}_1) \\ f(\mathbf{x}_2) \\ \vdots \\ f(\mathbf{x}_N) \end{pmatrix}$$

7.3.5 Pruning equations

If some terms in the Taylor series are specified, one can prune equations from the systems. E.g. if $f(\mathbf{x})$ happens to be known, the system of equations can be rewritten as

$$\begin{pmatrix} w_1(x_1 - x) & w_N(y - y_1) \\ w_2(x_2 - x) & w_N(y - y_2) \\ \vdots & \vdots \\ w_N(x_N - x) & w_N(y - y_N) \end{pmatrix} \begin{pmatrix} \partial_x f \\ \partial_y f \end{pmatrix}(\mathbf{x}) = \begin{pmatrix} w_1 f(\mathbf{x}_1) - w_1 f(\mathbf{x}) \\ w_2 f(\mathbf{x}_1) - w_2 f(\mathbf{x}) \\ \vdots \\ w_N f(\mathbf{x}_1) - w_N f(\mathbf{x}) \end{pmatrix}$$

Again, following the benefits of lexicographical ordering it is straightforward to write an arbitrary order system of equations in the form $\mathbf{WAu} = \mathbf{Wb}$, even with an arbitrary number of terms pruned from the Taylor series. However, note that the result of the least squares solve is now in the format

$$\begin{pmatrix} \partial_x f \\ \partial_y f \end{pmatrix}(\mathbf{x}) = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1N} \\ C_{21} & C_{22} & \dots & C_{2N} \end{pmatrix} \begin{pmatrix} f(\mathbf{x}_1) - f(\mathbf{x}) \\ f(\mathbf{x}_2) - f(\mathbf{x}) \\ \vdots \\ f(\mathbf{x}_N) - f(\mathbf{x}) \end{pmatrix}.$$

Thus, when evaluating the terms in the polynomial expansion the user must account for the modified right-hand side due to equation pruning. The modification to the right-hand side also depends on which terms are pruned from the expansion.

7.3.6 Source code

The source code for the least squares routines is found in `$DISCHARGE_HOME/Source/Utilities/CD_LeastSquares.*`, and the neighborhood algorithms are found in `$DISCHARGE_HOME/Source/Utilities/CD_VofUtils.*`.

CHAPTER

EIGHT

CONTRIBUTING

8.1 Contributions

We welcome feedback, bug reports, and contributions to `chombo-discharge`. If you have feedback, questions, or general types of queries, use the issue tracker or discussion tab at https://github.com/chombo-discharge.

If you want to submit code to `chombo-discharge`, use the pull request system at https://github.com/chombo-discharge.

8.2 Convergence testing

8.3 Continuous integration

`chombo-discharge` uses continuous integration at GitHub. These tests consist of cloning, installing, and running various `chombo-discharge` tests.

8.3.1 GitHub actions

The tests defined in `$DISCHARGE_HOME/Exec/Tests` are automatically run when opening a pull request. In general, all tests must pass before a pull request can be merged. The tests status can be observed either in the pull request, or at https://github.com/chombo-discharge/chombo-discharge/actions. GitHub actions usually take 1-2 hours to complete.

8.3.2 Running tests locally

The tests are defined in `$DISCHARGE_HOME/Exec/Tests` and can be run using various configurations. Generally, it is a good idea to run the tests locally before running them at GitHub (which can be slower).

Running all tests

To do a clean compile and run of all tests, navigate to `$DISCHARGE_HOME/Exec/Tests` and execute the following:

```
python3 tests.py --compile --clean --silent --no_compare --parallel -cores X
```

where `X` is the number of cores to use when compiling and running. If the tests should be run in serial, omit the `--parallel -cores X` flag.

Test suite options

The following options are available for running the various tests:

Using benchmark files

The test suite can generate benchmark files which can later be compared against new test suite output files. This can be a good idea if one wants to ensure that changes the `chombo-discharge` code does not affect output files. In this case one can run the test suite and generate benchmark files *before* adding changes to `chombo-discharge`. Once the code development is completed, the benchmark files can later be bit-wise compared against the results of a later test suite.

This consists of the following steps:

1. *Before* making changes to `chombo-discharge`, generate benchmark files with

```
python3 tests.py --compile --clean --silent --parallel -cores X --benchmark
```

2. Make the required changes to the `chombo-discharge` code.
3. Run the test suite again, and compare benchmark and output files as follows:

```
python3 tests.py --compile --clean --silent --parallel -cores X
```

8.4 Code standard

When submitting new code to `chombo-discharge`, the following guidelines below show be followed.

8.4.1 C++ standard

We are currently at `c++14`.

8.4.2 Namespace

All code in `chombo-discharge` is embedded in a namespace `ChomboDischarge`. Embedding into a namespace is done by including header file `CD_NamespaceHeader.H` that contain the necessary definitions. This is done by including after any other file includes. In addition, files must include `CD_NamespaceFooter.H` at the end.

8.4.3 File names

Each file should contain only one class definition, and the file name must be name of the class prepended by `CD_`. For example, if you are contributing a class `MyClass` the header files for this class must be named `CD_MyClass.H` and the implementation file must be named `CD_MyClass.cpp`. If your code contains templates or inlined functions, these should be defined in files appended by `Implem`, e.g. `CD_MyClassImplem.H`.

8.4.4 File headers

Each file shall begin with the following note:

```
/* chombo-discharge
* Copyright © <Copyright holder 1>
* Copyright © <Copyright holder 2>
* Please refer to Copyright.txt and LICENSE in the chombo-discharge root directory.
*/
```

where <*Copyright holder 1*>, <*Copyright holder 2*>, etc. are replaced by the copyright holder.

This file header shall be followed by a brief Doxygen documentation, containing at least `@file`, `@brief`, and `@author`.

