chombo-discharge Documentation

Nov 13, 2025

CONTENTS

Important: This is a beta release. Development is still in progress, and various bugs may be present. Minor changes to the user interface can still occur.

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

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

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

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

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

CHAPTER

ONE

INTRODUCTION

1.1 Using this documentation

This documentation is the user documentation for `chombo-discharge`. It includes an explanation of the data structures, algorithms, and code design. It is built as a part of the continuous integration (CI) at GitHub. The HTML, PDF, and doxygen documentation obtained from GitHub is automatically kept up-to-date with the latest version of `chombo-discharge`.

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 tools, users should nonetheless be willing to invest sufficient time and energy to understand how the code operates. 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 OpenMP, MPI, or MPI+OpenMP.
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*). Particle and fluid kernels can thus be individually load balanced. Many abstractions are in place so that user can describe a new set of physics, or write entirely new solvers into `chombo-discharge` without affecting the embedded boundary formalism. Of course, `chombo-discharge` provides several physics modules for solving various types of problems.

1.2.3 Organization

The `chombo-discharge` source files are organized as follows:

Table 1.2.2: 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, including tests, convergence tests, and user examples.

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.
- VisIt (https://visit-dav.github.io/visit-website), which used for visualization and analysis.

1.3.2 Cloning chombo-discharge

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

- 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 recursively with submodules, both Chombo and nlohmann/json will be placed in the Submodules folder in $DISCHARGE_HOME.

Tip: 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}
```

Alternatively, if cloning using https:

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

git clone --recursive https://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.

1.3.3 Test build

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

1. Copy the GNU configuration file

   ```
   cp $DISCHARGE_HOME/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`.

1.3.4 Full configuration

`chombo-discharge` is compiled using GNU Make, following the `Chombo` configuration methods.

Important: 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, a full configuration consists of specifying

- Fortran and C++ compilers
- Specifying build configurations. E.g., serial or parallel builds, and compiler flags for optimized and debug builds.
- 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. `chombo-discharge` will run substantially slower with `DEBUG=TRUE`.
- `OPT = FALSE/TRUE/HIGH`. Setting `OPT=TRUE/HIGH` enables optimized builds.
- `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 or disables MPI.
- `MPICXX = <MPI compiler>` This sets the MPI compiler.
- `CXXSTD = 14` For specifying 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_MT=TRUE/FALSE` Configures `Chombo` with memory tracking functionality. Not supported with OpenMP, and enabling memory tracking together with OpenMP will trigger a preprocessor error.
- `USE_HDF5 = TRUE/FALSE` This enables or disables HDF5 output.
- `OPENMPCC = TRUE/FALSE` Turn on or off OpenMP threading.

MPI

To enable MPI, make sure that `MPI` is set to true and that the `MPICXX` compiler is set. For GNU installations, one will usually have `MPICXX = mpicxx` or `MPICXX = mpic++`, while for Intel builds one will usually have `MPICXX = mpiicpc`.

Note: The MPI layer distributes grid patches among processes, i.e. uses *domain decomposition*.

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).

Warning: `Chombo` only supports HDF5 APIs at version 1.10 and below. To use a newer version of HDF5 together with the 1.10 API, add `-DH5_USE_110_API` to the HDFINC flags.

OpenMP

To turn on OpenMP threading one can set the `OPENMPCC` to `TRUE`. When compiled with OpenMP all loops over grid patches uses threading in the form

```
#pragma omp parallel for schedule(runtime)
for (int mybox = 0; mybox < nbox; mybox++) {

}
```

Warning: Memory tracking is currently not supported together with threading. When compiling `chombo-discharge` make sure that memory tracking is turned off (see *Main settings*).

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)

Finally, note that the `cxxoptflags` and `foptflags` are enabled when using optimized builds. Corresponding flags exist for builds with `DEBUG=TRUE` in the form of `cxxdbgflags` and `foptdbgflags`.

Pre-defined configuration files

Some commonly used configuration files are found in `$DISCHARGE_HOME/Lib/Local`, and most of these are given as both serial and MPI versions, and with or without HDF5. The user needs to further configure the `Chombo` makefile to ensure that the `chombo-discharge` is properly configured for the system being compiled for. 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.
   - Serial and parallel versions of HDF5.

   The serial and parallel HDF5 are normally installed in different locations, 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).

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.

Configuration files for GitHub

`chombo-discharge` uses GitHub actions for continuous integration and testing. These tests run on Linux for a selection of GNU and Intel compilers. The configuration files are located in `$DISCHARGE_HOME/Lib/Local/GitHub`.

1.3.5 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 remove the `chombo-discharge` installation first by running

```
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
  ```

- Incomplete perl installations.

  `Chombo` may occasionally complain about incomplete perl modules. These error messages are unrelated to `Chombo` and `chombo-discharge`, but the user may need to install additional perl modules before compiling `chombo-discharge`.

1.4 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.

1.4.1 Running `chombo-discharge`

How one runs `chombo-discharge` depends on the type of parallelism. Below, we consider basic examples for serial and parallel execution.

Serial

If the application was compiled for serial execution one runs it with:

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

where <input_file> is your input file. Output from `chombo-discharge` will then be given directly to the terminal, or alternatively can be redirected to a file using `>& status.dat`, e.g.,

```
./<application_executable> <input_file> >& status.dat
```

Parallel with OpenMP

When running with OpenMP one must specify the number of threads, and possibly also pin threads to CPUs. `chombo-discharge` is compiled with run-time thread scheduling (which defaults to static scheduling). Prior to running with OpenMP, one should define the number of threads, thread palcement, and the scheduling. For example

```
export OMP_NUM_THREADS=8
export OMP_PLACES=cores
export OMP_PROC_BIND=true
export OMP_SCHEDULE="dynamic, 4"

./<application_executable> <input_file>
```

Parallel with MPI

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

```
mpirun -np 8 <application_executable> <input_file>
```

On clusters, this is a little bit different and usually requires passing the above command through a batch system. Normally, the MPI installation will map processes to cores. One can use `--report-bindings` to verify the mapping. E.g.,

```
mpirun --report-bindings -np 8 <application_executable> <input_file>
```

Parallel with MPI+OpenMP

When running with both MPI and OpenMP the user must

1. Bind each MPI rank to a specified resource (e.g., a node, socket, or list of CPUs).
2. Bind OpenMP threads to resources available to each MPI rank.

For example, the following may not work as expected (because threads may bind to the same CPUs):

```
export OMP_NUM_THREADS=4
mpiexec -n 2 ./<application_executable> <input_file>
```

Each MPI rank may spawn threads on the same physical cores. With OpenMPI one can map each rank to a specified number of CPUs, and bind threads to those CPUs. For example, on a local workstation one might do

```
export NRANKS=2
export OMP_NUM_THREADS=4
export OMP_PLACES=cores
export OMP_PROC_BIND=true

mpiexec --bind-to core --map-by slot:PE=$OMP_NUM_THREADS -n $NRANKS ./<application_executable> <input_file>
```

MPI bindings are here bound to cores and will spawn threads only on the cores they are associated with. It is often useful to verify this by reporting the bindings as follows:

```
mpiexec --report-bindings --bind-to core --map-by slot:PE=$OMP_NUM_THREADS -n $NRANKS ./<application_executable> <input_file>
```

Important: More sophisticated architectures (e.g., clusters with NUMA nodes) require careful specification of MPI and thread placement (e.g., binding of MPI ranks to sockets).

1.4.2 Simulation I/O

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.

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 simulation meta-information to separate files so that the simulation progress can be individually for each rank.

`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 Table 1.4.1.

Table 1.4.1: 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:

```
CdrCTU.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 `CdrCTU.plt_vars = phi`.

Warning: An empty entry like `CdrCTU.plt_vars =` will lead to run-time errors, so if you do not want a class to provide plot data you should put `CdrCTU.plt_vars = none`.

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). To see information regarding the latest time steps, simply print a few lines in these files, e.g.

```
tail -200 pout.0
```

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.

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.

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 any sense from a run-time perspective, 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.

1.4.3 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 Examples

In chombo-discharge, applications are set up so that they use the chombo-discharge source code and one chombo-discharge physics module. These are normally set up through Python interfaces accompanying each module. Several example applications are given in $DISCHARGE_HOME/Exec/Examples, which are organized by example type (e.g., plasma simulation, electrostatics, radiative transfer, etc). If chombo-discharge built successfully, it will usually be sufficient to compile the example by navigating to the folder containing the program file (program.cpp) and compiling it:

```
make -s -j4 program
```

To see how these programs are run, see *Controlling chombo-discharge*.

1.5.1 Positive streamer in air

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, use DIM=3 either on the command-line or in the configuration file. 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.

Tip: One can track the simulation progress through the pout.* files, see *Parallel processor verbosity*.

1.6 Code testing

To ensure the integrity of chombo-discharge, we include tests in

- $DISCHARGE_HOME/Exec/Tests for a functional test suite.
- $DISCHARGE_HOME/Exec/Convergence for a code verification tests.

1.6.1 Test suite

To make sure `chombo-discharge` compiles and runs as expected, we include a test suite that runs functional tests of many `chombo-discharge` components. The tests are defined in `$DISCHARGE_HOME/Exec/Tests`, and are organized by application type.

Running the test suite

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_exec -cores X
```

where `X` is the number of cores to use when compiling. By default, this will compile without MPI and HDF5 support.

Advanced configuration

The following options are available for running the various tests:

- `--compile` Compile all tests.
- `--clean` Do a clean recompilation.
- `--silent` Turn off terminal output.
- `--benchmark` Generate benchmark files.
- `--no_exec` Compile, but do not run the test.
- `--compare` Run, and compare with benchmark files.
- `-dim <number>` Run only the specified 2D or 3D tests.
- `-mpi <true/false>` Use MPI or not.
- `-hdf <true/false>` Use HDF5 or not.
- `-cores <number>` Run with specified number of cores.
- `-suites <string>` Run a specific application test suite.
- `-tests <string` Run a specific test.

For example, to compile with MPI and HDF5 enabled:

```
python3 tests.py --compile --clean --silent --no_exec -mpi=true -hdf=true -cores 8
```

If one only wants to compile a specified test suite:

```
python3 tests.py --compile --clean --silent --no_exec -mpi=true -hdf=true -cores 8 -suites Electrostatics
```

One can also restrict the tests to specific dimensionality, e.g.

```
python3 tests.py --compile --clean --silent --no_exec -mpi=true -hdf=true -cores 8 -suites Electrostatics -dim=2
```

Using benchmark files

The test suite can generate benchmark files which can later be compared against new test suite output files. This is often a good idea if one wants to ensure that modifications to the `chombo-discharge` source code does not unintentionally change the output of computer simulations. 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 (using h5diff) 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 --benchmark -mpi=true -hdf=true -cores 8
```

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 --compare -mpi=true -hdf=true -cores 8
```

When running the tests this way, the output files are bit-wise compared and a warning is issued if the files do not exactly match.

1.6.2 Automated testing

On GitHub, the test suite is integrated with GitHub actions and are automatically run when opening a pull request for review. In general, all tests must pass before a pull request can be merged. The test status can be observed either in the pull request, or at https://github.com/chombo-discharge/chombo-discharge/actions. The automated tests run `chombo-discharge` with `DEBUG=TRUE` and `OPT=FALSE` in order to catch assertion errors or other places where the code might break. They usually take around 1 hour to complete.

The automated tests will clone, build, and run the `chombo-discharge` test suite for various configurations:

- Parallel and serial.
- With or without HDF5.
- In 2D and 3D.

The tests are run with the following compiler suites:

- GNU.
- Intel oneAPI.

1.6.3 Convergence testing

To ensure that the various components in `chombo-discharge` converge at desired truncation order, many modules are equipped with their own convergence tests. These are located in `$DISCHARGE_HOME/Exec/Convergence`. The tests are too extensive to include in continuous integration, and they must be run locally like a regular `chombo-discharge` application. Our approach for convergence testing is found in *Verification and validation*.

1.6.4 Performance profiling

There are two ways to run performance profiling of chombo-discharge:

- A posteriori profiling using Chombo macros. Most routines in chombo-discharge use these macros and they will compute the wall clock time spent in each routine.

  To enable these timers, set CH_TIMER=1 in the shell where you run your application. E.g,

  ```
  export CH_TIMER=1
  ```

  The Chombo will not only compute the time spent in each function, but also figure out the parent-child relation between function, and present the output as a hierarchical structure. This is often useful when optimizing functions at the development stage.

  Warning: Chombo's timers are not meant to use with many time steps. For efficient use, it is best to use it for a single time step.

- In-place profiling using the chombo-discharge Timer class (see https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classTimer.html for the C++ API).

  Some classes in chombo-discharge use the Timer class, and can be used for run-time evaluations of function costs and load imbalance.

  Warning: The Timer class incurs large performance penalties at high concurrencies (1K CPU cores and above).

1.7 Acknowledgements

Substantial efforts have been made in writing chombo-discharge. Publications that arise from the use of chombo-discharge must acknowledge it's usage and cite the appropriate methodology papers. Currently, if you use chombo-discharge you should cite the following method paper:

```
@article{Marskar_chombo-discharge_2023,
   author = {Marskar, Robert},
   doi = {10.21105/joss.05335},
   journal = {Journal of Open Source Software},
   month = may,
   number = {85},
   pages = {5335},
   title = {{chombo-discharge: An AMR code for gas discharge simulations in complex geometries}},
   url = {https://joss.theoj.org/papers/10.21105/joss.05335},
   volume = {8},
   year = {2023}
}
```

CHAPTER

TWO

DESIGN

2.1 Overview

A 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 conform to. 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. 2.1.1 shows a basic design diagram 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 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. 2.1.1.

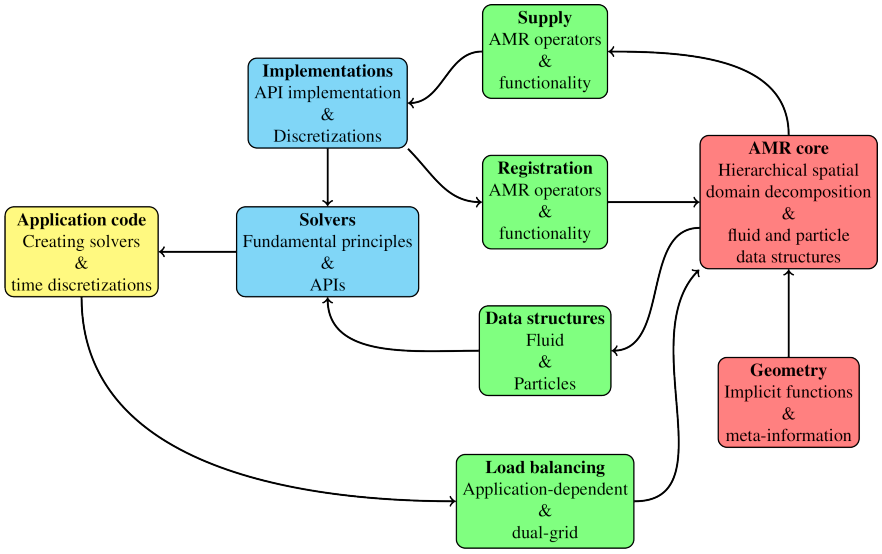

Fig. 2.1.1: Concept design sketch for `chombo-discharge`.

`chombo-discharge` also uses *loosely coupled* solvers as a foundation for the code design, where a *solver* indicates a piece of code for solving an equation. For example, solving the Laplace equation $\nabla^2\Phi = 0$ is encapsulated by one of

the chombo-discharge solvers. Some solvers in chombo-discharge have a null-implemented API, i.e., where we have enforced a strict separation of the solver interface and the solver implementation. This constraint exists because while new features may be added to a discretization, we do not want such changes to affect upstream application code. An example of this is the FieldSolver, which conceptualizes a numerical solver for solving for electrostatic field problems. The FieldSolver is an API with no fundamental discretization – it only contains high-level routines for understanding the type of solver being dealt with. Yet, it is the FieldSolver API which is used by most application codes (rather than the implementing subclass).

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*, which we also call *physics modules*. These modules encapsulate the time advancement of either individual or coupled solvers. One such module is the CdrPlasma module, which implements a conventional drift-diffusion model for streamer (and other types of) discharges. Solvers only interact with one another through these modules.

The top-level classes that represent the larger components in chombo-discharge are:

1. *Driver* for running simulations.
2. *AmrMesh* for encapsulating (almost) all AMR and EB functionality in a common core class.
3. *TimeStepper* for integrating the equations of motion.
4. *ComputationalGeometry* for representing computational geometries (such as electrodes and dielectrics).
5. *CellTagger* for flagging cells for refinement and coarsening.

2.2 Driver

The Driver class is the top-level class which runs chombo-discharge simulations. The primary purpose of the Driver class is to

1. Coordinate the simulation advance, i.e., run the relevant *TimeStepper* that is supplied. This can also involve restarting simulations from a user-provided time step.
2. Initiate and coordinate regrid operations.
3. Coordinate HDF5 I/O operations.

The constructor for this class is

```
Driver(const RefCountedPtr<ComputationalGeometry>& a_computationalGeometry,
       const RefCountedPtr<TimeStepper>&           a_timeStepper,
       const RefCountedPtr<AmrMesh>&               a_amrMesh,
       const RefCountedPtr<CellTagger>&            a_cellTagger = RefCountedPtr<CellTagger>(nullptr));
```

Here, the Driver class takes as input a user-provided geometry a_computationalGeometry, a user-provided physics module a_timestepper, and the AMR infrastructure a_amrMesh. The Driver class does not *require* an instance of *CellTagger*, which is the responsible for flagging cells for refinement. If users decide to omit a cell tagger as in the above constructor, regridding functionality is completely turned off and only the initially generated grids will be used throughout the entire simulation.

Tip: Here is the Driver C++ API.

Warning: Usage of injection of the AmrMesh object is likely to change in future versions of chombo-discharge.

2.2.1 Simulation setup

For setting up an running simulations, only a single routine is used:

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

This routine will set up and run a simulation.

New simulations

If a simulation starts from the first time step, the `Driver` class will perform the following steps in `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 and generate the initial AMR infrastructure. This initial grid is always generated by flagging cells for refinement along solid boundaries. If no such cells exist, a uniform grid on the coarsest domain is generated. 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.
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. After the regrid, the solvers are re-filled with initial data. I.e., initial data is always regenerated on the finest grid and is not interpolated from coarse grid.
5. Let *TimeStepper* perform a *post-initialization* routine. Whether or not this post-initialization does anything depends on the physics module begin used.

Restarting simulations

If a simulation uses the restart functionality (i.e., *does not start* from the first time step), `Driver` will execute the following steps in `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 `TimeStepper` to perform a *post-checkpoint* step. 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 obtained by taking the gradient.
4. Perform the number of initial regrids that the user asks for.

2.2.2 Simulation advancement

The algorithm for running a simulation is conceptually simple. `Driver` calls `TimeStepper::computeDt` for computing a reasonable time step for advancing the equations, and uses `Real TimeStepper::advance(Real dt)` to actually perform the advance. Note that these functions are abstract functions in *TimeStepper*, so the actual contents of these will differ between physics modules.

Regrids, plot files, and checkpoint files are written at certain step intervals, or specified time intervals. In brief, the algorithm looks like this:

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

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

      tryDt    = TimeStepper->computeDt()
      actualDt = TimeStepper->advance(tryDt)

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

      KeepRunningTheSimulation = true or false
   }
}
```

2.2.3 Regridding

Regrids are called by the `Driver` class and proceed as follows:

1. *CellTagger* generates tags for grid refinement and coarsening.
2. *TimeStepper* stores data that is required during regrids. This is necessary because we often need storage containers to store the solver states on both the old and the new grids. For mesh-based data, data is usually generated by interpolating data from the old grid to the new one.
3. *AmrMesh* generates the new grid boxes and associated EB information, and performs an initial load balancing.
4. *TimeStepper* checks if the application should re-balance the grids.
5. *AmrMesh* reinstantiates the EB and AMR grid infrastructure (e.g., interpolation operators).
6. *TimeStepper* performs a regrid operation of the physics module.
7. *TimeStepper* 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()

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

   if(loadBalance) {
     TimeStepper->loadBalance();
   }

   // AmrMesh finalizes the EBAMR grids
   AmrMesh->regridOperators()

   // Regrid timestepper
   TimeStepper->regrid()

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

2.2.4 Class options

Various class options are available for adjusting the behavior of the `Driver` class. Below, we include the current template options file for the `Driver` class.

Listing 2.2.1: Template input options for the `Driver` class. Runtime adjustable options are highlighted.

```
# ================================================================================================
# Driver class options
# ================================================================================================
Driver.verbosity                    = 2                 ## Driver verbosity.
Driver.geometry_generation          = chombo-discharge  ## Grid generation method, 'chombo-discharge' or 'chombo'
Driver.geometry_scan_level          = 0                 ## Geometry scan level for chombo-discharge geometry generator
Driver.ebis_memory_load_balance     = false             ## If using Chombo geo-gen, use memory as loads for EBIS generation
Driver.output_dt                    = -1.0              ## Output interval (values <= 0 enforces step-based output)
Driver.plot_interval                = 10                ## Plot interval
Driver.checkpoint_interval          = 100               ## Checkpoint interval
Driver.regrid_interval              = 10                ## Regrid interval
Driver.write_regrid_files           = false             ## Write or do not write plot files during regrids.
Driver.write_restart_files          = false             ## Write or do not write plot files during restart or not.
Driver.initial_regrids              = 0                 ## Number of initial regrids.
Driver.do_init_load_balance         = false             ## If true, load balance the first step in a fresh simulation.
Driver.start_time                   = 0                 ## Start time (fresh simulations only).
Driver.stop_time                    = 1.0               ## Stop time.
Driver.max_steps                    = 100               ## Maximum number of steps.
Driver.geometry_only                = false             ## Special option that ONLY plots the geometry.
Driver.write_memory                 = false             ## Write MPI memory report.
Driver.write_loads                  = false             ## Write (accumulated) computational loads.
Driver.output_directory             = ./                ## Output directory.
Driver.output_names                 = simulation        ## Simulation output names.
Driver.max_plot_depth               = -1                ## Restrict maximum plot depth (-1 => finest simulation level).
Driver.max_chk_depth                = -1                ## Restrict chechkpoint depth (-1 => finest simulation level).
Driver.num_plot_ghost               = 1                 ## Number of ghost cells to include in plots.
Driver.plt_vars                     = levelset          ## 'tags', 'mpi_rank', 'levelset', 'loads'.
Driver.restart                      = 0                 ## Restart step (less or equal to 0 implies fresh simulation).
Driver.allow_coarsening             = true              ## Allows removal of grid levels according to CellTagger.
Driver.grow_geo_tags                = 2                 ## How much to grow tags when using geometry-based refinement.
Driver.refine_angles                = 15.               ## Refine cells if angle between elements exceed this value.
Driver.refine_electrodes            = 0                 ## Refine electrode surfaces to specified level ( < 0 will refine␣
↪everything)
Driver.refine_dielectrics           = 0                 ## Refine dielectric surfaces to specified level ( < 0 will refine␣
↪everything)
```

We point out that the `output_directory` directory variable is *only* for the HDF5 plot files, whereas other files like `pout.*` are put in the directory in which the executable was ran.

From the user perspective, the most commonly adjusted parameters concern the I/O operations (plot and checkpoint intervals), and geometric refinement parameters.

2.3 ComputationalGeometry

`ComputationalGeometry` is the class that implements geometries in `chombo-discharge`. In principle, geometries consist of electrodes and dielectrics but there are many problems where the actual nature of the EB is irrelevant (such as fluid flow). For other problems, such as ones involving electric fields, the classification into electrodes and dielectric are obviously important.

Tip: Here is the ComputationalGeometry C++ API.

`ComputationalGeometry` is *not* an abstract class. The default implementation is an empty geometry, i.e., a geometry without any solid objects. Several pre-defined geometries are included in `$DISCHARGE_HOME/Geometries`.

Making a non-empty `ComputationalGeometry` class requires that you inherit from `ComputationalGeometry` and instantiate the class members specified in Listing 2.3.1.

Listing 2.3.1: List of all data member of the `ComputationalGeometry` base class. Highlighted members must be instantiated by the user in order to create a new geometry.

```
/*!
  @brief Background permittivity
*/
Real m_eps0;

/*!
  @brief True if we use the chombo-discharge geometry generation utility.
*/
bool m_useScanShop;

/*!
  @brief Grid level where we begin using ScanShop
*/
ProblemDomain m_scanDomain;

/*!
  @brief Maximum number of ghost cells that we will ever need
*/
int m_maxGhostEB;

/*!
  @brief List of dielectrics
*/
Vector<Dielectric> m_dielectrics;

/*!
  @brief List of electrodes
*/
Vector<Electrode> m_electrodes;

/*!
  @brief The gas-phase implicit function (i.e. outside electrodes and dielectrics).
*/
RefCountedPtr<BaseIF> m_implicitFunctionGas;

/*!
  @brief The solid-phase implicit function (i.e. the inside of the dielectrics).
*/
RefCountedPtr<BaseIF> m_implicitFunctionSolid;
```

Here, `m_eps0` is the *relative gas permittivity* (which is virtually always 1), `m_electrodes` are the electrodes for the geometry and `m_dielectrics` are the dielectrics for the geometry. These are described in detail below.

2.3.1 Implicit functions and compound geometries

`ComputationalGeometry` always uses implicit functions for describing the geometry, see *Geometry representation*. These functions are analytic functions that describe the inside and outside regions of a solid object, and usually appear in the form $f : \mathbb{R}^3 \rightarrow \mathbb{R}$. We point out that there *is* support in `chombo-discharge` for turning surface meshes into such functions, and arbitrary complex geometries can therefore be generated. In `chombo-discharge` we always use signed distance functions wherever we can.

When geometries are created, the `ComputationalGeometry` class will first create the (approximations to the) compound 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 both the electrodes and the dielectrics.

Because of this, there is a corresponding logic that underpins the partition into the three relevant domains. Let $A \cup B$ be the union of two objects A and B, and $A \cap B$ be the intersection of the same objects. Furthermore, let $f_1, f_2, \ldots$ denote the implicit functions for the electrodes. The electrode region is then given by

$$f = f_1 \cup f_2 \cup \ldots. \tag{2.3.1}$$

For the dielectric region, the canonical definition in `chombo-discharge` is that this region is composed of all regions

that are inside a dielectric but outside of a provided electrode (note that the user can override this behavior in his/her implementation of `ComputationalGeometry`). Specifically, if $g_1, g_2, \ldots$ are the dielectric implicit functions, the dielectric region is given by

$$g = (g_1 \cup g_2 \cup \ldots) \cap f^{\complement} \tag{2.3.2}$$

The region occupied by the gas is then given by $(f \cup g)^{\complement}$.

2.3.2 Electrode

The `Electrode` class is responsible for describing an electrode and also its boundary electrostatic boundary condition. Internally, this class is lightweight and consists only of a tuple that holds an implicit function and an associated boolean value that tells whether or not the level-set function is at a live voltage or not. The constructor for the electrode class is given in Listing 2.3.2.

Listing 2.3.2: Constructor for the the `Electrode` object.

```
Electrode(const RefCountedPtr<BaseIF>& a_baseIF, const bool a_live, const Real a_voltageFraction = 1.0);
```

In the code block above, `a_baseIF` argument is the implicit function for the electrode object, and the `a_live` argument is used by some solvers in order to determine if the electrode is at a live voltage or not. For example, if `a_live` is set to false, *FieldSolver* will fetch this value and determine that the electrode is at ground. Otherwise, if `a_live` is set to true then *FieldSolver* will determine that the electrode is at live voltage, and the `a_fraction` argument is an optional argument that allows the user to set the potential to a specified fraction of the live voltage. This is also used throughout other physics modules in `chombo-discharge`. The user can also set a specified fraction of the live voltage but setting the `a_voltageFraction` parameter to a relative fraction of the applied voltage.

Tip: Here is the Electrode C++ API

2.3.3 Dielectric

The `Dielectric` class describes a dielectric similar to how `Electrode` describes an electrode object. This class is lightweight and consists of a tuple that holds a level-set function and the relative associated permittivity (just like `Electrode` holds an implicit function and a relative voltage). The constructors for this class are

Listing 2.3.3: Constructors for the the `Dielectric` object.

```
/*!
  @brief Full constructor which uses constant permittivity.
  @param[in] a_baseIF       Implicit function
  @param[in] a_permittivity Constant permittivity
  @note Calls the define function for constant permittivity.
*/
Dielectric(const RefCountedPtr<BaseIF>& a_baseIF, const Real a_permittivity);

/*!
  @brief Full constructor which uses variable permittivity.
  @param[in] a_baseIF       Implicit function
  @param[in] a_permittivity Variable permittivity
  @note Calls the define function for variable permittivity.
*/
Dielectric(const RefCountedPtr<BaseIF>& a_baseIF, const std::function<Real(const RealVect a_pos)>& a_permittivity);
```

where the `a_baseIF` argument is the level-set function and the second argument sets a the permittivity. In the constructor, the relative permittivity can be set to a constant (first constructor) or to a spatially varying value (second constructor). Several solvers will then use the permittivities were specified by the user.

Tip: Here is the Dielectric C++ API

2.3.4 Retrieving parts

It is possible to retrieve the implicit functions for the electrodes and dielectrics through the following member functions:

Listing 2.3.4: Member functions for retrieving the defined electrodes and dielectrics.

```
/*!
  @brief Get dielectrics
  @return Dielectrics (m_dielectrics)
*/
const Vector<Dielectric>&
getDielectrics() const;

/*!
  @brief Get electrodes
  @return Electrodes (m_electrodes)
*/
const Vector<Electrode>&
getElectrodes() const;
```

Obtaining the implicit functions for each part can be useful when determining which object is closest to some physical location $\mathbf{x}$. This is frequently used when, e.g., colliding particles with the embedded boundaries and we want to determine which electrode or dielectric the particle collided with.

2.3.5 Retrieving compound implicit functions

When generating the geometry we compute the implicit functions for each *phase*, the gas-phase, the dielectric-phase, and the electrodes. We outlined this process in Eq. 2.3.1 and Eq. 2.3.2.

To retrive the implicit function corresponding to a particular phase, use

```
const RefCountedPtr<BaseIF>& getImplicitFunction(const phase::which_phase a_phase) const;
```

where `a_phase` will be `phase::gas` or `phase::solid`. This function will return the implicit function corresponding to all boundaries that contain the gas phase (`phase::gas`) or dielectric phase (`phase::solid`). There is no corresponding function for the interior of the electrodes as the solutions are uninteresting in these regions.

2.4 TimeStepper

`TimeStepper` is essentially the problem solving class `chombo-discharge`. The class is abstract and is subclassed by the programmer in order to solve new problems (or change time discretizations). Typically, `TimeStepper` owns all numerical solvers, has responsibility for setting up solvers, regridding its internal state, and advancing the equations of motion. Because `TimeStepper` and not *Driver* owns the solvers as class members, it will also partially coordinate I/O by providing data that *Driver* will add to HDF5 files.

Tip: Here is the TimeStepper C++ API

2.4.1 Basic functions

There are numerous functions that must be implemented and coordinated in order to provide a full-fledged `TimeStepper` for various problems. Below, we include the current header file for `TimeStepper`.

```
/* chombo-discharge
 * Copyright © 2021 SINTEF Energy Research.
 * Please refer to Copyright.txt and LICENSE in the chombo-discharge root directory.
 */

/*!
  @file   CD_TimeStepper.H
  @brief  Declaration of main (abstract) time stepper class.
  @author Robert Marskar
*/

#ifndef CD_TimeStepper_H
#define CD_TimeStepper_H

// Chombo includes
#include <CH_HDF5.H>

// Our includes
#include <CD_ComputationalGeometry.H>
#include <CD_MultiFluidIndexSpace.H>
#include <CD_AmrMesh.H>
#include <CD_NamespaceHeader.H>

/*!
  @brief Base class for advancing equations.
  @details This class is used by Driver for advancing sets of equations. In short, this class should implement a time-stepping␣
↪routine for a
  set of solvers. One should also implement routines for setting up the solvers, allocating necessary memory, regridding etc.
*/
class TimeStepper
{
public:
  /*!
    @brief Default constructor (does nothing)
  */
  TimeStepper();

  /*!
    @brief Default destructor (does nothing)
  */
  virtual ~TimeStepper();

  /*!
    @brief Set AmrMesh
    @param[in] a_amr AmrMesh
  */
  void
  setAmr(const RefCountedPtr<AmrMesh>& a_amr);

  /*!
    @brief Set the computational geometry
    @param[in] a_computationalGeometry The computational geometry.
  */
  void
  setComputationalGeometry(const RefCountedPtr<ComputationalGeometry>& a_computationalGeometry);

  /*!
    @brief Set up solvers
  */
  virtual void
  setupSolvers() = 0;

  /*!
    @brief Allocate data for the time stepper and solvers.
  */
  virtual void
  allocate() = 0;

  /*!
    @brief Fill solvers with initial data
  */
  virtual void
  initialData() = 0;
```

(continues on next page)

(continued from previous page)

```
  /*!
    @brief Post-initialize operations to be performed at end of setup stage.
  */
  virtual void
  postInitialize() = 0;

  /*!
    @brief Post-initialize operations to be performed after filling solvers with data read from checkpoint files.
  */
  virtual void
  postCheckpointSetup() = 0;

  /*!
    @brief Register realms to be used for the simulation.
  */
  virtual void
  registerRealms() = 0;

  /*!
    @brief Register operators to be used for the simulation
  */
  virtual void
  registerOperators() = 0;

  /*!
    @brief Parse runtime options
    @details Override this routine if your time stepper can use run-time configuration of the solvers that it advances. This␣
↪can e.g.
    be the CFL condition.
  */
  virtual void
  parseRuntimeOptions();

#ifdef CH_USE_HDF5
  /*!
    @brief Read header data from checkpoint file.
    @param[inout] a_header HDF5 header.
  */
  virtual void
  readCheckpointHeader(HDF5HeaderData& a_header);
#endif

#ifdef CH_USE_HDF5
  /*!
    @brief Write header data to checkpoint file.
    @param[inout] a_header HDF5 header.
  */
  virtual void
  writeCheckpointHeader(HDF5HeaderData& a_header) const;
#endif

#ifdef CH_USE_HDF5
  /*!
    @brief Write checkpoint data to file
    @param[inout] a_handle HDF5 file
    @param[in]    a_lvl    Grid level
    @details Implement this routine for checkpointing data for restarts. This will
    typically call the solvers' checkpointing routine, but more data can be added.
  */
  virtual void
  writeCheckpointData(HDF5Handle& a_handle, const int a_lvl) const = 0;
#endif

#ifdef CH_USE_HDF5
  /*!
    @brief Read checkpoint data from file
    @param[inout] a_handle HDF5 file
    @param[in]    a_lvl    Grid level
    @details Implement this routine for reading data for restarts. This will typically call the solvers' checkponiting routine.
  */
  virtual void
  readCheckpointData(HDF5Handle& a_handle, const int a_lvl) = 0;
#endif

  /*!
    @brief Get the number of plot variables for this time stepper.
    @details This is necessary because Driver, not TimeStepper, is responsible for allocating the necessary memory.
```

(continues on next page)

(continued from previous page)

```
  @return Returns number of plot variables that will be written during writePlotData
*/
virtual int
getNumberOfPlotVariables() const = 0;

/*!
  @brief Get plot variable names
*/
virtual Vector<std::string>
getPlotVariableNames() const = 0;

/*!
  @brief Write plot data to output holder.
  @param[inout] a_output      Output data holder.
  @param[inout] a_icomp       Starting component in a_output to begin at.
  @param[in]    a_outputRealm Realm where a_output belongs
  @param[in]    a_level       Grid level
*/
virtual void
writePlotData(LevelData<EBCellFAB>& a_output,
              int&                  a_icomp,
              const std::string     a_outputRealm,
              const int             a_level) const = 0;

/*!
  @brief An option for calling special functions prior to plotting data.
  Called by Driver in the IMMEDIATELY before writing the plot file.
*/
virtual void
prePlot();

/*!
  @brief An option for calling special functions prior to plotting data.
  Called by Driver in the IMMEDIATELY after writing the plot file.
*/
virtual void
postPlot();

/*!
  @brief Get computational loads to be checkpointed.
  @details This is used by Driver both for setting up load-balanced restarts AND for plotting the computational loads to a␣
↪file. This routine is
  disjoint from loadBalanceBoxes because this routine is not part of a regrid. This means that we are not operating with␣
↪temporarily load balanced
  grids where the but the final ones.
  @note The default implementation uses the box volume as a proxy for the load. You should overwrite this if you load␣
↪balance your application, and also
  make sure that the loads returned from this routine are consistent with what you put in loadBalanceBoxes.

  Also note that the return vector has the same ordering as the DisjointBoxLayout's boxes on the input grid level. See the␣
↪implementation for further details.
  @param[in] a_realm Realm
  @param[in] a_level Grid level
  @return Returns computational loads for each box on grid level a_level.

*/
virtual Vector<long int>
getCheckpointLoads(const std::string a_realm, const int a_level) const;

/*!
  @brief Compute a time step to be used by Driver.
*/
virtual Real
computeDt() = 0;

/*!
  @brief Advancement method. The implementation of this method should advance all equations of motion
  @param[in] a_dt Time step to be used for advancement
  @return    Returns the time step that was used.
  @note The return value does not need to equal a_dt. Adaptive time stepping methods will generally return != a_dt.
*/
virtual Real
advance(const Real a_dt) = 0;

/*!
  @brief Synchronzie solver times and time steps
  @param[in] a_step Time step
  @param[in] a_time Time (in seconds)
```

(continues on next page)

(continued from previous page)

```
    @param[in] a_dt   Time step that was used.
  */
  virtual void
  synchronizeSolverTimes(const int a_step, const Real a_time, const Real a_dt) = 0;

  /*!
    @brief Print a step report.
    @details This is called by Driver after time step. The routine can be used to display use information about the simulation␣
↪progress.
  */
  virtual void
  printStepReport() = 0;

  /*!
    @brief Perform pre-regrid operations.
    @details This should include all copying all data which should be interpolated to the new grids. It can also include␣
↪deallocating memory in case the regrid
    operation takes a lot of memory.
    @param[in] a_lmin           The coarsest level that changes
    @param[in] a_oldFinestLevel The finest level before the regrid.
  */
  virtual void
  preRegrid(const int a_lmin, const int a_oldFinestLevel) = 0;

  /*!
    @brief Time stepper regrid method.
    @param[in] a_lmin           The coarsest level that changed.
    @param[in] a_oldFinestLevel The finest level before the regrid.
    @param[in] a_newFinestLevel The finest level after the regrid.
  */
  virtual void
  regrid(const int a_lmin, const int a_oldFinestLevel, const int a_newFinestLevel) = 0;

  /*!
    @brief Perform post-regrid operations.
    @details This includes all operations to be done AFTER interpolating data to new grids.
  */
  virtual void
  postRegrid() = 0;

  /*!
    @brief Fuction which can have Driver do regrids at arbitrary points in the simulation.
    @details This is called by Driver at every time step and if it returns true, Driver will regrid, regardless of whether or␣
↪not it is
    on the regular "regrid intervals".
  */
  virtual bool
  needToRegrid();

  /*!
    @brief Load balancing query for a specified realm. If this returns true for a_realm, load balancing routines will be␣
↪called during regrids.
    @param[in] a_realm Realm name
  */
  virtual bool
  loadBalanceThisRealm(const std::string a_realm) const;

  /*!
    @brief Load balance grid boxes for a specified realm.
    @param[out] a_procs       MPI ranks owning the various grid boxes.
    @param[out] a_boxes       Grid boxes on every level (obtain them with a_grids[lvl].boxArray())
    @param[in]  a_realm       Realm identifier
    @param[in]  a_grids       Original grids
    @param[in]  a_lmin        Coarsest grid level that changed
    @param[in]  a_finestLevel New finest grid level
    @details This is only called by Driver if TimeStepper::loadBalanceThisRealm(a_realm) returned true. The default␣
↪implementation
    uses volume-based loads for the grid patches. If the user wants to load balance boxes on a realm, this routine must be␣
↪overwritten and
    he should compute loads for the various patches in a_grids and call LoadBalancing::makeBalance on each level. It is up to␣
↪the user/programmer
    to decide if load balancing should be done independently on each level, or if loads per MPI rank are accumulated across␣
↪levels.
  */
  virtual void
  loadBalanceBoxes(Vector<Vector<int>>&            a_procs,
                   Vector<Vector<Box>>&            a_boxes,
                   const std::string               a_realm,
```

(continues on next page)

(continued from previous page)

```
                  const Vector<DisjointBoxLayout>& a_grids,
                  const int                        a_lmin,
                  const int                        a_finestLevel);

protected:
  /*!
    @brief Class verbosity
  */
  int m_verbosity;

  /*!
    @brief Time step
  */
  int m_timeStep;

  /*!
    @brief TIme
  */
  Real m_time;

  /*!
    @brief Previous time step size
  */
  Real m_dt;

  /*!
    @brief AmrMesh.
  */
  RefCountedPtr<AmrMesh> m_amr;

  /*!
    @brief Computational geometry.
  */
  RefCountedPtr<ComputationalGeometry> m_computationalGeometry;
};

#include <CD_NamespaceFooter.H>

#endif
```

2.4.2 Setup routines

Here, we consider the various setup routines in `TimeStepper`. The routines are used by *Driver* in the simulation setup step, both for fresh simulation setups as well as restarts.

registerRealms

`chombo-discharge` permits things to happen on different sets of grids where the the grids themselves cover the same physical region, but where MPI ownership of grids might change between grid sets (see *Realm* for details). To register a *Realm*, users will have `TimeStepper` register realms in the `registerRealms()` routine, as follows:

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

The above code will ensure that `chombo-discharge` generates three *Realm*, which can be individually load balanced. 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.

setupSolvers

`setupSolvers` is used for instantiating the solvers. This routine is called *prior* to creating grids, so it is not possible to allocate mesh data for the solvers inside this routine. The rationale for this design is that *AmrMesh* must know relatively early which part of the AMR infrastructure that will be instantiated, so the solvers are created before allocating the grids. It is still quite possible to parse lots of data into the solvers, e.g., setting input variables.

To provide an example, the code snippet below shows the implementation of this routine for the `TimeStepper` implementation for the *Advection-diffusion model*:

Listing 2.4.1: Implementation of the `setupSolvers` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::setupSolvers()
{
  CH_TIME("AdvectionDiffusionStepper::setupSolvers");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::setupSolvers" << endl;
  }

  CH_assert(!m_solver.isNull());

  // Instantiate the species.
  m_species = RefCountedPtr<AdvectionDiffusionSpecies>(
    new AdvectionDiffusionSpecies(m_initialData, m_mobile, m_diffusive));

  // Prep the solver.
  m_solver->setVerbosity(m_verbosity);
  m_solver->setSpecies(m_species);
  m_solver->parseOptions();
  m_solver->setPhase(m_phase);
  m_solver->setAmr(m_amr);
  m_solver->setComputationalGeometry(m_computationalGeometry);
  m_solver->setRealm(m_realm);

  if (!m_solver->isMobile() && !m_solver->isDiffusive()) {
    MayDay::Error("AdvectionDiffusionStepper::setupSolvers - can't turn off both advection AND diffusion");
  }
}
```

registerOperators

Internally, an instantiation of *Realm* will provide access to the grids and cut-cell information on that *Realm*, as well as any operators that the user has seen fit to *register*. Various operators are available for, e.g., computing gradients, conservative coarsening, ghost cell interpolation, filling a patch with interpolation data, redistribution, particle-mesh operations, 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 automatically instantiate those operators during initialization or regrid operations. 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()`, but this is usually done by letting the solvers register what they need. Failure to do so will issue a run-time error. Solvers will typically allocate a subset of these operators, but for multiphysics code that use both fluid and particles, most of these will be in use. An example of this is given in Listing 2.4.2, which shows how this routine is implemented for the *Advection-diffusion model*:

Listing 2.4.2: Implementation of the `registerOperators` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::registerOperators()
{
  CH_TIME("AdvectionDiffusionStepper::registerOperators");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::registerOperators" << endl;
  }

  // Let the solver do this -- it knows what it needs.
  m_solver->registerOperators();
}
```

allocate

`allocate` is used for allocating particle and mesh data that is required during simulations. This step is done *after* the grids have been initialized by *AmrMesh and* during regrids. Again using *Advection-diffusion model* as an example,

Listing 2.4.3: Implementation of the `allocate` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::allocate()
{
  CH_TIME("AdvectionDiffusionStepper::allocate");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::allocate" << endl;
  }

  m_solver->allocate();
}
```

In the above snippet, `TimeStepper` only calls the solver allocation function (solvers generally know how to allocate their own internal data). For more complex problems this routine will probably allocate additonal data that only lives within `TimeStepper` (and not the solvers).

initialData

`initialData` is called by *Driver* setup routines after the `allocate` step, and has responsibility of setting up the problem with initial data. This can occasionally be simple, or for coupled problems it might be highly complex. For discharge problems, this can involve filling the solvers with initial densities, and solving the Poisson equation for obtaining the electric field. A simpler example is again given by *Advection-diffusion model*:

Listing 2.4.4: Implementation of the `initialData` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::initialData()
{
  CH_TIME("AdvectionDiffusionStepper::initialData");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::initialData" << endl;
  }

  // Fill the solver with initial data from the species.
  m_solver->initialData();

  // Set velocity, diffusion coefficient, and boundary conditions.
  m_solver->setSource(0.0);
  m_solver->setEbFlux(0.0);
  if (m_solver->isDiffusive()) {
    m_solver->setDiffusionCoefficient(m_diffCo);
```

(continues on next page)

(continued from previous page)

```
  }
  if (m_solver->isMobile()) {
    m_solver->setVelocity(m_velocity);
  }

  // Set flux functions
  auto fluxFunc = [](const RealVect a_pos, const Real a_time) {
    return 0.0;
  };

  //  m_solver->setDomainFlux(fluxFunc);
}
```

The above code defers the initialization of the density in the advection-diffusion-reaction solver to the actual solver implementation. Here, one could equally well have fetched the density directly from the solver and set it to something else by simply iterating through the grid cells (or setting it through other means). In addition, source terms are set to zero, as are boundary fluxes.

postInitialize

`postInitialize` is a special routine that is called *after* *Driver* has filled the solvers with initial data *and* *Driver* is done with all the initial regrids. While most data initialization steps can, however, be done in `initialData`, the function is put there as an open door to the programmer for performing certain post-initialization functions that do not not need be performed in `initialData` (which is called once per initial regrid). For example, the *Discharge inception model* uses this function to compute several relevant quantities after the electric has been obtained in `initialData`.

postCheckpointSetup

During simulation restarts, *Driver* will open an HDF5 file and have `TimeStepper` fill solvers and its own internal data with data from that file. `postCheckpointSetup` is a routine which is called immediately after the solvers have performed this step. Several gas discharge models use this function to compute the electric field from the potential that was saved in the HDF5 file.

2.4.3 I/O routines

`TimeStepper` contains I/O routines primarily serves two purposes:

1. To provide data for HDF5 plot files, used for post-processing analysis.
2. To read and write data from HDF5 checkpoint files, which are used to restart simulations from a specified time step.

In general, plot and checkpoint data do not contain the same data, and `TimeStepper` therefore requires that plot and checkpoint files are filled separately. We discuss these below:

getNumberOfPlotVariables

`getNumberOfPlotVariables` must return the number of components that will be plotted by `TimeStepper`. The reason why this routine exists is the *Driver* will pre-allocate the necessary memory on each AMR level, and `TimeStepper` will then copy solver data into this data holder. Specifically, if `TimeStepper` will plot a single scalar, it must return a value of one. If it plots a single vector, it must return a value of `SpaceDim`. Below we include the implementation of this routine for the *Advection-diffusion model*:

Listing 2.4.5: Implementation of the `getNumberOfPlotVariables` routine for a simple advection-diffusion problem.

```
int
AdvectionDiffusionStepper::getNumberOfPlotVariables() const
{
  CH_TIME("AdvectionDiffusionStepper::getNumberOfPlotVariables");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::getNumberOfPlotVariables" << endl;
  }

  // Not plotting anything of our own, so return whatever the solver wants to plot.
  return m_solver->getNumberOfPlotVariables();
}
```

getPlotVariableNames

`getPlotVariableNames` provides a list of plot variable names for the HDF5 file. This list must have the same length as the returned value of `getNumberOfPlotVariables`. Below we include the implementation of this routine for the *Advection-diffusion model*:

Listing 2.4.6: Implementation of the `getPlotVariableNames` routine for a simple advection-diffusion problem.

```
Vector<std::string>
AdvectionDiffusionStepper::getPlotVariableNames() const
{
  CH_TIME("AdvectionDiffusionStepper::getPlotVariableNames");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::getPlotVariableNames" << endl;
  }

  return m_solver->getPlotVariableNames();
}
```

Tip: When writing some vector data F to the HDF5-file, one can write the variables as `x-F`, `y-F`, and `z-F`, and VisIt visualization will automatically recognize F as a vector field.

writePlotData

`writePlotData` will write the plot data to the provided data holder. The function signature is

```
/*!
  @brief Write plot data to output holder.
  @param[inout] a_output      Output data holder.
  @param[inout] a_icomp       Starting component in a_output to begin at.
  @param[in]    a_outputRealm Realm where a_output belongs
  @param[in]    a_level       Grid level
*/
virtual void
writePlotData(LevelData<EBCellFAB>& a_output,
              int&                  a_icomp,
              const std::string     a_outputRealm,
```

In this function, `a_output` is pre-allocated block of memory that `TimeStepper` will write its components to (beginning at `a_icomp`). Note that if `TimeStepper` writes N components, the implementation must increment `a_icomp` by N. Usually, solvers will have their own `writePlotData` routines which lets `TimeStepper` simply call the solver functions. An example is given below for the *Advection-diffusion model*:

Listing 2.4.7: Implementation of the `writePlotData` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::writePlotData(LevelData<EBCellFAB>& a_output,
                                         int&                  a_icomp,
                                         const std::string     a_outputRealm,
                                         const int             a_level) const
{
  CH_TIME("AdvectionDiffusionStepper::writePlotData");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::writePlotData" << endl;
  }

  CH_assert(a_level >= 0);
  CH_assert(a_level <= m_amr->getFinestLevel());

  m_solver->writePlotData(a_output, a_icomp, a_outputRealm, a_level);
}
```

writeCheckpointData

`writeCheckpointData` must write necessary data for checkpointing the simulation state. This data is used when restarting simulations from a checkpoint file. Note that checkpoint data is written on a level-by-level basis. The function signature is

```
/*!
  @brief Write checkpoint data to file
  @param[inout] a_handle HDF5 file
  @param[in]    a_lvl    Grid level
  @details Implement this routine for checkpointing data for restarts. This will
  typically call the solvers' checkpointing routine, but more data can be added.
*/
virtual void
writeCheckpointData(HDF5Handle& a_handle, const int a_lvl) const = 0;
```

Usually, the solvers know themselves what data to put in the checkpoint files and these routines are then pretty simple. Below, we again include an example for the *Advection-diffusion model*:

Listing 2.4.8: Implementation of the `writeCheckpointData` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::writeCheckpointData(HDF5Handle& a_handle, const int a_lvl) const
{
  CH_TIME("AdvectionDiffusionStepper::writeCheckpointData");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::writeCheckpointData" << endl;
  }

  m_solver->writeCheckpointLevel(a_handle, a_lvl);
}
```

When implementing the function, it is important to add any data that is required when restarting the simulation. It is also beneficial to *not* add data that is not required since this will just lead to larger files. An example of this is the *FieldSolver* class, which only checkpoints the potential and not the electric field, as the latter is simply obtained by taking the gradient.

Tip: Computational particles can also be added to the checkpoint files.

readCheckpointData

`readCheckpointData` is the function that will read data from an HDF5 checkpoint file and populate `TimeStepper` with this data. The data is read on a level-by-level basis, with a function signature

```
/*!
  @brief Read checkpoint data from file
  @param[inout] a_handle HDF5 file
  @param[in]    a_lvl    Grid level
  @details Implement this routine for reading data for restarts. This will typically call the solvers' checkponiting routine.
*/
virtual void
readCheckpointData(HDF5Handle& a_handle, const int a_lvl) = 0;
```

Solvers will normally already know what data to read into their data members. E.g., the example for the *Advection-diffusion model* is

Listing 2.4.9: Implementation of the `readCheckpointData` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::readCheckpointData(HDF5Handle& a_handle, const int a_lvl)
{
  CH_TIME("AdvectionDiffusionStepper::readCheckpointData");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::readCheckpointData" << endl;
  }

  m_solver->readCheckpointLevel(a_handle, a_lvl);
}
```

2.4.4 Advance routines

computeDt

`computeDt` is a routine that will compute a trial time step when calling the `advance` method. We have chosen to call this a trial time step because

1. *Driver* might choose to use a smaller time step in order to write plot files at specific times.
2. When calling the actual advance method (see below), it is possible to return a different time step than the one computed through `computeDt`.

The calculation of a time step can be quite involved, depending on the application being imlemented. Moreover, many `TimeStepper` implementations will provide hooks for swapping algorithms, and in this case the time step might be limited differently. For the *Advection-diffusion model* the implementation is as follows:

Listing 2.4.10: Implementation of the `computeDt` routine for a simple advection-diffusion problem.

```
Real
AdvectionDiffusionStepper::computeDt()
{
  CH_TIME("AdvectionDiffusionStepper::computeDt");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::computeDt" << endl;
  }

  // TLDR: If we run explicit advection but implicit diffusion then we are only limited by the advective CFL. Otherwise,
  //       if diffusion is also explicit we need the advection-diffusion limited time step.

  // A weird thing, but sometimes we want to be able to force the CFL so that
  // we override run-time configurations of the CFL number. This code does that.
  Real cfl = 0.0;
  if (m_forceCFL > 0.0) {
```

(continues on next page)

(continued from previous page)

```
    cfl = m_forceCFL;
  }
  else {
    cfl = m_cfl;
  }

  Real dt = std::numeric_limits<Real>::max();

  switch (m_integrator) {
  case Integrator::Heun: {
    dt = cfl * m_solver->computeAdvectionDiffusionDt();

    break;
  }
  case Integrator::IMEX: {
    dt = cfl * m_solver->computeAdvectionDt();

    break;
  }
  default: {
    MayDay::Error("AdvectionDiffusionStepper::computeDt - logic bust");

    break;
  }
  }

  dt = std::max(dt, m_minDt);
  dt = std::min(dt, m_maxDt);

  return dt;
}
```

In the code above, `TimeStepper` supports both fully explicit advection-diffusion advances as well as split-step advances with implicit diffusion. Depending on how the user chooses to run the code, the time step is therefore computed differently. At the bottom of the above code, hard limits on the time step are also enforced.

advance

The `advance` method has responsibility for advancing physics module one time step, and is called by `Driver`. The function signature is

```
/*!
  @brief Advancement method. The implementation of this method should advance all equations of motion
  @param[in] a_dt Time step to be used for advancement
  @return    Returns the time step that was used.
  @note The return value does not need to equal a_dt. Adaptive time stepping methods will generally return != a_dt.
*/
virtual Real
advance(const Real a_dt) = 0;
```

As mentioned in the documentation for this method, the function takes a trial time step `a_dt` which is the physical time step. This time step is the one computed by `computeDt`. It is, however, quite possible to advance the equations of motion over a time that does not equal `a_dt`, which is generally the case for adaptive time stepping methods.

The implementation of the `advance` method is usually the most time-consuming part of implementation a new `TimeStepper`, and the implementation of this routine can become substantially complicated. For simpler problems this routine is relatively straightforward to implement, however, also in a way that involves AMR and cut-cells.

Listing 2.4.11: Implementation of the `advance` routine for a simple advection-diffusion problem.

```
Real
AdvectionDiffusionStepper::advance(const Real a_dt)
{
  CH_TIME("AdvectionDiffusionStepper::advance");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::advance" << endl;
```

(continues on next page)

(continued from previous page)

```
  }

  // State to be advanced.
  EBAMRCellData& state = m_solver->getPhi();

  switch (m_integrator) {
  case Integrator::Heun: {
    const bool conservativeOnly = false;
    const bool addEbFlux        = true;
    const bool addDomainFlux    = true;

    // Transient storage
    EBAMRCellData yp;
    EBAMRCellData k1;
    EBAMRCellData k2;

    m_amr->allocate(yp, m_realm, m_phase, 1);
    m_amr->allocate(k1, m_realm, m_phase, 1);
    m_amr->allocate(k2, m_realm, m_phase, 1);

    // Compute k1 coefficient
    m_solver->computeDivJ(k1, state, 0.0, conservativeOnly, addEbFlux, addDomainFlux);
    DataOps::copy(yp, state);
    DataOps::incr(yp, k1, -a_dt);

    // Compute k2 coefficient and final state
    m_solver->computeDivJ(k2, yp, 0.0, conservativeOnly, addEbFlux, addDomainFlux);
    DataOps::incr(state, k1, -0.5 * a_dt);
    DataOps::incr(state, k2, -0.5 * a_dt);

    break;
  }

  m_amr->conservativeAverage(state, m_realm, m_phase);
  m_amr->interpGhost(state, m_realm, m_phase);

  return a_dt;
}
```

In the above code we have included the part of the `advance` routine that executes Heun's method for advancing the scalar. This code also includes allocation of temporaries for computing the coefficients, storage for the intermediate state, and enforcement of boundary conditions, all of which include AMR. At the way out of the routine the solution is coarsened and the ghost cells are updated, and the trial time step (`a_dt`) is returned.

synchronizeSolverTimes

`synchronizeSolverTimes` is called after the `advance` method and is used to update the simulation time for all solvers. Again, this routine exists because there is often a physical time to be tracked by the solvers (e.g., enforcement of time-dependent voltage applications). This routine simply ensures that `TimeStepper` and all solvers that `TimeStepper` owns see the same physical time, number of steps, and time step sizes. The implementation for *Advection-diffusion model* is

Listing 2.4.12: Implementation of the `synchronizeSolverTimes` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::synchronizeSolverTimes(const int a_step, const Real a_time, const Real a_dt)
{
  CH_TIME("AdvectionDiffusionStepper::synchronizeSolverTimes");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::synchronizeSolverTimes" << endl;
  }

  m_timeStep = a_step;
  m_time     = a_time;
  m_dt       = a_dt;

  m_solver->setTime(a_step, a_time, a_dt);
}
```

printStepReport

`printStepReport` is called after the `advance` method, and provides extra information printed to the `pout.*` files (see *Controlling chombo-discharge*). This function is called by *Driver* after performing a time step, and can be used to print extra information not covered by *Driver*, such as how the time step was limited, or other information that is useful for monitoring the behavior of `TimeStepper`. For example, the current gas discharge models in `chombo-discharge` print the maximum electric field and density at each time step. Note that `printStepReport` has (or should have!) no side-effects that affect the simulation state.

2.4.5 Regrid routines

The regrid routines in `TimeStepper` must, in combination, be able to transfer the simulation between old and new grids. For an explanation to how regridding occurs in `chombo-discharge`, see *Regridding*. In particular, when regrids occur the old grids are eventually destroyed so it is necessary to cache the old-grid simulation states so that we have something to interpolate from whan transfer the state to the new grids.

preRegrid

`preRegrid` should any necessary pre-regrid operations that are necessary in order to call the `regrid`. This will virtually always include caching the old-grid simulation state, both for the solvers and also for internal data in `TimeStepper` Solvers usually know how to do this, and in some cases this function can be deceptively simple, as illustrated by the implementation of this function in *Advection-diffusion model*:

Listing 2.4.13: Implementation of the `preRegrid` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::preRegrid(const int a_lbase, const int a_oldFinestLevel)
{
  CH_TIME("AdvectionDiffusionStepper::preRegrid");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::preRegrid" << endl;
  }

  m_solver->preRegrid(a_lbase, a_oldFinestLevel);
}
```

Other implementations of `TimeStepper` may have substantially more complicated `preRegrid` routines.

regrid

`regrid` is the function that performs an actual regrid operation. At the time when `regrid` is called, the old grids are already destroyed and are only available through the cached data. Solvers are usually implemented with their own regrid routine, and if the only things that need to be regridded are the solvers, the implementation of this routine can be comparatively simple, as illustrated below for the *Advection-diffusion model*:

Listing 2.4.14: Implementation of the `regrid` routine for a simple advection-diffusion problem.

```
void
AdvectionDiffusionStepper::regrid(const int a_lmin, const int a_oldFinestLevel, const int a_newFinestLevel)
{
  CH_TIME("AdvectionDiffusionStepper::regrid");
  if (m_verbosity > 5) {
    pout() << "AdvectionDiffusionStepper::regrid" << endl;
  }

  // Regrid CDR solver and set up the flow fields
```

(continues on next page)

(continued from previous page)

```
  m_solver->regrid(a_lmin, a_oldFinestLevel, a_newFinestLevel);
  m_solver->setSource(0.0);
  m_solver->setEbFlux(0.0);

  if (m_solver->isDiffusive()) {
    m_solver->setDiffusionCoefficient(m_diffCo);
  }
  if (m_solver->isMobile()) {
    m_solver->setVelocity(m_velocity);
  }
}
```

For other `TimeStepper` implementations this routine can become much more complex. The *Îto-KMC plasma model*, for example, will regrid not only solver data but also internal mesh and particle data, recompute conductivities, deposit particles, handle superparticles, and prepare the simulation state for the next time step.

postRegrid

The `postRegrid` is called after `regrid` has completed and can be used to perform any post-regrid specific procedures. This function is not a pure function, and an implementation of this function is therefore not a requirement.

2.4.6 Load balancing routines

The default load-balancing method in `chombo-discharge` is to distribute grid patches equally among ranks, respecting a space-filling Morton curve on each grid level. However, during the regrid step, *Driver* will check if meshes should be load balanced using different heuristics. This load balancing can be done separately for each *Realm*, and in this case the MPI ranks will have different patch ownership in different grid sets.

If a realm should be load balanced with a different method than the default load balancing scheme, then `TimeStepper` can take a `DisjointBoxLayout` which originally load balanced using the patch volume, and regenerate the patch-to-rank ownership for the grids. This functionality is implemented through two routines:

1. `loadBalanceThisRealm` which checks if a specific *Realm* should be load balanced.
2. `loadBalanceBoxes` which load balances the boxes on the specified *Realm*.

Note that these functions are not pure functions, and it is perfect fine to use their default implementation, in which case each MPI rank gets approximately the same number of grid patches.

loadBalanceThisRealm

The function signature for this function is

```
/*!
  @brief Load balancing query for a specified realm. If this returns true for a_realm, load balancing routines will be called␣
↪during regrids.
  @param[in] a_realm Realm name
*/
virtual bool
loadBalanceThisRealm(const std::string a_realm) const;
```

This function must return true if the input *Realm* (`a_realm`) should be load balanced.

loadBalanceBoxes

If `loadBalanceThisRealm` returns true, the following function is responsible for actually regenerating the grids:

```
/*!
  @brief Load balance grid boxes for a specified realm.
  @param[out] a_procs       MPI ranks owning the various grid boxes.
  @param[out] a_boxes       Grid boxes on every level (obtain them with a_grids[lvl].boxArray())
  @param[in]  a_realm       Realm identifier
  @param[in]  a_grids       Original grids
  @param[in]  a_lmin        Coarsest grid level that changed
  @param[in]  a_finestLevel New finest grid level
  @details This is only called by Driver if TimeStepper::loadBalanceThisRealm(a_realm) returned true. The default␣
↪implementation
  uses volume-based loads for the grid patches. If the user wants to load balance boxes on a realm, this routine must be␣
↪overwritten and
  he should compute loads for the various patches in a_grids and call LoadBalancing::makeBalance on each level. It is up to␣
↪the user/programmer
  to decide if load balancing should be done independently on each level, or if loads per MPI rank are accumulated across␣
↪levels.
*/
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);
```

This is called if `loadBalanceThisRealm` evaluates to true, and in this case the `TimeStepper` should compute a new set of rank ownership for the input grid boxes. Observe that `loadBalanceBoxes` occurs for the entire AMR hierarchy, where the outer vector of `a_procs` and `a_boxes` is the grid level, and the inner vectors describe the ownership of each box. The default implementation of this function ensures that when we load balance a level, we account for the accumulated load on coarser levels (see *Default implementation of loadBalanceBoxes.*).

Listing 2.4.15: Default implementation of `loadBalanceBoxes`.

```
/*!
  @brief Load balancing query for a specified realm. If this returns true for a_realm, load balancing routines will be called␣
↪during regrids.
  @param[in] a_realm Realm name
*/
virtual bool
loadBalanceThisRealm(const std::string a_realm) const;
```

In the above, we use the `Loads` class to hold the computational load for each rank, and on each level we compute the load for each patch to be equal to the number of grid cells in the patch. This is later load balanced in `LoadBalancing::makeBalance`, which is a routine that ensures that when we assign boxes on some grid level l, we account for loads already assigned on coarser grid levels.

2.5 AmrMesh

`AmrMesh` handles virtually the entire AMR and cut-cell infrastructure in `chombo-discharge`, and has the following responsibility:

1. Generate grids form a set of cell tags (potentially more than one set of grids).
2. Load balance the grids if necessary.
3. Instantiate the cut-cell information on grids.
4. Provide a mask which tells the user if a cell is covered by a finer grid.
5. Generate operators that are required for handling AMR data, e.g.,
    - Ghost cell interpolators.

- Coarsening operators.
- Stencils for interpolation and extrapolation near the embedded boundaries.
- Fine-to-coarse grid interpolators.
- Infrastructure for particle-mesh operators.
- ... and many others.

In addition to these, `AmrMesh` contains function for actually allocating data with user-defined centering (e.g., cell, face, EB) on a specified *Realm*. It also has responsibility for allocating particle containers.

One thing that `AmrMesh` is *not*, is a numerical discretization holder for PDEs, which is a responsibility that is generally deferred to solvers. In some cases `AmrMesh` certainly holds operators that have an underlying discretization, e.g., gradient operators which have a specific discretization based on least squares reconstruction around the embedded boundaries. But in virtually all cases one should simply think of these operators as AMR operators that handle specific types of common data operations on mesh and particle data.

`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 its available input parameters.

Tip: Here is the AmrMesh C++ API

2.5.1 Class options

Various class options are available for adjusting the behavior of the `Driver` class. Below, we include the current template options file for the `Driver` class.

Listing 2.5.1: Template input options for the `AmrMesh` class. Runtime adjustable options are highlighted.

```
# ================================================================================================
# AmrMesh class options
# ================================================================================================
AmrMesh.lo_corner            = -1 -1 -1       ## Low corner of problem domain
AmrMesh.hi_corner            = 1  1  1        ## High corner of problem domain
AmrMesh.verbosity            = -1             ## Controls verbosity.
AmrMesh.coarsest_domain      = 128 128 128    ## Number of cells on coarsest domain
AmrMesh.max_amr_depth        = 0              ## Maximum amr depth
AmrMesh.max_sim_depth        = -1             ## Maximum simulation depth
AmrMesh.fill_ratio           = 1.0            ## Fill ratio for grid generation
AmrMesh.buffer_size          = 2              ## Number of cells between grid levels
AmrMesh.grid_algorithm       = tiled          ## Berger-Rigoustous 'br' or 'tiled' for the tiled algorithm
AmrMesh.box_sorting          = morton         ## 'none', 'shuffle', 'morton'
AmrMesh.blocking_factor      = 16             ## Blocking factor.
AmrMesh.max_box_size         = 16             ## Maximum allowed box size
AmrMesh.max_ebis_box         = 16             ## Maximum allowed box size for EBIS generation.
AmrMesh.ref_rat              = 2 2 2 2 2 2    ## Refinement ratios (mixed ratios are allowed).
AmrMesh.num_ghost            = 2              ## Number of ghost cells.
AmrMesh.lsf_ghost            = 2              ## Number of ghost cells when writing level-set to grid
AmrMesh.eb_ghost             = 2              ## Set number of of ghost cells for EB stuff
AmrMesh.mg_interp_order      = 2              ## Multigrid interpolation order
AmrMesh.mg_interp_radius     = 2              ## Multigrid interpolation radius
AmrMesh.mg_interp_weight     = 1              ## Multigrid interpolation weight (for least squares)
AmrMesh.centroid_interp      = minmod         ## Centroid interp stencils. linear, lsq, minmod, etc
AmrMesh.eb_interp            = minmod         ## EB interp stencils. linear, taylor, minmod, etc
AmrMesh.redist_radius        = 1              ## Redistribution radius for hyperbolic conservation laws
```

For users, modifying the behavior of `AmrMesh` is comparatively easy. We have listed the most commonly adjusted grid parameters above, which include the specification of the physical domain, the maximum number of AMR levels that are permitted, as well as how grids are generated.

2.5.2 Coarse-grid decomposition

`AmrMesh.lo_corner` and `AmrMesh.hi_corner` are the physical corners of the simulation domain. `AmrMesh.coarsest_domain` is the number of grid cells on the coarsest grid level (i.e., without AMR). It is important that cell sizes are uniform, so one must always have $\Delta x = \Delta y = \Delta z$. Usually, this means that `AmrMesh.lo_corner`, `AmrMesh.hi_corner`, and `AmrMesh.coarsest_domain` must all be consistently defined. Moreover, it is normally desirable to make `AmrMesh.coarsest_domain` a factor of 2 (e.g., 64, 128, 256, etc.), since this permits arbitrarily deep multigrid coarsening. This is not a requirement, however, although we do note that `AmrMesh.coarsest_domain` *must* be divisible by `AmrMesh.blocking_factor`.

2.5.3 Domain decomposition

With Cartesian AMR, each grid level is decomposed into grid blocks of constant size, or sizes that potentially vary between some min/max size along each dimension. In `chombo-discharge` this is encapsulated by `AmrMesh.blocking_factor`, which is the smallest grid box that can be generated when meshing the domain. Likewise, `AmrMesh.max_box_size` is the maximum box size that can be produced, but usage of a constant box size is common an increased requirement in `chombo-discharge`.

Tip: Use fixed box sizes where `AmrMesh.blocking_factor` and `AmrMesh.max_box_size` are the same.

The flag `AmrMesh.max_ebis_box` indicates essentially the blocking factor when generating the cut-cell information at the start of a simulation. It may happen for very large simulations that one has to increase the box size (e.g., to 32) in order to trim grid metadata.

`AmrMesh.ref_rat` determines the refinement ratio between grid levels, and factors of 2 and 4 are supported. We want to point out that mixed refinement factors are supported.

Tip: If `AmrMesh.max_amr_depth` is greater than the number of refinement ratios specified in `AmrMesh.ref_rat`, `chombo-discharge` will automatically fill in the remaining refinement ratios (padding with the last entry). I.e., one obtains factor 2 refinement every if `AmrMesh.ref_rat = 2`.

Two gridding algorithms are supported, called Tiled mesh refinement the classical Berger-Rigoutsos refinement algorithm. These are discussed in *Mesh generation*, and the user can specify which one to use by setting `AmrMesh.grid_algorithm` accordingly. In general, the tiled algorithm is exceedingly more performant at larger scales.

2.5.4 Ghost cells

It is normally not necessary to adjust the number of ghost cells in `chombo-discharge` simulations since most discretizations require at most 2 ghost cells. Substantial efforts have been made to avoid increasing the required number of ghost cells. Regardless, the number of ghost cells can be adjusted by setting `AmrMesh.num_ghost` to a specified value. A companion parameter is `AmrMesh.eb_ghost`, which specifies the maximum number of ghost cells that are used when computing the cut-cell discretization.

Warning: One must always have `AmrMesh.eb_ghost` > `AmrMesh.num_ghost`.

Finally, it is possible to evaluate implicit functions on the mesh (e.g., for figuring out how far a cell is from a boundary). It can be useful to include a larger ghost region in these data holders, which is adjusted by `AmrMesh.lsf_ghost`.

2.5.5 Multigrid interpolation

Multigrid interpolation is done using least squares reconstruction, as discussed in *Multigrid ghost cell interpolation*. The user can set the order, radius, and least squares weighting for this interpolation by setting `AmrMesh.mg_interp_order`, `AmrMesh.mg_interp_radius`, and `AmrMesh.mg_interp_weight`. Note that specifying `AmrMesh.mg_interp_radius > AmrMesh.eb_ghost` is sure to lead to a run-time error (possibly a segfault).

2.5.6 Interpolation near the EB

Most data in `chombo-discharge` is cell-centered, although data can be interpolated or extrapolated to cell centroids and EB centroids. The interpolation type is done by setting `AmrMesh.centroid_interp` and `AmrMesh.eb_interp` accordingly. Currently, we support the following options:

1. `constant` i.e., use the cell centered value.
2. `linear`, using bi/tri-linear interpolation.
3. `taylor`, using the Taylor series evaluated at the cell center.
4. `lsq`, using a first order unweighted least squares polynomial.
5. `pwl`, using piecewise linear reconstruction.
6. `minmod`, using a minmod slope limiter.
7. `superbee`, using a superbee slope limiter.
8. `monotonized_central`, using a van Leer slope limiter.

Typically, one can just leave this option at `minmod`.

2.6 CellTagger

The `CellTagger` class is responsible for flagging grid cells for refinement or coarsening, and is thus the main class that determines what the grids look like. By default, `CellTagger` does not actually flag anything for refinement, so if the user wants to implement a new refinement or coarsening routine, one must 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* and the user is responsible for providing `CellTagger` with these dependencies.

Tip: Here is the CellTagger C++ API

Refinement flags live in a data holder called `EBAMRTags`, which is a typedef:

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

See *Chombo-3 basics* for an explanation of how the individual templates. Briefly, the outer vector in `EBAMRTags` indicates the grid level, whereas `LayoutData` is a data holder on each grid level and indexes the grid patches. That is, `LayoutData<DenseIntVectSet` is a distribution of `DenseIntVectSet` on a level, whereas `DenseIntVectSet` is a data holder that stores the refinement flags on a grid patch. For performance reasons, `DenseIntVectSet` onlys store refinement cells on a per-patch basis. It is not possible to add a grid cell to a `DenseIntVectSet` if it falls outside the grid patch. Note that `EBAMRTags` is *not* set up for communication, and so it is not possible to fill ghost cells regions from other patches. The refinement flags themselves are owned by *Driver*, and are used to generate new grids that cover all the cell tags.

A part of the current C++ header file for `CellTagger` is included below, where we highlight the functions that must be implemented in order to create a new refinement method.

Listing 2.6.1: Header file for `CellTagger`.

```
/*!
  @brief Regrid function for cell tagger (in case it uses transient storage to do things)
  @details This function exists because implementations may require data to be allocated on a mesh. This function is
  called by Driver to make sure data is reallocated when it needs to.
*/
virtual void
regrid() = 0;

/*!
  @brief Parse class options.
  @note This function is called by Driver.
*/
virtual void
parseOptions() = 0;

/*!
  @brief Tag cells
  @param[inout] a_tags Tags on grid levels
  @details EBAMRTags is a data-type Vector<RefCountedPtr<LayoutData<DenseIntVectSet> > >. The vector indicates the grid level,␣
↪the LayoutData indicates data ownership (in
  much the same way as LevelData. The DenseIntVectSet is essentially an IntVectSet restricted to the patch (i.e. one cannot␣
↪add IntVects that are outside the patch).
  This function cells for refinement or coarsening. The user's responsibility is to add (or remove) tags from a_tags.
*/
virtual bool
tagCells(EBAMRTags& a_tags) = 0;
```

2.6.1 Implementing a new `CellTagger`

To implement a new `CellTagger`, the pure functions listed in Listing 2.6.1 must be implemented. Below, we discuss these in turn:

regrid

`regrid` is called by *Driver* during regrids. The existence of this routine is due to the common usage pattern where `CellTagger` holds some auxiliary mesh (or particle) data that is used when evaluating refinement and coarsening criteria. This routine should reallocate this storage during regrids.

tagCells

When the regrid routine enters, the `CellTagger` will be asked to generate the refinement flags through the `tagCells` function in Listing 2.6.1. This routine should add cells that will be refined, and remove cells that will be coarsened. The return value of this function is a boolean that should return `false` if no new tags were found. While it is possible to skip this test, we note that returning `true` also when no new refinement regions were found will trigger a full regrid.

parseOptions

`parseOptions` is called by *Driver* when setting the `CellTagger`. This routine should parse options into the `CellTagger` subclass. Note that it is also useful to overwrite the function `parseRuntimeOptions` if one needs run-time configuration of the refinement criteria.

Plot data

It is also possible to have `CellTagger` write data to plot files, and the interface for this is identical to the plot file interface in *TimeStepper*. The three functions below must then be implemented:

```
/*!
  @brief Get number of plot variables that will be written to file (by Driver).
  @return Returns number of plot variables that Driver will write to plot files.
*/
virtual int
getNumberOfPlotVariables() const;

/*!
  @brief Get plot variable names.
*/
virtual Vector<std::string>
getPlotVariableNames() const;

/*!
  @brief Write plot data.
  @param[inout] a_output Output data holder
  @param[inout] a_icomp Starting variable in a_output where we begin appending data.
  @param[in] a_outputRealm Realm where a_output belongs
  @param[in] a_level Grid level
*/
virtual void
writePlotData(LevelData<EBCellFAB>& a_output, int& a_icomp, const std::string a_outputRealm, const int a_level) const;
```

The interpretation of these functions is exactly the same as for *TimeStepper*, and we refer to the *TimeStepper* documentation for a detailed explanation.

2.6.2 Manual refinement and restriction

The user can add manual refinement by specifying Cartesian spatial regions to be refined down to some grid level, by specifying the physical corners and the refinement level. The input parameters in this case are

- `num_ref_boxes` for specifying how many such boxes will be parsed.
- `ref_box<num>_lo` and `ref_box<num>_hi` that determine the Cartesian region to be refined. Here, `<num>` is a placeholder for an integer. If the user specifies `num_ref_boxes = 2`, then `ref_box1_lo`, `ref_box1_hi`, `ref_box2_lo`, and `ref_box2_hi` must all be defined.
- `ref_box<num>_lvl`, which specifies the refinement depth corresponding to box `<num>`.

An example of the manual refinement syntax is given in Listing 2.6.2.

Listing 2.6.2: Default class options for `CellTagger`, including the manual refinement syntax and buffer region definition.

```
# ============================================================================================
# CellTagger class options
# ============================================================================================
CellTagger.buffer     = 0    ## Grow tagged cells

CellTagger.num_tag_boxes     = 0              ## Number of allowed tag boxes (0 = tags allowe everywhere)
CellTagger.tag_box1_lo       = 0.0 0.0 0.0    ## Only allow tags that fall between
CellTagger.tag_box1_hi       = 0.0 0.0 0.0    ## these two corners

CellTagger.num_ref_boxes     = 0              ## Number of specified refinement boxes
CellTagger.ref_box1_lo       = 0.0 0.0 0.0    ## Refine region between
CellTagger.ref_box1_hi       = 0.0 0.0 0.0    ## these two corners.
CellTagger.ref_box1_lvl      = 0              ## Refine to this level.
```

It is possible to prevent `CellTagger` from adding refinement flags in specified regions by specifying `num_tag_boxes`. The default behavior is to add a number of boxes where refinement and coarsening is allowed, and prevent tags from being generated outside of these regions. If no boxes are defined, tagging is allowed everywhere. The syntax is the same as for `num_ref_boxes`, e.g., one must define `tag_box<num>_lo` and `tag_box<num>_hi`. By adding restrictive

boxes, tagging will only be allowed inside the specified box corners `tag_box1_lo` and `tag_box1_hi`. More boxes can be specified by following the same convention, e.g., `tag_box2_lo` and `tag_box2_hi`.

2.6.3 Adding a buffer

By default, each MPI rank can only tag grid cells where it owns data, which has has been enforced due to 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. Frequently, however, it is necessary to add *buffer regions* to ensure that an area-of-interest around the tagged region is also refined. This is done by growing the final generated tags by specifying the `buffer`. The syntax for this is given in Listing 2.6.2. Just before passing the flags into `AmrMesh` grid generation routines, the tagged cells are put into a different data holder (`IntVectSet`), and this data holder *can* contain cells that are outside the patch boundaries.

CHAPTER

THREE

DISCRETIZATION

3.1 Spatial discretization

3.1.1 Cartesian AMR

`chombo-discharge` uses Cartesian patch-based structured adaptive mesh refinement (AMR) provided by `Chombo` [Colella *et al.*, 2004]. In patch-based AMR the domain is subdivided into a collection of hierarchically nested grid levels, see Fig. 3.1.1, where each grid level consist of a set of rectangular grid blocks. I.e., 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 its parent (i.e., coarser) grid. In other words, grid levels $l-1$ and $l+1$ are spatially separated by a non-zero number of grid cells on level l.

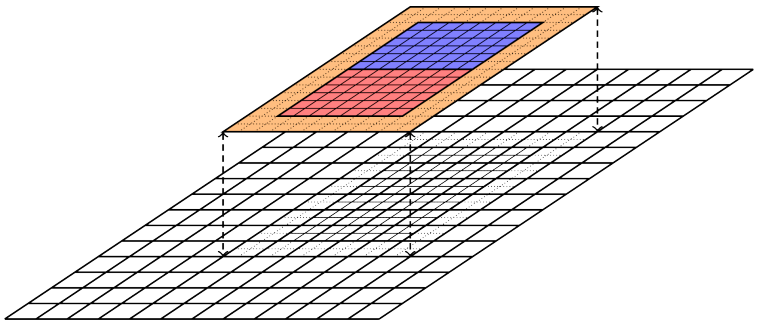

Fig. 3.1.1: Cartesian patch-based refinement showing two grid levels. The fine-grid level lives on top of the coarse level, and consists of two patches (red and blue colors) with two layers of ghost cells (dashed lines and orange shaded region).

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

Important: `chombo-discharge` only supports refinement factors of 2 and 4, with a few limitations for factor 4 refinement.

3.1.2 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. Fig. 3.1.2 shows an example of patch-based grid refinement for a complex surface.

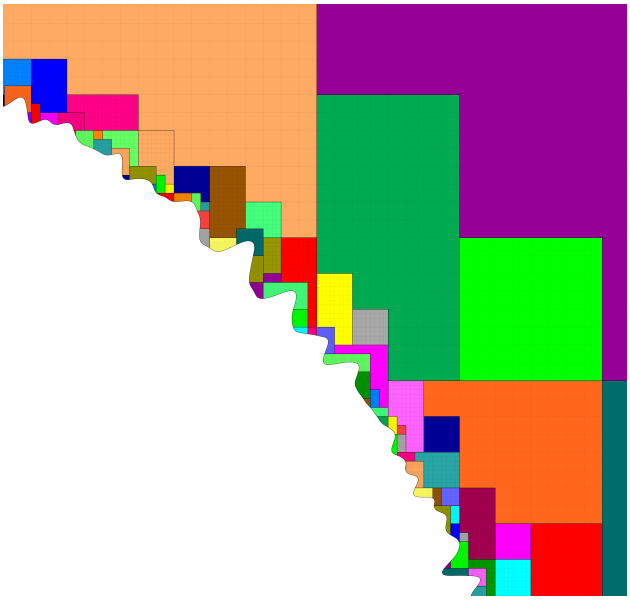

Fig. 3.1.2: Patch-based refinement (factor 4 between levels) of a complex surface. Each color shows a patch, which is a rectangular computational unit.

Since EBs are directly intersected by the geometry, pathological cases can arise where coarsening of a grid cell leads to a coarser cell consisting of multiple volumes. One can easily envision this case by intersecting a thin body with a Cartesian grid, as shown in Fig. 3.1.3. 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 when coarsening is involved. 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). However, because of multi-cells, EB data structures are not purely Cartesian. Data structures need to live on more complex graphs that describe the multi-cells and, furthermore, describe the cell connectivity between cut-cell volumes. Without multi-cells it would be impossible to describe most complex geometries, and it is extremely difficult to obtain performant geometric multigrid methods which instrinsically rely on this type of coarsening.

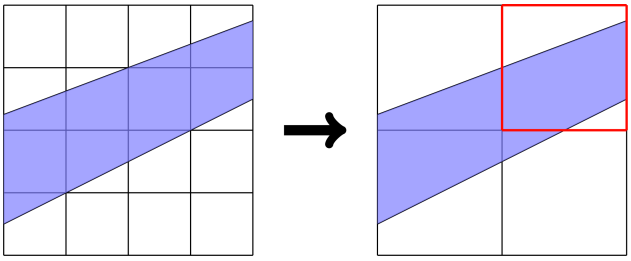

Fig. 3.1.3: Example of how multi-valued cells occur during grid coarsening. Left: Original grid. Right: Coarsened grid where one grid cell is multi-valued.

3.1.3 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 for the following reasons: 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, and permit mesh-less ray-tracing of computational particles.

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

$$d_1(\mathbf{x}) = R - |\mathbf{x}|, \tag{3.1.1}$$

$$d_2(\mathbf{x}) = R^2 - \mathbf{x} \cdot \mathbf{x}. \tag{3.1.2}$$

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 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.

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.

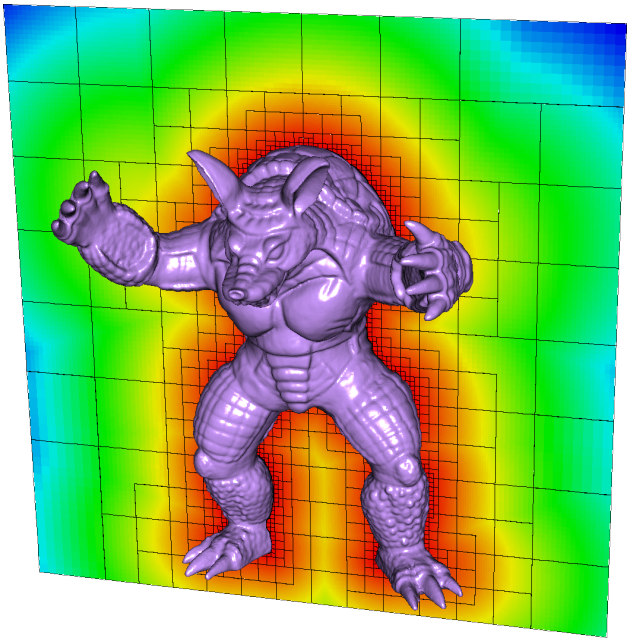

Fig. 3.1.4: Example of an SDF reconstruction and cut-cell grid from a surface tesselation in `chombo-discharge`.

3.1.4 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 $\mathcal{O}\left(N^2\right)$. However, as N becomes large, say $N = 10^5$, geometric queries of this type quickly become a bottleneck and the algorithm becomes practically unusable.

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 for geometries consisting of relatively small object surface areas.

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

$$|\nabla f| = 1, \tag{3.1.3}$$

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. 3.1.5. We know that cut cells do not exist in the grid patch if $|f\left(\mathbf{x_i}\right)| > \frac{\sqrt{3}\Delta x}{2}$ 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, so the problem lends itself naturally to parallelization. By default, partitioning of subregions is done using a Morton curve. 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. 3.1.5. 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.

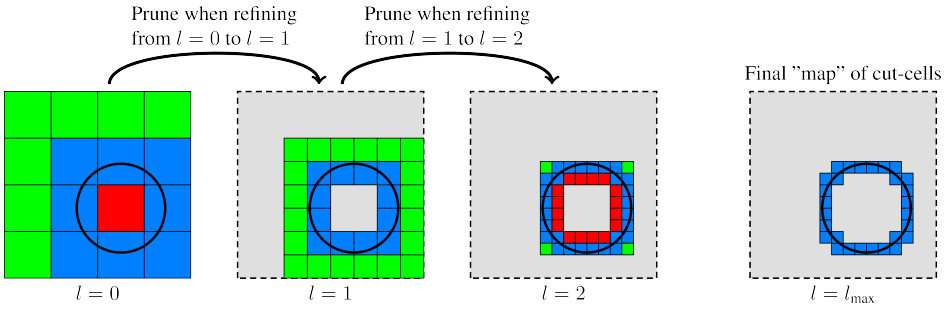

Fig. 3.1.5: Pruning cut-cells with the signed distance field. Red-colored grid patches are grid patches entirely contained inside the EB. Green-colored grid patches are entirely outside the EB, while blue-colored grid patches contain cut-cells.

The above load balancing strategy is very simple, and it reduces the original $O(N^3)$ complexity in 3D to $O(N^2)$ complexity, while 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.

3.1.5 Mesh generation

`chombo-discharge` supports two grid generation algorithms:

1. The classical Berger-Rigoutsos algorithm [Berger and Rigoutsos, 1991].
2. A *tiled* algorithm [Gunney and Anderson, 2016].

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 not fundamentally required for quadtree and octree grids, but is nevertheless usually imposed. 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 $l+1$, which greatly simplifies the definition of numerical stencils on each level. For example, ghost cells on level $l+1$ can then be interpolated from data only on levels l and $l+1$.

Berger-Rigoutsos algorithm

The Berger-Rigoustous grid algorithm is implemented in `Chombo` and can be 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 [Berger and Rigoutsos, 1991] for implementation details. Typically, it is not used at large scale in 3D due to its memory consumption.

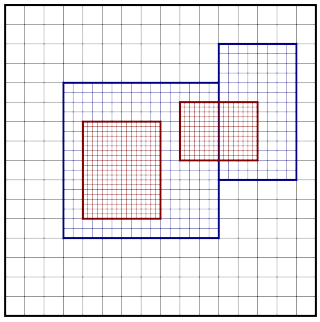

Fig. 3.1.6: Classical cartoon of patch-based refinement. Bold lines indicate entire grid blocks.

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.

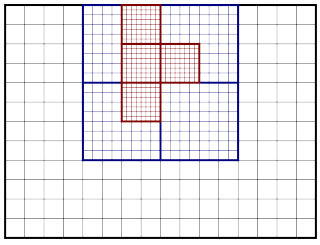

Fig. 3.1.7: Classical cartoon of tiled patch-based refinement. Bold lines indicate entire grid blocks.

3.1.6 Cell refinement philosophy

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

1. Refine all embedded boundaries down to a specified refinement level. See *Driver*.
2. Refine embedded boundaries based on estimations of the surface curvature in the cut-cells. See *Driver*.
3. Manually add refinement flags (by specifying boxes where cells will be refined). See *CellTagger*.
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. See *CellTagger*.

The first two cases are covered by the *Driver* class in chombo-discharge. In the first case *Driver* 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 *CellTagger* classes.

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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, and set the centering to face centered.

`Box` is frequently used throughout `chombo-discharge` when iterating over grid cells.

3.2.5 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` which is a Cartesian array, and a additional data structure that holds data in multi-valued grid cells. Doing arithmetic with `EBCellFAB` usually requires one to iterate over all the cell in the `FArrayBox`, and then also to iterate over the *irregular cells* (i.e., cut-cells) later. A `VoFIterator` is an object designed to iterate over the irregular cells, and must be defined by the set of cells that it will iterate over and the graph that describes the cell connectivity.

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

3.2.6 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`.

3.2.7 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 is 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.

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

3.2.8 DisjointBoxLayout

The `DisjointBoxLayout` class describes a grid on an AMR level where all the boxes are *disjoint*, i.e., boxes which do not 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` is not a distributed data structure, and 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 its own part of the `DisjointBoxLayout`.

```
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.

A related data iterators is `LayoutIterator`, which will 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 operations, e.g., count the number of cells in the grid. However, trying to use `LayoutIterator` to retrieve data that was allocated by different MPI rank is an error.

3.2.9 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 within `LevelData<T>` structure you must also implement a factory method for `T`, as well as an appropriate linearization function for `T`.

The `LevelData<T>` object provides the domain decomposition method in `Chombo` and `chombo-discharge`. Often, `T` is an `EBCellFAB`.

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.

3.2.10 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, normal vectors, 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 `LayoutData<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.ok(); ++dit){
   const Box      cellBox = dbl[dit()];
   const EBISBox& ebisbox = ebisl [dit()];

   EBCellFAB& patchData = myData[dit()];

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

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

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

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

3.2.11 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.

3.3 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. A requirement on the datatype `T` is that it must be linearizable so that it can be communicated across MPI ranks. 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`. Fig. 3.3.1 shows a sketch of the data layout in a two-level AMR hierarchy.

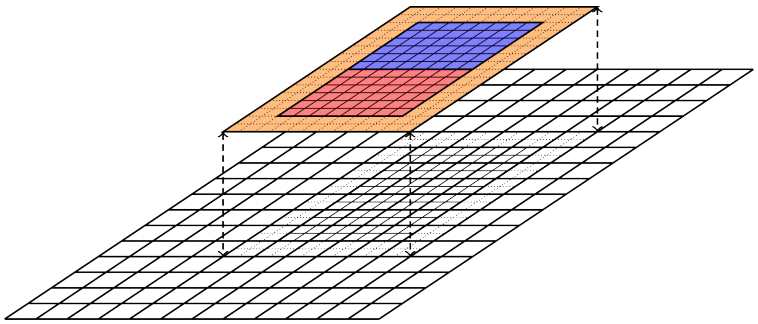

Fig. 3.3.1: Cartesian patch-based refinement showing two grid levels. The finer level consists of two patches (red and blue zones), and these zones each have a ghost cell layer 2 cells wide. The data lies on top of a coarse-grid data, i.e., data simultaneously exists on both the fine and the coarse levels. This data type is encapsulated by `EBAMRData<T>`.

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* has functionality for constructing 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 data holder that holds data on each EB centroid (or boundary centroid) across the entire AMR hierachy. In the same way, `EBAMRIFData` holds data on each face of all cut-cells in the hierarchy.

3.3.1 Allocating mesh data

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

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

Here, `numCOmps` determine the number of 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 numComps = 1;
EBAMRCellData myData;
m_amr->allocate(myData, "myRealm", phase::gas, numComps, 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.

3.3.2 Iterating over the AMR hierarchy

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

```
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()];
   }
}
```

Throughout `chombo-discharge` it will be common to see the above implemented explicitly as a loop that supports OpenMP:

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

   const DisjointBoxLayout& levelGrids = levelData.disjointBoxLayout();
   const DataIterator& dataIterator    = levelGrids.dataIterator();

   const int numBoxes = dataIterator.size();

#pragma omp parallel for schedule(runtime)
   for (int currentBox = 0; currentBox < numBoxes; currentBox++) {
      const DataIndex& dataIndex = dataIterator[currentBox];

      EBCellFAB& patchData = levelData[dataIndex];
   }
}
```

3.3.3 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`. Iterating over the cut-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.

3.3.4 Coarsening data

Coarsening of data implies replacing the coarse-grid data that lies underneath a fine grid by some average of the fine-grid data. We currently support the following coarsening algorithms:

- Arithmetic coarsening, in which the coarse-grid value is simply the average of the fine-grid values.
- Conservative coarsening, in which the coarse-grid value is the conservative average of the fine-grid values. This implies that the total mass on the coarse-grid cell is identical to the total mass in the fine-grid cells from which one coarsened.
- Harmonic, in which the coarse-grid value is the harmonic average of the fine-grid cell values.

These functions are available for both cell-centered data, cut-cell data, and face-centered data. Multiply signatures for this functionality exists, see the code-block below.

```
/*!
  @brief Average multifluid data over a specified realm
  @param[inout] a_data    Data to be coarsened.
  @param[in]    a_realm   Realm name
  @param[in]    a_average Averaging method
*/
void
average(MFAMRCellData& a_data, const std::string a_realm, const Average& a_average) const;

/*!
  @brief Average multifluid data over a specified realm
  @param[inout] a_data    Data to be coarsened.
  @param[in]    a_realm   Realm name
  @param[in]    a_average Averaging method
*/
void
average(MFAMRFluxData& a_data, const std::string a_realm, const Average& a_average) const;

/*!
  @brief Average down on specific realm and phase.
  @param[inout] a_data  Data to be coarsened.
  @param[in]    a_realm Realm name
  @param[in]    a_phase Phase (gas or solid)
  @param[in]    a_average Averaging method
*/
void
average(EBAMRCellData&           a_data,
        const std::string        a_realm,
        const phase::which_phase a_phase,
        const Average&           a_average) const;
```

See the AmrMesh API for further details.

3.3.5 Filling ghost cells

Filling ghost cells is done using the `interpGhost(...)` functions in *AmrMesh*. This process adheres to the following rules:

1. Within a grid level, cells are always filled from neighboring grid patches without interpolation.
2. Around the halo zone (see Fig. 3.3.1), ghost cells are filled using slope-limited interpolation *from the coarse grid only*. Currently, this slope is calculated with a minmod limiter, although support for superbee, piecewise constant, and van Leer limiters are also implemented.

The signatures for updating the ghost cells are:

```
/*!
  @brief Interpolate ghost vectors over a realm, using the default ghost cell interpolation method.
  @param[inout] a_data  Data to be interpolated.
  @param[in]    a_realm Realm name
  @param[in]    a_phase Phase (gas or solid)
*/
void
interpGhost(EBAMRCellData& a_data, const std::string a_realm, const phase::which_phase a_phase) const;
```

As one alternative, one can update ghost cells on a single grid level:

```
/*!
  @brief Interpolate ghost cells over a realm, using the default ghost cell interpolation method on a specific level.
  @param[inout] a_fineData Fine grid data
  @param[inout] a_coarData Coarse grid data
  @param[in]    a_level    The grid level corresponding to a_fineData
  @param[in]    a_realm    Realm name
  @param[in]    a_phase    Phase (gas or solid)
*/
void
interpGhost(LevelData<EBCellFAB>&       a_fineData,
            const LevelData<EBCellFAB>& a_coarData,
            const int                   a_level,
            const std::string           a_realm,
            const phase::which_phase    a_phase) const;
```

Strictly speaking it is also possible to update ghost cells using the multigrid interpolator, but this will only fill a single layer of ghost cells around the halo zone (except near the cut-cells where additional cells are filled).

3.3.6 Interpolating from the coarse grid

Coarse-grid interpolation occurs, e.g., when the AMR hierarchy changes. If one needs data on a grid level where no data already exists, it is possible to fill this data by interpolating from the coarse grid to a finer one.

Important: This type of interpolation is distinctly different from the ghost cell interpolation, as it affects data across the whole grid patch.

The interpolation function that fill fine-grid data from a coarse grid has the following signature:

```
/*!
  @brief Interpolate data to new grids
  @details This is called when requiring data to be interpolated to new grids. Takes old data as argument
  and fills the new grid data with an interpolation of the old grid data.
  @param[out] a_newData        New grid data.
  @param[in]  a_oldData        Old grid data.
  @param[in]  a_phase          Phase on which we regrid.
  @param[in]  a_lmin           Coarsest level that did not change (but distribution may have changed).
  @param[in]  a_oldFinestLevel Previous finest level.
  @param[in]  a_newFinestLevel New finest level.
  @param[in]  a_type Interpolation type
*/
void
interpToNewGrids(EBAMRCellData&                  a_newData,
                 const EBAMRCellData&            a_oldData,
                 const phase::which_phase        a_phase,
                 const int                       a_lmin,
                 const int                       a_oldFinestLevel,
                 const int                       a_newFinestLevel,
                 const EBCoarseToFineInterp::Type a_type);
```

Here, the user must supply both the old data and the new data, as well as on which grid levels the interpolation will take place. The final argument `a_type` is the interpolation type. We currently support the following interpolation methods:

- `Type::PWC`, which is piecewise-constant interpolation where the fine-cell data is filled with the coarse-cell values.
- `Type::ConservativePWC`, which is a piecewise-constant interpolation that is also conservative (i.e., volume-weighted).
- `Type::ConservativeMinMod`, which is a conservative interpolation method that uses the minmod limiter.
- `Type::ConservativeMonotonizedCentral`, which is a conservative interpolation method that uses the van Leer limiter.
- `Type::Superbee`, which is a conservative interpolation method that uses the superbeed limiter.

Note that there is "correct" interpolation method, but we note that we typically use a conservative minmod limiter in `chombo-discharge`.

3.3.7 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 EB where the coarse-side stencil will avoid using the coarsened data beneath the fine level. This is shown in Fig. 3.3.2 which shows the typical 5-point stencil in regular grid regions, and also a much larger and more complex stencil.

In Fig. 3.3.2 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 a conservative 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.

Note: It is up to the user to ensure that ghost cells are filled prior to computing the gradient. This can be done either using multigrid interpolation (see *Multigrid ghost cell interpolation*), or standard interpolation (see *Filling ghost cells*).

Fig. 3.3.2 also shows a much larger stencil (blue stencil) on the coarse side of the refinement interface. 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 that entirely avoids using coarsened data. In this case we fetch a sufficiently large neighborhood of cells for computing a least squares minimization of a local solution reconstruction in the neighborhood of the coarse cell. In order to avoid fetching potentially badly coarsened data, this neighborhood of cells only uses *valid* grid cells, i.e., the stencil does not reach underneath the fine level at all. Once this neighborhood of cells is obtained, we compute the gradient using the procedure in *Least squares*.