File inclusions should use the follow standards for C++, Chombo, and chombo-discharge

1. *C++*. Use brackets, e.g. `#include <memory>`.
2. Chombo. Use brackets, e.g. `#include <LevelData.H>`.
3. chombo-discharge. Use brackets and the file name, e.g. `#include <CD_FieldSolver.H>`.

Here is a complete example of a header file in chombo-discharge:

```
/* chombo-discharge
* Copyright © <Copyright holder 1>
* Copyright © <Copyright holder 2>
* Please refer to Copyright.txt and LICENSE in the chombo-discharge root directory.
*/

/*!
  @file   CD_MyClass.H
  @brief  This file contains ...
  @author Author name
*/

#ifndef CD_MyClass_H
#define CD_MyClass_H

// Std includes (e.g.)
#include <memory>

// Chombo includes (e.g.)
#include <LevelData.H>

// Our includes (e.g.)
#include <CD_EBAMRData.H>
#include <CD_NamespaceHeader.H>

/*!
  @brief This class does the following: ....
*/
class MyClass
{
public:

//...
};

#include <CD_NamespaceFooter.H>

#include <CD_MyClassImplem.H> // Inline and template code included at the end.

#endif
```

8.4.5 Code syntax

Class names, functions, and variables

We use the following syntax:

1. Class names, structs, and namespaces should be in Pascal case where the first letter of every word is capitalized. E.g. a class is called `MyClass`.
2. Class functions should be in Camel case where the first letter of every word but the first is capitalized. E.g. functions should be named `MyClass::myFunction`
3. Variables should use Pascal-case, with the following requirements:
   - Arguments to functions should be prepended by `a_`. For example `MyClass::myFunction(int a_inputVariable)`.
   - Class members should always be prepended by `m_`, indicating it is a member of a class. For example `MyClass::m_functionMember`.
   - Static variables are prepended by `s_`. For example `MyClass::s_staticFunctionMember`.
   - Global variables are prepended by `//`.

Code formatting

We use `clang-format` for formatting the source code. Before opening a pull request for review, navigate to `$DISCHARGE_HOME` and format the code using

```
find Source Physics Geometries Exec \( -name "*.H" -o -name "*.cpp" \) -exec clang-format -i {} +
```

8.4.6 Options files

Options files are named using the same convention as class files, e.g. `CD_MyClass.options`. It is the responsibility of `MyClass` to parse these variables correctly.

Everything in the options file should be lower-case, with the exception of the class name which should follow the class name syntax. If you need a separator for the variable, use an underscore `_`. For variables that should be grouped under a common block, use a dot `.` for grouping them. For a class `MyClass` and options file might look something like

```
MyClass.input_variable = 1.0
MyClass.bc.x.lo        = dirichlet 1.0
```

8.5 References

BIBLIOGRAPHY

[1] Marsha Berger and Isidore Rigoutsos. An Algorithm for Point Clustering and Grid Generation. *IEEE Transactions on Systems, Man and Cybernetics*, 1991. doi:10.1109/21.120081.

[2] A Bourdon, V P Pasko, N Y Liu, S Célestin, P Ségur, and E Marode. Efficient models for photoionization produced by non-thermal gas discharges in air based on radiative transfer and the Helmholtz equations. *Plasma Sources Science and Technology*, 16(3):656–678, aug 2007. URL: http://stacks.iop.org/0963-0252/16/i=3/a=026?key=crossref.26470bbe9c1f765777a162c80aa57bb7, doi:10.1088/0963-0252/16/3/026.

[3] O. Chanrion and T. Neubert. A PIC-MCC code for simulation of streamer propagation in air. *Journal of Computational Physics*, 2008. doi:10.1016/j.jcp.2008.04.016.

[4] P Colella, D T Graves, T J Ligocki, G Miller, D Modiano, P O Schwartz, B Van Straalen, J Pilliod, D Trebotich, M Barad, B Keen, A Nonaka, and C Shen. EBChombo software package for cartesian grid, embedded boundary applications. Technical Report, Lawrence Berkeley National Laboratory, 2004.

[5] Brian T.N. Gunney and Robert W. Anderson. Advances in patch-based adaptive mesh refinement scalability. *Journal of Parallel and Distributed Computing*, 2016. doi:10.1016/j.jpdc.2015.11.005.

[6] Edward W. Larsen, Guido Thömmes, Axel Klar, Seaid Mohammed, and Thomas Götz. Simplified PN Approximations to the Equations of Radiative Heat Transfer and Applications. *Journal of Computational Physics*, 183(2):652–675, dec 2002. URL: https://www.sciencedirect.com/science/article/pii/S0021999102972104, doi:10.1006/JCPH.2002.7210.

[7] V R Soloviev and V M Krivtsov. Surface barrier discharge modelling for aerodynamic applications. *Journal of Physics D: Applied Physics*, 42(12):125208, jun 2009. URL: http://stacks.iop.org/0022-3727/42/i=12/a=125208?key=crossref.6a72c0ae60b829dd552a56b7f9dfa90c, doi:10.1088/0022-3727/42/12/125208.

[8] David Trebotich and Daniel Graves. An adaptive finite volume method for the incompressible Navier–Stokes equations in complex geometries. *Commun. Appl. Math. Comput. Sci.*, 10(1):43–82, 2015. URL: https://doi.org/10.2140/camcos.2015.10.43, doi:10.2140/camcos.2015.10.43.

[9] E. H. Twizell, A. B. Gumel, and M. A. Arigu. Second-order,L 0-stable methods for the heat equation with time-dependent boundary conditions. *Advances in Computational Mathematics*, 6(1):333–352, 1996. URL: http://link.springer.com/10.1007/BF02127712, doi:10.1007/BF02127712.