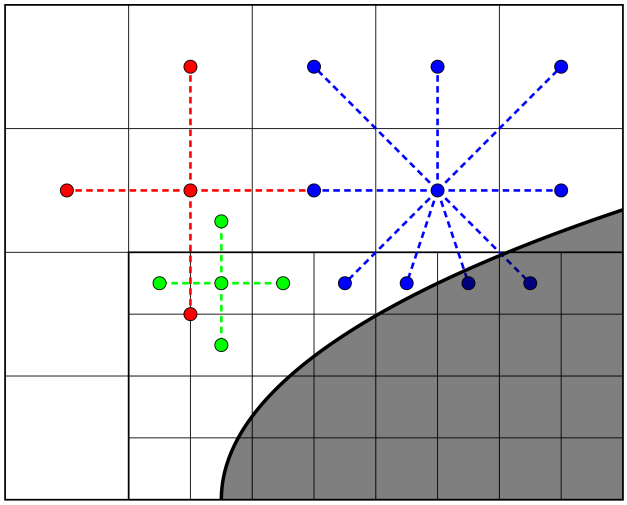

Fig. 3.3.2: Example of stencils for computing gradients near embedded boundaries. The red stencil shows a regular 5-point stencil for computing the gradient on the coarse side of the refinement boundary; it reaches into the coarsened data beneath the fine level. The green stencil shows a similar 5-point 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.

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

```
/*!
  @brief Compute cell-centered gradient over an AMR hierarchy.
  @param[out] a_gradient Cell centered gradient.
  @param[in]  a_phi      The scalar for which the gradient is computed.
  @param[in]  a_realm    Name of the realm where the data lives.
```

(continues on next page)

(continued from previous page)

```
  @param[in]  a_phase    Phase on which the data lives.
  @note This routine will reach into ghost cells and across refinement boundaries. The user must make sure that
  ghost cells are updated before using this routine.
*/
void
computeGradient(EBAMRCellData&           a_gradient,
                const EBAMRCellData&     a_phi,
                const std::string        a_realm,
                const phase::which_phase a_phase) const;
```

We reiterate that ghost cells must be updated *before* calling this routine. See *AmrMesh* or refer to the AmrMesh API for further details.

3.3.8 Copying data

To copy data between data holders, one may use the `AmrMesh<T>::copyData(...)` function *or* `DataOps::copy` (see *DataOps*).

The simplest way of copying data between data holders is via `DataOps::copy`, which does a *local-only* direct copy that also includes ghost cells. This version requires that the source and destination data holders are defined on the same realm, and does not invoke MPI calls.

A more general version is supplied by *AmrMesh*, and has the following structure:

```
/*!
  @brief Method for copying from a source container to a destination container. User supplies information
  @param[inout] a_dst        Destination data
  @param[in]    a_src        Source data
  @param[in]    a_dstComps   Destination components
  @param[in]    a_srcComps   Source components
  @param[in]    a_toRegion   Region we copy into
  @param[in]    a_fromRegion Region we copy from
  @note If the user requests copying into ghosted regions, the Copiers MUST USE THE CORRECT NUMBER OF GHOST CELLS
  @note This routine will not work if copying between grids before/after regrids.
*/
template <typename T>
void
copyData(EBAMRData<T>&       a_dst,
         const EBAMRData<T>& a_src,
         const Interval&     a_dstComps,
         const Interval&     a_srcComps,
         const CopyStrategy& a_toRegion   = CopyStrategy::Valid,
         const CopyStrategy& a_fromRegion = CopyStrategy::Valid) const noexcept;
```

In the above code, `a_dst` and `a_src` are the destination and source data holders for the copy. These need not be defined on the same *Realm*. Similarly, the `a_dstComps` and `a_srcComps` indicate the source and destination variables to be copied, which must have the same size. The final two arguments indicate which regions will be copied from. These are enums that are either `CopyStrategy::Valid` or `CopyStrategy::ValidGhost`, and indicates whether or not we will perform the copy only into *valid* cells (`CopyStrategy::Valid`) or also into the ghost cells (`CopyStrategy::ValidGhost`).

3.3.9 DataOps

We have prototyped functions for many common data operations in a static class `DataOps`.

Tip: For the full `DataOps` API, see the DataOps documentation.

`DataOps` contains numerous functions for operating on various template specializations of `EBAMRData<T>` (see *Mesh data*). For example, `DataOps` contains functions for scaling data, incrementing data, averaging cell-centered data onto faces, and many more.

Important: `DataOps` is designed to operate only within a single realm. This means that *all* arguments into the `DataOps` functions *must* be defined on the same realm.

3.4 Particles

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

- Particle-mesh operations, i.e., deposition and interpolation of particle variables to/from the mesh.
- Particle distribution and remapping with MPI.
- Rudimentary particle output to H5Part files.

Particle support is generally speaking templated, so that users can define new particle types that contain a desired set of variables. Typically, these will are derived from *GenericParticle*, which is discussed below.

3.4.1 GenericParticle

`GenericParticle` is a default particle usable by the `Chombo` particle library. The particle type is essentially a template

```
template <size_t M, size_t N>
class GenericParticle
```

where `M` and `N` are the number of `Real` and `RealVect` variables for the particle. The `GenericParticle` always stores the position of the particle, which is available through `GenericParticle<M,N>::position`.

To fetch the `Real` and `RealVect` variables, *GenericParticle* has member functions

```
template <size_t K>
inline Real&
GenericParticle<M,N>::real();

template <size_t K>
inline RealVect&
GenericParticle<M,N>::vect();
```

If using `GenericParticle` directly, the correct C++ way of fetching one of these variables is

```
GenericParticle<2,2> p;

Real& s = p.template real<0>();
```

Note that one must include the template keyword.

GenericParticle can also store the local (per MPI rank) particle ID and the MPI rank ID on the particle. These are available through member functions `particleID()` and `rankID`.

Tip: The `GenericParticle` C++ API is found at https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classGenericParticle.html.

Linearization functions

GenericParticle has linearization functions communicating particles with MPI, as well as for supplying output to HDF5 checkpoint files. By default, *GenericParticle* will communicate all particle properties (including rank ID and particle ID), and include all `Real` member values in the HDF5 checkpoint files. For particles that do not need to checkpoint all particle properties when writing HDF5 checkpoint files, it may be beneficial to reduce the file size by only exporting the necessary particle properties (e.g., the position and particle weight).

3.4.2 Custom particles

To create a simple custom particle class with more sane signatures, one can inherit from `GenericParticle` and specify new function signatures that return the appropriate fields. An example of this is given in the code-block below, where we define `KineticParticle` to be a particle that contains the th5ree additional fields on top of `GenericParticle` (weight, velocity, and acceleration).

```
class KineticParticle : public GenericParticle<1,2>
{
public:
   inline
   Real& weight() {
      return this->real<0>();
   }

   inline
   RealVect& velocity() {
      return this->vect<0>();
   }

   inline
   RealVect& acceleration() {
      return this->vect<1>();
   }
};
```

There are many particles in `chombo-discharge`, see the `GenericParticle` C++ API for more information.

3.4.3 ParticleContainer

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

1. Stores computational particles of type `P` over an AMR hierchy.
2. Provides infrastructure for remapping particles.
3. Provides functionality for getting a list of particles within a specified grid patch.
4. Provides functionality that is required during regrids.
5. Other types of functionality, like grouping particles into grid cells, set and get functions for assigning particle variables, etcl.

`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 *GenericParticle*, which will fulfill all template constraints.

3.4.4 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.

`AMRParticles<P>` always lives within `ParticleContainer<P>`, and is the class member of `ParticleContainer<P>` that actually holds the particles.

3.4.5 Basic usage

Here, we give some examples of basic usage of `ParticleContainer`. For the full API, see the `ParticleContainer` C++ API https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classParticleContainer.html.

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();
```

3.4.6 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.

3.4.7 Allocating particles

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

```
/*!
  @brief Template class for generic allocation of particle containers.
  @param[out] a_container Particle container to be allocated
  @param[out] a_realm     Realm on which the particles will be allocated.
*/
template <typename T>
void
allocate(ParticleContainer<T>& a_container, const std::string a_realm) const;
```

which will allocate a `ParticleContainer` on the realm `a_realm`.

3.4.8 Particle mapping

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

```
/*!
  @brief Remap over the entire AMR hierarchy
*/
void
remap();
```

This is simply used as follows:

```
ParticleContainer<P> myParticles;

myParticles.remap();
```

During remapping, the following steps are performed for each MPI rank:

1. Collect all particles from this rank onto thread-local particles (if using OpenMP).
2. Iterate through those particles and locate their position in the AMR hierarchy (level, grid index, and owning MPI rank).
3. The particles that will move off each MPI rank are put in separate data containers.
4. Assign local particles first, i.e., particles that moved off a grid patch into another grid patch owned by the same rank.
5. Scatter the particles with MPI.
6. Assign the scattered particles to each MPI rank.

3.4.9 Regridding

As with mesh data, `ParticleContainer<P>` requires storing the old-grid data before assigning data on the new grids. This is relatively simple to achieve, and is done as follows:

1. *Before* creating the new grids, each MPI rank collects *all* particles on a single `List<P>` by calling

```
/*!
  @brief Cache particles before calling regrid.
  @param[in] a_base Coarsest grid level which will not change.
*/
void
preRegrid(const 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.

   This is done by calling the `ParticleContainer<P>` regrid function:

```
/*!
  @brief Regrid function. a_base is the coarsest grid level which did not change
  @param[in] a_grids          AMR grids
  @param[in] a_domains        AMR domains
  @param[in] a_dx             Grid resolutions
  @param[in] a_refRat         Refinement ratios
  @param[in] a_validMask      Valid cells
  @param[in] a_levelTiles     Tiled AMR grids
  @param[in] a_base           Coarsest grid level that did not change.
  @param[in] a_newFinestLevel New finest grid level
*/
void
regrid(const Vector<DisjointBoxLayout>&         a_grids,
       const Vector<ProblemDomain>&             a_domains,
       const Vector<Real>&                      a_dx,
       const Vector<int>&                       a_refRat,
       const Vector<ValidMask>&                 a_validMask,
       const Vector<RefCountedPtr<LevelTiles>>& a_levelTiles,
       const int                                a_base,
       const int                                a_newFinestLevel);
```

Warning: One *must* call `preRegrid` before the `regrid`. Failure to do so will lead to loss of all particles.

3.4.10 Masked particles

`ParticleContainer<P>` also supports the concept of *masked particles*, where one can fetch a subset of particles that live only in specified grid cells. Typically, this "specified region" is the refinement boundary, but the functionality is generic and might prove useful also in other cases. This functionality is unlikely to be used directly by users of chombo-discharge, but it is nonetheless fruitful to understand the concept in order to more easily fathom how deposition across refinement boundaries proceed.

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 this is

```
/*!
  @brief Copy particles to mask particle data holder.
  @param[in] a_mask Mask
  @note If the mask is nullptr on any of the levels, those levels will be ignored.
*/
```

(continues on next page)

(continued from previous page)

```
void
copyMaskParticles(const Vector<RefCountedPtr<LevelData<BaseFab<bool>>>>& 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 retrieved through a call

```
/*!
  @brief Get the mask particles.
  @return m_maskParticles
*/
AMRParticles<P>&
getMaskParticles();
```

In the above functions 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 function

```
/*!
  @brief Clear the "mask" particles.
*/
void
clearMaskParticles() const;
```

An example pseudocode for working with masked particles is given below:

```
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();
```

3.4.11 Boundary interaction

`ParticleContainer<P>` is EB-agnostic and has no information about the embedded boundary and only partial information about the domain boundary. This means the following:

1. Particles remap just as if the embedded boundary was not there.
2. Particles that completely fall off the domain are deleted when calling the remapping function.

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);
}
```

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

intersection point can be computed and the particle can be deposited on the boundary, or bounced off it. See *AmrMesh* for details on how to obtain the distance function.

Domain edges

By default, the `ParticleContainer` remapping function will discard particles that fall outside of the domain. The user can also check if this happen by checking if the particle position is outside the computational domain:

```
GenericParticle<0,0> p;

const RealVect pos    = p.position();
const RealVect probLo = m_amr->getProbLo();
const RealVect probHi = m_amr->getProbHi();

bool outside = false;
for (int dir = 0; dir < SpaceDim; dir++) {
   if(pos[dir] < probLo[dir] || pos[dir] > probHi[dir]) {
      outside = true;
   }
}
```

Particle intersection

It is occasionally useful to catch particles that hit an EB or crossed a domain side. Assuming that the particle type `P` also has a member function that stores the starting position of the particle, one can compute the intersection point between the particle trajectory and the EB or domain sides. Currently, *AmrMesh* supports two methods for computing this

- Using a bisection algorithm with a user-specified step.
- Using a ray-casting algorithm.

These algorithms differ in the sense that the bisection approach will check for a particle crossing between two positions $\mathbf{x}_0$ and $\mathbf{x}_1$ using a pre-defined tolerance. The ray-casting algorithm will check if the particle can move from $\mathbf{x}_0$ towards $\mathbf{x}_1$ by using a variable step along the particle trajectory. This step is selected from the signed distance from the particle position to the EB such that it uses a large step if the particle is far away from the EB. Conversely, if the particle is close to the EB a small step will be used.

The algorithm that intersect the particles are a part of *AmrMesh*, and are called as follows:

```
/*!
  @brief Particle intersection algorithm based on ray-casting.
  @details This routine will iterate through all the particles and check if they intersect the geometry. The template
  parameter indicates the particle type -- it MUST have const RealVec& position() const and  const RealVect& oldPosition()␣
↪const
  functions that determine the start and stop position of the particle trajectory. This routine uses a ray-casting method
  to check for intersections with the EB (the domain side is much easier). If the particles are closer to the EB than
  a_tolerance, they are absorbed and placed on the EB. Their position are updated and they are placed in the
  a_ebParticles argument. This routine uses a ray-casting method where it computes the distance from the EB
  (assuming that the implicit function is a signed distance function). Particles are then moved that
  distance along their trajectory and we then update the new distance to the EB. This is done recursively until the particles
  have either moved the entire length or been absorbed by the EB or domain side.
  @param[inout] a_activeParticles     Particles to be intersected with geometry
  @param[out]   a_ebParticles         Particles that intersected with the EB
  @param[out]   a_domainParticles     Particles that intersected with the domain faces
  @param[in]    a_phase               Phase where the input particles live
  @param[in]    a_bisectionStep       Length of the bisection step
  @param[in]    a_deleteParticles     If true, particles will be removed from a_activeParticles if they intersect the geometry.
  @param[in]    a_nonDeletionModifier Optional input argument for letting the user manipulate particles that were intersected␣
↪but not deleted
*/
template <class P>
void
intersectParticlesRaycastIF(
  ParticleContainer<P>&         a_activeParticles,
  ParticleContainer<P>&         a_ebParticles,
  ParticleContainer<P>&         a_domainParticles,
```

(continues on next page)

(continued from previous page)

```
  const phase::which_phase      a_phase,
  const Real                    a_tolerance,
  const bool                    a_deleteParticles,
  const std::function<void(P&)> a_nonDeletionModifier = [](P&) -> void {
    return;
  }) const noexcept;

/*!
  @brief Particle intersection algorithm based on bisection.
  @details This routine will iterate through all the particles and check if they intersect the geometry. The template
  parameter indicates the particle type -- it MUST have const RealVec& position() const and  const RealVect& oldPosition()␣
↪const
  functions that determine the start and stop position of the particle trajectory. This routine uses a bisection method
  to check for intersections with the EB (the domain side is much easier). Their position are updated and they are placed
  in the a_ebParticles argument.
  @param[inout] a_activeParticles     Particles to be intersected with geometry
  @param[out]   a_ebParticles         Particles that intersected with the EB
  @param[out]   a_domainParticles     Particles that intersected with the domain faces
  @param[in]    a_phase               Phase where the input particles live
  @param[in]    a_bisectionStep       Length of the bisection step
  @param[in]    a_deleteParticles     If true, particles will be removed from a_activeParticles if they intersect the geometry.
  @param[in]    a_nonDeletionModifier Optional input argument for letting the user manipulate particles that were intersected␣
↪but not deleted
*/
template <class P>
void
intersectParticlesBisectIF(
  ParticleContainer<P>&         a_activeParticles,
  ParticleContainer<P>&         a_ebParticles,
  ParticleContainer<P>&         a_domainParticles,
  const phase::which_phase      a_phase,
  const Real                    a_bisectionStep,
  const bool                    a_deleteParticles,
  const std::function<void(P&)> a_nonDeletionModifier = [](P&) -> void {
    return;
  }) const noexcept;
```

The above two functions take as input/output arguments the particles to be iterated through (`a_activeParticles`). When calling the intersection functions, the intersected particles are put into EB-intersected particels (`a_ebParticles`) and domain-intersected particles (`a_domainParticles`). The user can choose whether or not to remove intersected particles from `a_activeParticles` by adjusting `a_deleteParticles`. The final argument lets the user supply a lambda that modifies particles that were intersected.

Important: The intersection functions require that `P` has a member function `oldPosition` which supplies the starting position of the particle.

Both the bisection and ray-casting algorithm have weaknesses. The bisection algorithm algorithm requires a user-supplied step in order to operate efficiently, while the ray-casting algorithm is very slow when the particle is close to the EB and moves tangentially along it. Future versions of `chombo-discharge` will likely include more sophisticated algorithms.

Tip: `AmrMesh` also stores the implicit function on the mesh, which could also be used to resolved particle collisions with the EB/domain.

3.4.12 Particle-mesh

Particle-mesh operations are required when particles interact with the mesh and vice-versa. There are two main operations involved:

1. *Deposition*, where particle properties are transferred to the mesh.
2. *Interpolation*, where mesh properties are transferred to the particles.

Particle deposition

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

```
/*!
  @brief Deposit scalar particle quantities on the mesh.
  @details P is the particle type, Ret is the returned value of the particle member function, and must be a Real or a
  RealVect. MemFunc is a pointer to a member function of P. E.g., depositParticles<P, const RealVect&, &P::position>.
  @param[out] a_meshData             Mesh data. Must have exactly one compnent.
  @param[in]  a_realm                Realm where data is registered.
  @param[in]  a_phase                Phase where data is registered.
  @param[in]  a_particles            Particle container. Must be in "usable state" for deposition.
  @param[in]  a_depositionType       Specification of deposition kernel (e.g., CIC)
  @param[in]  a_coarseFineDeposition Specification of handling of coarse-fine boundaries.
  @param[in]  a_forceIrregNGP        Force NGP deposition in irregular cells or not.
*/
template <class P, typename Ret, Ret (P::*MemberFunc)() 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 = false);

/*!
  @brief Deposit scalar particle quantities on the mesh.
  @details P is the particle type, Ret is the returned value of the particle member function, and must be a Real or a
  const Real&. MemFunc is a pointer to a member function of P. E.g., depositParticles<P, const Real&, &P::weight>.
  @param[out] a_meshData   Mesh data. MUST have exactly one compnent.
  @param[in]  a_realm      Realm where data is registered.
  @param[in]  a_phase      Phase where data is registered.
  @param[in]  a_particles  Particle container. Must be in "usable state" for deposition.
*/
template <class P, typename Ret, Ret (P::*MemberFunc)() const>
void
depositParticles(EBAMRIVData&                a_meshData,
                 const std::string&          a_realm,
                 const phase::which_phase&   a_phase,
                 const ParticleContainer<P>& a_particles) const noexcept;
```

Here, the template parameter `P` is the particle type and the template parameter `MemberFunc` is a C++ pointer-to-member-function that returns `Ret`, which must either be a `Real` or a `RealVect`. In addition, the function will accept `const Real&` or `const RealVect&`. The pointer-to-member `MemberFunc` indicates the variable to be deposited on the mesh, and must have return type `Ret`. This function pointer does not need to return a member in the particle class, but it must be marked `const`.

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`). 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 then ensure that mass is never deposited into covered grid cells.

The input argument `a_depositionType` indicates the deposition method, while `a_coarseFineDeposition` deposition modifications near refinement boundaries. These are discussed below.

Base deposition

The base deposition scheme is specified by an enum `DepositionType` with valid values:

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

`chombo-discharge` supports all of the above methods, which can be combined with various types of modifications near refinement boundaries.

Coarse-fine deposition

The input argument `a_coarseFineDeposition` determines how deposition near the coarse-fine deposition is handled. Refinement boundaries introduce additional complications in the deposition scheme due to

1. Fine-grid particles whose deposition clouds hang over the refinement boundary and onto the coarse level.
2. Coarse-grid particles whose deposition clouds stick underneath the fine-level.

In addition, there can be complicated near physical boundaries, such as domain or embedded boundaries.

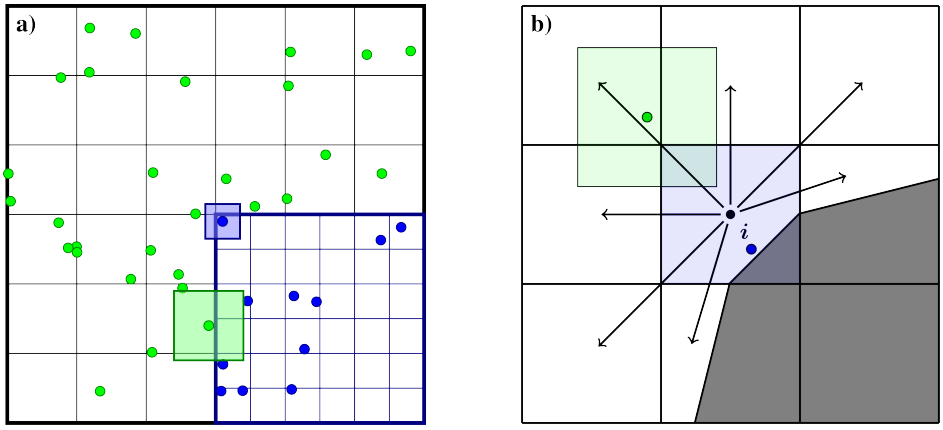

Fig. 3.4.1: Sketch of deposition schemes near refinement boundaries and cut-cells.

`chombo-discharge` support various ways of handling deposition across the refinement boundary. In all of these methods, the mass on the fine grid particles whose deposition clouds hang over the refinement boundaries is simply added to the coarse grid. The main modifications to the deposition scheme is performed for the coarse-grid particles that live around the refinement boundary (see Fig. 3.4.2). For the coarse-grid particles the following processes then occur:

The following coarse-fine deposition methods are currently supported:

- `CoarseFineDeposition::Interp` This method permits the coarse-grid particles to deposit into the region underneath the fine grid. The deposited mass that falls underneath the fine grid is then interpolated from the coarse grid to the fine grid. For example, see the indicated coarse-grid particle cloud in the left panel Fig. 3.4.1. While this particle has a width given by the coarse-grid cell size, it will deposit into the coarse grid cells underneath the fine grid. The mass that ends up in these cells is interpolated to the fine grid, which in this case will inject mass into two layers of fine-grid cells.
- `CoarseFineDeposition::Halo` This method extracts the coarse-grid particles that live on the refinement boundary and deposit them with their original width on both the coarse and fine levels. This is done by first

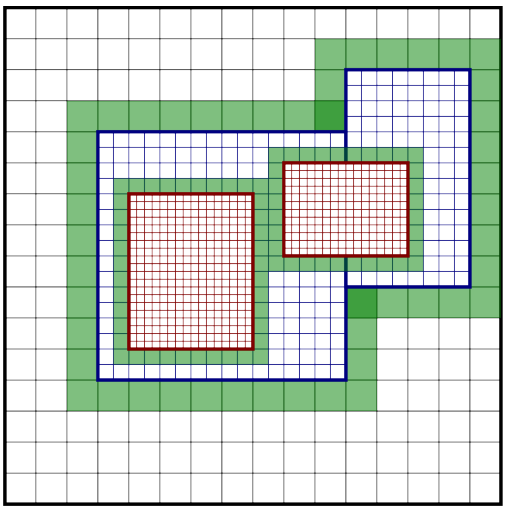

Fig. 3.4.2: Example regions containing coarse-grid particles that deposit with custom deposition rules.

depositing the particles on the coarse level, and then transferring them to the fine level and redepositing them there with the original particle width. Taking the left panel in Fig. 3.4.1 as an example, the green particle will then deposit into the coarse-grid cell as well as the first layer of fine-grid cells.

- `CoarseFineDeposition::HaloNGP` Similar to `CoarseFineDeposition::Halo` discussed above, this method also extracts the coarse-grid particles on the coarse side of the refinement boundary. However, rather than using the original deposition scheme, these particles are deposited with an NGP scheme.
- `CoarseFineDeposition::Transition` This is a method that was developed in order to minimize spurious gradients in the density across the EB. This method operates by extracting the coarse-grid particles that live around the refinement zone (within some radius), and *depositing them with the fine-grid particle width*.

Important: Most coarse-fine particle deposition schemes exhibit some artifacts around the refinement boundary, especially when the particle width exceeds the grid cell size (e.g., for TSC). The `CoarseFineDeposition::Transition` method is the one that we recommend, especially when used with CIC, as it eliminates spurious gradients across the refinement boundary.

Particle interpolation

To interpolate mesh data onto a particle property, the user can call the `AmrMesh` member functions

```
/*!
  @brief Interpolate mesh data onto a particle position.
  @details P is the particle type, Ret is the returned value of the particle member function, and must be a Real& or a
  RealVect&. MemFunc is a pointer to a member function of P. E.g., interpolate<P, Real&, &P::weight>.
  @param[inout] a_particles            Particles to be interpolated.
  @param[in]    a_realm                Realm where data is registered.
  @param[in]    a_phase                Phase where data is registered.
  @param[in]    a_meshScalarField      Scalar field on the mesh
  @param[in]    a_interpType           Interpolation type.
  @param[in]    a_forceIrregNGP        Force NGP interpolation in cut-cells.
*/
template <class P, class Ret, Ret (P::*MemberFunc)()>
void
interpolateParticles(ParticleContainer<P>&     a_particles,
                     const std::string&        a_realm,
                     const phase::which_phase& a_phase,
                     const EBAMRCellData&      a_meshScalarField,
```

(continues on next page)

(continued from previous page)

```
                    const DepositionType      a_interpType,
                    const bool                a_forceIrregNGP = false) 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, RealVect&, &MyParticleClass::acceleration>(...)
```

Note: 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 `SpaceDim` components.

Example

Assume that we have some particle class `KineticParticle` defined as

```
class KineticParticle : public GenericParticle<1,2>
{
public:
   inline
   Real& weight() {
      return this->real<0>();
   }

   inline
   RealVect& velocity() {
      return this->vect<0>();
   }

   inline
   RealVect& acceleration() {
      return this->vect<1>();
   }

   inline
   RealVect momentum() const {
      return this->weight() * this->velocity();
   }
};
```

To deposit the weight, velocity, and momentum on the grid we would call

```
RefCountedPtr<AmrMesh> amr;

amr->depositParticles<KineticParticle, const Real&,     &KineticParticle::mass >(...);
amr->depositParticles<KineticParticle, const RealVect&, &KineticParticle::velocity>(...);
amr->depositParticles<KineticParticle, const RealVect&, &KineticParticle::momentum>(...);
```

Likewise, to interpolate onto these fields we can call

```
RefCountedPtr<AmrMesh> amr;

amr->interpolateParticles<KineticParticle, Real&,     &KineticParticle::mass >(...);
amr->interpolateParticles<KineticParticle, RealVect&, &KineticParticle::velocity>(...);
```

3.4.13 Particle visualization

Note: Particle visualization is currently a work in progress with limited functionality.

Simple particle visualization can be performed by writing `H5Part` compatible files which can be read by VisIt. This is done through the function `writeH5Part` in the `DischargeIO` namespace, with the following signature:

```
/*!
  @brief Write a particle container to an H5Part file. Good for quick and dirty visualization of particles
  @details Use case is pretty straightforward but the user might need to cast particle types. E.g. call

  writeH5Part<M,N>(a_filename, (const ParticleContainer<GenericParticle<M,N>>&) a_particles, ...)

  Template substitution is not straightforward for this one.
  @param[in] a_filename  File name
  @param[in] a_particles Particles. Particle type must derive from GenericParticle<M, N>
  @param[in] a_realVars  Variable names for the M real variables
  @param[in] a_vectVars  Variable names for the N vector variables
  @param[in] a_shift     Particle position shift
  @param[in] a_shift     Time
*/
template <size_t M, size_t N>
void
writeH5Part(const std::string                                a_filename,
            const ParticleContainer<GenericParticle<M, N>>& a_particles,
            const std::vector<std::string>                   a_realVars = std::vector<std::string>(),
            const std::vector<std::string>                   a_vectVars = std::vector<std::string>(),
            const RealVect                                   a_shift    = RealVect::Zero,
            const Real                                       a_time     = 0.0) noexcept;
```

This routine permits particles to be written (in parallel, when using MPI) into a file readable by VisIt. The optional arguments `a_realVars` and `a_vectVars` permit the user to set the output variable names for the `M` scalar variables and the `N` vector variables. The argument `a_shift` will simply shift the particle positions in the output HDF5 file.

3.4.14 Superparticles

Often, merging or splitting of particles is required. In the most general case, users can simply interact directly with the particle list to modify the particles, which can be done either on a per-patch basis or within individual grid cells. In each case one starts with a list `List<P>` that needs to be modified.

`chombo-discharge` has rather elementary support for handling superparticles. Currently, we only support reinitialization of particles, or agglomeration of particles using kD-trees, as discussed below.

kD-trees

`chombo-discharge` has functionality for spatially partitioning particles using kD-trees, which can be used as a basis for particle merging and splitting. kD-trees operate by partitioning a set of input primitives into spatially coherent subsets. At each level in the tree recursion one chooses an axis for partitioning one subset into two new subsets, and the recursion continues until the partitioning is complete. Fig. 3.4.3 shows an example where a set of initial particles are partitioned using such a tree.

Tip: The source code for the kD-tree functionality is given in `$DISCHARGE_HOME/Source/Particle/CD_SuperParticles.H`.

The kD-tree partitioner requires a user-supplied criterion for particle partitioning. Only the partitioner `PartitionEqualWeight` is currently supported, and this partitioner will divide the original subset into two new subsets such that the particle weights in the two halves differs by at most one physical particle. This partitioner is imlemented as

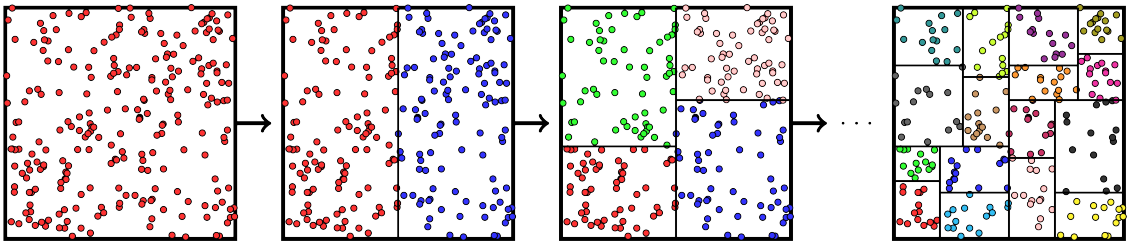

Fig. 3.4.3: Example of a kD-tree partitioning of particles in a single cell.

```
template <class P, Real& (P::*weight)(), const RealVect& (P::*position)() const>
typename KDNode<P>::Partitioner PartitionEqualWeight;
```

Here, `P` is the particle type, and this class *must* have function members `Real& P::weight()` and `const RealVect& P::position()` which return the particle weight and position.

Warning: `PartitionEqualWeight` will usually split particles to ensure that the weight in the two subsets are the same (thus creating new particles). In this case any other members in the particle type are copied over into the new particles.

The particles in each leaf of the kD-tree can then be merged into new particles. Since the weight in the nodes of the tree differ by at most one, the resulting computational particles also have weights that differ by at most one.

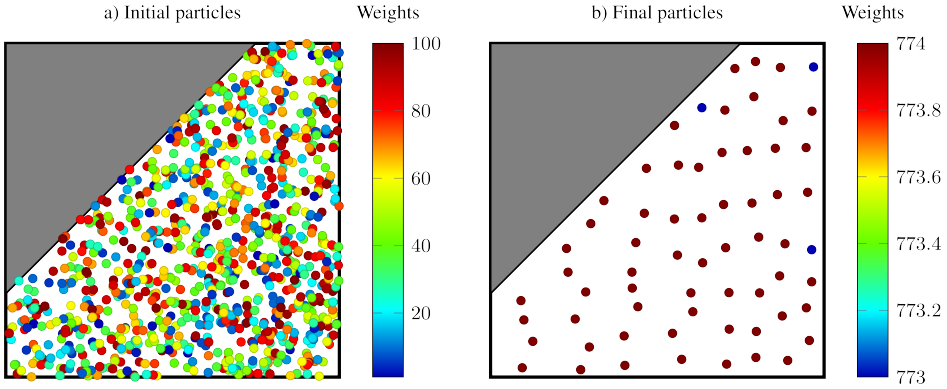

Fig. 3.4.4: kD-tree partitioning of particles into new particles whose weight differ by at most one. Left: Original particles with weights between 1 and 100. Right: Merged particles.

3.5 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*, i.e., 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.

3.5.1 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 load balance the grids for the fluid and particle algorithms 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 only difference lies primarily in the assignment of MPI ranks to grids, i.e., the load-balancing.

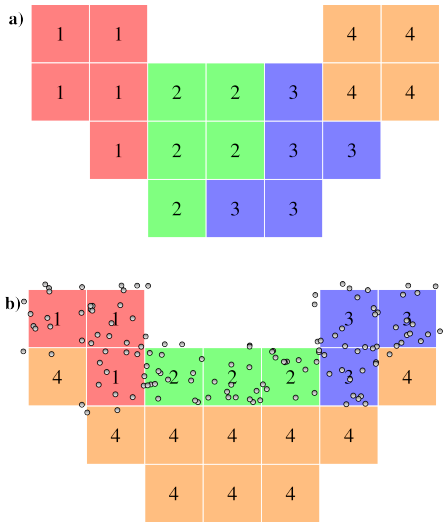

Fig. 3.5.1: Sketch of dual grid approach. Each rectangle represents a grid patch and the numbers show MPI ranks. a) Load balancing with the number of grid cells. b) Load balancing with the number of particles.

Fig. 3.5.1 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.

3.5.2 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. It is important, however, to keep track of what data is allocated where. Fortunately, *AmrMesh* will issue plenty of warnings if the user calls a function where the input arguments are realm-wise inconsistent.

3.6 Linear solvers

3.6.1 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 is defined on face centers and EB faces, while $a\left(\mathbf{x}\right)$ is defined on cell centers. 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.

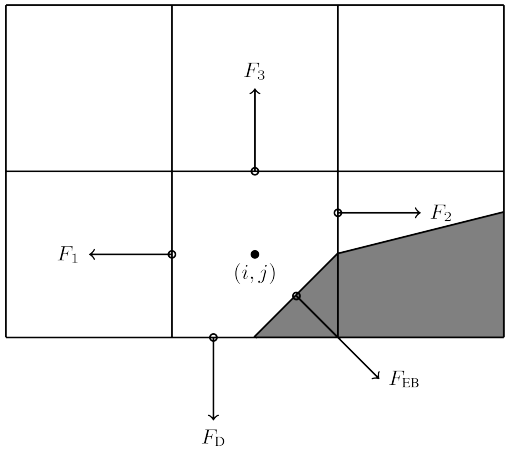

Fig. 3.6.1: Location of fluxes for finite volume discretization.

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

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

The other fluxes, such as F_2 requires interpolation of face-centered fluxes to the face centroids.

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. I.e., the fluxes F_{EB} and F_{D} are directly specified in Fig. 3.6.1.

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 where there is no EB the fluxes are face-centered and we therefore use finite differencing for obtaining a second order accurate approximation to the flux at the boundary. If the EB intersects the domain side, we interpolate face-centered fluxes to face centroids.

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 using 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. 3.6.2. If we can not find enough equations, several fallback options are in place to ensure that we obtain a sufficient number of equations.

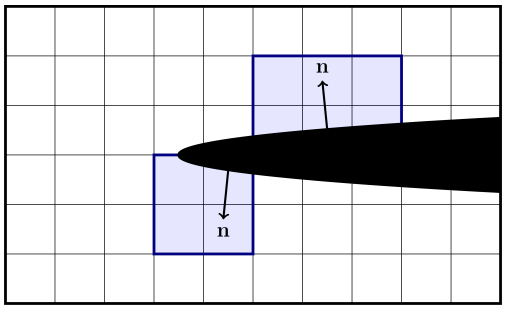

Fig. 3.6.2: Examples of neighborhoods (quadrant and half-space) used for gradient reconstruction on the EB.

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 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. 3.6.3 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.

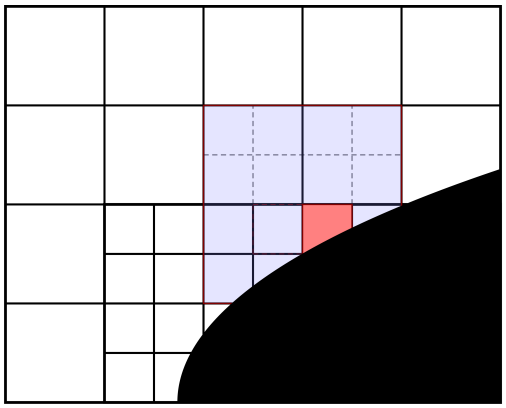

Fig. 3.6.3: Example of the region of a second order stencil for the Laplacian operator with second order gradient reconstruction on the embedded boundary.

Here, we rely on multigrid interpolation (see *Multigrid ghost cell interpolation*) to fill the required number of ghost cells.

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 to avoid forming an ill-conditioned system).

Multigrid ghost cell interpolation

With AMR, multigrid requires ghost cells on the refinement boundary. The interior stencils for the Helmholtz operator have a radius of one and thus only require a single layer of ghost cells (and no corner ghost cells). These ghost cells are filled using a finite-difference stencil, see Fig. 3.6.4.

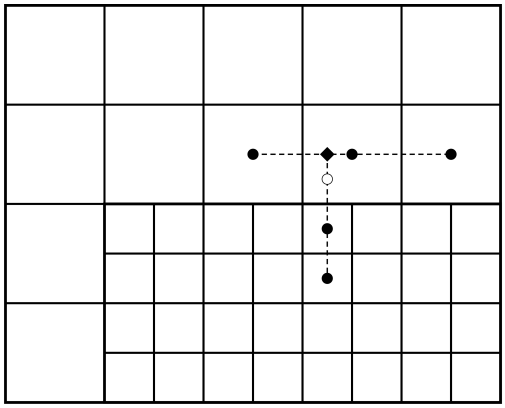

Fig. 3.6.4: Standard finite-difference stencil for ghost cell interpolation (open circle). We first interpolate the coarse-grid cells to the centerline (diamond). The coarse-grid interpolation is then used together with the fine-grid cells (filled circles) for interpolation to the ghost cell (open circle).

Embedded boundaries introduce many pathologies for multigrid:

1. Cut-cell stencils may have a large radius (see Fig. 3.6.3) 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. 3.6.5 shows a typical interpolation stencil for the stencil in Fig. 3.6.3. 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 requires at least 6 cells in 2D).

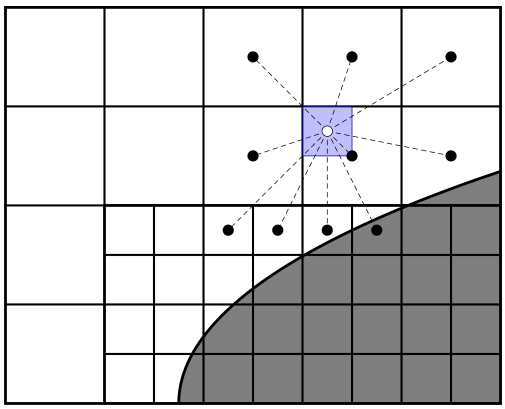

Fig. 3.6.5: Multigrid interpolation for refinement boundaries away from and close to an embedded boundary.

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.

Tip: Red-black Gauss-Seidel usually provide the best convergence rates. The multi-colored kernels are twice as expensive as red-black Gauss-Seidel relaxation in 2D, and four times as expensive in 3D, and then to only marginally improve convergence rates.

3.6.2 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 across the EB, e.g. for

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

where $b\left(\mathbf{x}\right)$ is only piecewise constant. This is the natural boundary conditions on a dielectric surface, for example.

Jump conditions

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

$$b_1\partial_{n_1}\Phi+b_2\partial_{n_2}\Phi=\sigma, \tag{3.6.1}$$

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. The jump factor is σ, and can be thought of as the surface charge density on the dielectric.

Discretization

To incorporate the jump condition in the Helmholtz discretization, we use a gradient reconstruction to obtain an approximation of Φ on the boundary, using Eq. 3.6.1. We then 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. 3.6.6) 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.

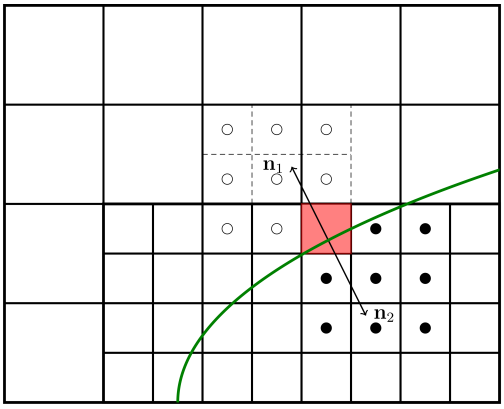

Fig. 3.6.6: Example of cells and stencils that are involved in discretizing the jump condition. Open and filled circles indicate cells in separate phases.

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).

3.6.3 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` uses its own elliptic operators, 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. Typically, smoothers tend to work sufficiently well but improved convergence rates can occasionally be achieved by using a conjugate gradient solver.

3.7 Verification and validation

We strive to include convergence testing (verification), and in some cases comparison with various types of experimental (validation). Below, we discuss our approach to spatial and temporal convergence testing.

3.7.1 Spatial convergence

Assume that we have some evolution problem which provides a solution on the mesh as $\phi_{\mathbf{i}}^k\left(\Delta x, \Delta t\right)$ where Δx is a uniform grid resolution and Δt is the time step used for evolving the state from $t = 0$ to $t = k\Delta t$.

To estimate the spatial order of convergence for the discretization we can use Richardson extrapolation to estimate the error, using the results from a finer grid resolution as the "exact" solution. We solve the problem on grids with resolutions Δx_c and a finer resolution Δx_f and estimate the error in the coarse-grid solution as

$$E_{\mathbf{i}}^k\left(\Delta x_c\right) = \phi_{\mathbf{i}}^k\left(\Delta x_c, \Delta t\right) - \left\{\mathcal{A}_{\Delta x_f \rightarrow \Delta x_c}\left[\phi^k\left(\Delta x_f, \Delta t\right)\right]\right\}_{\mathbf{i}}.$$

where $\mathcal{A}_{\Delta x_f \rightarrow \Delta x_c}$ is an averaging operator which coarsens the solution from the fine grid (Δx_f) to the coarse grid (Δx_c). The error norms are computed from $E_{\mathbf{i}}^k\left(\Delta x_c; \Delta x_f\right)$. Specifically:

$$\begin{aligned}
L_\infty\left[\phi^k\left(\Delta x_c, \Delta t\right)\right] &= \max\left|E_{\mathbf{i}}^k\left(\Delta x_c\right)\right|, \\
L_1\left[\phi^k\left(\Delta x_c, \Delta t\right)\right] &= \frac{1}{\sum_{\mathbf{i}}}\sum_{\mathbf{i}}\left|E_{\mathbf{i}}^k\left(\Delta x_c\right)\right|, \\
L_2\left[\phi^k\left(\Delta x_c, \Delta t\right)\right] &= \sqrt{\frac{1}{\sum_{\mathbf{i}}}\sum_{\mathbf{i}}\left|E_{\mathbf{i}}^k\left(\Delta x_c\right)\right|^2}.
\end{aligned}$$

where the sums run over the grid points.

Tip: If one has an exact solution $\phi_e(\mathbf{x}, T)$ available, one can replace $\phi_{\mathbf{i}}^k\left(\Delta x, \Delta t\right)$ by $\phi_e\left(\mathbf{i}\Delta x, k\Delta t\right)$.

3.7.2 Temporal convergence

For temporal convergence we compute the errors in the same way as for the spatial convergence, replacing Δt and Δx as fixed parameters. The solution error is computed as

$$E_{\mathbf{i}}^k\left(\Delta t_c\right) = \phi_{\mathbf{i}}^k\left(\Delta x, \Delta t_c\right) - \phi_{\mathbf{i}}^{k'}\left(\Delta x, \Delta t_f\right),$$

where $k\Delta t_c = T$ and $k'\Delta t_f = T$. Temporal integration errors are computed as

$$\begin{aligned}
L_\infty\left[\phi^k\left(\Delta x, \Delta t_c\right)\right] &= \max\left|E_{\mathbf{i}}^k\left(\Delta t_c\right)\right|, \\
L_1\left[\phi^k\left(\Delta x, \Delta t_c\right)\right] &= \frac{1}{\sum_{\mathbf{i}}}\sum_{\mathbf{i}}\left|E_{\mathbf{i}}^k\left(\Delta t_c\right)\right|, \\
L_2\left[\phi^k\left(\Delta x, \Delta t_c\right)\right] &= \sqrt{\frac{1}{\sum_{\mathbf{i}}}\sum_{\mathbf{i}}\left|E_{\mathbf{i}}^k\left(\Delta t_c\right)\right|^2}.
\end{aligned}$$

Tip: If an exact solution as available, one can replace $\phi_{\mathbf{i}}^{k'}\left(\Delta x, \Delta t_f\right)$ by $\phi_e\left(\mathbf{i}\Delta x, k'\Delta t_f\right)$.

CHAPTER

FOUR

SOLVERS

4.1 Convection-Diffusion-Reaction

Here, we discuss the discretization of the equation

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{v}\phi - D\nabla\phi) = S. \tag{4.1.1}$$

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 and D is the (isotropic) diffusion coefficient.

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.

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

4.1.1 CdrSolver

The `CdrSolver` class contains the interface for solving advection-diffusion-reaction problems. `CdrSolver` is an abstract class and does not contain any specific advective or diffusive discretization (these are added by implementation classes). However, `CdrSolver` supplies a useful interface that includes implementations of regrid functions and I/O capabilities, such that only the advective and diffusive discretizations need to be provided.

The implementation layers of the CDR capabilities consist of the following:

1. *CdrMultigrid*, which inherits from `CdrSolver` and adds a second order accurate discretization for the diffusion operator. This includes both explicit and implicit discretizations, where the implicit discretization uses geometric multigrid to resolve the diffusion problem.
2. *CdrCTU* and *CdrGodunov* which inherit from `CdrMultigrid` and add a second order accurate spatial discretization for the advection operator.

Currently, we mostly use the *CdrCTU* class which contains a second order accurate discretization with slope limiters. *CdrGodunov* is a similar operator, but the advection code for this is distributed by the `Chombo` team. The C++ API for these classes can be obtained from https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classCdrSolver.html and references therein.

Explicit advance methods for Eq. 4.1.1 only require an approximation to the right-hand side of the equation. These approximations are encapsulated by the following functions:

Listing 4.1.1: List of functions responsible for calculating approximations to divergence operators.

```
/*!
  @brief Compute div(J) explicitly, where J = nV - D*grad(n)
  @param[out] a_divJ             Divergence term, i.e. finite volume approximation to
  @param[in]  a_phi              Cell-centered state
  @param[in]  a_extrapDt         Extrapolation in time, i.e. shifting of div(J) towards e.g. half time step. Only affects the␣
↪advective term.
  @param[in]  a_conservativeOnly If true, we compute div(J) = 1/dx*sum(fluxes), which does not involve redistribution.
  @param[in]  a_ebFlux           If true, the embedded boundary flux will injected and included in div(J)
  @param[in]  a_domainFlux       If true, the domain flux will injected and included in div(J)
  @note a_phi is non-const because ghost cells will be re-filled
*/
virtual void
computeDivJ(EBAMRCellData& a_divJ,
            EBAMRCellData& a_phi,
            const Real     a_extrapDt,
            const bool     a_conservativeOnly,
            const bool     a_ebFlux,
            const bool     a_domainFlux) = 0;

/*!
  @brief Compute div(v*phi) explicitly
  @param[out] a_divF             Divergence term, i.e. finite volume approximation to Div(v*phi), including redistribution␣
↪magic.
  @param[in]  a_phi              Cell-centered state
  @param[in]  a_extrapDt         Extrapolation in time, i.e. shifting of div(F) towards e.g. half time step. Only affects the␣
↪advective term.
  @param[in]  a_conservativeOnly If true, we compute div(F)=1/dx*sum(fluxes), which does not involve redistribution.
  @param[in]  a_domainBc         How to set domain fluxes
  @param[in]  a_ebFlux           If true, the embedded boundary flux will injected be included in div(F)
  @param[in]  a_domainFlux       If true, the domain flux will injected and included in div(F)
  @note a_phi is non-const because ghost cells will be interpolated in this routine. Valid data in a_phi is not touched.
*/
virtual void
computeDivF(EBAMRCellData& a_divF,
            EBAMRCellData& a_phi,
            const Real     a_extrapDt,
            const bool     a_conservativeOnly,
            const bool     a_ebFlux,
            const bool     a_domainFlux) = 0;

/*!
  @brief Compute div(D*grad(phi)) explicitly
  @param[out] a_divF             Divergence term, i.e. finite volume approximation to Div(D*Grad(phi)).
  @param[in]  a_phi              Cell-centered state
  @param[in]  a_domainBc         Flag for setting domain boundary conditions
  @param[in]  a_conservativeOnly If true, we compute div(D) = 1/dx*sum(fluxes), which does not involve redistribution.
  @param[in]  a_useEbFlux        If true, the embedded boundary flux will be injected and included in div(D)
  @param[in]  a_domainFlux       If true, the domain flux will injected and included in div(D)
  @note a_phi is non-const because ghost cells will be interpolated in this routine. Valid data in a_phi is not touched.
*/
virtual void
computeDivD(EBAMRCellData& a_divD,
            EBAMRCellData& a_phi,
            const bool     a_conservativeOnly,
            const bool     a_ebFlux,
            const bool     a_domainFlux) = 0;
```

These functions are not implemented in `CdrSolver`, but in subclasses. In the current structure, *CdrMultigrid* supplies the diffusive discretization and *CdrCTU* supplies the advective discretization.

The implicit advance methods for `CdrSolver` are encapsulated by the following functions:

Listing 4.1.2: List of current implicit diffusion advance functions.

```
/*!
  @brief Implicit diffusion Euler advance with source term.
  @param[inout] a_newPhi Solution at time t + dt
  @param[in]    a_oldPhi Solution at time t
  @param[in]    a_source Source term.
```

(continues on next page)

(continued from previous page)

```
  @param[in]    a_dt     Time step
  @note For purely implicit Euler the source term should be centered at t+dt (otherwise it's an implicit-explicit method)
  @details This solves the implicit diffusion equation equation a_newPhi - a_oldPhi = dt*Laplacian(a_newPhi) + dt*a_source.
*/
virtual void
advanceEuler(EBAMRCellData&       a_newPhi,
             const EBAMRCellData& a_oldPhi,
             const EBAMRCellData& a_source,
             const Real           a_dt) = 0;

/*!
  @brief Implicit diffusion Crank-Nicholson advance with source term.
  @param[inout] a_newPhi Solution at time t + dt
  @param[in]    a_oldPhi Solution at time t
  @param[in]    a_source Source term.
  @param[in]    a_dt     Time step
*/
virtual void
advanceCrankNicholson(EBAMRCellData&       a_newPhi,
                      const EBAMRCellData& a_oldPhi,
                      const EBAMRCellData& a_source,
                      const Real           a_dt) = 0;
```

4.1.2 Discretization details

Explicit divergences and redistribution

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

$$\phi^{k+1} = \phi^k - \frac{\Delta t}{\kappa \Delta x^{\mathrm{DIM}}} \int_V \nabla \cdot \mathbf{G} dV, \tag{4.1.2}$$

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.

However, taking $\left[\nabla \cdot \mathbf{G}^k\right]$ 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. 4.1.2. 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 $D_{\mathbf{i}}^{nc}$

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

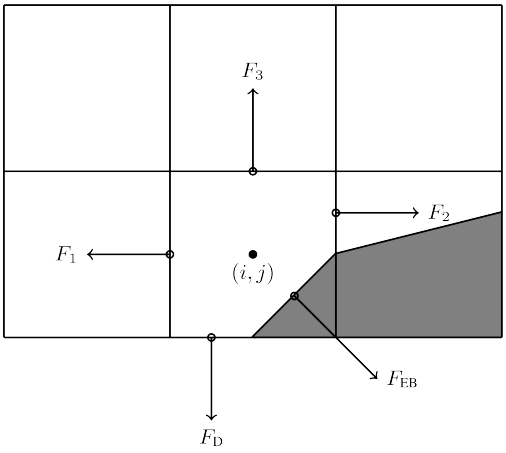

Fig. 4.1.1: Location of centroid fluxes for cut cells.

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 $\delta M_{\mathbf{i}} = \kappa_{\mathbf{i}} \left(1-\kappa_{\mathbf{i}}\right) \left(D_{\mathbf{i}}^{c} - D_{\mathbf{i}}^{nc}\right)$. 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

```
/*!
  @brief Compute div(G) where G is a general face-centered flux on face centers and EB centers. This can involve mass␣
↪redistribution.
  @param[in]    a_divG             div(G) or kappa*div(G).
  @param[inout] a_G                Vector field which contains face-centered fluxes on input. Contains face-centroid fluxes on␣
↪output.
  @param[in]    a_ebFlux           Flux on the EB centroids
  @param[in]    a_conservativeOnly If true, we compute div(G)=1/dx*sum(fluxes), which does not involve redistribution.
*/
virtual void
computeDivG(EBAMRCellData& a_divG, EBAMRFluxData& a_G, const EBAMRIVData& a_ebFlux, const bool a_conservativeOnly);
```

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_ebFlux` is the flux through 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.

Caution: Mass redistribution has the effect of not being monotone and thus not TVD, and the discretization order is formally $\mathcal{O}(\Delta x)$.

Explicit advection

Scalar advection updates follows the computation of the explicit divergence discussed in *Explicit divergences and redistribution*. 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 (`computeDivF`) was given in Listing 4.1.1.

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. In the simplest case, these fluxes are simply calculated using an upwind rule, potentially extrapolated with slope-limiters from the cell-center. In more complex cases, we use both the normal and transverse slopes in a cell. The argument `a_extrapDt` is the time step size. It is not needed in a method-of-lines context, but it is used in e.g. *CdrCTU* for computing transverse derivatives in order to expand the stability region of the discretization (i.e., permit larger CFL numbers).

For example, in order to perform an advective advance with the explicit Euler rule over a time step Δt, one would perform the following:

```
CdrSolver* solver;

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

Explicit diffusion

Explicit diffusion is performed in much the same way as explicit 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 (`computeDivD`) was given in Listing 4.1.1.

For example, to use an explicit Euler update of a diffusion-only problem, we increment in the same way as for explicit advection:

```
CdrSolver* solver;

EBAMRCellData& phi  = solver->getPhi();            // Cell-centered state
EBAMRCellData& divD = solver->getScratch();        // Scratch storage in solver
solver->computeDivD(divD, phi, false, true, true); // Computes divD including BCs
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 and diffusion, but aggregated as a function `computeDivJ`. The function signature is given in Listing 4.1.1.

For example, in order to perform an advection-diffusion advance over a time step Δt, one would perform the following:

```
CdrSolver* solver;

EBAMRCellData& phi  = solver->getPhi();                 // Cell-centered state
EBAMRCellData& divJ = solver->getScratch();             // Scratch storage in solver
solver->computeDivJ(divJ, phi, 0.0, false, true, true); // Computes divJ including BCs
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(...);     // Computes divF = div(n*phi)
DataOps:incr(phi, divF, -dt); // makes phi -> phi - dt*divF

solver->computeDivD(...);     // 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 then be performed twice.

Implicit diffusion

Usage of implicit diffusion can occasionally be necessary, especially if the diffusive time step leads to numerical stiffness and severe time step limitations. The convection-diffusion-reaction solvers support two basic implicit diffusion solves which were given in *List of current implicit diffusion advance functions.*.

As an example, perform a split step Godunov method with the Euler rules for explicit advection and implicit diffusion is done as follows:

```
// Compute phi = phi - dt*div(F)
solver->computeDivF(divF, phi, ...);
DataOps:incr(phi, divF, -dt);

// Implicit diffusion advance over a time step dt.
DataOps::copy(phiOld, phi);
solver->advanceEuler(phi, phiOld, dt);
```

4.1.3 CdrMultigrid

`CdrMultigrid` 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, `CdrMultigrid` adds two implicit time-integrators, an implicit Euler method and a Crank-Nicholson method.

The `CdrMultigrid` layer uses the Helmholtz discretization discussed in *Helmholtz equation*. It implements the pure functions required by *CdrSolver* but introduces a new pure function

```
/*!
  @brief Advection-only extrapolation to faces
  @param[out] a_facePhi Face-centered states
  @param[in] a_phi Cell-centered states
  @param[in] a_extrapDt Extrapolating time step.
*/
virtual void
advectToFaces(EBAMRFluxData& a_facePhi, const EBAMRCellData& a_phi, const Real a_extrapDt) override = 0;
```

The faces states defined by the above function are used when forming a finite-volume approximation to the divergence operators. In the simplest case, the face-centered values of ϕ are formed by an upwind approximation of the cell-centered states.

4.1.4 CdrCTU

`CdrCTU` is an implementation class that uses the corner transport upwind (CTU) discretization. The CTU discretization uses information that propagates over corners of grid cells when calculating the face states. It can combine this with use various limiters:

- No limiter.
- Minmod.
- Superbee.
- Monotonized central differences.

In addition, `CdrCTU` can turn off the transverse terms in which case the discretization reduces to the donor cell method where only normal slopes are used. A standard CFL condition will apply in this case.

Our motivation for using the CTU discretization lies in the time step selection for the CTU and donor-cell methods, see *Time step limitation*. Typically, we want to achieve a dimensionally independent time step that is the same in 1D, 2D, and 3D, but without directional splitting.

Face extrapolation

The finite volume discretization uses an upstream-centered Taylor expansion that extrapolates the cell-centered term to half-edges and half-steps:

$$\phi_{i+1/2,j,}^{n+1/2} = \phi_{i,j,k}^{n} + \frac{\Delta x}{2}\frac{\partial \phi}{\partial x} + \frac{\Delta t}{2}\frac{\partial \phi}{\partial t} + \mathcal{O}\left(\Delta t^2\right) + \mathcal{O}\left(\Delta t \Delta x\right)$$

Note that the truncation order is $\Delta t^2 + \Delta x \Delta t$ where the latter term is due to the cross-derivative $\frac{\partial^2 \phi}{\partial t \partial x}$. The resulting expression in 2D for a velocity field $\mathbf{v} = (u, v)$ is

$$\phi_{i\pm 1/2,j}^{n+1/2,+} = \phi_{i,j}^{n} \pm \frac{1}{2}\min\left[1, 1 \mp \frac{\Delta t}{\Delta x}u_{i,j}^{n}\right](\Delta^x \phi)_{i,j}^{n} - \frac{\Delta t}{2\Delta x}v_{i,j}^{n}(\Delta^y \phi)_{i,j}^{n},$$

Here, Δ^x are the regular (normal) slopes whereas Δ^y are the transverse slopes. The transverse slopes are given by

$$(\Delta^y \phi)_{i,j}^{n} = \begin{cases} \phi_{i,j+1}^{n} - \phi_{i,j}^{n}, & v_{i,j}^{n} < 0 \\ \phi_{i,j}^{n} - \phi_{i,j-1}^{n}, & v_{i,j}^{n} > 0 \end{cases}$$

Slopes

For the normal slopes, the user can choose between the minmod, superbee, and monotonized central difference (MC) slopes. Let $\Delta_l = \phi^n_{i,j} - \phi^n_{i-1,j}$ and $\Delta_r = \phi^n_{i+1,j} - \phi^n_{i,j}$. The slopes are given by:

$$\text{minmod: } (\Delta^x\phi)^n_{i,j} = \begin{cases} \Delta_l & |\Delta_l| < |\Delta_r| \text{ and } \Delta_l\Delta_r > 0 \\ \Delta_r & |\Delta_l| > |\Delta_r| \text{ and } \Delta_l\Delta_r > 0 \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{MC: } (\Delta^x\phi)^n_{i,j} = \text{sgn}\left(\Delta_l + \Delta_r\right) \min\left(\left|\frac{\Delta_l + \Delta_r}{2}\right|, 2\left|\Delta_l\right|, 2\left|\Delta_r\right|\right),$$

$$\text{superbee: } (\Delta^x\phi)^n_{i,j} = \begin{cases} \Delta_1 & |\Delta_1| > |\Delta_2| \text{ and } \Delta_1\Delta_2 > 0 \\ \Delta_2 & |\Delta_1| < |\Delta_2| \text{ and } \Delta_1\Delta_2 > 0 \\ 0 & \text{otherwise,} \end{cases}$$

where for the superbee slope we have $\Delta_1 = \text{minmod}\left(\Delta_l, 2\Delta_r\right)$ and $\Delta_2 = \text{minmod}\left(\Delta_r, 2\Delta_l\right)$.

Note: When using transverse slopes, monotonicity is not guaranteed for the CTU discretization. If transverse slopes are turned off, however, the scheme is guaranteed to be monotone.

Time step limitation

The stability region for the donor-cell and corner transport upwind methods are:

$$\text{Donor-cell} : \Delta t \leq \frac{\Delta x}{|v_x| + |v_y| + |v_z|}$$

$$\text{CTU} : \Delta t \leq \frac{\Delta x}{\max\left(|v_x|, |v_y|, |v_z|\right)}$$

Note that when the flow is diagonal to the grid, i.e., $|v_x| = |v_y| = |v_z|$, the CTU can use a time step that is three times larger than for the donor-cell method.

Class options

When running the `CdrCTU` solver the user can adjust the advective algorithm by turning on/off slope limiters and the transverse term through the class options: Several options are available for adjusting the behavior of the discretization, as given below:

```
# ================================================================================================
# CdrCTU solver settings.
# ================================================================================================
CdrCTU.bc.x.lo                  = wall                      ## 'data', 'function', 'wall', 'outflow', or 'solver'
CdrCTU.bc.x.hi                  = wall                      ## 'data', 'function', 'wall', 'outflow', or 'solver'
CdrCTU.bc.y.lo                  = wall                      ## 'data', 'function', 'wall', 'outflow', or 'solver'
CdrCTU.bc.y.hi                  = wall                      ## 'data', 'function', 'wall', 'outflow', or 'solver'
CdrCTU.bc.z.lo                  = wall                      ## 'data', 'function', 'wall', 'outflow', or 'solver'
CdrCTU.bc.z.hi                  = wall                      ## 'data', 'function', 'wall', 'outflow', or 'solver'
CdrCTU.slope_limiter            = minmod                    ## Slope limiter. 'none', 'minmod', 'mc', or 'superbee'
CdrCTU.use_ctu                  = true                      ## If true, use CTU. Otherwise it's DTU.
CdrCTU.plt_vars                 = phi vel src dco ebflux    ## Plot variables. Options are 'phi', 'vel', 'dco', 'src'
CdrCTU.plot_mode                = density                   ## Plot densities 'density' or particle numbers ('numbers')
CdrCTU.blend_conservation       = true                      ## Turn on/off blending with nonconservative divergenceo
CdrCTU.which_redistribution     = volume                    ## Redistribution type. 'volume', 'mass', or 'none' (turned off)
CdrCTU.use_regrid_slopes        = true                      ## Turn on/off slopes when regridding
```

(continues on next page)

(continued from previous page)

```
CdrCTU.gmg_verbosity            = -1                    ## GMG verbosity
CdrCTU.gmg_pre_smooth           = 12                    ## Number of relaxations in GMG downsweep
CdrCTU.gmg_post_smooth          = 12                    ## Number of relaxations in upsweep
CdrCTU.gmg_bott_smooth          = 12                    ## NUmber of relaxations before dropping to bottom solver
CdrCTU.gmg_min_iter             = 5                     ## Minimum number of iterations
CdrCTU.gmg_max_iter             = 32                    ## Maximum number of iterations
CdrCTU.gmg_exit_tol             = 1.E-10                ## Residue tolerance
CdrCTU.gmg_exit_hang            = 0.2                   ## Solver hang
CdrCTU.gmg_min_cells            = 16                    ## Bottom drop
CdrCTU.gmg_bottom_solver        = bicgstab              ## Bottom solver type. Valid options are 'simple' and 'bicgstab'
CdrCTU.gmg_cycle                = vcycle                ## Cycle type. Only 'vcycle' supported for now
CdrCTU.gmg_smoother             = red_black             ## Relaxation type. 'jacobi', 'multi_color', or 'red_black'
```

We do not discuss all options here, but note the following:

- `CdrCTU.slope_limiter`, which must be *none*, *minmod*, *mc*, or *superbee*.
- `CdrCTU.use_ctu`, which must be *true* or *false*. If setting this to false, transverse terms are turned off the `CdrCTU` will use the donor-cell scheme and time step restriction.
- All options that begin with `gmg_` indicate control over how the geometric multigrid algorithm operates, e.g., number of smoothings on each level or the bottom solver type.
- `CdrCTU.plt_vars` indicate which variables are added to plot files.

4.1.5 CdrGodunov

`CdrGodunov` inherits from `CdrMultigrid` and adds advection code for Godunov methods. This class borrows from `Chombo` internals (specifically, `EBLevelAdvectIntegrator`) and can do second-order advection with time-extrapolation.

`CdrGodunov` supplies (almost) the same discretization as `CdrCTU` with the exception that the underlying discretization can also be used for the incompressible Navier-Stokes equation. However, it only supports the monotonized central difference limiter.

Caution: `CdrGodunov` will be removed from future versions of `chombo-discharge`.

4.1.6 Using CdrSolver

Tip: For a complete example, see the *Advection-diffusion model*.

Filling the solver

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

```
/*!
  @brief Get the cell-centered phi
  @return m_phi
*/
virtual EBAMRCellData&
getPhi();

/*!
  @brief Get the source term
  @return m_source
*/
virtual EBAMRCellData&
```

(continues on next page)

(continued from previous page)

```
getSource();

/*!
  @brief Get the cell-centered velocity
  @return m_cellVelocity
*/
virtual EBAMRCellData&
getCellCenteredVelocity();

/*!
  @brief Get the face-centered velocities
  @return m_faceVelocity
*/
virtual EBAMRFluxData&
getFaceCenteredVelocity();

/*!
  @brief Get the eb-centered velocities
  @return m_ebVelocity
*/
virtual EBAMRIVData&
getEbCenteredVelocity();

/*!
  @brief Get the cell-centered diffusion coefficient
*/
virtual EBAMRCellData&
getCellCenteredDiffusionCoefficient();

/*!
  @brief Get the face-centered diffusion coefficient
  @return m_faceCentereDiffusionCoefficient
*/
virtual EBAMRFluxData&
getFaceCenteredDiffusionCoefficient();

/*!
  @brief Get the EB-centered diffusion coefficient
  @return m_ebCentereDiffusionCoefficient
*/
virtual EBAMRIVData&
getEbCenteredDiffusionCoefficient();

/*!
  @brief Get the eb flux data holder
  @return m_ebFlux
*/
virtual EBAMRIVData&
getEbFlux();

/*!
  @brief Get the domain flux data holder
  @return m_domainFlux
*/
virtual EBAMRIFData&
getDomainFlux();
```

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, so there is normally no need to fill these directly.

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. One can use `DataOps` functions to fill the data directly using a C++ lambda:

```
/*!
  @brief Polymorphic set value function. Assumes that a_lhs has SpaceDim components and sets all those components from the␣
↪input function.
  @param[out] a_lhs      Data to set
  @param[in]  a_function Function to use for setting the value.
```

(continues on next page)

(continued from previous page)

```
  @param[in]  a_probLo   Lower-left corner of physical domain.
  @param[in]  a_dx       Grid resolutions
  @param[in]  a_comp     Component to set
  @note Uses a VofIterator everywhere to this might be slow!
*/
static void
setValue(EBAMRCellData&                                 a_lhs,
         const std::function<RealVect(const RealVect)>& a_function,
         const RealVect                                 a_probLo,
         const Vector<Real>&                            a_dx);
```

This would typically look something like this:

```
EBAMRCellData& veloCell = m_solver->getCellCenteredVelocity();

auto veloFunc = [=](const RealVect x) -> RealVect
{
  return RealVect::Unit;
};

DataOps::setValue(veloCell, veloFunc, ...);
```

The same procedure goes for the source terms, diffusion coefficients, boundary conditions and so on.

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. For example, for the `CdrCTU` implementation the following variables are available:

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

4.1.7 CdrSpecies

The `CdrSpecies` class is a supporting class that passes information and initial conditions into `CdrSolver` instances. `CdrSpecies` specifies whether or not the advection-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). However, it does not contain any discretization.

Note: Click here for the `CdrSpecies` C++ API.

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

Tip: It is also possible to use computational particles as an initial condition in `CdrSpecies`. In this case you need to fill `m_initialParticles`, and these are then deposited with a nearest-grid-point scheme when instantiating the

solver. See *Particles* for further details.

4.1.8 Example application(s)

Example applications that use the `CdrSolver` are:

- *Advection-diffusion model*.
- *CDR plasma model*.

4.2 Electrostatic solver

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 subclass is *FieldSolverGMG*, which supplies a finite-volume discretization that is solved with geometric multigrid.

Tip: See the FieldSolver C++ API for the complete interface.

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 away from 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 one side of the boundary and let σ be the free parameter. When using `FieldSolverGMG` (see *FieldSolverGMG*), 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. This usually involves several steps, such as instantiating the solver and providing proper references to *AmrMesh* and the computational geometry. In addition, the user must set up a voltage curve and associated boundary conditions.

Tip: A complete example of setting up electrostatic problems and solving them is given in *Electrostatics model*.

4.2.3 Setting the voltage

The function signature for setting the voltage on various parts is

```
/*!
  @brief Set potential dependence in time.
  @param[in] a_voltage Function pointer which sets the voltage travel curve.
  @details If you want something more complex, the voltage can be set individually for each electrode using␣
↪setElectrodeDirichletFunction.
*/
virtual void
setVoltage(std::function<Real(const Real a_time)> a_voltage);
```

This allows setting a time-dependent voltage on electrodes and domain boundaries. As shown above, one can easily use `std::function<Real(const Real)>` or lambdas to set the voltage. It is relatively straightforward to turn tabulated data (see *LookupTable1D*) into lambda functions. A simple example that returns a linearly varying voltage is given below:

```
FieldSolver* fieldSolver;

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

fieldSolver->setVoltage(myVoltage);
```

4.2.4 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

```
FieldSolverGMG.bc.x.low  = dirichlet_custom
FieldSolverGMG.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:

```
FieldSolverGMG.bc.x.low  = neumann   0.0
FieldSolverGMG.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 f = [&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. For Dirichlet boundary conditions the solver will always multiply the provided input function by the voltage waveform. That is, 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

```
FieldSolverGMG.bc.y.low  = dirichlet 0.0
FieldSolverGMG.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 the live voltage. Specifically, on the upper y-plane this specification will generate a potential boundary condition function of the type

```
auto func = [](const RealVect a_pos, const Real a_time) {return 1.0};
dirichletFraction = 1.0;

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

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 `dirichlet <float>` and set a different multiplier on the domain edge (face). As an example. by specifying `bc.y.high = dirichlet 1.234` in the input script AND setting the multiplier on the wall as follows:

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

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

we end up with 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 $\partial_n \Phi$.

4.2.5 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, `FieldSolverGMG` 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, going through all electrodes and setting the electrode to be spatially and temporally varying. The member function that does this is

```
/*!
  @brief Set embedded boundary Dirichlet function on a specific electrode.
  @param[in] a_electrode electrode index. Follows the same order as ComputationalGeometry.
  @param[in] a_function  Voltage on the electrode.
*/
virtual void
setElectrodeDirichletFunction(const int a_electrode, const ElectrostaticEbBc::BcFunction& a_function);
```

Here, the type `ElectrostaticEbBc::BcFunction` is just an alias of `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 principle, one can then also set spatially varying voltages along an electrode.

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);
```

In summary, 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`, as discussed above).

Dielectrics

On dielectrics, we enforce the jump boundary condition directly, see *Jump conditions*.

4.2.6 Calling the solve function

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

```
/*!
  @brief Solves Poisson equation onto a_phi using a_rho and a_sigma as right-hand sides.
  @param[inout] a_potential Potential
  @param[in]    a_rho Space charge density
  @param[in]    a_sigma Surface charge density.
  @param[in]    a_zeroPhi Set a_potential to zero first.
  @return True if we found a solution and false otherwise.
  @note a_sigma must be defined on the gas phase.
*/
virtual bool
solve(MFAMRCellData& a_phi, const MFAMRCellData& a_rho, const EBAMRIVData& a_sigma, const bool a_zerophi = false) = 0;
```

where `a_phi` is the resulting potential that was computing with the space charge density `a_rho`, and surface charge density `a_sigma`.

4.2.7 FieldSolverGMG

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

The discretization used by `FieldSolverGMG` is described in *Linear solvers*. The underlying solver type is a Helmholtz solver, but `FieldSolverGMG` considers only the Laplacian term. For further details on the spatial discretization, see *Linear solvers*.

Solver configuration

`FieldSolverGMG` 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

Listing 4.2.1: Input options for the `FieldSolverGMG` class. Runtime adjustable options are highlighted.

```
# ================================================================================================
# FieldSolverGMG class options
# ================================================================================================
FieldSolverGMG.verbosity             = -1               ## Class verbosity
FieldSolverGMG.jump_bc               = natural          ## Jump BC type ('natural' or 'saturation_charge')
FieldSolverGMG.bc.x.lo               = dirichlet 0.0    ## Bc type (see docs)
FieldSolverGMG.bc.x.hi               = dirichlet 0.0    ## Bc type (see docs)
FieldSolverGMG.bc.y.lo               = dirichlet 0.0    ## Bc type (see docs)
FieldSolverGMG.bc.y.hi               = dirichlet 0.0    ## Bc type (see docs)
FieldSolverGMG.bc.z.lo               = dirichlet 0.0    ## Bc type (see docs)
FieldSolverGMG.bc.z.hi               = dirichlet 0.0    ## Bc type (see docs)
FieldSolverGMG.plt_vars              = phi rho E        ## Plot variables: 'phi', 'rho', 'E', 'res', 'perm', 'sigma', 'Esol'
FieldSolverGMG.use_regrid_slopes     = true             ## Use slopes when regridding or not
FieldSolverGMG.kappa_source          = true             ## Volume weighted space charge density or not (depends on algorithm)
FieldSolverGMG.filter_rho            = 0                ## Number of filterings of space charge before Poisson solve
FieldSolverGMG.filter_potential      = 0                ## Number of filterings of potential after Poisson solve

FieldSolverGMG.gmg_verbosity                = -1           ## GMG verbosity
FieldSolverGMG.gmg_use_default_settings     = true         ## Use default multigrid settings or not (see documentation)
FieldSolverGMG.gmg_pre_smooth               = 12           ## Number of relaxations in downsweep
FieldSolverGMG.gmg_post_smooth              = 12           ## Number of relaxations in upsweep
FieldSolverGMG.gmg_bott_smooth              = 0            ## Number of at bottom level (before dropping to bottom solver)
FieldSolverGMG.gmg_precond_smooth           = 12           ## Number of smoothing steps in the preconditioner
FieldSolverGMG.gmg_min_iter                 = 5            ## Minimum number of iterations
FieldSolverGMG.gmg_max_iter                 = 50           ## Maximum number of iterations
FieldSolverGMG.gmg_exit_tol                 = 1.E-10       ## Residue tolerance
FieldSolverGMG.gmg_exit_hang                = 0.2          ## Solver hang
FieldSolverGMG.gmg_min_cells                = 16           ## Bottom drop
FieldSolverGMG.gmg_drop_order               = 0            ## Drop stencil order to 1 if domain is coarser than this.
FieldSolverGMG.gmg_bc_order                 = 1            ## Boundary condition order for multigrid
FieldSolverGMG.gmg_bc_weight                = 4            ## Boundary condition weights (for least squares)
FieldSolverGMG.gmg_jump_order               = 1            ## Boundary condition order for jump conditions
FieldSolverGMG.gmg_jump_weight              = 4            ## Boundary condition weight for jump conditions (for least squares)
FieldSolverGMG.gmg_reduce_order             = true         ## If true, always use order=1 EB stencils in coarsened cells
FieldSolverGMG.gmg_bottom_solver            = bicgstab     ## Bottom solver type. 'simple', 'bicgstab', or 'gmres'
FieldSolverGMG.gmg_cycle                    = vcycle       ## Cycle type. Only 'vcycle' supported for now.
FieldSolverGMG.gmg_smoother                 = red_black    ## Relaxation type. 'jacobi', 'multi_color', or 'red_black'
FieldSolverGMG.gmg_relax_factor             = 1.5          ## Adjustable SOR factor
```

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 *Saturation charge BC* 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`, `FieldSolverGMG` 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:

- `FieldSolverGMG.gmg_verbosity`. Controls the multigrid verbosity. Setting it to a number > 0 will print multigrid convergence information.
- `FieldSolverGMG.gmg_use_default_settings`. Use default multigrid settings. This tends to make most problems converge.
- `FieldSolverGMG.gmg_pre_smooth`. Controls the number of relaxations on each level during multigrid down-sweeps.
- `FieldSolverGMG.gmg_post_smooth`. Controls the number of relaxations on each level during multigrid up-sweeps.
- `FieldSolverGMG.gmg_bott_smooth`. Controls the number of relaxations before entering the bottom solve.
- `FieldSolverGMG.gmg_min_iter`. Sets the minimum number of iterations that multigrid will perform.
- `FieldSolverGMG.gmg_max_iter`. Sets the maximum number of iterations that multigrid will perform.
- `FieldSolverGMG.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 $\Phi = 0$.
- `FieldSolverGMG.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.
- `FieldSolverGMG.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.

- `FieldSolverGMG.gmg_bc_order`. Sets the stencil order for Dirichlet boundary conditions (on electrodes). Note that this is also the stencil radius.
- `FieldSolverGMG.gmg_bc_weight`. Sets the least squares stencil weighting factor for least squares gradient reconstruction on EBs. See *Least squares* for details.
- `FieldSolverGMG.gmg_jump_order`. Sets the stencil order when performing least squares gradient reconstruction on dielectric interfaces. Note that this is also the stencil radius.
- `FieldSolverGMG.gmg_jump_weight`. Sets the least squares stencil weighting factor for least squares gradient reconstruction on dielectric interfaces. See *Least squares* for details.
- `FieldSolverGMG.gmg_bottom_solver`. Sets the bottom solver type.
- `FieldSolverGMG.gmg_cycle`. Sets the multigrid method. Currently, only V-cycles are supported.
- `FieldSolverGMG.gmg_smoother`. Sets the multigrid smoother.
- `FieldSolverGMG.gmg_relax_factor`. Sets the multigrid relaxation factor.

Tip: Enabling this setting tends to make most problems converge quite well.

Warning: 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., `FieldSolverGMG.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:

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

Saturation charge BC

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

```
FieldSolverGMG.which_jump = natural
```

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

```
FieldSolverGMG.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.

4.2.8 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.

4.2.9 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).

4.2.10 Example application(s)

Example applications that use the electrostatics capabilities are:

- *Electrostatics model*.
- *CDR plasma model*.

4.3 Îto diffusion

The Îto diffusion model advances computational particles as drifting Brownian walkers

$$\Delta \mathbf{X} = \mathbf{V} \Delta t + \sqrt{2D\Delta t}\mathbf{W} \tag{4.3.1}$$

where $\mathbf{X}$ is the spatial position of a particle, $\mathbf{V}$ the particle drift velocity, and D is the diffusion coefficient *in the continuum limit*. The vector term $\mathbf{W}$ indicates a random number sampled from a Gaussian distribution with mean value of 0 and standard deviation of 1.

Tip: The code for Îto diffusion is given in `/Source/ItoDiffusion`.

4.3.1 ItoParticle

The `ItoParticle` is used as the underlying particle type for running the Ito drift-diffusion solvers. It derives from *GenericParticle* as follows:

```
class ItoParticle : public GenericParticle<5, 3>
```

From the signature one can see that `ItoParticle` contains a number of extra class `Real` and `RealVect` class members. These extra fields are used for storing the following information in the particle:

1. Particle weight, mobility, diffusion coefficient, energy (not currently used), and a holder for a scratch storage.
2. The previous particle position, the velocity, and a holder for a `RealVect` scratch storage.

Tip: Several member functions are available for obtaining the particle properties. See the full ItoParticle C++ API

4.3.2 ItoSolver

The `ItoSolver` class encapsulates the implementation of Eq. 4.3.1 in `chombo-discharge`. This class can advance a set of computational particles (see *ItoParticle*) with the following functionality:

1. Move particles the a microscopic drift-diffusion model.
2. Compute particle intersection with embedded boundaries and domain edges.
3. Deposit particles and other particle types on the mesh.
4. Interpolate velocities and diffusion coefficients to the particle positons.
5. Manage superparticle splitting and merging.

Internally, `ItoSolver` stores its particles in various `ParticleContainer<ItoParticle>` containers. Although the particle velocities and diffusion coefficients can be manually assigned, they can also be interpolated from the mesh. `ItoSolver` stores the following properties on the mesh:

1. Mobility.
2. Diffusion coefficient.
3. Velocity function.

The reason for storing both the mobility and velocity function is to simply to improve flexibility when assigned the particle velocity $\mathbf{V}$. Note that the velocity function does *not* have to represent the particle velocity. When using both the mobility and velocity function, one can compute the particle velocity as $\mathbf{V} = \mu\mathbf{v}$, where $\mathbf{v}$ is a velocity field. This is typically done for discharge simulations where for simplicity we assign $\mathbf{v}$ to be the electric field, and μ to the the field-dependent mobility. Additional information is available in *Particle interpolation*.

4.3.3 ItoSpecies

`ItoSpecies` is a class for parsing solver information into `ItoSolver`, e.g., whether or not the particle type is mobile or not. The constructor for the `ItoSpecies` class is

```
/*!
  @brief Full constructor
  @param[in] a_name         Species name
  @param[in] a_chargeNumber Charge number
  @param[in] a_mobile       Mobile species or not
  @param[in] a_diffusive    Diffusive species or not
*/
ItoSpecies(const std::string a_name, const int a_chargeNumber, const bool a_mobile, const bool a_diffusive);
```

Here, `a_name` indicates a variable name for the solver. This variable will be used in, e.g., error messages and I/O functionality. `a_chargeNumber` indicates the charge number of the species and the two booleans `a_mobile` and `a_diffusive` indicates whether or not the solver is mobile or diffusive.

Note: The C++ `ItoSpecies` API is available at https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classItoSpecies.html.

Supplying initial data

Initial data for the `ItoSolver` is provided through `ItoSpecies` by providing it with the following:

1. Initial particles specified from a list (`List<ItoParticle>`) of particles.
2. Provide a density description from which initial particles are stochastically sampled within each grid cell.

In particular, there are two data members that must be populated:

```
/*!
  @brief Initial particles
*/
List<ItoParticle> m_initialParticles;

/*!
  @brief Initial density, in case the user wants to generate particles from a density distribution
*/
std::function<Real(const RealVect& x, const Real& t)> m_initialDensity;
```

These can either be populated during construction, or explicitly supplied via the following set functions:

```
/*!
  @brief Set the initial species density
  @param[in] a_initialDensity Initial density.
*/
virtual void
setInitialDensity(const std::function<Real(const RealVect& x, const Real& t)>& a_initialDensity);

/*!
  @brief Get initial particles -- this is called by ItoSolver when filling the solver with initial particles.
  @return Returns m_initialParticles
*/
List<ItoParticle>&
getInitialParticles();
```

When `ItoSolver` initializes the data in the solver, it will copy the particle list `m_initialParticles` from the species and into the solver.

Tip: When using MPI, the user must ensure that each MPI rank does not provide duplicate particles. The `ParticleOps` class contains lots of supporting functionality for sampling particles with MPI, see the ParticleOps C++ API

When sampling particles from a mesh-based density, the solver will generate the particles so that the specified density is approximately reached within each grid cell. If the density that is supplied does not lead to an integer number of particles in the grid cell (which is virtually always the case), the evaluation of the number of particles is stochastically evaluated. E.g., if the density is ϕ and then grid cell volume is ΔV, and $\phi \Delta V = 1.2$, then there is a 20% chance that there will be generated two particles within the grid cell, and 80% chance that only one particle will be generated.

Tip: The number of initially sampled particles is set through `ItoSolver.ppc_restart`.

4.3.4 Particle containers

Internally, `ItoSolver` contains several `ParticleContainer<ItoParticle>` for storing various categories of particles. These categories exist because the transport kernel will almost always lead to particles that leave the domain or intersect the EB. Chemistry models that use `ItoSolver` for tracking particles might also require *new* particles to be added into the domain.

`ItoSolver` defines an enum `WhichContainer` for classification of `ParticleContainer<ItoParticle>` data holders for holding particles that live on:

- Main particles (`WhichContainer::Bulk`).
- The embedded boundary (`WhichContainer::EB`).
- On the domain edges/faces (`WhichContainer::Domain`).
- Representing ''source particles" (`WhichContainer::Source`).
- Particles that live *inside* the EB (`WhichContainer::Covered`).

The particles are available from the solver through the function

```
/*!
  @brief Get a general particle container
  @param[in] a_container Which container to fetch.
*/
virtual ParticleContainer<ItoParticle>&
getParticles(const WhichContainer a_container);
```

Usually, `ItoSolver` will perform a drift-diffusion advance and the user will then check if some of the particles crossed into the EB. The solver can then automatically fill the boundary particles containers, see *Particle intersection*.

4.3.5 Remapping particles

`ItoSolver` has two functions for remapping particles:

```
/*!
  @brief Remap the bulk particle container.
*/
virtual void
remap();

/*!
  @brief Remap all particles in the input container
  @param[in] a_container Particle container
*/
virtual void
remap(const WhichContainer a_container);
```

The bottom function lets the user remap any `ParticleContainer<ItoParticle>` that lives in the solver. Here, `a_container` indicates which particle container to remap.

4.3.6 Particle deposition

`ItoSolver` contains several member functions for depositing various particle properties onto the mesh. The most general version is given below:

```
/*!
  @brief Generic deposition function which deposits a particle field onto the mesh using a specified deposition method.
  @details The template parameters indicate the particle type and quantity to be deposited. The second template parameter must␣
↪be a pointer to a member function in
  the particle class with signature 'const Real& P::function() const'. E.g. 'const Real& P::mass() const' which is the default␣
↪quantity to be deposited.
  @param[out] a_phi                 Mesh data -- must have exactly one component.
```

(continues on next page)

(continued from previous page)

```
  @param[in]  a_particles            Particles to be deposited
  @param[in]  a_deposition           Deposition method
  @param[in]  a_coarseFineDeposition Coarse-fine deposition strategy
*/
template <class P, class Ret, Ret (P ::*MemberFunc)() const>
void
depositParticles(EBAMRCellData&             a_phi,
                 ParticleContainer<P>&      a_particles,
                 const DepositionType       a_deposition,
                 const CoarseFineDeposition a_coarseFineDeposition) const;
```

This version permits the user to select any particle container `a_particles` and deposit them onto some pre-allocated mesh storage `a_phi`. Note that the template type `P` does not need to be `ItoParticle`, although this is the most common use case.

Important: The `ItoSolver` deposition methods are specified in the input script, see *Input options*. Both the base deposition scheme (e.g., NGP or CIC) must be specified, as well as the handling near refinement boundaries.

A simpler version that deposits the bulk particles as a density on the mesh is

```
/*!
  @brief Deposit particles onto mesh.
  @details This will deposit the mass (i.e., computational weight) "bulk" particles into m_phi.
  @note Calls the other version with a_container = WhichContainer::Bulk
*/
virtual void
depositParticles();
```

The particles are deposited into the class member `m_phi`, which stores the particle density on the mesh. This data can then be fetched with

```
/*!
  @brief Get the mesh data.
  @return Returns m_phi
*/
virtual EBAMRCellData&
getPhi();
```

For the full list of available deposition functions, see the `ItoSolver` C++ API https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classItoSolver.html.

Deposition of other quantities

One can also deposit the following quantities on the mesh:

- Conductivity, which deposits μW.
- Diffusivity, which deposits DW.

Here, W is the particle weight, μ is the particle mobility, D is the particle diffusion coefficient. It is up to the user to first interpolate or directly set the particle mobilities and diffusion coefficients before depositing the conductivity onto the mesh.

Functionality for the above deposited quantities exist as the following functions:

```
/*!
  @brief Deposit conductivities (i.e. mass*mobility / volume)
  @details This deposits mass*mobility (not multiplied by charge)
  @param[out] a_phi       Mesh data
  @param[in]  a_particles Particle data
*/
virtual void
depositConductivity(EBAMRCellData& a_phi, ParticleContainer<ItoParticle>& a_particles) const;
```

(continues on next page)

(continued from previous page)

```
/*!
  @brief Deposit diffusivity (i.e. mass*D/volume)
  @details This deposits mass*diffusion (not multiplied by charge)
  @param[out] a_phi       Mesh data
  @param[in]  a_particles Particle data
  @note Calls the other versions with a_deposition = m_deposition
*/
virtual void
depositDiffusivity(EBAMRCellData& a_phi, ParticleContainer<ItoParticle>& a_particles) const;
```

4.3.7 Particle interpolation

Interpolating particle velocities for `ItoSolver` is done by interpolating the mobility and particle velocities to the mesh,

$$\mathbf{V} = \mu\left(\mathbf{X}\right)\mathbf{v}\left(\mathbf{X}\right).$$

There is, however, some freedom in choosing how the mobility coefficient is calculated, which is discussed below. In either case, there is some interpolation from a mesh-based variable onto the particle position $\mathbf{X}$. This interpolation method is always parsed from an options file, and is usually an NGP or CIC scheme.

Important: When interpolating particle properties from the mesh, the user must first ensure that ghost cells are properly updated.

The separation into a mobility function and a velocity field is motivated by the introduction of an electric conductivity that permits a rather simple velocity velocity relation as $\mathbf{v} = \mu\mathbf{E}$, where $\mathbf{E}$ is the electric field. Complete interpolation of the particle velocity consists of calling two functions:

```
/*!
  @brief Interpolate mobilities
  @details This will switch between the two ways of computing the particle mobility.
*/
virtual void
interpolateMobilities();

/*!
  @brief Interpolate the particle velocities.
  @details This will compute the particle velocities as v = mu * V(Xp) where mu is the particle mobility and V(Xp) is the␣
↪interpolation of m_velocityFunction
  to the particle position.
*/
virtual void
interpolateVelocities();
```

Here, the calling sequence is such that the mobilities must be interpolated first, and then the velocity fields.

Mobility coefficient interpolation

The mobility coefficient of a particle is usually interpolated directly, i.e.,

$$\mu = \mu\left(\mathbf{X}\right).$$

The other option is to compute the mobility as

$$\mu = \frac{\left(\mu\left|\mathbf{v}\right|\right)\left(\mathbf{X}\right)}{\left|\mathbf{v}\left(\mathbf{X}\right)\right|}.$$

This method ensures that the particle velocity becomes $\mathbf{V} = \left(\mu\mathbf{v}\right)\left(\mathbf{X}\right)$.

Tip: One can switch between the two interpolation methods in the `ItoSolver` run-time input options.

Diffusion coefficient interpolation

Interpolation of the diffusion coefficient is always done using an interpolation method

$$D = D\left(\mathbf{X}\right).$$

The function signatures is

```
/*!
  @brief Interpolate the diffusion field to the particle positions.
  @details This computes D_p = Df(X_p) where Df is the diffusion field on the mesh.
*/
virtual void
interpolateDiffusion();
```

4.3.8 Particle intersections

It will happen that particles occasionally hit the embedded boundary or leave through the domain sides. In this case one might want to keep the particles in separate data holders rather than discard them. `ItoSolver` supplies several functions for transferring the particles to separate data containers when they intersect the EB or domain. The most relevant function is

```
/*!
  @brief Do boundary intersection tests.
  @details This will intersect the particles in the "bulk" particles data holder with the domain faces and EBs. If a particle␣
↪crossed the EB it will
  be put into the "EB" particle data holder and likewise for the particles that crossed the domain side.
  @param[in] a_ebIntersection      Enum for switching between various types of intersection tests.
  @param[in] a_deleteParticles     If true, the origin particle will also be removed from the bulk particle data holder.
  @param[in] a_nonDeletionModifier Optional input argument for letting the user manipulate particles that were intersected but␣
↪not deleted
  @note This will call the other version with WhichContainer::Bulk, WhichContainer::EB, and WhichContainer::Domain.
*/
virtual void
intersectParticles(
  const EBIntersection                    a_ebIntersection,
  const bool                              a_deleteParticles,
  const std::function<void(ItoParticle&)> a_nonDeletionModifier = [](ItoParticle&) -> void {
    return;
  });
```

Here, `EbIntersection` is a just an enum for putting logic into how the intersection is computed. Valid options are `EbIntersection::Bisection` and `EbIntersection::Raycast`. These algorithms are discussed in *Boundary interaction*. The flag `a_deleteParticles` specifies if the original particles should be deleted when populating the other particle containers (again, see *Boundary interaction*).

After calling `intersectParticles`, the particles that crossed the EB or domain walls are available through the `getParticles` routine, see *ItoSolver* and can then be parsed separately by user code.

4.3.9 Computing time steps

While `ItoSolver` has no fundamental requirement on the time steps that can be used, several functions are available for computing various types of drift and diffusion related time steps.

Important: All time step calculations below are imposed on the particles and not on the mesh variables.

Advective time step

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. This routine is implemented as

```
/*!
  @brief Compute advection time step dt = dx/vMax where vMax is the largest velocity component of the particle.
*/
virtual Real
computeAdvectiveDt() const;
```

which returns a CFL-like condition

$$\Delta t = \frac{\Delta x}{\max(|v_x|, |v_y|, |v_z|)}.$$

Diffusive time step

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

```
/*!
  @brief Compute the diffusive dt. This computes dt = dx*dx/(2*SpaceDim*D) for all particles
*/
virtual Real
computeDiffusiveDt() const;
```

which returns a CFL-like condition

$$\Delta t = \frac{\Delta x^2}{2dD},$$

where d is the spatial dimension and D is the particle diffusion coefficient.

Advective-diffusive time step

A combination of the advection and diffusion time step routines also exists as

```
/*!
  @brief Compute a time step for the advance -- this calls the level function.
  @details This computes the time step differently whether or not diffusion and advection are active. The Ito particle model␣
↪does not have a fundamental
  time step limitation, so these limits "replicate" the time step selections in a 1D fluid model.
  If we only use advection advection the time step is computed as dt = dx/sum(|V_i|) = dtA.
  If only diffusion is active the time step is computed as dt = (dx*dx)/(2*SpaceDim*D) = dtD.
  If both advection and diffusion are active the time step is computed as dt = 1/(1/dtA + 1/dtD).
*/
virtual Real
computeDt() const;
```

This time step limitation is inspired by fully explicit and non-split fluid models, and is calculated as

$$\Delta t = \frac{1}{\frac{\Delta x}{|v_x|+|v_y|+|v_z|} + \frac{\Delta x^2}{2dD}}.$$

4.3.10 Superparticle management

It can occasionally be necessary to merge or split computational particles. This occurs in, e.g., plasma simulations where chemical reactions lead to exponential growth of particles. `ItoSolver` can currently handle superparticles through several internal functions, and is also equipped with an interface in which the user can inject an external particle-handling routine. The function for splitting and merging the particles is in all cases

```
/*!
  @brief Make superparticles for a full container -- this is the AMR version that users will usually call.
  @param[in] a_container        Which container to repartition into new superparticles
  @param[in] a_particlesPerCell Target number of particles per cell
*/
virtual void
makeSuperparticles(const WhichContainer a_container, const int a_particlesPerCell);
```

Calling this function will merge/split the particles.

Important: Particle merging is currently performed within each grid cell, and particles must therefore be sorted by their cell index before calling the merging routine.

In order to specify the merging algorithm the user must set the `ItoSolver.merge_algorithm` to one of the following:

- `none` - No particle merging/splitting is performed.
- `equal_weight_kd` Use a kD-tree with bounding volume hierarchies to partition and split/merge the particles. This conserves the particle center-of-mass.
- `reinitialize` Re-initialize the particles in each grid cell, ensuring that weights are as uniform as possible.
- `reinitialize_bvh` Re-initialize the particles in each node of a kD tree. Weights are as uniform as possible.
- `external` Use an externally injected particle merging algorithm. In order to use this feature the user must supply one through

The user can set the merging algorithm through the input script (see *Input options*), or supply one externally by setting the merge algorithm to `external`. In addition, the user must first supply a particle merging function:

```
/*!
  @brief Set the particle merger. This will get called when merging particles using makeSuperparticles.
  @param[in] a_particleMerger Particle merger
*/
virtual void
setParticleMerger(const ParticleManagement::ParticleMerger<ItoParticle>& a_particleMerger) noexcept;
```

In the code above, `ParticleManagement::ParticleMerger<P>` is an alias:

```
/*!
  @brief Concept for splitting/merging particles
  @param[inout] a_particles Particles to be merged/split
  @param[in] a_cellInfo Cell info
  @param[in] a_numTargetParticles Number of target particles
*/
template <class P>
using ParticleMerger = std::function<
  void(List<P>& a_particles, const CellInfo& a_cellInfo, const int a_numTargetParticles)>;
```

Tip: `ItoSolver` uses the kD-node implementation from *Superparticles* and partitioners for splitting the particles into two subsets with equal weights.

4.3.11 Example transport kernel

Transport kernels for the particles within `ItoSolver` will typically be imposed externally by the user through a `TimeStepper` subclass that advances the particles. For completeness, we here include a simple transport kernel for the `ItoSolver` which simply consists of a drift-diffusion kick:

```
List<ItoParticle> particles;

for (ListIterator<ItoParticle>& lit(particles); lit.ok(); ++lit) {
   ItoParticle& p = lit();

   p.oldPosition() = p.position();
   p.position()   += p.velocity() * a_dt + sqrt(2.0*p.diffusion()*a_dt) * this->randomGaussian();
}
```

The function `randomGaussian` implements a diffusion hopping and returns a 2D/3D dimensional vector with values drawn from a normal distribution with standard width of one and mean value of zero. The implementation uses the random number generators in *Random numbers*.

4.3.12 I/O

Plot files

For a complete list of available plot variables, see *Input options*.

4.3.13 Input options

Several input options are available for configuring the run-time configuration of `ItoSolver`, which are listed in Listing 4.3.1.

Listing 4.3.1: Input options for the `ItoSolver` class. All options are run-time configurable.

```
# ================================================================================================
# ItoSolver class options
# ================================================================================================
ItoSolver.verbosity              = -1                ## Class verbosity
ItoSolver.merge_algorithm        = equal_weight_kd   ## Particle merging algorithm. Either 'reinitialize', 'equal_weight_kd', or
↪'reinitialize_bvh'
ItoSolver.plt_vars               = phi vel dco       ## 'phi', 'vel', 'dco', 'part', 'eb_part', 'dom_part', 'src_part', 'energy_density',
↪ 'energy'
ItoSolver.intersection_alg       = bisection         ## Intersection algorithm for EB-particle intersections.
ItoSolver.bisect_step            = 1.E-4             ## Bisection step length for intersection tests
ItoSolver.normal_max             = 5.0               ## Maximum value (absolute) that can be drawn from the exponential␣
↪distribution.
ItoSolver.redistribute           = false             ## Turn on/off redistribution.
ItoSolver.blend_conservation     = false             ## Turn on/off blending with nonconservative divergenceo
ItoSolver.checkpointing          = particles         ## 'particles' or 'numbers'
ItoSolver.ppc_restart            = 32                ## Maximum number of computational particles to generate for restarts.
ItoSolver.irr_ngp_deposition     = true              ## Force irregular deposition in cut cells or not
ItoSolver.irr_ngp_interp         = true              ## Force irregular interpolation in cut cells or not
ItoSolver.mobility_interp        = direct            ## How to interpolate mobility, 'direct' or 'velocity', i.e. either mu_p =␣
↪mu(X_p) or mu_p = (mu*E)(X_p)/E(X_p)
ItoSolver.plot_deposition        = cic               ## Cloud-in-cell for plotting particles.
ItoSolver.deposition             = cic               ## Deposition type.
ItoSolver.deposition_cf          = transition        ## 'interp', 'halo', 'halo_ngp', 'transition'.
```

Plot file variables

Plot variables are specified using `ItoSolver.plt_vars`, see *Plot files*). To add a variable to HDF5 output files, one can modify the `ItoSolver.plt_vars` input variable to include, e.g., the following variables:

- ϕ, i.e. the deposited particle weights (`ItoSolver.plt_vars = phi`)
- $\mathbf{v}$, the advection field (`ItoSolver.plt_vars = vel`).
- D, the diffusion coefficient (`ItoSolver.plt_vars = dco`).

Particle-mesh configuration

To specify the mobility interpolation, use `ItoSolver.mobility_interp`. Valid options are `direct` and `velocity`, see *Particle interpolation*.

Deposition and coarse-fine deposition (see *Particle-mesh*) is controlled using the flags

- `ItoSolver.deposition` for the base deposition scheme. Valid options are `ngp`, `cic`, and `tsc`.
- `ItoSolver.deposition_cf` for the coarse-fine deposition strategy. Valid options are `interp`, `halo`, or `halo_ngp`.

To modify the deposition scheme in cut-cells, one can enforce NGP interpolation and deposition through

- `ItoSolver.irr_ngp_deposition` for enforcing NGP deposition. Valid options are `true` or `false`.
- `ItoSolver.irr_ngp_interp` for enforcing NGP interpolation. Valid options are `true` or `false`.

Checkpoint-restart

Available input options for the `ItoSolver` are listed below:

4.3.14 Example application(s)

Example applications that use `ItoSolver` are found in

- `$DISCHARGE_HOME/Physics/BrownianWalker`, see *Brownian walker*.
- `$DISCHARGE_HOME/Physics/ItoKMC`, see *Îto-KMC plasma model*.

4.4 Kinetic Monte Carlo

Kinetic Monte Carlo (KMC) algoritms are composed of various methods for stochastically simulating chemically reacting systems. While various flavors of KMC are encountered in different fields of science, KMC in the context of `chombo-discharge` is primarily associated with chemistry kernels.

4.4.1 Concept

In `chombo-discharge` the Kinetic Monte Carlo solver advances a state (or multiple states) represented e.g. as state vectors

$$\vec{X}(t) = \begin{pmatrix} X_1(t) \\ X_2(t) \\ X_3(t) \\ \vdots \end{pmatrix}$$

Each row in $\vec{X}$ represents the population of some chemical species, and is indexed by an integer. Reactions between species are represented stoichiometrically as

$$X_A + X_B + \ldots \xrightarrow{k} X_C + X_D + \ldots,$$

where k is the reaction rate. Each such reaction is associated with a state change in $\vec{X}$, e.g.

$$\vec{X} \xrightarrow{r} \vec{X} + \vec{\nu}_r,$$

where r is the reaction type and ν_r is the state change associated with the firing of *one* reaction of type r. A set of such reactions is called the *reaction network* $\vec{R}$.

Propensities $a_r\left(\vec{X}\right)$ are defined such that $a_r\left(\vec{X}\right)\mathrm{d}t$ is the probability that exactly one reaction of type r occurs in the time interval $[t, t+\mathrm{d}t]$. For unimolecular reactions of the type

$$X_A + X_B + \ldots \xrightarrow{k} \emptyset$$

with $A \neq B \neq \ldots$ the propensity function is $kX_AX_B\ldots$. For bimolecular of the type

$$2X_A \xrightarrow{k} \emptyset$$

the propensity is $k\frac{1}{2}X_A(X_A-1)$ because there are $\frac{1}{2}X_A(X_A-1)$ unique pairs of molecules of type A. Propensities for higher-order reactions can then be expanded using the binomial theorem. For example, for a third-order reaction $3X_A \xrightarrow{k} \emptyset$ the propensity function is $k\frac{1}{6}X_A(X_A-1)(X_A-2)$.

Various algorithms can be used for advancing the state $\vec{X}$ for an arbitrary reaction network $\vec{R}$.

1. The *Stochastic simulation algorithm* (SSA). The SSA is also known as the Gillespie algorithm [Gillespie, 1977], and is an exact stochastic solution to the above problem. However, it becomes inefficient as the number of reactions per unit time grows.
2. *Tau leaping*, which is an approximation to the SSA which uses Poisson sampling of the underlying reactions.
3. Hybrid advance that switches between the SSA and tau-leaping in their respective limits, see *Hybrid algorithm*. The hybrid algorithm is taken from Cao *et al.* [2006], and switches between tau leaping and the SSA in their respective limits.

4.4.2 Stochastic simulation algorithm

For the SSA we compute the time until the next reaction by

$$T = \frac{1}{\sum_{r\in\vec{R}} a_r} \ln\left(\frac{1}{u_1}\right)$$

where $A = \sum_{r\in\vec{R}} a_r$ and u_1 is a uniformly distributed random variable between 0 and 1. The type of reaction that fires is deterimined from

$$r_c = \text{smallest integer satisfying } \sum_{r'=1}^{r_c} a_{r'} > u_2 A,$$

where and u_2 is another uniformly distributed random variable between 0 and 1. The state is then advanced as

$$\vec{X}(t+T) = \vec{X}(t) + \vec{\nu}_{r_c}.$$

4.4.3 Tau leaping

With tau-leaping the state is advanced over a time Δt as

$$\vec{X}\left(t+\Delta t\right) = \vec{X}\left(t\right) + \sum_{r\in\vec{R}} \vec{\nu}_r \mathcal{P}\left(a_r\left[\vec{X}\left(t\right)\right]\Delta t\right),$$

where $\mathcal{P}$ is a Poisson-distributed random variable. Note that tau leaping may fail to give a thermodynamically valid state, and should thus be used in combination with step rejection.

Tau-leaping variants

The following forms of tau-leaping are also supported:

1. Midpoint tau-leaping.
2. Poisson random-corrections tau-leaping.
3. Implicit Euler-type tau-leaping.

These methods can be used either as standalone methods or together with the hybrid algorithm.

Warning: We do not recommend implicit methods for reactive problems. The reason for this is that exponential growth follows the equation

$$\partial_t X = kX,$$

where $k > 0$ is a growth rate. Application of the implicit Euler rule to this system yields

$$X^{n+1} = \frac{X^n}{1-k\Delta t},$$

which has a pole at $\Delta t = k^{-1}$, and which is only non-negative for $k\Delta t < 1$. Similarly, the discretization can then lead to a large overshoot. In practice, the time step then has to be limited to $\Delta t < k^{-1}$.

On the other hand, an explicit Euler update would yield

$$X^{n+1} = \left(1+k\Delta t\right)X^n,$$

which is stable for any Δt (provided k is positive).

4.4.4 Hybrid algorithm

The hybrid algorithm is taken from Cao *et al.* [2006]. Assume that we wish to integrate over some time Δt, which proceeds as follows:

1. Let $\tau = 0$ be the simulated time within Δt.
2. Partition the reaction set $\vec{R}$ into *critical* and *non-critical* reactions. The critical reactions are defined as the subset of $\vec{R}$ that are within N_{crit} firings away from exhausting one of its reactants. The non-critical reactions are defined as the remaining subset.

3. Compute time steps until the firing of the next critical reaction, and a time step such that the propensities of the non-critical reactions do not change by more than some relative factor ϵ. Let these time steps be given by $\Delta\tau_{\mathrm{c}}$and $\Delta\tau_{\mathrm{nc}}$.
4. Select a reactive substep within Δt from

$$\Delta\tau = \min\left[\Delta t - \tau, \min\left(\Delta\tau_{\mathrm{c}}, \Delta\tau_{\mathrm{nc}}\right)\right]$$

5. Resolve reactions as follows:
   a. If $\Delta\tau_{\mathrm{c}} < \Delta\tau_{\mathrm{nc}}$ and $\Delta\tau_{\mathrm{c}} < \Delta t - \tau$ then one critical reaction fires. Determine the reaction type using the SSA algorithm.

      Next, advance the state using tau leaping for the non-critical reaction.

   b. Otherwise: No crical reactions fire. Advance the state using tau-leapnig for the non-critical reactions only. An exception is made if $A\Delta\tau$ is smaller than some specified threshold in which case we switch to SSA advancement (which is more efficient in this limit).
6. Check if $\vec{X}$ is a thermodynamically valid state.
   a. If the state is valid, accept it and let $\tau \rightarrow \tau + \Delta\tau$.
   b. If the state is invalid, reject the advancement. Let $\Delta\tau_{\mathrm{nc}} \rightarrow \Delta\tau_{\mathrm{nc}}/2$ and return to step 4).
7. If $\tau < \Delta t$, return to step 2.

The Cao *et al.* [2006] algorithm requires algorithmic specifications as follows:

- The factor ϵ which determines the non-critical time step.
- The factor N_{crit} which determines which reactions are critical or not.
- Factors for determining when and how to switch to the SSA-based algorithm in step 5b.

4.4.5 Implementation

In `chombo-discharge`, the KMC solver is implemented as

```
/*!
  @brief Class for running Kinetic Monte-Carlo simulations.
  @details The template parameter State is the underlying state type that KMC operators on. There are rather simple required␣
↪member functions on the State parameter,
  but the reaction type (template parameter R) MUST be able to operate on the state through the following functions:

  1. Real R::propensity(State) const -> Computes the reaction propensity for the input state.
  2. T R::computeCriticalNumberOfReactions(State) const -> Computes the minimum number of reactions that exhausts one of the␣
↪reactants.
  3. void R::advanceState(State&, const T numReactions) const -> Advance state by numReactions
  4. std::<some_container> getReactants() const -> Get reactants involved in the reactions.
  5. static T R::population(const <some_type> reactant, const State& a_state) -> Get the population of the input reactant in␣
↪the input state.

  The template parameter T should agree across both both R, State, and KMCSolver. Note that this must be a signed integer type.
*/
template <typename R, typename State, typename T = long long>
class KMCSolver
{
public:
  using ReactionList = std::vector<std::shared_ptr<const R>>;

  /*!
    @brief Full constructor.
    @param[in] a_reactions List of reactions.
  */
  inline KMCSolver(const ReactionList& a_reactions) noexcept;
```

Here, the template parameters are:

- R is the type of reaction to advance with.
- State is the state vector that the KMC and reactions will advance.
- T is the internal floating point or integer representation.

Tip: The KMCSolver C++ API is found at https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classKMCSolver.html.

KMCSolver is designed to operate with the possibility of separating the solver from the reaction and state types. Several template constraints exist on the reaction type R as well as the state type State.

State

The State representation *must* have a member function

```
/*!
  @brief Check if state is a valid state.
  @return Returns false if any populations are negative.
*/
inline bool
isValidState() const noexcept;
```

This function should return true if the state is a valid one (e.g., no negative populations) and false otherwise. The functionality is used when using the hybrid advancement algorithm, see *Hybrid algorithm*.

Reaction(s)

The reaction representation R *must* have the following member functions:

```
/*!
  @brief Compute the propensity function for this reaction type.
  @param[in] a_state State vector
  @note User should set the rate before calling this routine.
*/
inline Real
propensity(const State& a_state) const noexcept;

/*!
  @brief Compute the number of times the reaction can fire before exhausting one of the reactants.
  @param[in] a_state State vector
*/
inline T
computeCriticalNumberOfReactions(const State& a_state) const noexcept;

/*!
  @brief Get the reactants involved in the reaction.
  @return m_reactants
*/
inline std::list<size_t>
getReactants() const noexcept;

/*!
  @brief Get the state change due to a change in the input reactant species.
  @param[in] a_rectant Reactant species.
*/
inline T
getStateChange(const size_t a_reactant) const noexcept;

/*!
  @brief Advance the incoming state with the number of reactions.
  @param[in] a_state        State vector
  @param[in] a_numReactions Number of reactions.
*/
inline void
advanceState(State& a_state, const T& a_numReactions) const noexcept;
```

These template requirements exist so that users can define their states independent of their reactions. Likewise, reactions can be defined to operate flexibly on state, and the `KMCSolver` can be defined without deep restrictions on the states and reactions that are used.

Defining states

State representations `State` can be defined quite simply (e.g. just a list of indices). In the absolute simplest case a state can be defined by maintaining a list of populations like below:

```
class MyState {
public:
   MyState(const size_t numSpecies) {
      m_populations.resize(numSpecies);
   }

   bool isValidState() const {
      return true;
   }

   std::vector<long long> m_populations;
};
```

More advanced examples can distinguish between different *modes* of populations, e.g. between species that can only appear on the left/right hand side of the reactions.

Defining reactions

See *Implementation* for template requirements on state-advancing reactions. Using `MyState` above as an example, a minimal reaction that can advance $A \rightarrow B$ with a rate of $k = 1$ is

```
class MyStateReaction {
public:

   // List of reactants and products
   MyStateReaction(const size_t a_A, const size_t a_B) {
      m_A = a_A;
      m_B = a_B;
   }

   // Compute propensity
   Real propensity(const State& a_state) {
      return a_state[m_A];
   }

   // Never consider these reactions to be "critical"
   long long computeCriticalNumberOfReactions(const Mystate& a_state) {
      return std::numeric_limits<long long>::max();
   }

   // Get a vector/list/deque etc. of the reactant's. <some_container> can be e.g. std::vector<size_t>
   std::list<size_t> R::getReactants() const {
      return std::list<size_t>{m_A};
   }

   // Get population
   long long population(const size_t& a_reactant, const MyState& a_state) {
      return a_state.m_populations[a_reactant];
   }

   // Advance state with reaction A -> B
   void advanceState(const MyState& s, const long long& numReactions) const {
      s.populations[m_A] -= numReactions;
      s.populations[m_B] += numReactions;
   }

protected:
   size_t m_A;
   size_t m_B;
};
```

Advancement routines

Many advancement routines for the `KMCSolver` are defined internally. The most general one that uses the hybrid advance is

```
/*!
  @brief Advance using Cao et. al. hybrid algorithm over the input time. This can end up using substepping.
  @param[inout] a_state          State vector to advance
  @param[in]    a_dt            Time increment
  @param[in]    a_leapPropagator Which leap propagator to use.
  @note Calls the other version with m_reactions
*/
inline void
advanceHybrid(State&                  a_state,
              const Real              a_dt,
              const KMCLeapPropagator& a_leapPropagator = KMCLeapPropagator::ExplicitEuler) const noexcept;
```

When using the hybrid algorithm, the user should set the hybrid solver parameters through the function

```
/*!
  @brief Set solver parameters
  @param[in] a_numCrit Determines critical reactions. This is the number of reactions that need to fire before depleting a␣
↪reactant.
  @param[in] a_numSSA  Maximum number of SSA steps to run when switching from tau-leaping to SSA (hybrid algorithm only).
  @param[in] a_maxIter Maximum permitted number of iterations for the implicit solver
  @param[in] a_eps     Maximum permitted change in propensities when performing tau-leaping for non-critical reactions.
  @param[in] a_SSAlim  Threshold for switching from tau-leaping of non-critical reactions to SSA for all reactions (hybrid␣
↪algorithm only)
  @param[in] a_exitTol Exit tolerance for implicit solvres.
*/
inline void
setSolverParameters(const T    a_numCrit,
                    const T    a_numSSA,
                    const T    a_maxIter,
                    const Real a_eps,
                    const Real a_SSAlim,
                    const Real a_exitTol) noexcept;
```

4.4.6 State and reaction examples

`chombo-discharge` maintains some states and reaction methods that can be useful when solving problems with `KMCSolver`. The following two implementations are currently in use:

1. `KMCSingleState`, see the KMCSingleState C++ API and the KMCSingleStateReaction C++ API.
2. `KMCDualState`, see the KMCDualState C++ API and the KMCDualStateReaction C++ API.

4.4.7 Verification

Verification tests for `KMCSolver` are given in

- `$DISCHARGE_HOME/Exec/Convergence/KineticMonteCarlo/C1`
- `$DISCHARGE_HOME/Exec/Convergence/KineticMonteCarlo/C2`

C1: Avalanche model

An electron avalanche model is given in `$DISCHARGE_HOME/Exec/Convergence/KineticMonteCarlo/C1`. The problem solves for a reaction network

$$X + \emptyset \xrightarrow{k_i} X + X + \emptyset$$
$$X + \emptyset \xrightarrow{k_a} \emptyset$$

In the limit $X \gg 1$ the exact solution is

$$X(t) \approx X(0) \exp\left[(k_i - k_a)t\right].$$

Figure Fig. 4.4.1 shows the Kinetic Monte Carlo solution for $k_i = 2k_a = 2$ and $X(0) = 10$.

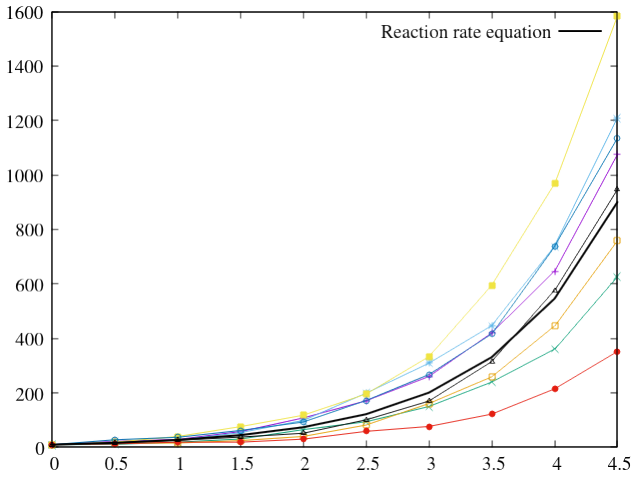

Fig. 4.4.1: Comparison of Kinetic Monte Carlo solution with reaction rate equation for an avalanche-like problem.

C2: Schlögl model

Solution the Schlögl model are given in `$DISCHARGE_HOME/Exec/Convergence/KineticMonteCarlo/C2`. For the Schlögl model we solve for a single population X with the reactions

$$B_1 + 2X \xrightarrow{c_1} 3X,$$
$$3X \xrightarrow{c_2} B1 + 2X,$$
$$B2 \xrightarrow{c_3} X,$$
$$X \xrightarrow{c_4} B2.$$

The states B_1 and B_2 are buffered states with populations that do not change during the reactions. Figure Fig. 4.4.1 shows the Kinetic Monte Carlo solutions for rates

$$c_1 = 3 \times 10^{-7},$$
$$c_2 = 10^{-4},$$
$$c_3 = 10^{-3},$$
$$c_4 = 3.5$$

and $B_1 = 10^5$, $B_2 = 2 \times 10^5$. The initial state is $X(0) = 250$.

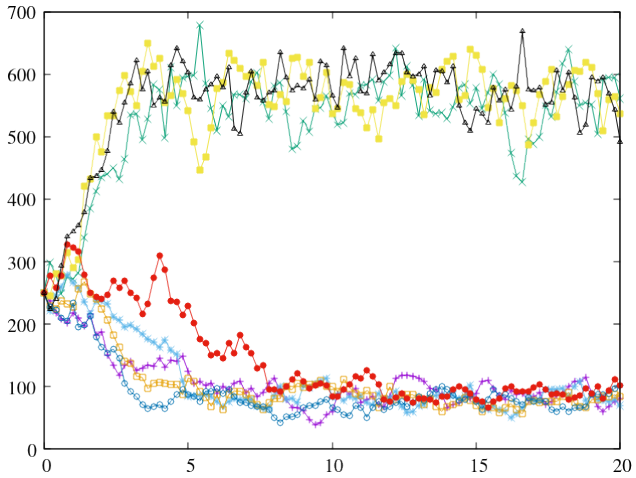

Fig. 4.4.2: Convergence to bi-stable states for the Schlögl model.

4.5 Mesh ODE solver

The `MeshODESolver<N>` implements a solver for

$$\frac{\partial \vec{\phi}}{\partial t} = \vec{S},$$

where $\vec{\phi}$ represents N unknowns on the mesh, and $\vec{S}$ is the corresponding source term. The class is templated as

```
/*!
  @brief Class for solving dy/dt = f on an AMR hierarchy.
  @details The template parameter is the number of variables in y and f.
*/
template <size_t N = 1>
class MeshODESolver
```

where `N` indicates the number of variables stored on the mesh.

`MeshODESolver<N>` is designed to store `N` variables in each grid cell, without any cell-to-cell coupling. To instantiate the solver, use the full constructor with reference to *AmrMesh*:

```
/*!
  @brief Full constructor.
  @param[in] a_amr AMR core.
*/
MeshODESolver(const RefCountedPtr<AmrMesh>& a_amr) noexcept;
```

Tip: Source code for the `MeshODESolver<N>` resides in `Source/MeshODESolver`. See https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classMeshODESolver.html for the full C++ API.

4.5.1 Setting $\vec{\phi}$

Mesh-based

To set $\vec{\phi}$, one can fetch the N mesh components from

```
/*!
  @brief Get the solution vector (left-hand side of equation).
*/
EBAMRCellData&
getPhi() noexcept;
```

This will return the data holder that holds the cell centered data $\vec{\phi}$ which the user can iterate through. See *Mesh data* for examples.

Analytic function

One can set $\vec{\phi}(\mathbf{x}) = \vec{f}(\mathbf{x})$ through the following functions:

```
/*!
  @brief Set phi for a specific component.
  @param[in] a_phiFunc Phi function
  @param[in] a_comp    Component
*/
virtual void
setPhi(const std::function<Real(const RealVect& a_pos)>& a_phiFunc, const size_t a_comp) noexcept;

/*!
  @brief Set phi everywhere.
  @param[in] a_phiFunc Phi function.
*/
virtual void
setPhi(const std::function<std::array<Real, N>(const RealVect& a_pos)>& a_phiFunc) noexcept;
```

These differ only in that the first function sets a specific component, whereas the second version sets all N components.

4.5.2 Setting $\vec{S}$

Mesh-based

For a general method of setting the source term one can fetch $\vec{S}$ through

```
/*!
  @brief Get the solution vector (left-hand side of equation).
*/
EBAMRCellData&
getRHS() noexcept;
```

This returns a reference to $\vec{S}$ which the user can iterate through and set the value in each cell. See *Mesh data* for explicit examples.

Spatially dependent

The source term can also be set on a component-by-component basis using

```
/*!
  @brief Set right-hand side for specified component.
  @param[in] a_srcFunc Source term function.
  @param[in] a_comp    Component
*/
virtual void
setRHS(const std::function<Real(const RealVect& a_pos)>& a_srcFunc, const size_t a_comp) noexcept;
```

As a function of $\vec{\phi}$

In order to compute the source term $\vec{S}$ as a function of $\vec{\phi}$, `MeshOdeSolver<N>` has a function

```
/*!
  @brief Alias for right-hand side
*/
using RHSFunction = std::function<std::array<Real, N>(const std::array<Real, N>&, const Real)>;

/*!
  @brief Compute right-hand side from left-hand side. I.e. compute f = f(y,t).
  @param[in] a_rhsFunction Function for computing the right-hand side.
*/
virtual void
computeRHS(const RHSFunction& a_rhsFunction) noexcept;
```

which computes the source term $\vec{S}$ as a function

$$\vec{S} = \vec{f}\left(\vec{\phi}, t\right).$$

An example which sets $\vec{S} = \vec{\phi}$ is given below

```
auto f = [](const std::array<Real, N>& phi, const Real t) -> std::array<Real, N> {
   const std::array<Real, N> S = phi;

   return S;
};

solver.computeRHS(f);
```

4.5.3 Regridding

When regridding the `MeshODESolver<N>`, one must first ensure that the mesh data on the old mesh is stored before calling the regrid function:

```
/*!
  @brief Perform pre-regrid operations.
  @param[in] a_lbase          Coarsest level that changed during regrid.
  @param[in] a_oldFinestLevel Finest grid level before the regrid operation.
  @note This copies m_phi onto m_cache
*/
virtual void
preRegrid(const int a_lbase, const int a_oldFinestLevel) noexcept;
```

This must be done *before* *AmrMesh* creates the new grids. This will store $\vec{\phi}$ on the old mesh. After *AmrMesh* has generated the new grids, $\vec{\phi}$ can be interpolated onto the new grids by calling

```
/*!
  @brief Regrid this solver.
  @param[in] a_lmin           Coarsest level where grids did not change.
  @param[in] a_oldFinestLevel Finest AMR level before the regrid.
  @param[in] a_newFinestLevel Finest AMR level after the regrid.
```

(continues on next page)

(continued from previous page)

```
  @details This linearly interpolates (potentially with limiters) m_phi to the new grids.
*/
virtual void
regrid(const int a_lmin, const int a_oldFinestLevel, const int a_newFinestLevel) noexcept;
```

Users can also choose to turn on/off slope limiters when putting the solution on the new mesh.

Important: The source term $\vec{S}$ is also allocated on the new mesh, but is not interpolated onto the new grids. It must therefore be set by the user after calling the regrid function.

4.5.4 Input options

Several input options are available for configuring the run-time configuration of `MeshODESolver<N>`, which are listed below

Listing 4.5.1: Input options for the `MeshODESolver<N>` class. All options are run-time configurable.

```
# ============================================================================================
# MeshODESolver class options
# ============================================================================================
MeshODESolver.verbosity             = -1         ## Verbosity level
MeshODESolver.plt_vars              = phi rhs    ## Stuff that can be plotted. 'phi' and/or 'rhs
MeshODESolver.use_regrid_slopes     = false      ## Use or don't use slopes when regridding
```

4.5.5 I/O

The user can add $\vec{\phi}$ and $\vec{S}$ to output files by specifying these in the input script. These variables are named

```
MeshODESolver.plt_vars = phi rhs
```

Only `phi` and `rhs` are recognized as valid arguments. If choosing to omit output variables for the solver, one can put e.g. `MeshODESolver.plt_vars = -1`.

Note: `MeshODESolver<N>` checkpoint files only contain $\vec{\phi}$.

4.6 Radiative transfer

4.6.1 RtSolver

Radiative transfer solvers are supported in the form of

- Diffusion solvers, i.e. first order Eddington solvers, which take the form of a Helmholtz equation.
- Particle solvers, which track photons are particles (e.g., Monte Carlo sampled solvers).

The solvers share a parent class `RtSolver`, and code that uses only the `RtSolver` interface will should be able to switch between the two implementations. Note, however, 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 instantaneous transport.

Tip: 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 solver*). 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.

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 reference to an `RtSpecies` instantiation such that the solvers can look up the required information. Currently, `RtSpecies` is a lightweight class where the user needs to implement the function

```
/*!
  @brief Get kappa (i.e. the inverse absorption length) at physical coordinates.
  @param[in] a_pos Physical coordinates.
*/
virtual Real
getAbsorptionCoefficient(const RealVect a_pos) const = 0;
```

This absorption coefficient is used in both the diffusion (see *Diffusion approximation*) and Monte Carlo (see *Monte Carlo solver*) solvers.

Important: Upon construction, one must set the class member `m_name`, which is the name passed to the actual solver.

Setting the source term

`RtSolver` stores a source term η on the mesh, which describes the number of photons that are generated produced per unit volume and time. This variable can be set through the following functions:

```
/*!
  @brief Set source term
  @param[in] a_source Source term
*/
virtual void
setSource(const EBAMRCellData& a_source);

/*!
  @brief Set source
  @param[in] a_source Source term
*/
virtual void
setSource(const Real a_source);

/*!
  @brief Set source
  @param[in] a_source Source term (varies in space)
*/
virtual void
setSource(const std::function<Real(const RealVect a_pos)> a_source);
```

The usage of η varies between the different solvers. It is possible, for example, to generate computational photons (particles) using η when using Monte Carlo sampling, but this is not a requirement.

4.6.2 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(s) of motion

In the diffusion approximation, the radiative transport equation is

$$\partial_t \Psi + \kappa \Psi - \nabla \cdot \left(\frac{1}{3\kappa} \nabla \Psi \right) = \frac{\eta}{c}, \tag{4.6.1}$$

where Ψ is the radiative intensity (i.e., photons absorbed per unit volume`. Here, $\kappa$ is the absorption coefficient (i.e., inverse absorption length). This value can be spatially dependent, and is passed in through the *RtSpecies* function `getAbsorptionCoefficient` that was discussed above. Note that in the context below, κ is *not* the volume fraction of a grid cell but the absorption coefficient. The above equation is called the Eddington approximation, with the closure relation being that the radiative flux is given by $F = -\frac{c}{3\kappa}\nabla\Psi$.

In the stationary case this reduces to a Helmholtz equation

$$\kappa \Psi - \nabla \cdot \left(\frac{1}{3\kappa} \nabla \Psi \right) = \frac{\eta}{c}, \tag{4.6.2}$$

Implementation

`EddingtonSP1` uses multigrid methods for solving Eq. 4.6.1 and Eq. 4.6.2, see *Linear solvers*. To advance the solution, one will call the member function

```
/*!
  @brief Advance RTE onto state a_phi
  @param[in]    a_dt      Time step
  @param[inout] a_phi     RTE solution
  @param[in]    a_source  Source term
  @param[in]    a_zeroPhi Set phi to zero in initial guess for multigrid solve
  @note If you're not doing a stationary solve, this does a backward Euler solve.
*/
virtual bool
advance(const Real a_dt, EBAMRCellData& a_phi, const EBAMRCellData& a_source, const bool a_zeroPhi = false) override;
```

Internally, this version will perform one of the following:

1. Solve Eq. 4.6.1 if using a *transient* solver. This is done using a backward Euler solve:

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

   This equation is a Helmholtz equation for Ψ^{k+1} which is solved using geometric multigrid, see *Linear solvers*.

2. Solve Eq. 4.6.2 if using instantaneous photon transport. This is done directly with a geometric multigrid solver, see *Linear solvers*.

Boundary conditions

Simplified domain boundary conditions

It is possible to set the following *simplified* boundary conditions on domain faces and embedded boundaries: All of these boundary condition specifications take the form `<type> <value>`.

1. Dirichlet, with a fixed value of Φ. E.g., `dirichlet 0.0`.
2. Neumann, using a fixed value of $\partial_n \Phi$. E.g., `neumann 0.0`.
3. A *Larsen-type* boundary condition, which is an absorbing boundary condition in the form

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

   where r_1 and r_2 are reflection coefficients and g is a surface source, see [Larsen *et al.*, 2002]. 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

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

   The user must then pass a value `larsen <value>`, where the `value` corresponds to the souce term g. Typically, this term is zero.

Tip: For radiative transfer, the Larsen boundary condition is usually the correct one as it approximately describes outflow of photons on the boundary. In this case the correct boundary condition is `larsen 0.0`.

Custom domain boundary conditions

It is possible to use more complex boundary conditions by passing in `dirichlet_custom`, `neumann_custom`, or `larsen_custom` options through the solver configuration options (see *Solver configuration*). In this case the `EddingtonSP1` solver will use a specified function at the domain edge/face which can vary spatially (and with time). To specify that function, `EddingtonSP1` has a member function

```
/*!
  @brief Set the boundary condition function on a domain side
  @param[in] a_dir      Coordinate direction.
  @param[in] a_side     Side (low/high)
  @param[in] a_function Boundary condition function.
  @details This sets a boundary condition for a particular domain side. The user must also specify how to use this BC in the␣
↪input script.
*/
virtual void
setDomainSideBcFunction(const int                                a_dir,
                        const Side::LoHiSide                     a_side,
                        const EddingtonSP1DomainBc::BcFunction& a_function);
```

Here, the `a_function` argument is simply an alias:

```
/*!
  @brief Function which maps f(R^3,t) : R. Used for setting the associated value and boundary condition type.
*/
using BcFunction = std::function<Real(const RealVect a_position, const Real a_time)>;
```

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);
```

Warning: A run-time error will occur if the user specifies one of the custom boundary conditions but does not actually set the function.

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.

The specification of these boundary conditions occurs in precise analogy with the domain boundary conditions, and are therefore not discussed further here.

Solver configuration

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

Listing 4.6.1: `EddingtonSP1` solver configuration options. Run-time configurable options are highlighted.

```
# ================================================================================================
# EddingtonSP1 class options
# ================================================================================================
EddingtonSP1.verbosity              = -1         ## Solver verbosity
EddingtonSP1.stationary             = true       ## Stationary solver
EddingtonSP1.reflectivity           = 0.         ## Reflectivity
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.use_regrid_slopes      = true       ## Slopes on/off when regridding

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    ## Bc on domain side. 'dirichlet', 'neuman', 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. Either 'simple <number>' and 'bicgstab'
EddingtonSP1.gmg_cycle              = vcycle       ## Cycle type. Only 'vcycle' supported for now
```

(continues on next page)

(continued from previous page)

```
EddingtonSP1.gmg_ebbc_weight       = 1            ## 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'
```

The multigrid options are analogous to the multigrid options for *FieldSolverGMG*, see *Tuning multigrid performance*.

4.6.3 Monte Carlo solver

`McPhoto` defines a class which can solve radiative transfer problems using discrete photons. The class derives from *RtSolver* and can thus be used also be used by applications that only require the *RtSolver* interface. `McPhoto` can provide a rather complex interaction with boundaries, such as computing the intersection between a photon path and a geometry, and thus capture shadows (which *EddingtonSP1* can not).

The Monte Carlo sampling is a particle-based radiative transfer solver, and particle-mesh operations (see *Particle-mesh*) are thus required in order to deposit the photons on a mesh if one wants to compute mesh-based absorption profiles.

Tip: The `McPhoto` class is defined in `$DISCHARGE_HOME/Source/RadiativeTransfer/CD_McPhoto.H`. See the McPhoto C++ API for further details.

The solver has multiple data holders for systemizing photons, which is especially useful during transport kernels where some of the photons might strike a boundary:

- In-flight photons.
- Bulk-absorbed photons, i.e., photons that were absorbed on the mesh.
- EB-absorbed photons, i.e., photons that struck the EB during a transport step.
- Domain-absorbed photons, i.e., photons that struck the domain edge/face during a transport step.
- Source photons, for letting the user pass in externally generated photons into the solver.

Various functions are in place for obtaining these particles:

```
/*!
  @brief Get m_photons
  @return m_photons
*/
virtual ParticleContainer<Photon>&
getPhotons();

/*!
  @brief Get bulk photons, i.e. photons absorbed on the mesh
  @return m_bulkPhotons
*/
virtual ParticleContainer<Photon>&
getBulkPhotons();

/*!
  @brief Get eb Photons, i.e. photons absorbed on the EB
  @return m_ebPhotons
*/
virtual ParticleContainer<Photon>&
getEbPhotons();

/*!
  @brief Get domain photons, i.e. photons absorbed on domain edges/faces
  @return m_domainPhotonsl
*/
virtual ParticleContainer<Photon>&
getDomainPhotons();

/*!
  @brief Get source photons
  @return m_sourcePhotons
```

(continues on next page)

(continued from previous page)

```
*/
virtual ParticleContainer<Photon>&
getSourcePhotons();
```

Photon particle

The `Photon` particle is a simple encapsulation of a computational photon which is used by `McPhoto`. It derives from `GenericParticle<2,1>` and stores (in addition to the particle position):

- The particle weight.
- The particle mean absorption coefficient.
- The particle velocity/direction.

Tip: The `Photon` class is defined in `$DISCHARGE_HOME/Source/RadiativeTransfer/CD_Photon.H`

When defining the `McPhoto` class, the particle's absorption coefficient can be computed from the implementation of the absorption function method in *RtSpecies*.

Generating photons

There are several ways users can generate computational photons that are to be transported by the solver.

1. Fetch the *source photons* by calling

   ```
   /*!
     @brief Get source photons
     @return m_sourcePhotons
   */
   virtual ParticleContainer<Photon>&
   getSourcePhotons();
   ```

   The source photons can then be filled and added to the other photons.

2. Add photons directly, by first obtaining the in-flight photons through

   ```
   /*!
     @brief Get m_photons
     @return m_photons
   */
   virtual ParticleContainer<Photon>&
   getPhotons();
   ```

   Photons can then be added directly.

3. If the source term η has been filled, the user can call `McPhoto::advance` to have the solver generate the computational photons and then advance them. This is the correct approach for, e.g., applications that always use mesh-based photon source terms and want to have the computational photons be generated on the fly.

   Warning: The `advance` function is *only* meant to be used together with a mesh-based source term that the user has filled prior to calling the method.

   When using the `advance`, the number of photons that are generated are limit to a user-specified number (see *Solver configuration* for further details).

Transport modes

McPhoto can be run as a fully transient, in which photons are tracked in time, or as an instantaneous solver. For the instantaneous mode, photon absorption positions are stochastically sampled with Monte Carlo procedure and the photons are immediately absorbed on the mesh. For the transient mode the photon advancement occurs over Δt, so there is a limited distance ($c\Delta t$) that the photons can propagate. In this case, only some of the photons will be absorbed on the mesh whereas the rest may continue their propagation.

Instantaneous transport

When using instantaneous transport, any photon generated in a time step is immediately absorbed on the boundary through the following steps:

1. Optionally, have the solver generate photons to be transport (or add them externally).
2. Draw a propagation distance r by drawing random numbers from an exponential distribution $p(r) = \kappa \exp(-\kappa r)$. Here, κ is computed by calling the underlying *RtSpecies* absorption function. The absorbed position of the photon is set to $\mathbf{x} = \mathbf{x}_0 + r\mathbf{n}$.

   Warning: In instantaneous mode photons might travel infinitely long, i.e. there is no guarantee that $c\Delta t \leq r$.

3. Deposit the photons on the mesh.

Transient transport

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. For each photon, do the following:

1. Compute an absorption length r by sampling the absorption function at the current photon position.
2. Each photon is advanced over the time step Δt such that the position is

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{c}\Delta t.$$

3. Check if $|\mathbf{x} - \mathbf{x}_0| < r$ and if it is, absorb the photon on the mesh.

Other transport kernels

In addition to the above two methods, the solver interface permits users to add e.g. source photons externally and add them to the solvers' transport kernel.

Solver configuration

McPhoto can be configured through its input options, see below:

```
# ================================================================================================
# McPhoto class options
# ================================================================================================
McPhoto.verbosity                = -1              ## Solver verbosity
McPhoto.instantaneous            = true            ## Instantaneous transport or not
McPhoto.max_photons_per_cell     = 32              ## Maximum no. generated in a cell (<= 0 yields physical photons)
McPhoto.num_sampling_packets     = 1               ## Number of sub-sampling packets for max_photons_per_cell. Only for␣
↪instantaneous=true
```

(continues on next page)

(continued from previous page)

```
McPhoto.blend_conservation       = false          ## Switch for blending with the nonconservative divergence
McPhoto.transparent_eb           = false          ## Turn on/off transparent boundaries. Only for instantaneous=true
McPhoto.plt_vars                 = phi src phot   ## Available are 'phi' and 'src', 'phot', 'eb_phot', 'dom_phot', 'bulk_phot', 'src_
↪phot'
McPhoto.intersection_alg         = bisection      ## EB intersection algorithm. Supported are: 'raycast' 'bisection'
McPhoto.bisect_step              = 1.E-4          ## Bisection step length for intersection tests
McPhoto.bc_x_low                 = outflow        ## Boundary condition. 'outflow', 'symmetry', or 'wall'
McPhoto.bc_x_high                = outflow        ## Boundary condition
McPhoto.bc_y_low                 = outflow        ## Boundary condition
McPhoto.bc_y_high                = outflow        ## Boundary condition
McPhoto.bc_z_low                 = outflow        ## Boundary condition
McPhoto.bc_z_high                = outflow        ## Boundary condition
McPhoto.photon_generation        = deterministic  ## Volumetric source term. 'deterministic' or 'stochastic'
McPhoto.source_type              = number         ## 'number'      -> Source term contains the number of photons produced
                                                  # 'volume'      -> Source terms contains the number of photons produced per␣
↪unit volume
                                                  # 'volume_rate' -> Source terms contains the volumetric rate
                                                  # 'rate'        -> Source terms contains the rate
McPhoto.deposition               = cic            ## 'ngp'  -> nearest grid point, 'cic' -> cloud-in-cell
McPhoto.deposition_cf            = transition     ## 'interp', 'halo', 'halo_ngp', 'transition'.
```

Tip: The `McPhoto` class includes a hidden input parameter `McPhoto.dirty_sampling = true/false` which enables a cheaper sampling method for discrete photons when calling the `advance` method. The caveat is that the method does not incorporate boundary intersect, only works for instantaneous propagation, and avoids filling the data holders that are necessary for load balancing.

Clarifications

When computational photons are generated through the solver, users might have filled the source term differently depending on the application. For example, users might have filled the source term with the number of photons generated per unit volume and time, or the *physical* number of photons to be generated. The two input options `McPhoto.photon_generation` and `McPhoto.source_type` contain the necessary specifications for ensuring that the user-filled source term can be translated properly for ensuring that the correct number of physical photons are generated. Firstly, `McPhoto.source_type` contains the specification of what the source term contains:

- `number` if the source term contains the physical number of photons.
- `volume` if the source terms contains the physical number of photons generated per unit volume.
- `volume_rate` if the source terms contains the physical number of photons generated per unit volume and time.
- `rate` if the source terms contains the physical number of photons generated per unit time.

When `McPhoto` calculates the number of physical photons in a cell, it will automatically determine from `McPhoto.source_type`, ΔV and Δt how many physical photons are to be generated in each grid cell.

`McPhoto.photon_generation` permits the user to turn on/off Poisson sampling when determining how many photons will be generated. If this is set to *stochastic*, the solver will first compute the number of physical photons $\overline{N}_\gamma^{\mathrm{phys}}$ following the procedure above, and then run a Poisson sampling such that the final number of physical photons is

$$N_\gamma^{\mathrm{phys}} = P\left(\overline{N}_\gamma^{\mathrm{phys}}\right).$$

Otherwise, if `McPhoto.photon_generation` is set to *deterministic* then the solver will generate

$$N_\gamma^{\mathrm{phys}} = \overline{N}_\gamma^{\mathrm{phys}}$$

photons. Again, these elements are important because users might have chosen to perform the Poisson sampling outside of `McPhoto`.

Important: All of the above procedures are done *per-cell*.

4.6.4 Example application

Example applications that use `RtSolver` are found in:

- *Radiative transfer*.
- *CDR plasma model*.
- *Îto-KMC plasma model*.

4.7 Surface ODE solver

`chombo-discharge` provides a simple solver for ODE equations

$$\frac{\partial \vec{\phi}}{\partial t} = \vec{F},$$

where $\vec{\phi}$ represents N unknowns on the EB. Note that the underlying data type for $\vec{\phi}$ and $\vec{F}$ is `EBAMRIVData`, see *Mesh data*. Such a solver is useful, for example, as a surface charge solver where ϕ is the surface charge density and F is the charge flux onto the EB.

The surface charge solver is implemented as

```
/*!
  @brief Surface ODE solver
  @details This is a basic solver that acts as an ODE solver on cut-cells.
*/
template <int N = 1>
class SurfaceODESolver
```

where N indicates the number of variables stored in each cut cell. This solver is analogous to *Mesh ODE solver*, with the exception that variables are only stored on cut-cells.

4.7.1 Instantiation

To instantiate the solver, use one of the following constructors:

```
static_assert(N > 0, "SurfaceODESolver<N> must have N > 0");

/*!
  @brief Default constructor. Must subsequently set AmrMesh.
  @details Sets realm to primal and phase to phase::gas.
*/
SurfaceODESolver();

/*!
  @brief Full constructor.
  @details Sets AmrMesh to input and sets realm to primal and phase to phase::gas.
  @param[in] a_amr      AmrMesh reference
*/
SurfaceODESolver(const RefCountedPtr<AmrMesh>& a_amr);
```

The solver also requires a reference to *AmrMesh*, and the computational geometry such that a full instantiation example is

```
SurfaceODESolver<1>* solver = new SurfaceODESolver<1>();

solver->setAmr(...);
solver->setComputationalGeometry(...);
```

4.7.2 Setting $\vec{\phi}$

Mesh-based

To set $\vec{\phi}$ on the mesh, one can fetch the underlying data by calling

```
/*!
  @brief Get internal state
  @return Returns m_phi
*/
virtual EBAMRIVData&
getPhi() noexcept;
```

This returns a reference to the underlying data which is defined on all cut-cells. The user can then iterate through this data and set the values accordingly, see *Iterating over the AMR hierarchy*.

Constant value

To set the data directly, `SurfaceODESolver<N>` defines functions

```
/*!
  @brief Convenience function for setting m_phi
  @param[in] a_phi Values for all components
*/
virtual void
setPhi(const Real a_phi);

/*!
  @brief Convenience function for setting m_phi
  @param[in] a_phi Component-wise values.
*/
virtual void
setPhi(const std::array<Real, N>& a_phi);
```

4.7.3 Setting $\vec{F}$

In order to set the right-hand side of the equation, functions exist that are entirely analogous to the function signatures for setting $\vec{\phi}$:

```
/*!
  @brief Convenience function for setting m_rhs
  @param[in] a_rhs Values for all components
*/
virtual void
setRHS(const Real a_rhs);

/*!
  @brief Convenience function for setting m_rhs
  @param[in] a_rhs Component-wise values.
*/
virtual void
setRHS(const std::array<Real, N>& a_rhs);

/*!
  @brief Convenience function for setting m_rhs
  @param[in] a_rhs Values per cell and component.
  @note a_rhs must have N components.
*/
virtual void
```

(continues on next page)

(continued from previous page)

```
setRHS(const EBAMRIVData& a_rhs);

/*!
  @brief Get internal state
  @return Returns m_rhs
*/
virtual EBAMRIVData&
getRHS();
```

4.7.4 Resetting cells

SurfaceODESolver<N> has functions for setting values in the subset of the cut-cells representing dielectrics or electrodes. The information about whether or not a cut-cell is on the dielectric or electrode is passed in through *ComputationalGeometry*.

The function signatures are

```
/*!
  @brief Reset m_phi on electrode cells.
  @details This will set a_data to the specified value (in electrode cells)
  @param[in] a_val  Value
  @note Does not include ghost cells.
*/
virtual void
resetElectrodes(const Real a_value) noexcept;

/*!
  @brief Reset the input data holder on electrode cells.
  @details This will set a_data to the specified value (in electrode cells)
  @param[in] a_data Input data to be set.
  @param[in] a_val  Value
  @note Does not include ghost cells.
*/
virtual void
resetElectrodes(EBAMRIVData& a_phi, const Real a_value) const noexcept;

/*!
  @brief Reset m_phi on dielectric cells.
  @details This will set a_data to the specified value (in dielectric cells)
  @param[in] a_val  Value
  @note Does not include ghost cells.
*/
virtual void
resetDielectrics(const Real a_value) noexcept;

/*!
  @brief Reset the input data holder on dielectric cells.
  @details This will set a_data to the specified value (in dielectric cells)
  @param[in] a_data Input data to be set.
  @param[in] a_val  Value
  @note Does not include ghost cells.
*/
virtual void
resetDielectrics(EBAMRIVData& a_phi, const Real a_value) const noexcept;
```

Note that one can always call SurfaceODESolver<N>::getPhi() to iterate over other types of cell subsets and set the values from there.

4.7.5 Regridding

When regridding the `SurfaceODESolver<N>`, one must first call call

```
/*!
  @brief Pre-regrid function.
  @details This stores the data on the old mesh so it can be regridded later.
  @param[in] a_lbase          Coarsest level which will change during regrids.
  @param[in] a_oldFinestLevel Finest level before the regrid operation.
*/
virtual void
preRegrid(const int a_lbase, const int a_oldFinestLevel) noexcept;
```

This must be done *before* *AmrMesh* creates the new grids, and will store $\vec{\phi}$ on the old mesh. After *AmrMesh* has generated the new grids, $\vec{\phi}$ can be interpolated onto the new grids by calling

```
/*!
  @brief Regrid function.
  @param[in] a_lmin           Coarsest level where grids did not change.
  @param[in] a_oldFinestLevel Finest AMR level before the regrid.
  @param[in] a_newFinestLevel Finest AMR level after the regrid.
  @details This interpolates or coarsens conservatively, e.g. sigma_c = sum(A_f * sigma_f)/A_c if we coarsen.
*/
virtual void
regrid(const int a_lmin, const int a_oldFinestLevel, const int a_newFinestLevel) noexcept;
```

Note that when interpolating to the new grids one can choose to initialize data in the new cells using the value in the underlying coarse cells, i.e.

$$\vec{\phi}_{\mathbf{i}_{\text{fine}}} = \vec{\phi}_{\mathbf{i}_{\text{coarse}}}$$

Alternatively one can initialize the fine-grid data such that the area-weighted value of $\vec{\phi}$ is conserved, i.e.

$$\sum_{\mathbf{i}_{\text{fine}}} \alpha_{\mathbf{i}_{\text{fine}}} \Delta x_{\text{fine}}^{D-1} \vec{\phi}_{\mathbf{i}_{\text{fine}}} = \alpha_{\mathbf{i}_{\text{coar}}} \Delta x_{\text{coar}}^{D-1} \vec{\phi}_{\mathbf{i}_{\text{coar}}}$$

which gives

$$\vec{\phi}_{\mathbf{i}_{\text{fine}}} = r^{D-1} \frac{\alpha_{\mathbf{i}_{\text{coar}}}}{\sum_{\mathbf{i}_{\text{fine}}} \alpha_{\mathbf{i}_{\text{fine}}}} \vec{\phi}_{\mathbf{i}_{\text{coar}}},$$

where $\mathbf{i}_{\text{fine}}$ is set of cut-cells that occur when refining the coarse-grid cut-cell $\mathbf{i}_{\text{coar}}$ and r is the refinement factor between the two grid levels. In this case $\vec{\phi}$ is strictly conserved. Users can switch between these two methods by specifying the proper configuration option in the configuration file.

4.7.6 Input options

Several input options are available for configuring the run-time configuration of `MeshODESolver`, which are listed below

Listing 4.7.1: Input options for the `SurfaceODESolver<N>` class. All options are run-time configurable.

```
# ============================================================================================
# SurfaceODESolver solver settings.
# ============================================================================================
SurfaceODESolver.verbosity     = -1              ## Chattiness
SurfaceODESolver.regrid        = conservative    ## Regrid method. 'conservative' or 'arithmetic'
SurfaceODESolver.plt_vars      = phi             ## Plot variables. Valid arguments are 'phi' and 'rhs'
```

Note: `SurfaceODESolver` checkpoint files only contain $\vec{\phi}$.

4.8 Tracer particles

Tracer particles are particles that move along a prescribed velocity field

$$\frac{\partial \mathbf{X}}{\partial t} = \mathbf{V}$$

where $\mathbf{X}$ is the particle position and $\mathbf{V}$ is the particle velocity. The velocity is interpolated from a mesh-based field as

$$\mathbf{V} = \mathbf{v}\left(\mathbf{X}\right),$$

where $\mathbf{v}$ is a velocity field defined on the mesh. Such particles are useful, for example, for numerical integration along field lines.

Tip: The `chombo-discharge` tracer particle functionality resides in `$DISCHARGE_HOME/Source/TracerParticles`.

4.8.1 TracerParticleSolver

The tracer particle solver is templated as

```
/*!
  @brief Base class for a tracer particle solver. This solver can advance particles in a pre-defined velocity field.
  @details The user can set the velocity field through public member functions. This class is templated so the user can
  switch tracer particle implementations.

  This is a single-phase solver -- i.e. the particles only live on one of the phases. Extensions to multiphase
  is certainly possible.

  The template requirements on the particle type P are:
  1) It must contain a function RealVect& position() (derived from BinItem)
  2) It must contain a function const Real& weight() const.
  3) It must contain a function RealVect& velocity().
*/
template <typename P>
class TracerParticleSolver
```

where `P` is the particle type used for the solver. The template constraints on `P` are

1. It *must* contain a function `RealVect& position()`
2. It *must* contain a function `const Real& weight() const`
3. It *must* contain a function `RealVect& velocity()`.

Users are free to provide their own particle type provided that it meets these template constraints. However, we also define a plug-and-play particle class that meets these requirements, see *TracerParticle*.

Note: The `TracerParticleSolver<P>` API is available at https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classTracerParticleSolver.html.

4.8.2 TracerParticle

The `TracerParticle` type inherits from *GenericParticle* particle class and is templated as

```
/*!
  @brief A tracer particle class. This is templated for holding extra storage (useful for kernels).
  @details The template parameters M and N determine extra storage allocated to the particle. M determines
  the number of allocated scalars (Reals) and N determines the number of allocated vectors (RealVects). These
  quantities are communicated when remapping particles.
*/
template <size_t M, size_t N>
class TracerParticle : public GenericParticle<M, N>
```

The class also defines two more members; the weight and a particle velocity. These are accessible as

```
/*!
  @brief Get the particle "weight"
  @return m_weight
*/
inline Real&
weight();

/*!
  @brief Get the particle "weight"
  @return m_weight
*/
inline const Real&
weight() const;

/*!
  @brief Get the particle velocity.
  @return m_velocity
*/
inline RealVect&
velocity();

/*!
  @brief Get the particle velocity.
  @return m_velocity
*/
inline const RealVect&
velocity() const;
```

Note that, just as for `GenericParticle`, the template arguments `M` and `N` indicates the number of scalars and vectors allocated to the particle, see *GenericParticle*. These data fields can be used by applications for, e.g., storing integration variables (such as intermediate positions in a Runge-Kutta code).

4.8.3 Initialization

To initialize the solver, one can use the full constructor

```
/*!
  @brief Full contructor
  @param[in] a_amr      Handle to AmrMesh.
  @param[in] a_compGeom Computational geometry.
*/
TracerParticleSolver(const RefCountedPtr<AmrMesh>& a_amr, const RefCountedPtr<ComputationalGeometry> a_compGeom);
```

4.8.4 Getting the particles

To obtain the solver particles, simply call

```
/*!
  @brief Get all particles.
  @return m_particles
*/
virtual ParticleContainer<P>&
getParticles();

/*!
  @brief Get all particles. Const version.
  @return m_particles
*/
virtual const ParticleContainer<P>&
getParticles() const;
```

This returns the `ParticleContainer<P>` holding the particles, see *ParticleContainer*.

4.8.5 Setting $\mathbf{v}$

To set the velocity field on the mesh, use

```
/*!
  @brief Set the tracer particle velocity field.
  @param[in] a_velocityField Velocity field.
*/
virtual void
setVelocity(const EBAMRCellData& a_velocityField);
```

This will associate the input velocity `a_velocityField` with $\mathbf{v}$.

4.8.6 Interpolating velocities

To compute $\mathbf{V} = \mathbf{v}\left(\mathbf{X}\right)$ for all particles that reside in the solver, use

```
/*!
  @brief Interpolate particles velocities.
*/
virtual void
interpolateVelocities();
```

This will interpolate the velocities to the particle positions using the user-defined interpolation method (see *Input options*).

If desirable, one can also interpolate a scalar field defined on the mesh onto the particle weight by calling

```
/*!
  @brief Interpolate a scalar field onto the particle weight
*/
virtual void
interpolateWeight(const EBAMRCellData& a_scalar) noexcept;
```

The interpolation function is set by the user, see *Input options*. See *Particle-mesh* for further details.

4.8.7 Deposit particles

To deposit the particles, call

```
/*!
  @brief Deposit particle weight on mesh.
  @param[out] a_phi Deposited weight.
*/
virtual void
deposit(EBAMRCellData& a_phi) const noexcept;
```

This will deposit the particle weights onto the input data holder.

The deposition function is set by the user, see *Input options*. Complete details regarding how the deposition functions work is available in *Particle-mesh*.

4.8.8 Input options

Available input options for the tracer particle solver are given in the listing below.

Listing 4.8.1: List on configuration options for `TracerParticleSolver<P>`. All options are run-time configurable.

```
# ================================================================================================
# TracerParticleSolver class options
# ================================================================================================
TracerParticleSolver.verbosity         = -1           ## Solver verbosity level.
TracerParticleSolver.deposition        = cic          ## Deposition method. Must be 'ngp' or 'cic'
TracerParticleSolver.interpolation     = cic          ## Interpolation method. Must be 'ngp' or 'cic'
TracerParticleSolver.deposition_cf     = transition   ## 'interp', 'halo', 'halo_ngp', 'transition'.
TracerParticleSolver.plot_weight       = true         ## Turn on/off plotting of the particle weight.
TracerParticleSolver.plot_velocity     = true         ## Turn on/off plotting of the particle velocities.
TracerParticleSolver.volume_scale      = false        ## If true, depositions yield density * volume instead of just volume
```

CHAPTER

FIVE

MULTI-PHYSICS APPLICATIONS

5.1 CDR plasma model

In the CDR plasma model we are solving

$$\begin{aligned}\nabla\cdot(\epsilon_r\nabla\Phi) &= -\frac{\rho}{\epsilon_0},\\ \frac{\partial\sigma}{\partial t} &= F_\sigma,\\ \frac{\partial n}{\partial t} + \nabla\cdot(\mathbf{v}n - D\nabla n) &= S,\end{aligned} \tag{5.1.1}$$

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 (see *RtSolver*). Diffusive RTE methods (see *EddingtonSP1*) 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 (see *Monte Carlo solver*). In general, discrete photon methods incorporate better physics (like shadows). They can also be more easily adapted to 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

$$\begin{aligned}\epsilon_r &= \epsilon_r(\mathbf{x}),\\ \mathbf{v} &= \mathbf{v}\left(t,\mathbf{x},\mathbf{E},n\right),\\ D &= \mathbf{v}\left(t,\mathbf{x},\mathbf{E},n\right),\\ S &= S\left(t,\mathbf{x},\mathbf{E},\nabla\mathbf{E},n,\nabla n,\Psi\right),\\ \eta &= \eta\left(t,\mathbf{x},\mathbf{E},n\right),\\ F &= F(t,\mathbf{x},\mathbf{E},n),\end{aligned} \tag{5.1.2}$$

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.

Tip: 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.

See CdrPlasmaStepper for the overall C++ API.

This module uses the following solvers:

1. Advection-diffusion-reaction solver, *CdrSolver*.
2. Electrostatics solvers, *FieldSolver*.
3. Radiative transfer solver (either Monte-Carlo or continuum approximation), *RtSolver*.
4. Surface charge solver, see *Surface ODE solver*.

5.1.1 Time discretizations

Here, we discuss two discretizations of Eq. 5.1.1. 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 [2015] 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 semi-implicit scheme is used, it is not necessary to restrict the time step by the dielectric relaxation time.

Solver configuration

`CdrPlasmaGodunovStepper` contains several options that can be configured when running the solver, which are listed below:

```
# ================================================================================================
# CdrPlasmaGodunovStepper options
# ================================================================================================
CdrPlasmaGodunovStepper.verbosity             = -1              ## Class verbosity
CdrPlasmaGodunovStepper.solver_verbosity      = -1              ## Individual solver verbosities
CdrPlasmaGodunovStepper.min_dt                = 0.              ## Minimum permitted time step
CdrPlasmaGodunovStepper.max_dt                = 1.E99           ## Maximum permitted time step
CdrPlasmaGodunovStepper.cfl                   = 0.8             ## CFL number
CdrPlasmaGodunovStepper.use_regrid_slopes     = false           ## Use slopes when regridding (or not)
CdrPlasmaGodunovStepper.filter_rho            = 0               ## Number of filterings of space charge
CdrPlasmaGodunovStepper.filter_compensate     = false           ## Use compensation step after filter or not
CdrPlasmaGodunovStepper.field_coupling        = semi_implicit   ## Field coupling. 'explicit' or 'semi_implicit'
CdrPlasmaGodunovStepper.advection             = muscl           ## Advection algorithm. 'euler', 'rk2', or 'muscl'
CdrPlasmaGodunovStepper.diffusion             = explicit        ## Diffusion. 'explicit', 'implicit', or 'auto'.
CdrPlasmaGodunovStepper.diffusion_thresh      = 1.2             ## Diffusion threshold. If dtD/dtA > this then we use␣
↪implicit diffusion.
CdrPlasmaGodunovStepper.diffusion_order       = 2               ## Diffusion order.
CdrPlasmaGodunovStepper.relax_time            = 100.            ## Relaxation time. 100 <= is usually a "safe" choice.
CdrPlasmaGodunovStepper.fast_poisson          = 1               ## Solve Poisson every this time steps. Mostly for debugging.
CdrPlasmaGodunovStepper.fast_rte              = 1               ## Solve RTE every this time steps. Mostly for debugging.
CdrPlasmaGodunovStepper.fhd                   = false           ## Set to true if you want to add a stochastic diffusion flux
CdrPlasmaGodunovStepper.source_comp           = interp          ## Interpolated interp, or upwind X for species X
CdrPlasmaGodunovStepper.floor_cdr             = true            ## Floor CDR solvers to avoid negative densities
CdrPlasmaGodunovStepper.debug                 = false           ## Turn on debugging messages. Also monitors mass if it was␣
↪injected into the system.
CdrPlasmaGodunovStepper.profile               = false           ## Turn on/off performance profiling.
```

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.

5.1.2 CdrPlasmaPhysics

Overview

CdrPlasmaPhysics is an abstract class which represents the plasma physics for the CDR plasma module, i.e. it provides the coupling functions in Eq. 5.1.2. 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 by using a JSON schema for description the physics.

Complete API

The full API for the CdrPlasmaPhysics class is given below:

```
/*!
  @brief Abstract base class for specifying plasma kinetics. This is the base class used by CdrPlasmaStepper when advancing␣
↪the minimal plasma model.
*/
class CdrPlasmaPhysics
{
public:
  /*!
    @brief Default constructor. Does nothing.
  */
  CdrPlasmaPhysics()
  {}

  /*!
    @brief Base destructor. Does nothing.
  */
  virtual ~CdrPlasmaPhysics()
  {}

  /*!
    @brief Parse run-time options
  */
  virtual void
  parseRuntimeOptions() {

  };

  /*!
    @brief Get number of plot variables for this physics class.
    @details This is used by CdrPlasmaStepper for pre-allocating data that will be put in a plot file. When overriding
    this method then this routine should return the number of plot variables that will be plotted. The returned number
    should thus be the same as the vectors that are returned from getPlotVariableNames and getPlotVariables.
  */
  virtual int
  getNumberOfPlotVariables() const
  {
    return 0;
  }

  /*!
    @brief Get plot variable names. The
    @details This function will return the plot variable names that CdrPlasmaPhysics will plot. The length of the returned
    vector must be the same as the returned value from getNumberOfPlotVariables. Note that the names/positions between this
    routine and getPlotVariables should be consistent.
  */
  virtual Vector<std::string>
  getPlotVariableNames() const
  {
    return Vector<std::string>(this->getNumberOfPlotVariables(), std::string("empty data"));
  }

  /*!
    @brief Provide plot variables. This is used by CdrPlasmaStepper when writing plot files.
    @details The length of the returned vector should be the same as the returned value from getNumberOfPlotVariables. The␣
↪names
    for the returned variables are given by the returned vector in getPlotVariableNames().
    @param[in]  a_cdrDensities  Grid-based density for particle species.
    @param[in]  a_cdrGradients  Grid-based gradients for particle species.
    @param[in]  a_rteDensities  Grid-based densities for photons.
    @param[in]  a_E             Electric field.
    @param[in]  a_pos           Position in space.
    @param[in]  a_dx            Grid resolution.
```

(continues on next page)

(continued from previous page)

```
   @param[in]  a_dt            Advanced time.
   @param[in]  a_time          Current time.
   @param[in]  a_kappa         Grid cell unit volume.
 */
 virtual Vector<Real>
 getPlotVariables(const Vector<Real>     a_cdrDensities,
                  const Vector<RealVect> a_cdrGradients,
                  const Vector<Real>     a_rteDensities,
                  const RealVect         a_E,
                  const RealVect         a_position,
                  const Real             a_dx,
                  const Real             a_dt,
                  const Real             a_time,
                  const Real             a_kappa) const
 {
   return Vector<Real>(this->getNumberOfPlotVariables(), 1.0);
 }

 /*!
   @brief Compute alpha. Should return Townsend ionization coefficient.
   @details This function is mostly used for the cell tagging classes
   @param[in] a_E        Electric field.
   @param[in] a_position Physical coordinates
 */
 virtual Real
 computeAlpha(const Real a_E, const RealVect a_position) const = 0;

 /*!
   @brief Compute eta. Should return Townsend attachment coefficient.
   @details This function is mostly used for the cell tagging classes
   @param[in] a_E        Electric field.
   @param[in] a_position Physical coordinates
 */
 virtual Real
 computeEta(const Real a_E, const RealVect a_position) const = 0;

 /*!
   @brief Routine intended for advancing a reaction network over a time a_dt.
   @details This routine assumes that the subsequent advance is in the form phi^(k+1) = phi^k + S*a_dt. Thus, this routine␣
↪exists such that users can EITHER
   fill a_cdrSources and a_rteSources directly with an explicit rule, OR they can perform a fully implicit advance within␣
↪this routine and set S from that.
   @param[out] a_cdrSources    Source terms for CDR equations.
   @param[out] a_rteSources    Source terms for RTE equations.
   @param[in]  a_cdrDensities  Grid-based density for particle species.
   @param[in]  a_cdrGradients  Grid-based gradients for particle species.
   @param[in]  a_rteDensities  Grid-based densities for photons.
   @param[in]  a_E             Electric field.
   @param[in]  a_pos           Position in space.
   @param[in]  a_dx            Grid resolution.
   @param[in]  a_dt            Advanced time.
   @param[in]  a_time          Current time.
   @param[in]  a_kappa         Grid cell unit volume.
 */
 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;

 /*!
   @brief Compute velocities for the CDR equations
   @param[in] a_time         Time
   @param[in] a_pos          Position
   @param[in] a_E            Electric field
   @param[in] a_cdrDensities CDR densities
   @return Returns the drift velocities for each CDR species. The vector ordering is the same as m_cdrSpecies.
 */
 virtual Vector<RealVect>
 computeCdrDriftVelocities(const Real         a_time,
                           const RealVect     a_pos,
```

(continues on next page)

(continued from previous page)

```
                          const RealVect     a_E,
                          const Vector<Real> a_cdrDensities) const = 0;

  /*!
    @brief Compute diffusion coefficients for the CDR equations.
    @param[in] a_time         Time
    @param[in] a_pos          Position
    @param[in] a_E            Electric field
    @param[in] a_cdrDensities CDR densities
    @return Returns the diffusion coefficients for each CDR species. The vector ordering is the same as m_cdrSpecies.
  */
  virtual Vector<Real>
  computeCdrDiffusionCoefficients(const Real         a_time,
                                  const RealVect     a_pos,
                                  const RealVect     a_E,
                                  const Vector<Real> a_cdrDensities) const = 0;

  /*!
    @brief Compute CDR fluxes on electrode-gas interfaces. This is used as a boundary condition in the CDR equations.
    @param[in] a_time            Time
    @param[in] a_pos             Position
    @param[in] a_normal          Boundary normal vector. This points into the gas phase.
    @param[in] a_E               Electric field
    @param[in] a_cdrVelocities   CDR velocities. Normal component only.
    @param[in] a_cdrDensities    CDR densities.
    @param[in] a_cdrGradients    Normal gradients of cdr densities
    @param[in] a_rteFluxes       RTE fluxes (normal component only)
    @param[in] a_extrapCdrFluxes Extrapolated fluxes from the gas side.
    @return Returns the flux on an electrode interface cell. The vector ordering must be the same as m_cdrSpecies.
    @note A positive flux is a flux INTO the domain.
  */
  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;

  /*!
    @brief Compute CDR fluxes on dielectric-gas interfaces. This is used as a boundary condition in the CDR equations.
    @param[in] a_time            Time
    @param[in] a_pos             Position
    @param[in] a_normal          Normal vector. This points into the gas phase.
    @param[in] a_E Electric      field
    @param[in] a_cdrDensities    CDR densities (on the EB)
    @param[in] a_cdrVelocities   Normal component of CDR velocities (on the EB).
    @param[in] a_cdrGradients    Normal gradients of cdr densities
    @param[in] a_rteFluxes       RTE fluxes (normal component only)
    @param[in] a_extrapCdrFluxes Extrapolated fluxes from the gas side.
    @return Returns the flux on a dielectric interface cell. The vector ordering must be the same as m_cdrSpecies.
    @note A positive flux is a flux INTO the domain.
  */
  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;

  /*!
    @brief Compute CDR fluxes through domain sides. This is used as a boundary condition in the CDR equations.
    @param[in] a_time            Time
    @param[in] a_pos             Position
    @param[in] a_dir             Direction (0 = x, 1=y etc)
    @param[in] a_side            Side (low or high side)
    @param[in] a_E               Electric field
    @param[in] a_cdrDensities    CDR densities.
    @param[in] a_cdrVelocities   CDR velocities (normal component only).
    @param[in] a_cdrGradients    CDR gradients (normal component only)
    @param[in] a_rteFluxes       RTE fluxes (normal component only)
```

(continues on next page)

(continued from previous page)

```
  @param[in] a_extrapCdrFluxes Extrapolated fluxes from the gas side.
  @note A positive flux is a flux INTO the domain.
*/
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,
                       const Vector<Real>   a_rteFluxes,
                       const Vector<Real>   a_extrapCdrFluxes) const = 0;

/*!
  @brief Set the initial surface charge
  @param[in] a_time Time
  @param[in] a_pos Position
*/
virtual Real
initialSigma(const Real a_time, const RealVect a_pos) const = 0;
```

5.1.3 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 *Setting up a new problem*). 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 solver*.

The user must use one of the following:

- 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.

- Alternatively, 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 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 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

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

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

$$\rho(h) = \frac{p'(h) N_{\mathrm{A}}}{R T(h)}$$

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

Initial data can be provided with

- Function based densities.
- Computational particles (deposited using a nearest-grid-point scheme).

Density functions

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 *LookupTable1D*). 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.

Initial particles

Initial particles can be included with the `initial particles` field. The current implementation supports

- `uniform` For drawing initial particles randomly distributed inside a box. The user must specify the two corners `lo corner` and `hi corner` that indicate the spatial extents of the box, and the `number` of computational particles to draw. The weight is specified by a field `weight`. For example:

```
{"plasma species":
  [
    {
      "name": "e",
      "Z":  -1,
      "mobile": true,
      "diffusive": true,
      "initial particles": {
        "uniform": {
          "lo corner": [0,0,0],
          "hi corner": [1,1,1],
          "number": 100,
          "weight": 1.0
        }
      }
    }
  ]
}
```

- `sphere` For drawing initial particles randomly distributed inside a sphere. Mandatory fields are
  - `center` for specifying the sphere center.
  - `radius` for specifying the sphere radius.
  - `number` for the number of computational particles.
  - `weight` for the initial particle weight.

```
{"plasma species":
  [
    {
      "name": "e",
      "Z":  -1,
      "mobile": true,
      "diffusive": true,
      "initial particles": {
        "sphere": {
          "center": [0,0,0],
          "radius": 1.0,
          "number": 100,
          "weight": 1.0
        }
      }
    }
  ]
}
```

- `copy` For using an already initialized particle distribution. The only mandatory fields is `copy`, e.g.

```
{"plasma species":
[
  {
    "name": "e",
    "Z":  -1,
    "mobile": true,
    "diffusive": true,
    "initial particles": {
      "sphere": {
        "center": [0,0,0],
        "radius": 1.0,
        "number": 100,
        "weight": 1.0
      }
    }
```

(continues on next page)

(continued from previous page)

```
  },
  {
    "name": "O2+",
    "Z": 1,
    "mobile": true,
    "diffusive": true,
    "initial particles": {
       "copy": "e"
    }
  }
]}
```

This will copy the particles from the species `e` to the species `O2+`.

Warning: The species one copies from must be defined *before* the species one copies *to*.

Complex example

For example, a species with complex initial data that combines density functions with initial particles can look like:

```
{"plasma species":
  [
    {
      "name": "N2+",
      "Z":  1,
      "mobile": false,
      "diffusive": false,
      "initial data": {
        "uniform": 1E10,
        "gauss2" :
          [
            {
               "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
        }
      },
      "initial particles": {
        "sphere": {
          "center": [0,0,0],
          "radius": 1.0,
```

(continues on next page)

(continued from previous page)

```
          "number": 100,
          "weight": 1.0
        }
      }
    }
  ]
}
```

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 *LookupTable1D*.

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 *LookupTable1D*.

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"
```

(continues on next page)

(continued from previous page)

```
    }
  }
 ]
}
```

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.* [2007] 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 [2008]. 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

$$R = kn_An_B\ldots$$

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"],
```

(continues on next page)

(continued from previous page)

```
      "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 $N_2 + N_2 + O_2 \rightarrow N_4^+ + O_2$. It is not possible to set different reaction rates for the two reactions.

Specifying reaction rates

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
    }
  ]
}
```

Single-temperature rates

- `functionT A` To set a rate dependent on a single species temperature in the form $k(T) = c_1 T^{c_2}$, set `lookup` to `functionT A`. The user must specify the species from which we compute the temperature T by including a field `T`. The constants c_1 and c_2 must also be included.

  For example, in order to add a reaction $e + O_2 \rightarrow \varnothing$ with rate $k = 1.138 \times 10^{-11} T_e^{-0.7}$ we can add the following:

  ```
  {"plasma reactions":
    [
      {
        "reaction": "e + M+ ->",
        "lookup": "functionT A",
        "T": "e",
        "c1": 1.138
        "c2": -0.7
      }
    ]
  }
  ```

Two-temperature rates

- `functionT1T2 A` 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 $e + O_2 + O_2 \rightarrow O_2^- + O2$ in the set, with this reaction having a rate

  $$k = 2.4 \times 10^{-41} \left(\frac{T_{O_2}}{T_e}\right),$$

  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
    }
  ]
}
```

Townsend ionization and attachment

To set standard Townsend ionization and attachment reactions, set `lookup` to `alpha*v` and `eta*v`, respectively. This will compute the rate constant $k = \alpha |\mathbf{v}|$ where $\mathbf{v}$ is the drift velocity of some species. To specify the species one includes the field `species`.

For example, to include the reactions $e \rightarrow e + e + M^+$ and $e \rightarrow M^-$ one can specify the reactions as

```
{"plasma reactions":
  [
    {
      "reaction": "e -> e + e + M+",
      "lookup": "alpha*v",
      "species": "e"
    },
    {
      "reaction": "e -> M-",
      "lookup": "eta*v",
      "species": "e"
    }
  ]
}
```

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,
      "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 *Mobilities*), 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.

Modifying reactions

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 [2009]. 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 [2009].

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

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.

Warnings and caveats

Note that the JSON interface *always* computes reactions as if they were specified by the deterministic reaction rate equation

$$\partial_t n_i = \sum_r k_r n_j n_k n_l \ldots,$$

where the fluid source term for any reaction r is $S_r = k_r n_j n_k n_l \ldots$. Caution should always be exercised when defining a reaction set.

Higher-order reactions

Usually, many rate coefficients depend on the output of other software (e.g., BOLSIG+) and the scaling of rate coefficients is not immediately obvious. This is particularly the case for three-body reactions with BOLSIG+ that may require scaling before running the Boltzmann solver (by scaling the input cross sections), or after running the Boltzmann solver, in which case the rate coefficients themselves might require scaling. In any case the user should investigate the cross-section file that BOLSIG+ uses, and figure out the required scaling.

Important: For two-body reactions, e.g. $A + B \rightarrow \varnothing$ the rate coefficient must be specified in units of $\mathrm{m^3 s^{-1}}$, while for three-body reactions $A + B + C \rightarrow \varnothing$ the rate coefficient must have units of $\mathrm{m^6 s^{-1}}$.

For three-body reactions the units given by BOLSIG+ in the output file may or may not be incorrect (depending on whether or not the user scaled the cross sections).

Townsend coefficients

Townsend coefficients are not fundamentally required for specifying the reactions, but as with the higher-order reactions some of the output rates for three-body reactions might be inconsistently represented in the BOLSIG+ output files. For example, some care might be required when using the Townsend attachment coefficient for air when the reaction $\mathrm{e} + \mathrm{O_2} + \mathrm{O_2} \rightarrow \mathrm{O_2^-} + \mathrm{O_2}$ is included because the rate constant might require proper scaling after running the Boltzmann solver, but this scaling is invisible to the BOLSIG+'s calculation of the attachment coefficient η/N.

Warning: The JSON interface *does not guard* against inconsistencies in the user-provided chemistry, and provision of inconsistent η/N and attachment reaction rates are quite possible.

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 *Plasma reactions*. 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
    }
  ]
```

5.1.4 Adaptive mesh refinement

The AMR functionality for `CdrPlasmaStepper` is implemented by subclassing `CellTagger` through a series of intermediate classes. Currently, we mostly use the `CdrPlasmaStreamerTagger` class for tagging cells based on the evaluation of the Townsend ionization coefficient as

$$(\alpha - \eta)\,\Delta x > \epsilon,$$

where ϵ is a refinement threshold. A similar threshold is used when coarsening grid cells.

Tip: See https://chombo-discharge.github.io/chombo-discharge/doxygen/html/classPhysics_1_1CdrPlasma_1_1CdrPlasmaStreamerTagger.html for the C++ API for `CdrPlasmaStreamerTagger`.

5.1.5 Setting up a new problem

New problems that use the `CdrPlasma` physics model are best set up by using the Python tools provided with the module. A full description is available in the `README.md` file contained in the folder:

```
# Physics/CdrPlasma
This physics module solves the minimal plasma model

The folders contains

* **PlasmaModels** Implementations of various plasma models.
* **TimeSteppers** Implementations of the various time integration algorithms.
* **CellTaggers** Implementations of various cell taggers.
* **Data** Various data (e.g. transport data).


## Setting up a new application
To set up a new problem, use the Python script. For example:

```shell
./setup.py -base_dir=MyApplications -app_name=MyPlasmaModel -geometry=CoaxialCable
```

The application will be installed to $DISCHARGE_HOME/MyApplications/MyPlasmaApplication
The user will need to modify the geometry and set the initial conditions through the inputs file.

Modifying the application
Users are free to modify this application, e.g. adding new initial conditions and flow fields.
```

To see the list of available options type

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

5.2 Discharge inception model

5.2.1 Overview

The discharge inception model computes the inception voltage and probability of discharge inception for arbitrary geometries and voltage forms. For estimating the streamer inception, the module resolves the electron avalanche integral

$$K\left(\mathbf{x}\right) = \int_{\mathbf{x}}^{\text{until } \alpha_{\text{eff}} \leq 0} \alpha_{\text{eff}}(E, \mathbf{x}')\mathrm{d}l \tag{5.2.1}$$

where $E = |\mathbf{E}|$ and where $\alpha_{\text{eff}}(E, \mathbf{x}) = \alpha(E, \mathbf{x}) - \eta(E, \mathbf{x})$ is the effective ionization coefficient. The integration runs along electric field lines.

The discharge inception model solves for $K = K\left(\mathbf{x}\right)$ for all $\mathbf{x}$, i.e., the inception integral is evaluated for all starting positions of the first electron. This differs from the conventional approach where the user will first extract electric field lines for post-processing. Note that $K\left(\mathbf{x}\right) = K(\mathbf{x}; U)$ where U is the applied voltage.

In addition to the above, the user can specify a critical threshold value for K_c which is used for computing

- The *critical volume* $V_c = \int_{K \geq K_c} \mathrm{d}V$.
- The inception voltage U_c.
- The probability of having the first electron in the critical volume, $dP(t, t + \Delta t)$.

The discharge inception model is implemented through the following files:

- `DischargeInceptionStepper`, which implements *TimeStepper*.
- `DischargeInceptionTagger`, which implements *CellTagger* and flags cells for refinement and coarsening.
- `DischargeInceptionSpecies`, which implements *CdrSpecies* for the negative ion transport description.

Tip: The source code for the discharge inception model is given in `$DISCHARGE_HOME/Physics/DischargeInception`. See DischargeInceptionStepper for the C++ API for this time stepper.

Townsend criterion

One may also solve for the Townsend inception criterion, which is formulated as follows:

$$T\left(\mathbf{x}\right) = \gamma\left(\exp\left[K\left(\mathbf{x}\right)\right] - 1\right) \geq 1.$$

The interpretation of this criterion is that each starting electron produces $\exp\left[K\left(\mathbf{x}\right)\right] - 1$ secondary electron-ion pairs. The residual ions will drift towards cathode surfaces and generate secondary ionization with a user-supplied efficiency $\gamma = \gamma\left(E, \mathbf{x}\right)$.

The discharge inception model can be run in two modes:

- A stationary mode, where one only calculations $K\left(\mathbf{x}\right)$ for a range of voltages, see *Stationary mode*.
- In transient mode, where $K\left(\mathbf{x}\right)$ is computed dynamically according to a user-supplied voltage shape. This mode can also be used to evaluate the inception probability for a given voltage curve, see *Transient mode*.

Implementation

The discharge inception model is implemented in `$DISCHARGE_HOME/Physics/DischargeInception` as

```
/*!
  @brief Class for streamer inception integral evaluations.
  @details P is the tracer particle type
  F is the field solver type.
  C is the convection-diffusion-reaction solver type.
*/
template <typename P = TracerParticle<2, 3>, typename F = FieldSolverGMG, typename C = CdrCTU>
class DischargeInceptionStepper : public TimeStepper
```

The template template parameters indicate which types of solvers to used within the compound algorithm. The template parameters indicate the following:

- `typename P` is the tracer particle solver (see *TracerParticleSolver*) for reconstructing the inception integral.
- `typename F` is the field solver (see *FieldSolver*) that is used when computing the electric field.
- `typename C` is the convection diffusion reaction solver (see *CdrSolver*) to use when resolving negative ion transport.

5.2.2 Stationary mode

The stationary mode resolves Eq. 5.2.1 for a range of voltages, including both positive and negative polarities. This procedure occurs through the following steps:

1. Solve for the background field with an applied voltage $U = 1$ on all live electrodes.
2. Obtain the critical field by obtaining the root $\alpha_{\text{eff}}(E) = 0$. This step is performed using Brent's method.
3. Scale the voltage to the lowest voltage that exceeds the critical field.
4. Solve for $K(\mathbf{x})$ from the lowest voltage that exceeds the critical field until a user-specified threshold. These calculations are performed for both polarities.

On output, the user is provided with the following:

- K values for positive and negative voltages, as well as the Townsend criterion.
- Critical volumes and surfaces.
- Inception voltage, both for streamer inception and Townsend inception, for both polarities.
- The *critical position*, i.e., the position corresponding to $K(\mathbf{x})$.

5.2.3 Transient mode

In transient mode the user can apply a voltage curve $U = U(t)$ reconstruct the K value at each time step, and recompute the critical volume so that we obtain

$$\begin{aligned} K &= K(\mathbf{x}, t) \\ T &= T(\mathbf{x}, t) \\ V_c &= V_c(t). \end{aligned}$$

The rationale for computing the above quantities is that one may describe the availability of negative ions within the critical volumes by augmenting the model with an ion transport description. Furthermore, if one also has a description of the electron detachment rate, one may then evaluate the probability that an electron detaches from a negative ion inside of the critical region.

Negative ion transport

We also assume that ions move as drifting Brownian walkers in the electric field (see *Îto diffusion*). This can be written in the fluctuating hydrodynamics limit as an evolution equation for the ion density

$$\frac{\partial n_-}{\partial t} = -\nabla\cdot(\mathbf{v}n_-) + \nabla\cdot(D\nabla n_-) + \sqrt{2Dn_-}\mathbf{Z},$$

where $\mathbf{Z}$ represents uncorrelated Gaussian white noise. Note that the above equation is a mere rewrite of the Ito process for a collection of particles; it is not really useful per se since it is a tautology for the original Ito process.

However, we are interested in the average ion distribution over many experiments, so by taking the ensemble average we obtain a regular advection-diffusion equation for the evolution of the negative ion distribution $\langle n_-\rangle$:

$$\frac{\partial \langle n_-\rangle}{\partial t} = -\nabla\cdot(\mathbf{v}\langle n_-\rangle) + \nabla\cdot(D\nabla\langle n_-\rangle)\,.$$

This equation is sensible only when $\langle n_-\rangle$ is interpreted as an ion density distribution (over many identical experiments).

Inception probability

The probability of discharge inception in a time interval $[t, t+\mathrm{d}t]$ is given by

$$\mathrm{d}P(t) = \left[1 - P\left(t\right)\right]\lambda(t)\mathrm{d}t,$$

where $\lambda(t)$ is a placeholder for the electron generation rate within the critical volume. The exact solution for $P(t)$ is

$$P(t) = 1 - \exp\left(-\int_{-\infty}^{t}\lambda\left(t'\right)\mathrm{d}t'\right).$$

An expression for the electron generation rate can be given by

$$\lambda(t) = \int_{V_c(t)}\left\langle\frac{\partial n_e}{\partial t}\right\rangle\left(1-\frac{\eta}{\alpha}\right)\mathrm{d}V + \int_{A_c(t)}\frac{\langle j_e\rangle}{e}\left(1-\frac{\eta}{\alpha}\right)\mathrm{d}A,$$

Inserting the expression for λ and integrating for $P(t)$ yields

$$P(t) = 1 - \exp\left[-\int_0^t\left(\int_{V_c(t')}\left\langle\frac{dn_\mathrm{e}}{dt'}\right\rangle\left(1-\frac{\eta}{\alpha}\right)\mathrm{d}V + \int_{A_c(t')}\frac{j_e}{q_\mathrm{e}}\left(1-\frac{\eta}{\alpha}\right)\mathrm{d}A\right)\mathrm{d}t'\right]. \tag{5.2.2}$$

Here, $\left\langle\frac{dn_\mathrm{e}}{dt}\right\rangle$ is the electron production rate from both background ionization and electron detachment, i.e.

$$\left\langle\frac{dn_\mathrm{e}}{dt}\right\rangle = S_\mathrm{bg} + k_d\left\langle n_-\right\rangle,$$

where S_bg is the background ionization rate set by the user, k_d is the negative ion detachment rate, and $\langle n_-\rangle$ is the negative ion distribution. The second integral is due to electron emission from the cathode and into the critical volume. Note that, internally, we always ensure that $j_\mathrm{e}dA$ evaluates to zero on anode surfaces.

Statistical time lag

We also compute the probability of a first electron appearing in the time interval $[t, t+\Delta t]$, given by

$$\Delta P(t, t+\Delta t) = \left[1 - P(t)\right]\left(\int_{V_c(t')}\left\langle\frac{dn_\mathrm{e}}{dt'}\right\rangle\left(1-\frac{\eta}{\alpha}\right)\mathrm{d}V + \int_{A_c(t')}\frac{j_e}{q_\mathrm{e}}\left(1-\frac{\eta}{\alpha}\right)\mathrm{d}A\right)\Delta t \tag{5.2.3}$$

When running in transient mode the user must set the voltage curve, and pay particular caution to setting the initial ion density, mobility, and detachment rates.

The statistical time lag, or average waiting time for the first electron, is available from the computed data, and is given by integrating $t\mathrm{d}P$, which yields

$$\tau = \frac{1}{P(t)} \int_0^\infty t\left[1 - P(t)\right] \lambda(t) \mathrm{d}t.$$

Other derived values (such as the standard deviation of the waiting time) is also available, and can be calculated from the $P(t)$ and $\lambda(t)$ similar to the procedure above. Numerically, this is calculated using the trapezoidal rule.

5.2.4 Input data

The input to the discharge inception model are:

1. Streamer inception threshold.
2. Townsend ionization and attachment coefficient.
3. Background ionization rate (e.g., from cosmic radiation).
4. Electron detachment rate from negative ions.
5. Negative ion mobility and diffusion coefficients.
6. Initial negative ion density.
7. Secondary emission coefficients.
8. Voltage curve (for transient simulations).
9. Space and surface charge if resolving a space-charge influenced field.

The input data to the discharge inception model is done by passing in C++-functions to the class. These functions are mainly in the forms of an `std::function`, so the user can pass in function pointers or lambdas for configuring the behavior of the model. The relevant user API for setting the above variables are listed below.

```
/*!
  @brief Set the voltage curve (used for transient mode)
  @param[in] a_voltageCurve Voltage curve
*/
virtual void
setVoltageCurve(const std::function<Real(const Real& a_time)>& a_voltageCurve) noexcept;

/*!
  @brief Set space charge distribution
  @param[in] a_rho Space charge distribution
*/
virtual void
setRho(const std::function<Real(const RealVect& x)>& a_rho) noexcept;

/*!
  @brief Set surface charge distribution
  @param[in] a_sigma Surface charge distribution
*/
virtual void
setSigma(const std::function<Real(const RealVect& x)>& a_sigma) noexcept;

/*!
  @brief Set the negative ion density
  @param[in] a_density Negative ion density
*/
virtual void
setIonDensity(const std::function<Real(const RealVect x)>& a_density) noexcept;

/*!
  @brief Set the negative ion mobility (field-dependent)
  @param[in] a_mobility Negative ion  mobility
```

(continues on next page)

(continued from previous page)

```
  */
  virtual void
  setIonMobility(const std::function<Real(const Real E)>& a_mobility) noexcept;

  /*!
    @brief Set the negative ion diffusion coefficient (field-dependent)
    @param[in] a_diffCo Negative ion diffusion coefficient
  */
  virtual void
  setIonDiffusion(const std::function<Real(const Real E)>& a_diffCo) noexcept;

  /*!
    @brief Set the ionization coefficient.
    @param[in] a_alpha Townsend ionization coefficient. E is the field in SI units.
  */
  virtual void
  setAlpha(const std::function<Real(const Real& E, const RealVect& x)>& a_alpha) noexcept;

  /*!
    @brief Set the attachment coefficient.
    @param[in] a_eta Townsend attachment coefficient. E is the field in SI units.
  */
  virtual void
  setEta(const std::function<Real(const Real& E, const RealVect& x)>& a_eta) noexcept;

  /*!
    @brief Get ionization coefficient
  */
  virtual const std::function<Real(const Real& E, const RealVect& x)>&
  getAlpha() const noexcept;

  /*!
    @brief Get attachment coefficient
  */
  virtual const std::function<Real(const Real& E, const RealVect& x)>&
  getEta() const noexcept;

  /*!
    @brief Set the background ionization rate (e.g. from cosmic radiation etc).
    @param[in] a_backgroundRate Background ionization rate (units of 1/m^3 s).
  */
  virtual void
  setBackgroundRate(const std::function<Real(const Real& E, const RealVect& x)>& a_backgroundRate) noexcept;

  /*!
    @brief Set the detachment rate for negative ions.
    @details The total detachment rate is dn_e/dt = k*n_ion; this sets the constant k.
    @param[in] a_detachmentRate Detachment rate (in units of 1/s).
  */
  virtual void
  setDetachmentRate(const std::function<Real(const Real& E, const RealVect& x)>& a_detachmentRate) noexcept;

  /*!
    @brief Set the field emission current
    @param[in] a_fieldEmission Field emission current density.
  */
  virtual void
  setFieldEmission(const std::function<Real(const Real& E, const RealVect& x)>& a_currentDensity) noexcept;

  /*!
    @brief Set the secondary emission coefficient
    @param[in] a_coeff Secondary emission coefficient.
  */
  virtual void
  setSecondaryEmission(const std::function<Real(const Real& E, const RealVect& x)>& a_coeff) noexcept;
```

Tip: It is relatively simple to use tabulated data input for setting the input (e.g., the transport coefficients). See *LookupTable1D*.

5.2.5 Algorithms

The discharge inception model uses a combination of electrostatic field solves, Particle-In-Cell, and fluid advection for resolving the necessary dynamics. The various algorithms involved are discussed below.

Field solve

Since the background field scales linearly with applied voltage, we require only a single field solve at the beginning of the simulation. This field solve is done with an applied voltage of $U = 1\,\mathrm{V}$ and the electric field is then later scaled by the appropriate voltage.

Inception integral

We use a Particle-In-Cell method for computing the inception integral $K\left(\mathbf{x}\right)$ for an arbitrary electron starting position. All grid cells where $\alpha_{\mathrm{eff}} > 0$ are seeded with one particle on the cell centroid and the particles are then tracked through the grid. The particles move a user-specified distance along field lines $\mathbf{E}$, using first or second order integration. If a particle leaves through a boundary (EB or domain boundary), or enters a region $\alpha_{\mathrm{eff}} \leq 0$, the integration for that particle is stopped. Once all the particle integrations have halted, we rewind the particles back to their starting position and deposit their weight on the mesh, which provides us with $K = K\left(\mathbf{x}\right)$.

Note: When tracking positive ions for evaluation of the Townsend criterion, the same algorithms are used.

Euler

For the Euler rule the particle weight for a particle p the update rule is

$$\begin{aligned}\mathbf{X}_p^{k+1} &= \mathbf{X}_p^k - \hat{\mathbf{E}}\left(\mathbf{X}_p^k\right)\Delta x \\ w_p^{k+1} &= w_p^k + \alpha_{\mathrm{eff}}\left(\left|\mathbf{E}\left(\mathbf{X}_p^k\right)\right|, \mathbf{X}_p^k\right)\Delta x,\end{aligned}$$

where Δx is an integration length.

Trapezoidal

With the trapezoidal rule we first update

$$\mathbf{X}_p' = \mathbf{X}_p^k - \hat{\mathbf{E}}\left(\mathbf{X}_p^k\right)\Delta x,$$

which is followed by

$$\begin{aligned}\mathbf{X}_p^{k+1} &= \mathbf{X}_p^k + \frac{\Delta x}{2}\left[\hat{\mathbf{E}}\left(\mathbf{X}_p^k\right) + \hat{\mathbf{E}}\left(\mathbf{X}_p'\right)\right]. \\ w_p^{k+1} &= w_p^k + \frac{\Delta x}{2}\left[\alpha_{\mathrm{eff}}\left(\left|\mathbf{E}\left(\mathbf{X}_p^k\right)\right|, \mathbf{X}_p^k\right) + \alpha_{\mathrm{eff}}\left(\left|\mathbf{E}\left(\mathbf{X}_p'\right)\right|, \mathbf{X}_p'\right)\right]\end{aligned}$$

Step size selection

The permitted tracer particle step size is controlled by user-specified maximum and minimum space steps:

$\Delta_{\text{phys,min}}$: Minimum physical step size
$\Delta_{\text{phys,max}}$: Maximum physical step size
$\Delta_{\text{grid,min}}$: Minimum grid-cell step size
$\Delta_{\text{grid,max}}$: Maximum grid-cell step size
Δ_α : Avalanche length
$\Delta_{\nabla\alpha}$: Rate-of-change of ionization coefficient.

The particle integration step size is then selected according to the following heuristic:

$$\begin{aligned}
\Delta X &= \min\left(\Delta_\alpha \frac{1}{\overline{\alpha}}, \Delta_{\nabla\alpha}\frac{\overline{\alpha}}{|\nabla\overline{\alpha}|}\right)\\
\Delta X &= \min\left(\Delta X, \Delta_{\text{phys,max}}\right)\\
\Delta X &= \max\left(\Delta X, \Delta_{\text{phys,min}}\right)\\
\Delta X &= \min\left(\Delta X, \Delta_{\text{grid,max}}\Delta x\right)\\
\Delta X &= \max\left(\Delta X, \Delta_{\text{grid,min}}\Delta x\right).
\end{aligned}$$

These parameters are implemented through the following input options:

```
# Particle integration controls
DischargeInceptionStepper.min_phys_dx      = 1.E-10         ## Minimum permitted physical step size
DischargeInceptionStepper.max_phys_dx      = 1.E99          ## Maximum permitted physical step size
DischargeInceptionStepper.min_grid_dx      = 0.5            ## Minimum permitted grid step size
DischargeInceptionStepper.max_grid_dx      = 5.0            ## Maximum permitted grid step size
DischargeInceptionStepper.alpha_dx         = 5.0            ## Step size relative to avalanche length
DischargeInceptionStepper.grad_alpha_dx    = 0.1            ## Maximum step size relative to alpha/grad(alpha)
DischargeInceptionStepper.townsend_grid_dx = 2.0            ## Space step to use for Townsend tracking
```

Note that the input variable `townsend_grid_dx` determines the spatial step (relative to the grid resolution Δx) when tracking ions for the Townsend region reconstruction.

Critical volume

The critical volume is computed as

$$V_c = \int_{K(\mathbf{x})\geq K_c \cup \gamma\exp[K(\mathbf{x})]\geq 1} \mathrm{d}V.$$

Note that the critical volume is both voltage and polarity dependent.

Critical surface

The critical surface is computed as

$$A_c = \int_{K(\mathbf{x})\geq K_c \cup \gamma(\exp[K(\mathbf{x})]-1)\geq 1} \mathrm{d}A.$$

Note that the critical surface is both voltage and polarity dependent, and is non-zero only on cathode surfaces.

Inception voltage

The inception voltage for starting a critical avalanche can be computed in the stationary solver mode (see *Stationary mode*). This is done separately for the streamer and Townsend inception voltages.

Streamer inception

For streamer inception we use $K\left(\mathbf{x};U\right)$ for a range of voltages $U \in U_1, U_2, \ldots$ and (linearly) interpolate between these values. If two values of the K integral bracket K_c, i.e.

$$K_a = K\left(\mathbf{x};U_a\right) \leq K_c$$
$$K_b = K\left(\mathbf{x};U_b\right) \geq K_c$$

then we can estimate the inception voltage for a starting electron at position $\mathbf{x}$ through linear interpolation as

$$U_{\text{inc, streamer}}\left(\mathbf{x}\right) = U_a + \frac{K_c - K_a}{K_b - K_a}\left(U_b - U_a\right)$$

Townsend inception

A similar method to the one used above is used for the Townsend criterion, using e.g. $T\left(\mathbf{x};U\right) = \gamma\left(\exp\left[K\left(\mathbf{x};U\right)\right] - 1\right)$, then if

$$T_a = T\left(\mathbf{x};U_a\right) \leq 1,$$
$$T_b = T\left(\mathbf{x};U_b\right) \geq 1,$$

then we can estimate the inception voltage for a starting electron at position $\mathbf{x}$ through linear interpolation as

$$U_{\text{inc, Townsend}}\left(\mathbf{x}\right) = U_a + \frac{1 - T_a}{T_b - T_a}\left(U_b - U_a\right)$$

Minimum inception voltage

The minium inception voltage is the minimum voltage required for starting a critical avalanche (or Townsend process) for an arbitrary starting electron. For any position $\mathbf{x}$, then

$$U_{\text{inc}}\left(\mathbf{x}\right) = \min\left[U_{\text{inc, streamer}}\left(\mathbf{x}\right), U_{\text{inc, Townsend}}\left(\mathbf{x}\right)\right]$$

From the above, this is simply

$$U_{\text{inc}}^{\text{min}} = \min_{\forall \mathbf{x}}\left[U_{\text{inc}}\left(\mathbf{x}\right)\right].$$

From the above we also determine

$$\mathbf{x}_{\text{inc}}^{\text{min}} \leftarrow \mathbf{x} \text{ that minimizes } U_{\text{inc}}\left(\mathbf{x}\right) \forall \mathbf{x},$$

which is the position of the first electron that enables a critical avalanche at the minimum inception voltage.

Inception probability

The inception probability is given by Eq. 5.2.2 and is computed using straightforward numerical quadrature:

$$\int_{V_c} \left\langle \frac{dn_e}{dt} \right\rangle \left(1 - \frac{\eta}{\alpha}\right) \mathrm{d}V \approx \sum_{\mathbf{i} \in K_{\mathbf{i}} > K_c} \left(\left\langle \frac{dn_e}{dt} \right\rangle \right)_{\mathbf{i}} \left(1 - \frac{\eta_{\mathbf{i}}}{\alpha_{\mathbf{i}}}\right) \kappa_{\mathbf{i}} \Delta V_{\mathbf{i}},$$

and similarly for the surface integral.

Important: The integration runs over *valid cells*, i.e. grid cells that are not covered by a finer grid.

Advection algorithm

The advection algorithm for the negative ion distribution follows the time stepping algorithms described in the advection-diffusion model, see *Advection-diffusion model*.

5.2.6 Adaptive mesh refinement

The discharge inception model runs its own mesh refinement routine, which refines the mesh if

$$\alpha_{\mathrm{eff}}\left(|\mathbf{E}|, \mathbf{x}\right) \Delta x > \lambda,$$

where λ is a user-specified refinement criterion. The user can control refinement buffers and criterion through the following input options (see *Solver configuration*).

- `DischargeInceptionTagger.buffer` Adds a buffer region around tagged cells.
- `DischargeInceptionTagger.max_voltage` Maximum voltage that will be simulated.
- `DischargeInceptionTagger.ref_alpha` Sets the refinement criterion λ as above.

Tip: When using transient mode, it may be useful to simply set the maximum voltage to the peak voltage of the voltage curve.

For stationary solves it might be difficult because the range of voltages is determined automatically during run-time. It may be beneficial to run the program twice, first using `max_voltage = 1`, and then running the program again using the peak voltage from the output file.

5.2.7 Solver configuration

The `DischargeInceptionStepper` class come with user-configurable input options which are given below.

```
# ================================================================================================
# DischargeInceptionStepper class options
# ================================================================================================
DischargeInceptionStepper.verbosity            = -1               ## Chattiness.
DischargeInceptionStepper.profile              = false            ## Turn on/off run-time profiling
DischargeInceptionStepper.full_integration     = true             ## Use full reconstruction of K-region or not
DischargeInceptionStepper.mode                 = stationary       ## Mode (stationary or transient)
DischargeInceptionStepper.eval_townsend        = true             ## Evaluate Townsend criterion or not
DischargeInceptionStepper.inception_alg        = trapz            ## Integration algorithm. Either euler or trapz
DischargeInceptionStepper.output_file          = report.txt       ## Output file
DischargeInceptionStepper.K_inception          = 12               ## User-specified inception value
DischargeInceptionStepper.plt_vars             = K T Uinc field   ## Plot variables
```

(continues on next page)

(continued from previous page)

```
# Particle integration controls
DischargeInceptionStepper.min_phys_dx          = 1.E-10    ## Minimum permitted physical step size
DischargeInceptionStepper.max_phys_dx          = 1.E99     ## Maximum permitted physical step size
DischargeInceptionStepper.min_grid_dx          = 0.5       ## Minimum permitted grid step size
DischargeInceptionStepper.max_grid_dx          = 5.0       ## Maximum permitted grid step size
DischargeInceptionStepper.alpha_dx             = 5.0       ## Step size relative to avalanche length
DischargeInceptionStepper.grad_alpha_dx        = 0.1       ## Maximum step size relative to alpha/grad(alpha)
DischargeInceptionStepper.townsend_grid_dx     = 2.0       ## Space step to use for Townsend tracking

# Static mode
DischargeInceptionStepper.rel_step_dU     = 0.05    ## Relative (%) voltage increase for each step
DischargeInceptionStepper.step_dK         = 1.0     ## Desired increase in K for each voltage step
DischargeInceptionStepper.limit_max_K     = 15.0    ## Limit the maximum K-value

# Dynamic mode
DischargeInceptionStepper.ion_transport     = true     ## Turn on/off ion transport
DischargeInceptionStepper.transport_alg     = heun     ## Transport algorithm. 'euler', 'heun', or 'imex'
DischargeInceptionStepper.cfl               = 0.8      ## CFL time step for dynamic mode
DischargeInceptionStepper.first_dt          = 1.E-9    ## First time step to be used.
DischargeInceptionStepper.min_dt            = 1.E-9    ## Minimum permitted time step
DischargeInceptionStepper.max_dt            = 1.E99    ## Maximum permitted time step
DischargeInceptionStepper.voltage_eps       = 0.02     ## Permitted relative change in V(t) when computing dt
DischargeInceptionStepper.max_dt_growth     = 0.05     ## Maximum relative change in dt when computing dt
```

In the above options, the user can select the integration algorithms, the mode, and where to place the output file (which contains, e.g., the values of the ionization integral). The user can also include the following data in the HDF5 output files, by setting the `plt_vars` configuration option:

- `field` - Potential, field, and charge distributions.
- `K` - Inception integral.
- `T` - Townsend criterion.
- `Uinc` - Inception voltage.
- `alpha` - Effective ionization coefficient.
- `eta` - Eta coefficient.
- `bg_rate` - Background ionization rate.
- `emission` - Field emission.
- `poisson` - Poisson solver.
- `tracer` - Tracer particle solver.
- `cdr` - CDR solver.
- `ions` - Ion solver.

Important: The interface for setting the transport data (e.g., ionization coefficients) occurs via the C++ interface.

5.2.8 Setting up a new problem

To set up a new problem, using the Python setup tools in `$DISCHARGE_HOME/Physics/DischargeInception` is the simplest way. A full description is available in the `README.md` file contained in the folder:

```
# Physics/DischargeInception
This physics module predicts discharge inception.
The user must supply transport coefficients, geometry, and electron sources.

## Setting up a new application
To set up a new problem, use the Python script. For example:

```shell
python3 setup.py -base_dir=MyApplications -app_name=MyDischargeInception -geometry=Cylinder
```

The application will then be installed to $DISCHARGE_HOME/MyApplications/MyDischargeInception.
```

To see available setup options, use

```
./setup.py --help
```

5.2.9 Example programs

Example programs that use the discharge inception model are given in

High-voltage vessel

- `$DISCHARGE_HOME/Exec/Examples/DischargeInception/Vessel`. This program is set up in 2D (stationary) and 3D (transient) for discharge inception in atmospheric air. The input data is computed using BOLSIG+.

Electrode with surface roughness

- `$DISCHARGE_HOME/Exec/Examples/DischargeInception/ElectrodeRoughness`. This program is set up in 2D (stationary) and 3D (transient) for discharge inception on an irregular electrode surface. We use SF6 transport data as input data, computed using BOLSIG+.

Electrode with surface roughness

- `$DISCHARGE_HOME/Exec/Examples/DischargeInception/ElectrodeRoughness`. This program is set up in 2D and 3D (stationary) mode, and includes the influence of the Townsend criterion.

5.3 Îto-KMC plasma model

Warning: This section is not very well documented. The following featurse are definitely missing:

- Use of hybrid models (PPC-fluid specifications)
- How the physics time step is restricted

5.3.1 Underlying model

Plasma transport

The Îto-KMC model uses an Îto solver for some of the species, i.e. the particles drift and diffuse according to

$$d\mathbf{X} = \mathbf{V}dt + \sqrt{2Ddt},$$

where $\mathbf{X}$ is the particle position and $\mathbf{V}$ and D is the particle drift velocity and diffusion coefficients, respectively. These are obtained by interpolating the macroscopic drift velocity and diffusion coefficients to the particle position. Further details regarding the Îto method are given in *ItoSolver*.

Not all species in the Îto-KMC model need to be defined by particle solvers, as some species can be tracked by more conventional advection-diffusion-reaction solvers, see *CdrSolver* for discretization details.

Photoionization

Photoionization in the Îto-KMC model is done using discrete photons. These are generated and advanced in every time step, and interact with the plasma through user-defined photo-reactions. The underlying solver is the discrete Monte Carlo photon transport solver, see *Monte Carlo solver*.

Field interaction

The Îto-KMC model uses an electrostatic approximation where the field is obtained by solving the Poisson equation for the potential. See *Electrostatic solver* for further details.

Chemistry

Kinetic Monte Carlo (KMC) is used within grid cells for resolving the plasma chemistry. The algorithmic concepts are given in *Kinetic Monte Carlo*.

5.3.2 Algorithms

Time stepping

The Îto-KMC model has adaptive time step controls and no fundamental CFL limitation. Currently, the time step can be restricted through several means:

1. A physics-based time step, which is proportional to the non-critical time step from the KMC integrator.
2. A particle advective, diffusive, or advective-diffusive CFL limitation, both upper and lower bounds.
3. A CFL-condition on the convection-diffusion-reaction solvers.
4. A limit that restricts the time step to a factor proportional to the dielectric relaxation time.
5. Hard limits, placing upper and lower bounds on the time step.

In addition, the user can specify the maximum permitted growth or reduction in the time step.

These limits are given by the following input variables:

```
ItoKMCGodunovStepper.min_particle_advection_cfl              = 0.0              ## Advective time step CFL restriction
ItoKMCGodunovStepper.max_particle_advection_cfl              = 1.0              ## Advective time step CFL restriction
ItoKMCGodunovStepper.min_particle_diffusion_cfl              = 0.0              ## Diffusive time step CFL restriction
ItoKMCGodunovStepper.max_particle_diffusion_cfl              = 1.E99            ## Diffusive time step CFL restriction
ItoKMCGodunovStepper.min_particle_advection_diffusion_cfl    = 0.0              ## Advection-diffusion time step CFL␣
↪restriction
ItoKMCGodunovStepper.max_particle_advection_diffusion_cfl    = 1.E99            ## Advection-diffusion time step CFL␣
↪restriction
ItoKMCGodunovStepper.fluid_advection_diffusion_cfl           = 0.5              ## Advection-diffusion time step CFL␣
↪restriction
ItoKMCGodunovStepper.relax_dt_factor                         = 100.0            ## Relaxation time step restriction.
ItoKMCGodunovStepper.min_dt                                  = 0.0              ## Minimum permitted time step
ItoKMCGodunovStepper.max_dt                                  = 1.E99            ## Maximum permitted time step
ItoKMCGodunovStepper.max_growth_dt                           = 1.E99            ## Maximum permitted time step increase (dt␣
↪* factor)
ItoKMCGodunovStepper.max_shrink_dt                           = 1.E99            ## Maximum permissible time step reduction␣
↪(dt/factor)
ItoKMCGodunovStepper.extend_conductivity                     = true             ## Permit particles to live outside the EB␣
↪to avoid bad gradients near EB
ItoKMCGodunovStepper.cond_filter_num                         = 0                ## Number of filterings for conductivity
```

Particle placement

The KMC algorithm resolves the number of particles that are generated or lost within each grid cell, leaving substantial freedom in how one distributes new particles (or remove older ones). We currently support three methods for placing new particles:

1. Place all particles on the cell centroid.
2. Randomly distribute the new particles within the grid cell.
3. Compute an upstream position within the grid cell and randomly distribute the particles in the downstream region.

The methods each have their advantages and disadvantages. Placing all new particles on the cell centroid has the advantage that there will be no spurious space charge effects arising in the grid cell. Randomly distributing the new particles has the advantage that it generates secondary particles more evenly over the reaction region. Both these methods (`centroid` and `random`) are, however, sources of numerical diffusion since the secondary particles are potentially placed in the wake of the primary particles (which is non-physical). The downstream method circumvents this source of numerical diffusion by only placing secondary particles in the downstream region of some user-defined species (typically the electrons). See *JSON interface* for instructions on how to assign the particle placement method.

Spatial filtering

It is possible to apply filtering of the space-charge density prior to the field advance in order to reduce the impact of discrete particle noise. Filters are applied as follows:

$$\rho_i = \alpha\rho_i + \frac{1-\alpha}{2}\left(\rho_{i+s} + \rho_{i-s}\right).$$

where α is a filtering factor and s a stride. Users can apply this filtering by adjusting the following input options:

```
ItoKMCGodunovStepper.rho_filter_num        = 0    # Number of filterings for the space-density
ItoKMCGodunovStepper.rho_filter_max_stride = 1    # Maximum stride for filter
ItoKMCGodunovStepper.rho_filter_alpha      = 0.5  # Filtering factor (0.5 is a bilinear filter)
```

Warning: Spatial filtering of the space-charge density is a work-in-progress which may yield unexpected results. Bugs may or may not be present. Users should exercise caution when using this feature.

Reaction network

Particle management

5.3.3 0D chemistry

The user input interface to the Îto-KMC model consists of a zero-dimensional plasma kinetics interface called `ItoKMCPhysics`. This interface consists of the following main functionalities:

1. User-defined species, and which solver types (Îto or CDR) to use when tracking them.
2. User-defined initial particles, reaction kinetics, and photoionization processes.

ItoKMCPhysics

The complete C++ interface specification is given below. Because the interface is fairly extensive, `chombo-discharge` also supplies a JSON-based implementation called `ItoKMCJSON` (see *JSON interface*) for defining these things through file input. The difference between `ItoKMCPhysics` and its implementation `ItoKMCJSON` is that the JSON-based implementation class only implements a subset of potential features supported by `ItoKMCPhysics`.

```
/* chombo-discharge
 * Copyright © 2021 SINTEF Energy Research.
 * Please refer to Copyright.txt and LICENSE in the chombo-discharge root directory.
 */

/*!
  @file   CD_ItoKMCPhysics.H
  @brief  Main file for describing Ito-based plasma physics
  @author Robert Marskar
*/

#ifndef CD_ItoKMCPhysics_H
#define CD_ItoKMCPhysics_H

// Std includes
#include <memory>
#include <vector>

// Chombo includes
#include <RealVect.H>
#include <RefCountedPtr.H>
#include <List.H>

// Our includes
#include <CD_ItoSpecies.H>
#include <CD_CdrSpecies.H>
#include <CD_RtSpecies.H>
#include <CD_Photon.H>
#include <CD_ItoParticle.H>
#include <CD_PointParticle.H>
#include <CD_ItoKMCPhotoReaction.H>
#include <CD_ItoKMCSurfaceReactionSet.H>
#include <CD_KMCDualState.H>
#include <CD_KMCDualStateReaction.H>
#include <CD_KMCSolver.H>
#include <CD_NamespaceHeader.H>

namespace Physics {
  namespace ItoKMC {

    /*!
      @brief Numerical representation of the KMC state. Can be floating-point or integer type
    */
    using FPR = Real;

    /*!
      @brief KMC state used in the Kinetic Monte Carlo advancement
    */
    using KMCState = KMCDualState<FPR>;
```

(continues on next page)

(continued from previous page)

```
    /*!
      @brief KMC reaction used in the Kinetic Monte Carlo advancement
    */
    using KMCReaction = KMCDualStateReaction<KMCState, FPR>;

    /*!
      @brief KMC solverl type used in the Kinetic Monte carlo advancement
    */
    using KMCSolverType = KMCSolver<KMCReaction, KMCState, FPR>;

    /*!
      @brief Basic function for diffusing a particle
    */
    using DiffusionFunction = std::function<RealVect(const ItoParticle& p, const Real& a_dt)>;

    /*!
      @brief Map to species type
      @details This is just for distinguishing between species that are treated with an Ito or CDR formalism
    */
    enum SpeciesType
    {
      Ito,
      CDR
    };

    /*!
      @brief Base class for interaction between Kinetic Monte Carlo and Ito-based plasma solvers.
      @details When using this class, the user should populate m_kmcReactions with the appropriate reactions AND implement␣
↪routines
      for computing the mobility/diffusion coefficeints.
    */
    class ItoKMCPhysics
    {
    public:
      /*!
        @brief Constructor. Does nothing.
      */
      inline ItoKMCPhysics() noexcept;

      /*!
        @brief Destructor. Does nothing.
      */
      inline virtual ~ItoKMCPhysics() noexcept;

      /*!
        @brief Define the KMC solver and state
      */
      inline void
      defineKMC() const noexcept;

      /*!
        @brief Kill the KMC solver
      */
      inline void
      killKMC() const noexcept;

      /*!
        @brief Get the neutral density at a position in space
        @param[in] a_pos Physical position
        @return Neutral density
      */
      virtual Real
      getNeutralDensity(const RealVect a_pos) const noexcept = 0;

      /*!
        @brief Compute Townsend ionization coefficient
        @param[in] a_E Electric field magnitude
        @param[in] a_x Physical coordinate
        @return Should return the Townsend ionization coefficient.
      */
      virtual Real
      computeAlpha(const Real a_E, const RealVect a_x) const = 0;

      /*!
        @brief Compute Townsend attachment coefficient
        @param[in] a_E Electric field magnitude
        @param[in] a_x Physical coordinate
        @return Should return the Townsend attachment coefficient.
```

(continues on next page)

(continued from previous page)

```
  */
  virtual Real
  computeEta(const Real a_E, const RealVect a_x) const = 0;

  /*!
    @brief Get all particle drift-diffusion species
    @return m_itoSpecies
  */
  const Vector<RefCountedPtr<ItoSpecies>>&
  getItoSpecies() const;

  /*!
    @brief Get all fluid drift-diffusion species
    @return m_cdrSpecies
  */
  const Vector<RefCountedPtr<CdrSpecies>>&
  getCdrSpecies() const;

  /*!
    @brief Get all photon species
    @return m_rtSpecies
  */
  const Vector<RefCountedPtr<RtSpecies>>&
  getRtSpecies() const;

  /*!
    @brief Get diffusion functions
  */
  const Vector<DiffusionFunction>&
  getItoDiffusionFunctions() const noexcept;

  /*!
    @brief Get number of plot variables
  */
  virtual Vector<std::string>
  getPlotVariableNames() const noexcept;

  /*!
    @brief Get plot variables
    @param[in] a_E       Electric field
    @param[in] a_pos     Physical position
    @param[in] a_phi     Plasma species densities
    @param[in] a_gradPhi Density gradients for plasma species.
    @param[in] a_dx      Grid resolution
    @param[in] a_kappa   Cut-cell volume fraction
  */
  virtual Vector<Real>
  getPlotVariables(const RealVect          a_E,
                   const RealVect          a_pos,
                   const Vector<Real>&     a_phi,
                   const Vector<RealVect>& a_gradPhi,
                   const Real              a_dx,
                   const Real              a_kappa) const noexcept;

  /*!
    @brief Get number of plot variables
  */
  virtual int
  getNumberOfPlotVariables() const noexcept;

  /*!
    @brief Return number of Ito solvers.
  */
  inline int
  getNumItoSpecies() const;

  /*!
    @brief Return number of CDR solvers.
  */
  inline int
  getNumCdrSpecies() const;

  /*!
    @brief Return total number of plasma species.
  */
  inline int
  getNumPlasmaSpecies() const;
```

(continues on next page)

(continued from previous page)

```
  /*!
    @brief Return number of RTE solvers.
  */
  inline int
  getNumPhotonSpecies() const;

  /*!
    @brief Return true/false if physics model needs species gradients.
  */
  virtual bool
  needGradients() const noexcept;

  /*!
    @brief Get the internal mapping between plasma species and Ito solvers
  */
  inline const std::map<int, std::pair<SpeciesType, int>>&
  getSpeciesMap() const noexcept;

  /*!
    @brief Parse run-time options
  */
  inline virtual void
  parseRuntimeOptions() noexcept;

  /*!
    @brief Set initial surface charge. Default is 0, override if you want.
    @param[in] a_time Simulation time
    @param[in] a_pos  Physical coordinate
  */
  inline virtual Real
  initialSigma(const Real a_time, const RealVect a_pos) const;

  /*!
    @brief Compute the Ito solver mobilities.
    @param[in] a_time Time
    @param[in] a_pos  Position
    @param[in] a_E    Electric field
    @return Must return a vector of non-negative mobility coefficients for the plasma species
  */
  virtual Vector<Real>
  computeMobilities(const Real a_time, const RealVect a_pos, const RealVect a_E) const noexcept = 0;

  /*!
    @brief Compute the Ito solver diffusion coefficients
    @param[in] a_time Time
    @param[in] a_pos  Position
    @param[in] a_E    Electric field
    @return Must return a vector of non-negative diffusion coefficients for the plasma species
  */
  virtual Vector<Real>
  computeDiffusionCoefficients(const Real a_time, const RealVect a_pos, const RealVect a_E) const noexcept = 0;

  /*!
    @brief Resolve secondary emission at the EB.
    @details Routine is here to handle charge injection, secondary emission etc.
    @param[out] a_secondaryParticles Outgoing plasma species particles.
    @param[out] a_cdrFluxes          CDR fluxes for CDR species.
    @param[out] a_secondaryPhotons   Photons injected through the EB.
    @param[in]  a_primaryParticles   Particles that left the computational domain through the EB.
    @param[in]  a_cdrFluxesExtrap    Extrapolated CDR fluxes.
    @param[in]  a_primaryPhotons     Photons that left the computational domain through the EB.
    @param[in]  a_newNumParticles    Total number of particles in the cut-cell AFTER the transport step.
    @param[in]  a_oldNumParticles    Total number of particles in the cut-cell BEFORE the transport step.
    @param[in]  a_electricField      Electric field.
    @param[in]  a_physicalCellCenter Physical position of the cell center.
    @param[in]  a_cellCentroid       Cell centroid relative to the cell center (not multiplied by dx).
    @param[in]  a_bndryCentroid      EB face centroid relative to the cell center (not multiplied by dx).
    @param[in]  a_bndryNormal        Cut-cell normal vector.
    @param[in]  a_bndryArea          Cut-cell boundary area - not multiplied by dx (2D) or dx^2 (3D).
    @param[in]  a_dx                 Grid resolution on this level.
    @param[in]  a_dt                 Time step.
    @param[in]  a_isDielectric       Dielectric or electrode.
    @param[in]  a_matIndex           Material index (taken from computationalGeometry).
  */
  virtual void
  secondaryEmissionEB(Vector<List<ItoParticle>>&       a_secondaryParticles,
                      Vector<Real>&                    a_cdrFluxes,
                      Vector<List<Photon>>&            a_secondaryPhotons,
```

(continues on next page)

(continued from previous page)

```
                    const Vector<List<ItoParticle>>& a_primaryParticles,
                    const Vector<Real>&              a_cdrFluxesExtrap,
                    const Vector<List<Photon>>&      a_primaryPhotons,
                    const RealVect&                  a_E,
                    const RealVect&                  a_physicalCellCenter,
                    const RealVect&                  a_cellCentroid,
                    const RealVect&                  a_bndryCentroid,
                    const RealVect&                  a_bndryNormal,
                    const Real                       a_bndryArea,
                    const Real                       a_dx,
                    const Real                       a_dt,
                    const bool                       a_isDielectric,
                    const int                        a_matIndex) const noexcept = 0;

  /*!
    @brief Advance the reaction network using the KMC algorithm.
    @param[inout] a_numParticles  Number of physical particles
    @param[out]   a_numNewPhotons Number of new physical photons to generate (of each type)
    @param[out]   a_physicsDt     Reasonable KMC time step computed at end of integration with eps = 1
    @param[in]    a_phi           Plasma species densities.
    @param[in]    a_gradPhi       Plasma species density gradients.
    @param[in]    a_dt            Time step
    @param[in]    a_E             Electric field
    @param[in]    a_pos           Physical position
    @param[in]    a_dx            Grid resolution
    @param[in]    a_kappa         Cut-cell volume fraction.
  */
  inline void
  advanceKMC(Vector<FPR>&            a_numParticles,
             Vector<FPR>&            a_numNewPhotons,
             Real&                   a_physicsDt,
             const Vector<Real>&     a_phi,
             const Vector<RealVect>& a_gradPhi,
             const Real              a_dt,
             const RealVect          a_E,
             const RealVect          a_pos,
             const Real              a_dx,
             const Real              a_kappa) const;

  /*!
    @brief Reconcile the number of particles.
    @details This will add/remove particles and potentially also adjust the particle weights.
    @param[inout] a_particles Computational particles
    @param[in]    a_newNumParticles New number of particles (i.e., after the KMC advance)
    @param[in]    a_oldNumParticles Previous number of particles (i.e., before the KMC advance)
    @param[in]    a_electricField   Electric field in cell
    @param[in]    a_cellPos         Cell center position
    @param[in]    a_centroidPos     Cell centroid position
    @param[in]    a_loCorner        Low corner of minimum box enclosing the cut-cell
    @param[in]    a_hiCorner        High corner of minimum box enclosing the cut-cell
    @param[in]    a_bndryCentroid   Cut-cell boundary centroid
    @param[in]    a_bndryNormal     Cut-cell normal (pointing into the domain)
    @param[in]    a_dx              Grid resolution
    @param[in]    a_kappa           Cut-cell volume fraction.
    @note Public because this is called by ItoKMCStepper
  */
  inline void
  reconcileParticles(Vector<List<ItoParticle>*>& a_particles,
                     const Vector<FPR>&          a_newNumParticles,
                     const Vector<FPR>&          a_oldNumParticles,
                     const RealVect              a_electricField,
                     const RealVect              a_cellPos,
                     const RealVect              a_centroidPos,
                     const RealVect              a_lo,
                     const RealVect              a_hi,
                     const RealVect              a_bndryCentroid,
                     const RealVect              a_bndryNormal,
                     const Real                  a_dx,
                     const Real                  a_kappa) const noexcept;

  /*!
    @brief Generate new photons.
    @details This will add photons
    @param[in] a_newPhotons      New photons
    @param[in] a_numNewPhotons   Number of physical photons to be generated.
    @param[in] a_cellPos         Cell center position
    @param[in] a_centroidPos     Cell centroid position
    @param[in] a_loCorner        Low corner of minimum box enclosing the cut-cell
```

(continues on next page)

(continued from previous page)

```
    @param[in] a_hiCorner        High corner of minimum box enclosing the cut-cell
    @param[in] a_bndryCentroid   Cut-cell boundary centroid
    @param[in] a_bndryNormal     Cut-cell normal (pointing into the domain)
    @param[in] a_dx              Grid resolution
    @param[in] a_kappa           Cut-cell volume fraction.
    @note Public because this is called by ItoKMCStepper
  */
  inline void
  reconcilePhotons(Vector<List<Photon>*>& a_newPhotons,
                   const Vector<FPR>&     a_numNewPhotons,
                   const RealVect         a_cellPos,
                   const RealVect         a_centroidPos,
                   const RealVect         a_lo,
                   const RealVect         a_hi,
                   const RealVect         a_bndryCentroid,
                   const RealVect         a_bndryNormal,
                   const Real             a_dx,
                   const Real             a_kappa) const noexcept;

  /*!
    @brief Reconcile photoionization reactions.
    @param[inout] a_itoPrticles       Particle products placed in Ito solvers.
    @param[inout] a_cdrParticles      Particle products placed in CDR solvers.
    @param[in]    a_absorbedPhotons Photons absorbed on the mesh.
    @details This runs through the photo-reactions and associates photo-ionization products.
  */
  inline void
  reconcilePhotoionization(Vector<List<ItoParticle>*>&   a_itoParticles,
                           Vector<List<PointParticle>*>& a_cdrParticles,
                           const Vector<List<Photon>*>&  a_absorbedPhotons) const noexcept;

protected:
  /*!
    @brief Enum for switching between KMC algorithms
    @details 'SSA' is the Gillespie algorithm, 'Tau' is tau-leaping and 'Hybrid' is the Cao et. al. algorithm.
  */
  enum class Algorithm
  {
    SSA,
    ExplicitEuler,
    Midpoint,
    PRC,
    ImplicitEuler,
    HybridExplicitEuler,
    HybridMidpoint,
    HybridPRC,
    HybridImplicitEuler
  };

  /*!
    @brief Enum for switching between various particle placement algorithms
  */
  enum class ParticlePlacement
  {
    Random,
    Centroid,
    Downstream
  };

  /*!
    @brief Algorithm to use for KMC advance
  */
  Algorithm m_algorithm;

  /*!
    @brief Particle placement algorithm
  */
  ParticlePlacement m_particlePlacement;

  /*!
    @brief Map for associating a plasma species with an Ito solver or CDR solver.
  */
  std::map<int, std::pair<SpeciesType, int>> m_speciesMap;

  /*!
    @brief Class name. Used for options parsing
  */
  std::string m_className;
```

(continues on next page)

(continued from previous page)

```
    /*!
      @brief Turn on/off debugging
    */
    bool m_debug;

    /*!
      @brief Is defined or not
    */
    bool m_isDefined;

    /*!
      @brief If true, increment onto existing particles rather than creating new ones
    */
    bool m_incrementNewParticles;

    /*!
      @brief Is the KMC solver defined or not.
    */
    static thread_local bool m_hasKMCSolver;

    /*!
      @brief Kinetic Monte Carlo solver used in advanceReactionNetwork.
    */
    static thread_local KMCSolverType m_kmcSolver;

    /*!
      @brief KMC state used in advanceReactionNetwork.
    */
    static thread_local KMCState m_kmcState;

    /*!
      @brief KMC reactions used in advanceReactionNetowkr
      @note This is set up via setupKMC in order toi ensure OpenMP thread safety when calling advanceReactionNetwork. The␣
↪vector
      is later depopulated in killKMC().
    */
    static thread_local std::vector<std::shared_ptr<const KMCReaction>> m_kmcReactionsThreadLocal;

    /*!
      @brief List of reactions for the KMC solver
    */
    std::vector<KMCReaction> m_kmcReactions;

    /*!
      @brief List of photoionization reactions
    */
    std::vector<ItoKMCPhotoReaction> m_photoReactions;

    /*!
      @brief List of reactions that are a part of the time step limitation.
    */
    std::vector<Real> m_reactiveDtFactors;

    /*!
      @brief Random number generators for photoionization pathways.
      @details The first index is the photon index, i.e. entry in m_photonSpecies. The second index in the
      map is an RNG generator for selecting photo-reactions, and a map which associates the returned
      reaction from the RNG generator with an index in m_photoReactions.
    */
    std::map<int, std::pair<std::discrete_distribution<int>, std::map<int, int>>> m_photoPathways;

    /*!
      @brief Surface reactions.
    */
    ItoKMCSurfaceReactionSet m_surfaceReactions;

    /*!
      @brief Diffusion functions for the various Ito species
    */
    Vector<DiffusionFunction> m_itoDiffusionFunctions;

    /*!
      @brief List of solver-tracked particle drift-diffusion species.
    */
    Vector<RefCountedPtr<ItoSpecies>> m_itoSpecies;

    /*!
```

(continues on next page)

(continued from previous page)

```
  @brief List of solver-tracked fluid drift-diffusion species.
*/
Vector<RefCountedPtr<CdrSpecies>> m_cdrSpecies;

/*!
  @brief List of solver-tracked photon species.
*/
Vector<RefCountedPtr<RtSpecies>> m_rtSpecies;

/*!
  @brief An internal integer describing which species is the "ionizing" species.
  @details This is used by the particle reconciliation routine when only placing secondary particles
  in the downstream region of the primary particles.
*/
int m_downstreamSpecies;

/*!
  @brief Maximum new number of particles generated by the chemistry advance
*/
int m_maxNewParticles;

/*!
  @brief Maximum new number of photons generated by the chemistry advance
*/
int m_maxNewPhotons;

/*!
  @brief Solver setting for the Cao et. al algorithm.
  @details Determines critical reactions. A reaction is critical if it is m_Ncrit firings away from depleting a reactant.
*/
int m_Ncrit;

/*!
  @brief Solver setting for the Cao et. al algorithm.
  @details Maximum number of SSA steps to run when switching into SSA-based advancement for non-critical reactions.
*/
int m_NSSA;

/*!
  @brief Maximum number of iterations for implicit KMC-leaping algorithms
*/
int m_maxIter;

/*!
  @brief Solver setting for the Cao et. al. algorithm.
  @details Equal to the maximum permitted change in the relative propensity for non-critical reactions
*/
Real m_SSAlim;

/*!
  @brief Solver setting for the Cao et. al. algorithm.
  @details Equal to the maximum permitted change in the relative propensity for non-critical reactions
*/
Real m_eps;

/*!
  @brief Exit tolerance for implicit KMC-leaping algorithms
*/
Real m_exitTol;

/*!
  @brief Define method -- defines all the internal machinery
*/
inline void
define() noexcept;

/*!
  @brief Build internal representation of how we distinguish the Ito and CDR solvers
  @note This should ALWAYS be called after initializing the species since ItoKMCStepper will rely on it.
*/
inline void
defineSpeciesMap() noexcept;

/*!
  @brief Define pathways for photo-reactions
*/
inline void
```

(continues on next page)

(continued from previous page)

```
  definePhotoPathways() noexcept;

  /*!
    @brief Parse the maximum number of particles generated per cell
  */
  inline void
  parsePPC() noexcept;

  /*!
    @brief Parse the maximum number of particles generated per cell
  */
  inline void
  parseDebug() noexcept;

  /*!
    @brief Parse reaction algorithm
  */
  inline void
  parseAlgorithm() noexcept;

  /*!
    @brief Update reaction rates
    @param[out] a_kmcReactions Reaction rates to be set.
    @param[in] a_E       Electric field
    @param[in] a_pos     Physical position
    @param[in] a_phi     Plasma species densities
    @param[in] a_gradPhi Density gradients for plasma species.
    @param[in] a_dt      Time step
    @param[in] a_dx      Grid resolution
    @param[in] a_kappa   Cut-cell volume fraction
    @note Must be implemented by the user.
  */
  virtual void
  updateReactionRates(std::vector<std::shared_ptr<const KMCReaction>>& a_kmcReactions,
                      const RealVect                                   a_E,
                      const RealVect                                   a_pos,
                      const Vector<Real>&                              a_phi,
                      const Vector<RealVect>&                          a_gradPhi,
                      const Real                                       a_dt,
                      const Real                                       a_dx,
                      const Real                                       a_kappa) const noexcept = 0;

  /*!
    @brief Remove particles from the input list.
    @details This will remove weight from the input particles if we can. Otherwise we remove full particles.
    @param[inout] a_particles   List of (super-)particles to remove from.
    @param[in]    a_numToRemove Number of physical particles to remove from the input list
  */
  inline void
  removeParticles(List<ItoParticle>& a_particles, const long long a_numToRemove) const;

  /*!
    @brief Compute the upstream position in a grid cell. Returns false if an upstream position was undefinable.
    @details This routine will compute the position of the "most upstream" particle in the input list.
    For all particles we assume that they advect in the direction of Z*E, where Z is the charge number.
    This routine also computes the minimum bounding box (in unit coordinates) that encloses the downstreamn
    region within the grid cell.
    @param[out] a_pos Upstream position.
    @param[out] a_lo  Minimum coordinate of the downstream region (relative to the unit cell)
    @param[out] a_hi  Maximum coordinate of the downstream region (relative to the unit cell)
    @param[in] a_Z Particle charge number
    @param[in] a_particles Particles where we look for an upstream position.
    @param[in] a_electricField Electric field vector
    @param[in] a_cellPos Physical location of the cell center
    @param[in] a_dx Grid resolution
  */
  inline bool
  computeUpstreamPosition(RealVect&               a_pos,
                          RealVect&               a_lo,
                          RealVect&               a_hi,
                          const int&              a_Z,
                          const List<ItoParticle>& a_particles,
                          const RealVect&         a_electricField,
                          const RealVect&         a_cellPos,
                          const Real&             a_dx) const noexcept;

  /*!
    @brief No diffusion function for a particle
```

(continues on next page)

(continued from previous page)

```
        @param[in] a_particle Particle to diffuse
        @param[in] a_dt Time step
        @return Returns the zero vector.
      */
      inline RealVect
      noDiffusion(const ItoParticle& a_particle, const Real a_dt) const noexcept;

      /*!
        @brief Isotropic diffusion function for a particle
        @param[in] a_particle Particle to diffuse
        @param[in] a_dt Time step
        @return Returns sqrt(2 * D * a_dt) * N(0,1), where N(0,1) is a normal distribution
      */
      inline RealVect
      isotropicDiffusion(const ItoParticle& a_particle, const Real a_dt) const noexcept;

      /*!
        @brief Quasi-isotropic diffusion function for a particle which does not permit backward diffusion.
        @param[in] a_particle Particle to diffuse
        @param[in] a_dt Time step
        @return Same as isotropic diffusion, but ensures that the particle cannot diffuse backward
      */
      inline RealVect
      forwardIsotropicDiffusion(const ItoParticle& a_particle, const Real a_dt) const noexcept;
    };
  } // namespace ItoKMC
} // namespace Physics

#include <CD_NamespaceFooter.H>

#include <CD_ItoKMCPhysicsImplem.H>

#endif
```

Species definitions

Species are defined either as input species for CDR solvers (see *CdrSolver*) or Îto solvers (see *ItoSolver*). It is sufficient to populate the `ItoKMCPhysics` species vectors `m_cdrSpecies` and `m_itoSpecies` if one only wants to initialize the solvers. See *ItoKMCPhysics* for their C++ definition. In addition to actually populating the vectors, users will typically also include initial conditions for the species. The interfaces permit initial particles and density functions for both both solver types. Additionally, one must define all species associated with radiative transfer solvers.

Plasma reactions

Plasma reactions in the Îto-KMC model are represented stoichiometrically as

$$A + B + \ldots \xrightarrow{c} C + D + \ldots$$

where c is the KMC rate coefficient, i.e. *not the rate coefficient from the reaction rate equation*. An arbitrary number of reactions is supported, but currently the reaction mechanisms are limited to:

- The left-hand side must consist only of species that are tracked by a CDR or Îto solver.
- The right-hand side can consist of species tracked by a CDR solver, an Îto solver, or a radiative transfer solver.

These rules apply only to the internal C++ interface; implementations of this interface will typically also allow species defined as ''background species", i.e. species that are not tracked by a solver. In that case, the interface implementation must absorb the background species contribution into the rate constant.

In the KMC algorithm, one operates with chemical propensities rather than rate coefficients. For example, the chemical propensity a_1 for a reaction $A + B \xrightarrow{c_1} \varnothing$ is

$$a_1 = c_1 X_A X_B,$$

and from the macroscopic limit $X_A \gg 1$, $X_B \gg 1$ one may in fact also derive that $c_1 = k_1/\Delta V$ where k_1 is the rate coefficient from the reaction rate equation, i.e. where $\partial_t n_A = -k_1 n_A n_B$. Note that if the reaction was $A + A \xrightarrow{c_2} \varnothing$ then the propensity is

$$a_2 = \frac{1}{2} c_2 X_A \left(X_A - 1\right)$$

since there are $\frac{1}{2} X_A \left(X_A - 1\right)$ distinct pairs of particles of type A. Likewise, the fluid rate coefficient would be $k_2 = c_2 \Delta V/2$.

The distinction between KMC and fluid rates is an important one; the reaction representation used in the Îto-KMC model only operates with the KMC rates c_j, and it is us to the user to ensure that these are consistent with the fluid limit. Internally, these reactions are implemented through the dual state KMC implementation, see *Kinetic Monte Carlo*. During the reaction advance the user only needs to update the c_j coefficients (typically done via an interface implementation); the calculation of the propensity is automatic and follows the standard KMC rules (e.g., the KMC solver accounts for the number of distinct pairs of particles). This must be done in the routine `updateReactionRates(...)`, see *ItoKMCPhysics* for the complete specification.

Photoionization

Photo-reactions are also represented stoichiometrically as

$$\gamma \xrightarrow{\xi} A + B + \ldots,$$

where ξ is the photo-reaction efficiency, i.e. the probability that the photon γ causes a reaction when it is absorbed on the mesh. Currently, photons can only be generated through *plasma reactions*, see *Plasma reactions*, and we do not permit reactions between the photon and plasma species.

Important: We currently only support constant photoionization probabilities ξ.

`ItoKMCPhysics` adds some flexibility when dealing with photo-reactions as it permits pre-evaluation of photoionization probabilities. That is, when generating photons through reactions of the type $A+B \rightarrow \gamma$, one may scale the reaction by the photoionization probability ξ so that only ionizing photons are generated and then write the photo-reaction as $\gamma \xrightarrow{\xi=1} A + B + \ldots$. However, this process becomes more complicated when dealing with multiple photo-reaction pathways, e.g.,

$$\gamma \xrightarrow{\xi_1} A + B,$$
$$\gamma \xrightarrow{\xi_2} A + B,$$

where ξ_1 and ξ_2 are the probabilities that the photon γ triggers the reaction. If only a single physical photon is absorbed, then one must stochastically determine which pathway is triggered. There are two ways of doing this:

1. Pre-scale the photon generation by $\xi_1 + \xi_2$, and then determine the pathway through the relative probabilities $\xi_1/\left(\xi_1 + \xi_2\right)$ and $\xi_2/\left(\xi_1 + \xi_2\right)$.
2. Do not scale the photon generation reaction and determine the pathway through the absolute probabilities ξ_1 and ξ_2. This can imply a large computational cost since one will have to track all photons that are generated.

The former method is normally the preferred way as it can lead to reductions in computational complexity and particle noise, but for flexibility `ItoKMCPhysics` supports both of these. The latter method can be of relevance if users wants precise descriptions of photons that trigger both photoionizing reactions and surface reactions (e.g., secondary electron emission).

An alternative is to split the photon type γ into volumetrically absorbed and surface absorbed photon species γ_v and γ_s. In this case one may pre-scale the photon-generating reactions by the photo-reaction probabilities ξ (for volume

absorption) and ζ (for surface absorption) as follows:

$$A + B \xrightarrow{k \rightarrow k\xi} \gamma_v,$$
$$A + B \xrightarrow{k \rightarrow k\zeta} \gamma_s.$$

Volumetric and surface absorption is then treated independently

$$\gamma_v \xrightarrow{(\xi=1)}, \ldots$$
$$\gamma_s \xrightarrow{(\zeta=1)} \ldots.$$

This type of pre-evaluation of the photo-reaction pathways is sensible in a statistical sense, but loses meaning if only a single photon is involved.

Surface reactions

Warning: Surface reactions are supported by *ItoKMCPhysics* but not implemented in the JSON interface (yet).

Transport coefficients

Species mobilities and diffusion coefficients should be computed just as they are done in the fluid approximation. The `ItoKMCPhysics` interface requires implementations of two functions that define the coefficients as functions of $\mu = \mu\left(t, \mathbf{x}, \mathbf{E}\right)$, and likewise for the diffusion coefficients. Note that these functions should return the *fluid coefficients*.

Important: There is currently no support for computing μ as a function of the species densities (e.g., the electron density), but this only requires modest extensions of the Îto-KMC module.

5.3.4 Time step calculation

The time step calculation in `ItoKMCPhysics` is based on the approximation of a reasonable time step as

$$\Delta t = \frac{X_i}{|\sum_r \nu_{ri} a_r|},$$

where ν_{ri} is the state change in species i due to firing of exactly one reaction of type r. By default, the above is evaluated for all reactions, but the user can specify a subset of reactions for which the above constraint is applied.

Caveats

There are some caveats with using the time step limitation given above. For example, if there are no electrons in the simulation region, the above method will (correctly) return a very large time step based on the behavior of the other species. But one may very have *detachment* of electrons enabled, and the combination of a detachment event and a large time step is quite unfortunate. Because of this, the above constraint is also applied on proxy-states of $\vec{X}$ that have been completely exhausted through critical reactions, which provides a much more reasonable time step.

A second factor involved in the time step calculation above is that one may have regions where X_i is very small, but where $\sum_r \nu_{ri} a_r$ is very high. Outside of electron avalanching regions, detachment may completely dominate the time step calculation and cause a much smaller time step than necessary. Our resolution to this behavior is to permit the user to manually specify which reactions are important when computing the time step, see *Time step calculation*.

5.3.5 JSON interface

The JSON-based chemistry interface (called `ItoKMCJSON`) simplifies the definition of the plasma kinetics and initial conditions by permitting specifications through a JSON file. In addition, the JSON specification will permit the definition of background species that are not tracked as solvers (but that simply exist as a density function). The JSON interface thus provides a simple entry point for users that have no interest in defining the chemistry from scratch.

Several mandatory fields are required when specifying the plasma kinetics, such as:

- Gas pressure, temperature, and number density.
- Background species (if any).
- Townsend coefficients.
- Plasma species and their initial conditions, which solver to use when tracking them.
- Photon species and their absorption lengths.
- Plasma reactions.
- Photoionization reactions.

These specifications follow a pre-defined JSON format which is discussed in detail below.

Tip: While JSON files do not formally support comments, the JSON submodule used by `chombo-discharge` allows them.

Gas law

The JSON gas law represents a loose definition of the gas pressure, temperature, and *number density*. Multiple gas laws can be defined in the JSON file, and the user can then select which one to use. This is defined within a JSON entry `gas` and a subentry `law`. In the `gas/law` field one must specify an `id` field which indicates which gas law to use. One may also set an optional field `plot` to true/false if one wants to include the pressure, temperature, and density in the HDF5 output files.

Important: Gas parameters are specified in SI units.

Each user-defined gas law exists as a separate entry in the `gas/law` field, and the minimum requirements for these are that they contain a `type` specifier. Currently, the supported types are

- `ideal` (for constant ideal gas)

An example JSON specification is given below.

```
{
   "gas" : {
      // Specify which gas law we use. The user can define multiple gas laws and then later specify which one to use.
      "law" : {
         "id" : "my_ideal_gas", // Specify which gas law we use.
         "plot" : true,         // Turn on/off plotting.
         "my_ideal_gas" : {
             // Definition for an ideal gas law. Temperature is always in Kelvin the pressure is in Pascal. The neutral density
             // is derived using an ideal gas law.
             "type" : "ideal",
             "temperature" : 300,
             "pressure" : 1E5
         }
      }
   }
}
```

Ideal gas

When specifying that the gas law `type` is `ideal`, the user must further specify the temperature and pressure of the gas.

```
{
   "gas" : {
      "law" : {
         "id" : "my_ideal_gas", // Specify which gas law we use.
         "my_ideal_gas" : {
             "type" : "ideal",    // Ideal gas law type
             "temperature" : 300, // Temperature must be in Kelvin
             "pressure" : 1.0     // Pressure must be in bar
         },
      }
   }
}
```

Background species

Background species i are defined as continuous background densities N_i given by

$$N_i\left(\mathbf{x}\right) = \chi_i\left(\mathbf{x}\right) N\left(\mathbf{x}\right),$$

where $\chi_i\left(\mathbf{x}\right)$ is the molar fraction of species i (typically, but not necessarily, a constant).

When specifying background species, the user must include an array of background species in the `gas` JSON field. For example,

```
{
   "gas" : {
      "background species" : [
         {
            // First species goes here
         },
         {
            // Second species goes here
         }
      ]
   }
}
```

The order of appearance of the background species does not matter.

Name

The species name is specified by including an `id` field, e.g.

```
{
   "gas" : {
      "background species" : [
         {
            "id": "O2"
         }
      ]
   }
}
```

Molar fraction

The molar fraction can be specified in various forms by including a field `molar fraction`. This field also requires the user to specify the *type* of molar fraction. In principle, the molar fraction can have a positional dependency.

> **Warning:** There's no internal renormalization of the molar fractions input by the user. Internal inconsitencies will occur if the user supplies inputs molar fractions that sum to a number different than one.
>
> Various checks will be enabled if the user compiles in debug-mode (see *Installation*), but these checks are not guaranteed to catch all cases.

Constant

To specify a constant molar fraction, set the `type` specifier to constant and specify the value. An example JSON specification is

```
{
   "gas" : {
      "background species" : [
         {
            "id": "O2"             // Species name
            "molar fraction": {    // Molar fraction specification
               "type": "constant", // Constant molar fraction
               "value": 0.2        // Molar fraction value
            }
         }
      ]
   }
}
```

Tabulated versus height

The molar fraction can be set as a tabulated value versus one of the Cartesian coordinate axis by setting the `type` specifier to `table vs height`. The input data should be tabulated in column form, e.g.

```
# height       molar fraction
0              0.1
1              0.1
2              0.1
```

The file parser (see *LookupTable1D*) will ignore the header file if it starts with a hashtag (#). Various other inputs are then also required:

- `file` File name containing the height vs. molar fraction data (required).
- `axis` Cartesian axis to associate with the height coordinate (required).
- `height column` Column in data file containing the height (optional, defaults to 0).
- `molar fraction column` Column in data file containing the molar fraction (optional, defaults to 1).
- `height scale` Scaling of height column (optional).
- `fraction scale` Scaling (optional).
- `min height` Truncate data to minimum height (optional, applied after scaling).
- `max height` Truncate data to maximum height (optional, applied after scaling).
- `num points` Number of points to keep in table representation (optional, defaults to 1000).

- `spacing` Spacing table representation (optional, defaults to "linear".
- `dump` Option for dumping tabulated data to file.

An example JSON specification is

```
{
   "gas" : {
      "background species" : [
         {
            "id": "O2"                          // Species name
            "molar fraction": {                 // Molar fraction specification
               "type": "table vs height",       // Constant molar fraction
               "file" : "O2_molar_fraction.dat", // File name containing molar fraction
               "dump" : "debug_O2_fraction.dat", // Optional debugging hook for dumping table to file
               "axis" : "y",                    // Axis which represents the "height"
               "height column" : 0,             // Optional specification of column containing the height data (defaults to 0)
               "molar fraction column" : 1,     // Optional specification of column containing the height data (defaults to 1)
               "height scale" : 1.0,            // Optional scaling of height column
               "fraction scale" : 1.0,          // Optional scaling of molar fraction column
               "min height" : 0.0,              // Optional truncation of minimum height kept in internal table (occurs after␣
↪scaling)
               "max height" : 2.0,              // Optional truncation of maximum height kept in internal table (occurs after␣
↪scaling)
               "num points" : 100,              // Optional specification of number of data points kept in internal table␣
↪(defaults to 1000)
               "spacing" : "linear"             // Optional specification of table representation. Can be 'linear' or
↪'exponential' (defaults to linear)
            }
         }
      ]
   }
}
```

Plotting

Background species densities can be included in HDF5 plots by including an optional field `plot`. For example

```
{
   "gas" : {
      "background species" : [
         {
            "id": "O2"             // Species name
            "molar fraction": {    // Molar fraction specification
               "type": "constant", // Constant molar fraction
               "value": 0.2        // Molar fraction value
            },
            "plot": true           // Plot number density in HDF5 or not
         }
      ]
   }
}
```

Townsend coefficients

Townsend ionization and attachment coefficients α and η must be specified, and have two usages:

1. Flagging cells for refinement (users can override this).
2. Potential usage in plasma reactions (which requires particular care, see *Warnings and caveats*.

These are specified by including JSON entries `alpha` and `eta`, e.g.

```
{
    "alpha": {
       // alpha-coefficient specification goes here
    },
    "eta" : {
       // eta-coefficient specification goes here
```

(continues on next page)

(continued from previous page)

```
    }
}
```

There are various way of specifying these, as discussed below:

Automatic

In auto-mode the Townsend coefficients for the electrons is automatically derived from the user-specified list of reactions. E.g., the Townsend ionization coefficient is computed as

$$\alpha = \frac{\sum k}{\mu E},$$

where $\sum k$ is the sum of all ionizing reactions, also incorporating the neutral density. The advantage of this approach is that one may choose to discard some of the reactions from e.g. BOLSIG+ output but without recomputing the Townsend coefficients.

To automatically set Townsend coefficients, set `type` to auto and specify the species that is involved, e.g.

```
{
   "alpha": {
      "type": "auto",
      "species": "e"
   }
}
```

The advantage of using auto-mode for the coefficients is that one automatically ensures consistency between the user-specified list of reactions and the

Constant

To set a constant Townsend coefficient, set `type` to constant and then specify the value, e.g.

```
{
    "alpha": {
       "type": "constant",
       "value": 1E5
    }
}
```

Tabulated vs E/N

To set the coefficient as functions $f = f\left(E/N\right)$, set the `type` specifier to `table vs E/N` and specify the following fields:

- `file` For specifying the file containing the input data, which must be organized as column data, e.g.

  ```
  # E/N   alpha/N
  0       1E5
  100     1E5
  200     1E5
  ```

- `header` For specifying where in the file one starts reading data. This is an optional argument intended for use with BOLSIG+ output data where the user specifies that the data is contained in the lines below the specified header.
- `E/N column` For setting which column in the data file contains the values of E/N (optional, defaults to 0).

- `alpha/N column` For setting which column in the data file contains the values of α/N. Note that this should be replaced by `eta/N column` when parsing the attachment coefficient. This is an optional argument that defaults to 1.
- `scale E/N` Optional scaling of the column containing the E/N data.
- `scale alpha/N` Optional scaling of the column containing the α/N data.
- `min E/N` Optional truncation of the table representation of the data (applied after scaling).
- `max E/N` Optional truncation of the table representation of the data (applied after scaling).
- `num points` Number of data points in internal table representation (optional, defaults to 1000).
- `spacing` Spacing in internal table representation. Can be `linear` or `exponential`. This is an optional argument that defaults to `exponential`.
- `dump` A string specifier for writing the table representation of $E/N, \alpha/N$ to file.

An example JSON specification that uses a BOLSIG+ output file for parsing the data is

```
{
   "alpha": {
      "type" : "table vs E/N",                                    // Specify that we read in alpha/N vs E/N
      "file" : "bolsig_air.dat",                                  // File containing the data
      "dump" : "debug_alpha.dat",                                 // Optional dump of internalized table to file. Useful for␣
↪debugging.
      "header" : "E/N (Td)\tTownsend ioniz. coef. alpha/N (m2)", // Optional argument. Contains line immediately preceding␣
↪the data to be read.
      "E/N column" : 0,                                           // Optional specification of column containing E/N (defaults␣
↪to 0)
      "alpha/N column" : 1,                                       // Optional specification of column containing alpha/N␣
↪(defaults to 1)
      "min E/N" : 1.0,                                            // Optional truncation of minium E/N kept when resampling␣
↪the table (occurs after scaling)
      "max E/N" : 1000.0,                                         // Optional truncation of maximum E/N kept when resampling␣
↪the table (happens after scaling)
      "num points" : 1000,                                        // Optional number of points kept when resamplnig the table␣
↪(defaults to 1000)
      "spacing" : "exponential",                                  // Optional spcification of table representation. Defaults␣
↪to 'exponential' but can also be 'linear'
      "scale E/N" : 1.0,                                          // Optional scaling of the column containing E/N
      "scale alpha/N" : 1.0                                       // Optional scaling
   }
}
```

Plotting

To include the Townsend coefficients as mesh variables in HDF5 files, include the `plot` specifier, e.g.

```
{
   "alpha": {
      "type": "auto",
      "species": "e",
      "plot": true
   }
}
```

Plasma species

Plasma species are species that are tracked using either a CDR or Îto solver. The user must specify the following information:

- An ID/name for the species.
- The species' charge number.
- The solver type, i.e. whether or not it is tracked by a particle or fluid solver.
- Whether or not the species is mobile/diffusive. If the species is mobile/diffusive, the mobility/diffusion coefficient must also be specified.
- Optionally, the species temperature. If not specified, the temperature will be set to the background gas temperature.
- Initial particles for the solvers.

Basic definition

Species are defined by an array of entries in a `plasma species` JSON field. Each species *must* specify the following fields

- `id` (string) For setting the species name.
- `Z` (integer) For setting the species' charge number.
- `solver` (string) For specifying whether or not the species is tracked by a CDR or Îto solver. Acceptable values are *cdr* and *ito*.
- `mobile` (boolean) For specifying whether or not the species is mobile.
- `mobile` (boolean) For specifying whether or not the species is diffusive.

An example specification is

```
"plasma species" :
[
    // List of plasma species that are tracked. This is an array of species
    // with various identifiers, some of which are always required (id, Z, type, mobile, diffusive) and
    // others which are secondary requirements.
    {
        "id": "e",          // Species ID
        "Z" : -1,           // Charge number
        "solver" : "ito",   // Solver type. Either 'ito' or 'cdr'
        "mobile" : true,    // Mobile or not
        "diffusive" : true  // Diffusive or not
    },
    {
        // Definition of O2+ plasma species.
        "id": "O2+",            // Species ID
        "Z" : 1,                // Charge number
        "solver" : "cdr",       // CDR solver.
        "mobile" : false,       // Not mobile.
        "diffusive" : false     // Not diffusive
    }
]
```

Note that the order of appearance of the various species is irrelevant. However, if a species is specified as mobile/diffusive, the user *must* also specify the corresponding transport coefficients.

Mobility and diffusion coefficients

Mobility and diffusion coefficients are specified by including fields `mobility` and `diffusion` that further specifies how the transport coefficients are calculated. Note that the input to these fields should be *fluid transport coefficients*. These fields can be specified in various forms, as shown below.

Constant

To set a constant coefficient, set the `type` specifier to constant and then assign the value. For example,

```
"plasma species" :
[
    {
        "id": "e",              // Species ID
        "Z" : -1,               // Charge number
        "solver" : "ito",       // Solver type. Either 'ito' or 'cdr'
        "mobile" : true,        // Mobile or not
        "diffusive" : true      // Diffusive or not,
        "mobility": {
           "type": "constant", // Set constant mobility
           "value": 0.02
        },
        "diffusion": {
           "type": "constant", // Set constant diffusion coefficient
           "value": 2E-4
        }
    }
]
```

Constant $*N$

To set a coefficient that is constant vs N, set the `type` specifier to `constant mu*N` or `constant D*N` and then assign the value. For example,

```
"plasma species" :
[
    {
        "id": "e",                  // Species ID
        "Z" : -1,                   // Charge number
        "solver" : "ito",           // Solver type. Either 'ito' or 'cdr'
        "mobile" : true,            // Mobile or not
        "diffusive" : true          // Diffusive or not,
        "mobility": {
           "type": "constant mu*N", // Set mu*N to a constant
           "value": 1E24
        },
        "diffusion": {
           "type": "constant D*N",  // Set D*N to a constant
           "value": 5E24
        }
    }
]
```

Table vs E/N

To set the transport coefficients as functions $f = f(E/N)$, set the `type` specifier to `table vs E/N` and specify the following fields:

- `file` For specifying the file containing the input data, which must be organized as column data, e.g.

```
# E/N   mu/N
0       1E5
100     1E5
200     1E5
```

- `header` For specifying where in the file one starts reading data. This is an optional argument intended for use with BOLSIG+ output data where the user specifies that the data is contained in the lines below the specified header.
- `E/N column` For setting which column in the data file contains the values of E/N (optional, defaults to 0).
- `mu*N column` For setting which column in the data file contains the values of μN (or alternatively DN for the diffusion coefficient). This is an optional argument that defaults to 1.
- `E/N scale` Optional scaling of the column containing the E/N data.
- `mu*N scale` Optional scaling of the column containing the μN data (or alternatively DN for the diffusion coefficient).
- `min E/N` Optional truncation of the table representation of the data (applied after scaling).
- `max E/N` Optional truncation of the table representation of the data (applied after scaling).
- `num points` Number of data points in internal table representation (optional, defaults to 1000).
- `spacing` Spacing in internal table representation. Can be `linear` or `exponential`. This is an optional argument that defaults to `exponential`.
- `dump` A string specifier for writing the table representation of $E/N, \mu N$ to file.

An example JSON specification that uses a BOLSIG+ output file for parsing the data for the mobility is

```
"plasma species" :
[
   {
      "id": "e",                                      // Species ID
      "Z" : -1,                                       // Charge number
      "solver" : "ito",                               // Solver type.
      "mobile" : true,                                // Mobile
      "diffusive" : false,                            // Not diffusive
      "mobility" : {
         "type" : "table vs E/N",                     // Specification of tabulated mobility lookup method
         "file" : "bolsig_air.dat",                   // File containg the mobility data
         "dump" : "debug_mobility.dat",               // Optional argument for dumping table to disk (useful for debugging)
         "header" : "E/N (Td)\tMobility *N (1/m/V/s)", // Line immediately preceding the colum data
         "E/N column" : 0,                            // Column containing E/N
         "mu*N column" : 1,                           // Column containing mu*N
         "min E/N" : 1,                               // Minimum E/N kept when resampling table
         "max E/N" : 2E6,                             // Maximum E/N kept when resampling table
         "points" : 1000,                             // Number of grid points kept when resampling the table
         "spacing" : "exponential",                   // Grid point spacing when resampling the table
         "E/N scale" : 1.0,                           // Optional argument for scaling mobility coefficient
         "mu*N scale" : 1.0                           // Optional argument for scaling the E/N column
      }
   }
]
```

Tip: The parser for the diffusion coefficient is analogous; simply replace `mu*N` by `D*N`.

Parallel diffusion

`ItoKMCJSON` can limit diffusion against the electric field (or strictly speaking, the particle drift direction). The species specification permits a flag `diffusion model` that defauls to isotropic diffusion. However, it is possible to set this to other types of diffusion models. Currently, the available options are

1. `isotropic`, which sets an isotropic diffusion model.
2. `forward isotropic`, which uses isotropic diffusion but does not permit diffusion against the drift direction (this component is set to zero).

The code block below shows an example:

```
"plasma species" :
[
   {
      "id": "e",
      "Z" : -1,
      "solver" : "ito",
      "mobile" : true,
      "diffusive" : true,
      "diffusion model": "forward isotropic"
   }
]
```

Warning: Changing the diffusion model to anything other than isotropic can have unintended physical side effects. Although this functionality can enhance stability in e.g. cathode sheaths, it can have other unintended consequences.

Temperature

The species temperature is only parametrically attached to the species (i.e., not solved for), and can be specified by the user. Note that the temperature is mostly relevant for transport coefficients (e.g., reaction rates). The temperature can be specified through the `temperature` field for each species, and various forms of specifying this is available.

Important: If the specifies temperature is *not* specified by the user, it will be set equal to the background gas temperature.

Background gas

To set the temperature equal to the background gas temperature, set the `type` specifier field to `gas`. For example:

```
"plasma species" :
[
   {
      "id": "O2+",      // Species ID
      "Z" : 1,          // Charge number
      "solver" : "ito", // Solver type.
      "temperature": {  // Specify temperature
         "type": "gas"  // Temperature same as gas
      }
   }
]
```

Constant

To set a constant temperature, set the `type` specifier to `constant` and then specify the temperature. For example:

```
"plasma species" :
[
   {
      "id": "O2+",           // Species ID
      "Z" : 1,               // Charge number
      "solver" : "ito",      // Solver type.
      "temperature": {       // Specify temperature
         "type": "constant", // Constant temperature
         "value": 300        // Temperature in Kelvin
      }
   }
]
```

Table vs E/N

To set the species temperature as a function $T = T\left(E/N\right)$, set the `type` specifier to `table vs E/N` and specify the following fields:

- `file` For specifying the file containing the input data, which must be organized as column data *versus the mean energy*, e.g.

  ```
  # E/N   energy (eV)
  0       1
  100     2
  200     3
  ```

- `header` For specifying where in the file one starts reading data. This is an optional argument intended for use with BOLSIG+ output data where the user specifies that the data is contained in the lines below the specified header.
- `E/N column` For setting which column in the data file contains the values of E/N (optional, defaults to 0).
- `eV column` For setting which column in the data file contains the energy vs E/N. This is an optional argument that defaults to 1.
- `E/N scale` Optional scaling of the column containing the E/N data.
- `eV scale` Optional scaling of the column containing the energy (in eV).
- `min E/N` Optional truncation of the table representation of the data (applied after scaling).
- `max E/N` Optional truncation of the table representation of the data (applied after scaling).
- `num points` Number of data points in internal table representation (optional, defaults to 1000).
- `spacing` Spacing in internal table representation. Can be `linear` or `exponential`. This is an optional argument that defaults to `exponential`.
- `dump` A string specifier for writing the table representation of $E/N, eV$ to file.

An example JSON specification that uses a BOLSIG+ output file for parsing the data for the mobility is

```
"plasma species" :
[
   {
      "id": "O2+",                          // Species ID
      "Z" : 1,                              // Charge number
      "solver" : "ito",                     // Solver type.
      "mobile" : true,                      // Not mobile
      "diffusive" : false,                  // Not diffusive
      "temperature": {                      // Specification of temperature
         "type": "table vs E/N",            // Tabulated
```

(continues on next page)

(continued from previous page)

```
        "file": "bolsig_air.dat",                // File name
        "dump": "debug_temperature.dat",         // Dump to file
        "header" : "E/N (Td)\tMean energy (eV)", // Header preceding data
        "E/N column" : 0,                        // Column containing E/N
        "eV column" : 1,                         // Column containing the energy
        "min E/N" : 10,                          // Truncation of table
        "max E/N" : 2E6,                         // Truncation of table
        "E/N scale": 1.0,                        // Scaling of input data
        "eV scale": 1.0,                         // Scaling of input data
        "spacing" : "exponential",               // Table spacing
        "points" : 1000                          // Number of points in table
      }
   }
]
```

Initial particles

Initial particles for species are added by including an `initial particles` field. The field consists of an array of JSON entries, where each array entry represents various ways of adding particles.

Important: The `initial particles` field is *incrementing*, each new entry will add additional particles.

For example, to add two particles with particle weights of 1, one may specify

```
"plasma species" :
[
   {
      "id": "e",                 // Species ID
      "Z" : 1,                   // Charge number
      "solver" : "ito",          // Solver type.
      "mobile" : false,          // Not mobile
      "diffusive" : false,       // Not diffusive
      "initial particles": [     // Initial particles
         {
            "single particle": { // Single physical particle at (0,0,0)
               "position": [0, 0, 0],
               "weight": 1.0
            }
         },
         {
            "single particle": { // Single physical particle at (0,1,0)
               "position": [0, 1, 0],
               "weight": 1.0
            }
         }
      ]
   }
]
```

The various supported ways of additional initial particles are discussed below.

Important: If a solver is specified as a CDR solver, the initial particles are deposited as densities on the mesh using an NGP scheme.

Single particle

Single particles are placed by including a `single particle` JSON entry. The user must specify

- `position` The particle position.
- `weight` The particle weight.

For example:

```
"plasma species" :
[
   {
      "id": "e",                 // Species ID
      "Z" : 1,                   // Charge number
      "solver" : "ito",          // Solver type.
      "mobile" : false,          // Not mobile
      "diffusive" : false,       // Not diffusive
      "initial particles": [     // Initial particles
         {
            "single particle": { // Single physical particle at (0,0,0)
               "position": [0, 0, 0],
               "weight": 1.0
            }
         }
      ]
   }
]
```

Uniform distribution

Particles can be randomly drawn from a uniform distribution (a rectangular box in 2D/3D) by including a `uniform distribution`. The user must specify

- `low corner` The lower left corner of the box.
- `high corner` The lower left corner of the box.
- `num particles` Number of computational particles that are drawn.
- `weight` Particle weights.

For example:

```
"plasma species" :
[
   {
      "id": "e",                                  // Species ID
      "Z" : 1,                                    // Charge number
      "solver" : "ito",                           // Solver type.
      "mobile" : false,                           // Not mobile
      "diffusive" : false,                        // Not diffusive
      "initial particles": [                      // Initial particles
         {
            "uniform distribution" : {            // Particles inside a box
               "low corner" : [ -0.04, 0, 0 ],    // Lower-left physical corner of sampling region
               "high corner" : [ 0.04, 0.04, 0 ], // Upper-right physical corner of sampling region
               "num particles": 1000,             // Number of computational particles
               "weight" : 1                       // Particle weights
            }
         }
      ]
   }
]
```

Sphere distribution

Particles can be randomly drawn inside a circle (sphere in 3D) by including a `sphere distribution`. The user must specify

- `center` The sphere center.
- `radius` The sphere radius.
- `num particles` Number of computational particles that are drawn.
- `weight` Particle weights.

For example:

```
"plasma species" :
[
   {
      "id": "e",                            // Species ID
      "Z" : 1,                              // Charge number
      "solver" : "ito",                     // Solver type.
      "mobile" : false,                     // Not mobile
      "diffusive" : false,                  // Not diffusive
      "initial particles": [                // Initial particles
         {
            "sphere distribution" : {       // Particles inside sphere
               "center" : [ 0, 7.5E-3, 0 ], // Sphere center
               "radius" : 1E-3,             // Sphere radius
               "num particles": 1000,       // Number of computational particles
               "weight" : 1                 // Particle weights
            }
         }
      ]
   }
]
```

Gaussian distribution

Particles can be randomly drawn from a Gaussian distribution by including a `gaussian distribution`. The user must specify

- `center` The sphere center.
- `radius` The sphere radius.
- `num particles` Number of computational particles that are drawn.
- `weight` Particle weights.

For example:

```
"plasma species" :
[
   {
      "id": "e",                            // Species ID
      "Z" : 1,                              // Charge number
      "solver" : "ito",                     // Solver type.
      "mobile" : false,                     // Not mobile
      "diffusive" : false,                  // Not diffusive
      "initial particles": [                // Initial particles
         {
            "gaussian distribution" : {     // Particles drawn from a gaussian
               "center" : [ 0, 7.5E-3, 0 ], // Center
               "radius" : 1E-3,             // Radius
               "num particles": 1000,       // Number of computational particles
               "weight" : 1                 // Particle weights
            }
         }
      ]
```

(continues on next page)

(continued from previous page)

```
   }
]
```

List/file

Particles can be read from a file by including a `list` specifier. The particles must be organized as rows containing the coordinate positions and weights, e.g.

```
# x   y   z    w
  0   0   0    1
```

The user must specify

- `file` File name.
- `x` Column containing the x coordinates (optional, defaults to 0).
- `y` Column containing the y coordinates (optional, defaults to 1).
- `z` Column containing the z coordinates (optional, defaults to 2).
- `w` Column containing the particle weight (optional, defaults to 3).

For example:

```
"plasma species" :
[
   {
      "id": "e",                          // Species ID
      "Z" : 1,                            // Charge number
      "solver" : "ito",                   // Solver type.
      "mobile" : false,                   // Not mobile
      "diffusive" : false,                // Not diffusive
      "initial particles": [              // Initial particles
         {
            "list": {
               "file": "initial_particles.dat", // File containing the particles
               "x column": 0,
               "y column": 1,
               "z column": 2,
               "w column": 3
            }
         }
      ]
   }
]
```

Initial densities

One may include an initial density by setting the `initial density` parameter. For example:

```
"plasma species" :
[
   {
      "id": "e",
      "Z" : 1,
      "solver" : "cdr",
      "mobile" : true,
      "diffusive" : true,
      "initial density": 1E10
   }
]
```

If the species is defined as a particle species, computational particles will be generated within each grid so that the density is approximately as specified.

Photon species

Photon species are species that are tracked using a radiative transfer solver. These are defined by an array of entries in a `photon species` JSON entry, and must define the following information:

- An ID/name for the species.
- The absorption coefficient for the photon type.

Basic definition

A basic definition of a single photon species is

```
"photon species":
 [
     {
         "id": "Y",              // Photon species id. Must be unique
         "kappa": {              // Specification of absorption coefficient.
             "type": "constant", // Specify constant absorption coefficient
             "value": 1E4        // Value of the absorption coefficeint
         }
     }
 ]
```

Multiple photon species are added by appending with more entries, e.g.

```
"photon species":
 [
     {
         "id": "Y1",             // Photon species id. Must be unique
         "kappa": {              // Specification of absorption coefficient.
             "type": "constant", // Specify constant absorption coefficient
             "value": 1E4        // Value of the absorption coefficeint
         }
     },
     {
         "id": "Y2",             // Photon species id. Must be unique
         "kappa": {              // Specification of absorption coefficient.
             "type": "constant", // Specify constant absorption coefficient
             "value": 1E4        // Value of the absorption coefficeint
         }
     }
 ]
```

Absorption coefficient

Constant

In order to set a constant absorption coefficient, set `type` to constant and then specify the `value` field. For example:

```
"photon species":
 [
     {
         "id": "Y",
         "kappa": {
             "type": "constant",
             "value": 1E4
         }
     }
 ]
```

stochastic A

The `stochastic A` specifier computes a random absorption length from the expression

$$\kappa = (\chi_{\text{min}} p_i) \left(\frac{\chi_{\text{max}}}{\chi_{\text{min}}} \right)^{\frac{f-f_1}{f_2-f_1}},$$

where p_i is the partial pressure of some species i, and $p_i\chi_{\text{min}}$ and $p_i\chi_{\text{max}}$ are minimum and maximum absorption lengths on the frequency interval $f \in [f_1, f_2]$. The user must specify χ_{min}, χ_{max}, f_1, f_2, and the background species used when computing p_i. This is done through fields `f1`, `f_2`, `chi min`, `chi max`, and `neutral`. For example:

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

stochastic B

The `stochastic B` specifier computes a random absorption length from the expression

$$\kappa = (\chi_{\text{min}} p_i) \left(\frac{\chi_{\text{max}}}{\chi_{\text{min}}} \right)^{u},$$

where p_i is the partial pressure of some species i, and $p_i\chi_{\text{min}}$ and $p_i\chi_{\text{max}}$ are minimum and maximum absorption lengths, and u is a random number between 0 and 1. Note that that this is just a simpler way of using the `stochastic A` specifier above. The user must specify χ_{min}, χ_{max}, and the background species used when computing p_i. This is done through fields `chi min`, `chi max`, and `neutral`. For example:

```
"photon species":
[
    {
        "id": "Y",
        "kappa": {
            "type": "stochastic B",
            "chi min": 2.625E-2,
            "chi max": 1.5,
            "neutral": "O2"
        }
    }
]
```

Plasma reactions

Plasma reactions are always specified stoichiometrically in the form $S_A + S_B + \ldots \xrightarrow{k} S_C + S_D + \ldots$. The user must supply the following information:

- Species involved on the left-hand and right-hand side of the reaction.
- How to compute the reaction rate.
- Optionally specify gradient corrections to the reaction.
- Optionally specify whether or not the reaction rate will be written to file.

Important: The JSON interface expects that the reaction rate k is the same as in the reaction rate equation. Internally, the rate is converted to a format that is consistent with the KMC algorithm.

Plasma reactions should be entered in the JSON file using an array (the order of reactions does not matter).

```
"plasma reactions": [
   {
   // First reaction goes here,
   },
   {
   // Next reaction goes here,
   }
]
```

Reaction specifiers

Each reaction must include an entry `reaction` in the list of reactions. This entry must consist of a single string with a left-hand and right-hand sides separated by `->`. For example:

```
"plasma reactions": [
   {
      "reaction": "e + N2 -> e + e + N2+"
   }
]
```

which is equivalent to the reaction $e + N_2 \rightarrow e + e + N_2^+$. Each species must be separated by whitespace and a addition operator. For example, the specification `A + B + C` will be interpreted as a reaction $A + B + C$ but the specification `A+B + C` will be interpreted as a reaction between one species `A+B` and another species `C`.

Internally, both the left- and right-hand sides are checked against background and plasma species. The program will perform a run-time exit if the species are not defined in these lists. Note that the right-hand side can additonal contain photon species.

Tip: Products not not defined as a species can be enclosed by parenthesis `(` and `)`.

The reaction string generally expects all species on each side of the reaction to be defined. It is, however, occasionally useful to include products which are *not* defined. For example, one may with to include the reaction $e + O_2^+ \rightarrow O + O$ but not actually track the species O. The reaction string can then be defined as the string `e + O2+ ->` but as it might be difficult for users to actually remember the full reaction, we may also enter it as `e + O2+ -> (O) + (O)`.

Rate calculation

Reaction rates can be specified using multiple formats (constant, tabulated, function-based, etc). To specify how the rate is computed, one must specify the `type` keyword in the JSON entry.

Constant

To use a constant reaction rate, the `type` specifier must be set to constant and the reaction rate must be specified through the `value` keyword. A JSON specification is e.g.

```
"plasma reactions": [
   {
      "reaction": "e + N2 -> e + e + N2+" // Specify reaction
      "type": "constant",                 // Reaction rate is constant
      "value": 1.E-30                     // Reaction rate
   }
]
```

Function vs E/N

Some rates given as a function $k = k\left(E/N\right)$ are supported, which are outlined below.

Important: In the below function-based rates, E/N indicates the electric field in units of Townsend.

function E/N exp A

This specification is equivalent to a fluid rate

$$k = c_1 \exp\left[-\left(\frac{c_2}{c_3 + c_4 E/N}\right)^{c_5}\right].$$

The user must specify the constants c_i in the JSON file. An example specification is

```
"plasma reactions": [
   {
      "reaction": "A + B -> "       // Example reaction string.
      "type": "function E/N exp A", // Function based rate.
      "c1": 1.0,                    // c1-coefficient
      "c2": 1.0,                    // c2-coefficient
      "c3": 1.0,                    // c3-coefficient
      "c4": 1.0,                    // c4-coefficient
      "c5": 1.0                     // c5-coefficient
   }
]
```

Temperature-dependent

Some rates can be given as functions $k = k(T)$ where T is some temperature.

function T A

This specification is equivalent to a fluid rate

$$k = c_1 \left(T_i\right)^{c_2}.$$

Mandatory input variables are c_1, c_2, and the specification of the species corresponding to T_i. This can correspond to one of the background species. An example specification is

```
"plasma reactions": [
   {
      "reaction": "A + B -> " // Example reaction string.
      "type": "function T A", // Function based rate.
      "c1": 1.0,              // c1-coefficient
      "c2": 1.0,              // c2-coefficient
      "T": "A"               // Which species temperature
   }
]
```

function T1T2 A

This specification is equivalent to a fluid rate

$$k = c_1 \left(\frac{T_1}{T_2} \right)^{c_2}.$$

Mandatory input variables are c_1, c_2, and the specification of the species corresponding to T_1 and T_2. This can correspond to one of the background species. An example specification is

```
"plasma reactions": [
   {
      "reaction": "A + B -> "    // Example reaction string.
      "type": "function T1T2 A", // Function based rate.
      "c1": 1.0,                 // c1-coefficient
      "c2": 1.0,                 // c2-coefficient
      "T1": "A",                 // Which species temperature for T1
      "T2": "B"                  // Which species temperature for T2
   }
]
```

Townsend rates

Reaction rates can be specified to be proportional to $\alpha \left| \mathbf{v}_i \right|$ where α is the Townsend ionization coefficient and $\left| \mathbf{v}_i \right|$ is the drift velocity for some species i. This type of reaction is normally encountered when using simplified chemistry, e.g.

$$\partial_t n_i = \alpha \left| \mathbf{v}_s \right| n_i = \alpha \mu \left| E \right| n_i.$$

which is representative of the reaction $S_i \rightarrow S_i + S_i$. To specify a Townsend rate constant, one can use the following:

1. `alpha*v` for setting the rate constant proportional to the Townsend ionization rate.
2. `eta*v` for setting the rate constant proportional to the Townsend attachment rate.

One must also specify which species is associated with $\left| \mathbf{v} \right|$ by specifying a species flag. A complete JSON specification is

```
{
   "plasma reactions":
      [
         {
            "reaction": "e -> e + e + M+", // Reaction string
            "type": "alpha*v",             // Rate is alpha*v
            "species": "e"                 // Species for v
         }
      ]
}
```

Warning: When using the Townsend coefficients for computing the rates, one should normally *not* include any neutrals on the left hand side of the reaction. The reason for this is that the Townsend coefficients α and η already incorporate the neutral density. By specifying e.g. a reaction string $\mathrm{e} + \mathrm{N}_2 \rightarrow \mathrm{e} + \mathrm{e} + \mathrm{N}_2^+$ together with the `alpha*v` or `eta*v` specifiers, one will end up multiplying in the neutral density twice, which will lead to an incorrect rate.

Tabulated vs E/N

Scaling

Reactions can be scaled by including a `scale` field in the JSON entry. This will scale the reaction coefficient by the input factor, e.g. modify the rate constant as

$$k \rightarrow \nu k,$$

where ν is the scaling factor. This is useful when scaling reactions from different units, or for completely turning off some input reactions. An example JSON specification is

```
"plasma reactions":
[
    {
        "reaction": "e -> e + e + M+", // Reaction string
        "type": "alpha*v",             // Rate is alpha*v
        "species": "e",                // Species for v,
        "scale": 0.0                   // Scaling factor
    }
]
```

Tip: If one turns off a reaction by setting `scale` to zero, the KMC algorithm will still use the reaction but no reactants/products are consumed/produced.

Efficiency

Reactions efficiencies can be modified in the same way as one do with the `scale` field, e.g. modify the rate constant as

$$k \rightarrow \nu k,$$

where ν is the reaction efficiency. Specifications of the efficiency can be achieved in the forms discussed below.

Constant efficiency

An example JSON specification for a constant efficiency is:

```
"plasma reactions":
[
    {
        "reaction": "e -> e + e + M+",
        "efficiency": 0.5
    }
]
```

Efficiency vs E/N

An example JSON specification for an efficiency computed versus E/N is

```
"plasma reactions":
[
    {
        "reaction": "e -> e + e + M+",
        "efficiency vs E/N": "efficiencies.dat"
    }
]
```

where the file `efficiencies.dat` must contain two-column data containing values of E/N along the first column and efficiencies along the second column. An example file is e.g.

```
0    0
100  0.1
200  0.3
500  0.8
1000 1.0
```

This data is then internally convered to a uniformly spaced lookup table (see *LookupTable1D*).

Efficiency vs E

An example JSON specification for an efficiency computed versus E is

```
"plasma reactions":
[
    {
        "reaction": "e -> e + e + M+",
        "efficiency vs E": "efficiencies.dat"
    }
]
```

where the file `efficiencies.dat` must contain two-column data containing values of E along the first column and efficiencies along the second column. This method follows the same as `efficiency vs E/N` where the data in the input file is put in a lookup table.

Quenching

Reaction quenching can be achieved in the following forms:

Pressure-based

The reaction rate can modified by a factor $p_q/\left[p_q + p\left(\mathbf{x}\right)\right]$ where p_q is a quenching pressure and $p\left(\mathbf{x}\right)$ is the gas pressure. This will modify the rate as

$$k \rightarrow k\frac{p_q}{p_q + p\left(\mathbf{x}\right)}.$$

An example JSON specification is

```
"plasma reactions":
[
    {
        "reaction": "e + N2 -> e + N2 + Y", // Reaction string
        "type": "alpha*v",                  // Rate is alpha*v
        "species": "e",                     // Species for v,
        "quenching pressure": 4000.0        // Quenching pressure
    }
]
```

Important: The quenching pressure must be specified in units of Pascal.

Rate-based

The reaction rate can be modified by a factor $k_r / [k_r + k_p + k_q]$. The intention behind this scaling is that reaction r occurs only if it is not predissociated (by rate k_p) or quenched (by rate k_q). Such processes can occur, for example, in excited molecules. This will modifiy the rate constant as

$$k \rightarrow k \frac{k_r}{k_r + k_p + k_q},$$

where the user will specifiy k_r, k_p, and k_q/N. The JSON specification must contain `quenching rates`, for example:

```
"plasma reactions":
[
    {
        "reaction": "e + N2 -> e + N2 + Y", // Reaction string
        "type": "alpha*v",                  // Rate is alpha*v
        "species": "e",                     // Species for v,
        "quenching rates": {                // Specify relevant rates
           "kr": 1E9,
           "kp": 1E9,
           "kq/N": 0.0
        }
    }
]
```

Gradient correction

In LFA-based models it is frequently convenient to include energy-corrections to the ionization rate. In this case one modifies the rate as

$$k \rightarrow k \left(1 + \frac{\mathbf{E} \cdot (D \nabla n_i)}{n_i \mu_i E^2} \right).$$

To include this correction one may include a specifier `gradient correction` in the JSON entry, in which case one must also enter the species n_i. A JSON specification that includes this

```
"plasma reactions":
[
    {
        "reaction": "e -> e + e + M+", // Reaction string
        "type": "alpha*v",             // Rate is alpha*v
        "species": "e",                // Species for v,
        "gradient correction": "e"     // Specify gradient correction using species "e"
    }
]
```

This is equivalent to a source term

$$\begin{aligned} S &= \alpha n_e \left| \mathbf{v}_e \right| \left(1 + \frac{\mathbf{E} \cdot (D_e \nabla n_e)}{n_e \mu_e E^2} \right) \\ &= \alpha \left[n_e \left| \mathbf{v}_e \right| + \hat{\mathbf{E}} \cdot (D_e \nabla n_e) \right]. \end{aligned}$$

One can recognize this term as a regular electron impact ionization source term (typically written as $\alpha \mu n_e E$). With the gradient correction, the ionization source term is essentially computed using the full electron flux, i.e., including the diffusive electron flux. Note that the full electron flux has a preferential direction, and the physical interpretation of this direction is that if there is net diffusion against the electric field, electrons lose energy and the impact ionization source term is correspondingly lower.

Understanding reaction rates

The JSON specification takes *fluid rates* as input to the KMC algorithm. Note that subsequent application of multiple scaling factors are multiplicative. For example, the specification

```
"plasma reactions":
[
    {
        "reaction": "e -> e + e + M+", // Reaction string
        "type": "alpha*v",             // Rate is alpha*v
        "species": "e",                // Species for v,
        "gradient correction": "e"     // Specify gradient correction using species "e"
        "quenching pressure": 4000,    // Quenching pressure
        "efficiency": 0.1              // Reaction efficiency
    }
]
```

is equivalent to the rate constant

$$k = \alpha \left| \mathbf{v}_e \right| \left(1 + \frac{\mathbf{E} \cdot (D_e \nabla n_e)}{n_e \mu_e E^2} \right) \frac{p_q}{p_q + p\left(\mathbf{x}\right)} \nu$$

Internally, conversions between the *fluid rate* k and the KMC rate c are made. The conversion factors depend on the reaction order, and ensure consistency between the KMC and fluid formulations.

Plotting rates

Reaction rates can be plotted by including an optional `plot` specifier. If the specifier included, the reaction rates will be included in the HDF5 output. The user can also include a description string which will be used when plotting the reaction.

The following two specifiers can be included:

- `plot` (true/false) for plotting the reaction.
- `description` (string) for naming the reaction in the HDF5 output.

If the plot description is left out, the reaction string will be used as a variable name in the plot file. A JSON specification that includes these

```
"plasma reactions":
[
    {
        "reaction": "e -> e + e + M+",       // Reaction string
        "type": "alpha*v",                   // Rate is alpha*v
        "species": "e",                      // Species for v,
        "plot": true,                        // Plot this reaction
        "description": "Townsend ionization" // Variable name in HDF5
    }
]
```

Printing rates

`ItoKMCJSON` can print the plasma reaction rates (including photon-generating ones) to file. This is done through an optional input argument `print_rates` in the input file (not the JSON specification). The following options are supported:

```
ItoKMCJSON.print_rates            = true
ItoKMCJSON.print_rates_minEN      = 1
ItoKMCJSON.print_rates_maxEN      = 1000
ItoKMCJSON.print_rates_num_points = 200
ItoKMCJSON.print_rates_spacing    = exponential
ItoKMCJSON.print_rates_filename   = fluid_rates.dat
ItoKMCJSON.print_rates_pos        = 0 0 0
```

Setting `ItoKMCJSON.print_rates` to true in the input file will write all reaction rates as column data $E/N, k(E/N)$. Here, k indicates the *fluid rate*, so for a reaction $A + B + C \xrightarrow{k} \ldots$ it will include the rate k. Reactions are ordered identical to the order of the reactions in the JSON specification. This feature is mostly used for debugging or development efforts.

Time step calculation

The user can manually specify which reactions are to be used when computing a chemistry time step Δt, as discussed in *Time step calculation*. To enable a time step calculation, specify the `include_dt_calc` flag in the reaction specifier, as shown below:

```
"plasma reactions":
[
    {
        "reaction": "e -> e + e + M+",
        "type": "alpha*v",
        "species": "e",
        "include_dt_calc": true
    }
]
```

Tip: It is normally sufficient to enable Δt calculations for the ionizing reactions.

Particle placement

Specification of secondary particle placement is done through the JSON file by specifying the `particle placement` field. Currently, new particles may be placed on the centroid, uniformly distributed in the grid cell, or placed randomly in the downstream region of some user-defined species. The method is specified through the `method` specifier, which must either be `centroid`, `random`, or `downstream`. If specifying `downstream`, one must also include a species specifier (which must correspond to one of the plasma species). The following three specifies are all valid:

```
"particle placement":
{
   "method": "centroid"
}

"particle placement":
{
   "method": "random"
}

"particle placement":
{
   "method": "downstream",
   "species": "e"
}
```

Photoionization

Photoionization reactions are specified by including a `photoionization` array of JSON entries that specify each reaction. The left-hand side of the reaction must contain a photon species (plasma and background species can be present but will be ignored), and the right-hand side can consist of any species except other photon species. Each entry *must* contain a reaction string specifier and optionally also an efficiency (which defaults to 1). Assume that photons are generated through the reaction

```
"plasma reactions":
[
    {
        "reaction": "e + N2 -> e + N2 + Y", // Reaction string
        "type": "alpha*v",                  // Rate is alpha*v
        "species": "e",                     // Species for v,
        "quenching pressure": 4000.0        // Quenching
        "efficiency": 0.6                   // Excitation events per ionization event
        "scale": 0.1                        // Photoionization events per absorbed photon
    }
]
```

where we assume an excitation efficiency of 0.6 and photoionization efficiency of 0.1. An example photoionization specification is then

```
"photoionization":
[
    {
        "reaction": "Y + O2 -> e + O2+"
    }
]
```

Note that the efficiency of this reaction is 1, i.e. the photoionization probabilities was pre-evaluated. As discussed in *ItoKMCPhysics*, we can perform late evaluation of the photoionization probability by specifically including a efficiency. In this case we modify the above into

```
"plasma reactions":
[
    {
        "reaction": "e + N2 -> e + N2 + Y", // Reaction string
        "type": "alpha*v",                  // Rate is alpha*v
        "species": "e",                     // Species for v,
        "quenching pressure": 4000.0        // Quenching
        "efficiency": 0.6                   // Excitation events per ionization event
    }
],
"photoionization":
[
    {
        "reaction": "Y + O2 -> e + O2+",
        "efficiency": 0.1
    },
    {
        "reaction": "Y + O2 -> (null)",
        "efficiency": 0.9
    }
]
```

where a null-absorption model has been added for the photon absorption. When multiple pathways are specified this way, and they have probabilities $\xi_1, \xi_2, \ldots$, the reaction is stochastically determined from a discrete distribution with relative probabilities $p_i = \xi_i/(\xi_1 + \xi_2 + \ldots)$.

Important: The null-reaction model is *not* automatically added when using late evaluation of the photoionization probability.

Secondary emission

Secondary emission is supported for particles, including photons, but not for fluid species. Specification of secondary emission is done separately on dielectrics and electrodes by specifying JSON entries `electrode emission` and `dielectric emission`. The internal specification for these are identical, so all examples below use `dielectric emission`. Secondary emission reactions are specified similarly to photoionization reactions. However, the left-hand side must consist of a single species (photon or particle), while the right-hand side can consist of multiple species. Wildcards `@` are supported, as is the `(null)` species which enables null-reactions, as further discussed below. An example JSON specification that enables secondary electron emission due to photons and ion impact is

```
"dielectric emission":
[
   {
      "reaction": "Y -> e",
      "efficiency": 0.1
   },
   {
      "reaction": "Y -> (null)",
      "efficiency": 0.9
   }
]
```

The `reaction` field specifies which reaction is triggered: The left hand side is the primary (outgoing) species and the left hand side must contain the secondary emissions. The left-hand side can consist of either photons (e.g., `Y`) or particle (e.g., `O2+`) species. The efficiency field specifies the efficiency of the reaction. When multiple reactions are specified, we randomly sample the reaction according to a discrete distribution with probabilities

$$p(i) = \frac{\nu_i}{\sum_i \nu_i},$$

where ν_i are the efficiencies. Note that the efficiencies do not need to sum to one, and if only a single reaction is specified the efficiency specifier has not real effect. The above reactions include the null reaction in order to ensure that the correct secondary emission probability is reached, where the `(null)` specifier implies that no secondary emission takes place. In the above example the probability of secondary electron emission is 0.1, while the probability of a null-reaction (outgoing particle is absorbed without any associated emission) is 0.9. The above example can be compressed by using a wildcard and an `efficiencies` array as follows:

```
"dielectric emission":
[
   {
   "reaction": "Y -> @",
      "@": ["e", "(null)"],
      "efficiencies": [1,9]
   }
]
```

where for the sake of demonstration the efficiencies are set to 1 and 9 (rather than 0.1 and 0.9). This has no effect on the probabilities $p(i)$ given above.

Field emission

Field emission for a specified species is supported by including a `field emission` specifier. Currently supported expressions are:

1. Fowler-Nordheim emission:

$$
\begin{aligned}
J(E) &= \frac{aF^2}{\phi}\exp\left(-v(f)b\phi^{3/2}F\right), \\
F &= (\beta E)\times 10^9, \\
a &= 1.541434\times 10^{-6}\,\mathrm{AeV/V^2}, \\
b &= 6.830890\,\mathrm{eV^{-3/2}Vnm^{-1}}, \\
v(f) &= 1 - f + (1/6)\,f\log(f), \\
f &\approx 1.439964\,\mathrm{eV^2V^{-1}nm}\times\frac{F}{\phi^2}.
\end{aligned}
$$

   Here, ϕ is the work function (in electron volts) and β is an empirical factor that describes local field amplifications. Note that the above expression gives J in units of $\mathrm{A/nm^2}$.

2. Schottky emission:

$$
\begin{aligned}
J(E) &= \lambda A_0 T^2\exp\left(-\frac{(\phi-\Delta\phi)\,q_\mathrm{e}}{k_\mathrm{B}T}\right), \\
F &= \beta E, \\
A_0 &\approx 1.201736\times 10^6\,\mathrm{Am^{-2}K^{-2}}, \\
\Delta\phi &= \sqrt{\frac{q_\mathrm{e}F}{4\pi\epsilon_0}}.
\end{aligned}
$$

   Here, T is the cathode temperature and ϕ is the work function (in electron volts). The value λ is typically around 0.5. As for the Fowler-Nordheim equation, a factor that describes local field amplifications is given in β.

Warning: The expressions above both use a correction factor β that describes local field amplifications. Note, however, that the number of emitted electrons is proportional to the area of the emitter, and there are no current models in `chombo-discharge` that can compute this effective area.

Below, we show an example that includes both Schottky and Fowler-Nordheim emission

```
"field emission":
[
    {
        "species": "e",
        "surface": "electrode",
        "type": "fowler-nordheim",
        "work": 5.0,
        "beta": 1
    },
    {
        "species": "e",
        "surface": "electrode",
        "type": "schottky",
        "lambda": 0.5
        "temperature": 300,
        "work": 5.0,
        "beta": 1,
    }
]
```

Warnings and caveats

Higher-order reactions

Usually, many rate coefficients depend on the output of other software (e.g., BOLSIG+) and the scaling of rate coefficients is not immediately obvious. This is particularly the case for three-body reactions with BOLSIG+ that may require scaling before running the Boltzmann solver (by scaling the input cross sections), or after running the Boltzmann solver, in which case the rate coefficients themselves might require scaling. In any case the user should investigate the cross-section file that BOLSIG+ uses, and figure out the required scaling.

Important: For two-body reactions, e.g. $A + B \rightarrow \varnothing$ the rate coefficient must be specified in units of $\mathrm{m^3 s^{-1}}$, while for three-body reactions $A + B + C \rightarrow \varnothing$ the rate coefficient must have units of $\mathrm{m^6 s^{-1}}$.

For three-body reactions the units given by BOLSIG+ in the output file may or may not be incorrect (depending on whether or not the user scaled the cross sections).

Townsend coefficients

Townsend coefficients are not fundamentally required for specifying the reactions, but as with the higher-order reactions some of the output rates for three-body reactions might be inconsistently represented in the BOLSIG+ output files. For example, some care might be required when using the Townsend attachment coefficient for air when the reaction $\mathrm{e} + \mathrm{O_2} + \mathrm{O_2} \rightarrow \mathrm{O_2^-} + \mathrm{O_2}$ is included because the rate constant might require proper scaling after running the Boltzmann solver, but this scaling is invisible to the BOLSIG+'s calculation of the attachment coefficient η/N.

Warning: The JSON interface *does not guard* against inconsitencies in the user-provided chemistry, and provision of inconsistent η/N and attachment reaction rates are quite possible.

Tips and tricks

As with fluid drift-diffusion models, numerical instabilities can also occur due to unbounded growth in the plasma density. This is a process which has been linked both to the local field approximation and also to the presence of numerical diffusion. Simulations that fail to stabilize, i.e., where the field strength diverges, may benefit from the following stabilizing features:

1. **Turn off parallel diffusion.**

   The *JSON interface* class permits various diffusion models, some of which turn off backward diffusion completely. Note that this also modifies the amount of *physical* diffusion in this direction.

2. **Use gradient corrections.**

   As discussed earlier, using a gradient correction can help limit non-physical ionization due to backwards-diffusing electrons.

3. **Use downstream particle placement.**

   Because the KMC algorithm solves for the number of particles in a grid cell, distributing new particles uniformly over a grid cell can lead to numerical diffusion where secondary electrons are placed in the wake of primary electrons. Using downstream particle placement often leads to slightly more stable simulations.

4. **Describe the primary species using an Ito solver.**

Similar to the point above, using a fluid solver for the ions may lead to upstream placement of the resulting positive charge.

5. **Use kd-tree particle merging.**

   Currently particle merging strategies are reinitialization and merging based on bounding volume hierarchies. If using reinitialization, new particles can be generated in the wake of old ones and can thus upset the charge distribution and cause numerical backwards diffusion (e.g., of electrons).

6. **Numerically limit reaction rates.**

   It is possible to specify that reaction rates will be numerically limited so that the rate does not exceed a specified threshold of Δt^{-1}. This is done through the JSON interface by setting `limit max k*dt` to some value. Note that this changes the physics of the model, but usually enhances stability at larger time steps. An example is given below:

```
"plasma reactions": [
   {
      "reaction": "e + N2 -> e + e + N2+"
      "type": "constant",
      "value": 1.E-12,
      "limit max k*dt" : 2.0
   }
]
```

This will limit the rate such that $k\left[N\right]_2\Delta t = 2$. I.e., all background species are first absorbed into the rate calculation before the rate is limited. We point out that limiting is not possible if both species on the left hand side are solver variables.

Important: The above features have been implemented in order to push the algorithm towards coarser grids and larger time steps. It is essential that the user checks that the model converges when these features are applied.

5.3.6 Example programs

Example programs that use the Îto-KMC module are given in

- `$DISCHARGE_HOME/Exec/Examples/ItoKMC/AirBasic` for a basic streamer discharge in atmospheric air.
- `$DISCHARGE_HOME/Exec/Examples/ItoKMC/AirDBD` for a streamer discharge over a dielectric.

CHAPTER

SIX

SINGLE-SOLVER APPLICATIONS

6.1 Advection-diffusion model

The advection-diffusion model simply advects a scalar quantity with EB and AMR. The equation of motion is

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{v}\phi - D\nabla\phi) = 0.$$

The full implementation for this model consists of the following classes:

- `AdvectionDiffusionStepper`, which implements *TimeStepper*.
- `AdvectionDiffusionSpecies`, which implements *CdrSpecies*, and thus sparses the initial condition into the problem.
- `AdvectionDiffusionTagger`, which implements *CellTagger* and flags cells for refinement and coarsening.

This module only uses *CdrSolver*.

Tip: The source code for this problem is found in `$DISCHARGE_HOME/Physics/AdvectionDiffusion`. See AdvectionDiffusionStepper for the C++ API for this time stepper.

6.1.1 Time stepping

Two time stepping algorithms are supported:

1. A second-order Runge-Kutta method (Heun's method).
2. An implicit-explicit method (IMEX) which uses corner-transport upwind (CTU, see *CdrCTU*) and a Crank-Nicholson diffusion solve.

   Important: The Crank-Nicholson method is known to be marginally stable.

See *Solver configuration* to see how to select between these two algorithms.

Heun's method

For Heun's method we perform the following steps: Let $\left[\nabla \cdot \mathbf{J}\left(\phi\right)\right]$ be the finite-volume approximation to $\nabla \cdot \left(\mathbf{v}\phi - D\nabla\phi\right)$, where $\mathbf{J}\left(\phi\right) = \mathbf{v}\phi - D\nabla\phi$. The Runge-Kutta advance is then

$$\begin{aligned}\phi' &= \phi^k - \Delta t\left[\nabla \cdot \mathbf{J}\left(\phi^k\right)\right] \\ \phi^{k+1} &= \phi^k - \frac{\Delta t}{2}\left(\left[\nabla \cdot \mathbf{J}\left(\phi^k\right)\right] + \left[\nabla \cdot \mathbf{J}\left(\phi'\right)\right]\right)\end{aligned}$$

Warning: Note that when diffusion and advecting is coupled in this way, we do not include the transverse terms in the *CdrCTU* discretization and limit the time step by

$$\Delta t \leq \frac{1}{\frac{\sum_{i=1}^{d}|v_i|}{\Delta x} + \frac{2dD}{\Delta x^2}},$$

where d is the dimensionality and D is the diffusion coefficient.

IMEX

For the implicit-explicit advance, we use the *CdrCTU* discretization to center the divergence at the half time step. We seek the update

$$\frac{\phi^{k+1} - \phi^k}{\Delta t} - \frac{1}{2}\left[\nabla\left(D\nabla\phi^k\right) + \nabla\left(D\nabla\phi^{k+1}\right)\right] = -\nabla \cdot \mathbf{v}\phi^{k+1/2},$$

which is a Crank-Nicholson discretization of the diffusion equation with a source term centered on $k+1/2$. We use the *CdrCTU* for obtaining the edge states $\phi^{k+1/2}$ and then complete the update by solving the corresponding Helmholtz equation

$$\phi^{k+1} - \frac{\Delta t}{2}\nabla\left(D\nabla\phi^{k+1}\right) = \phi^k - \Delta t\nabla \cdot \mathbf{v}\phi^{k+1/2} + \frac{\Delta t}{2}\nabla\left(D\nabla\phi^k\right).$$

In this case the time step limitation is

$$\Delta t \leq \frac{\Delta x}{\sum_i |v_i|}.$$

Warning: It is possible to use this module with any implementation of `CdrSolver`, but the IMEX discretization only makes sense if the hyperbolic term can be centered on $k + 1/2$.

If the truncation order of $\phi^{k+1/2}$ is $\mathcal{O}\left(\Delta t^2\right)$, the resulting IMEX discretization is globally second order accurate. For the *CdrCTU* discretization the edge states are accurate to $\mathcal{O}\left(\Delta t \Delta x\right)$, so the scheme is globally first order convergent (but with a small error constant).

6.1.2 Initial data

Default behavior

By default, the initial data for this problem is given by a super-Gaussian blob

$$\phi\left(\mathbf{x}, t=0\right) = \phi_0 \exp\left(-\frac{\left|\mathbf{x}-\mathbf{x}_0\right|^4}{2R^4}\right),$$

where ϕ_0 is an amplitude, $\mathbf{x}_0$ is the blob center and R is the blob radius. These are set by the input options

```
AdvectionDiffusion.blob_amplitude     = 1.0      ## Blob amplitude
AdvectionDiffusion.blob_radius        = 0.1      ## Blob radius
AdvectionDiffusion.blob_center        = 0 0 0    ## Blob center
```

Custom value

For a more general way of specifying initial data, `AdvectionDiffusionStepper` has a public member function

```
/*!
  @brief Set the initial data..
  @param[in] a_initData Initial data function.
*/
void
setInitialData(const std::function<Real(const RealVect& a_position)>& a_initData) noexcept;
```

6.1.3 Velocity field

Default behavior

The default velocity field for this class is

$$\begin{aligned} v_x &= -r\omega\sin\theta, \\ v_y &= r\omega\cos\theta, \\ v_z &= 0, \end{aligned}$$

where $r = \sqrt{x^2+y^2}$, $\tan\theta = \frac{x}{y}$. I.e. the flow field is a circulation around the Cartesian grid origin.

To adjust the velocity field through ω, simply set the following option:

```
AdvectionDiffusion.omega              = 1.0      ## Rotation velocity
```

Custom value

For a more general way of setting a user-specified velocity, `AdvectionDiffusionStepper` has a public member function

```
/*!
  @brief Set the velocity field.
  @param[in] a_velocity Velocity field.
*/
void
setVelocity(const std::function<RealVect(const RealVect& a_position)>& a_velocity) noexcept;
```

6.1.4 Diffusion coefficient

Default behavior

The default diffusion coefficient for this problem is set to a constant. To adjust it, ω, set

```
AdvectionDiffusion.diffco             = 1.0      ## Diffusion coefficient
```

to a chosen value.

Custom value

For a more general way of setting the diffusion coefficient, `AdvectionDiffusionStepper` has a public member function

```
/*!
  @brief Set the diffusion coeffieint.
  @param[in] a_diffusion Diffusion coefficient
*/
void
setDiffusionCoefficient(const std::function<Real(const RealVect& a_position)>& a_diffusion) noexcept;
```

6.1.5 Boundary conditions

At the EB, this module uses a wall boundary condition (i.e. no flux into or out of the EB). On domain edges (faces in 3D), the user can select between wall boundary conditions or outflow boundary conditions by selecting the corresponding input option for the solver. See *CdrCTU*.

6.1.6 Cell refinement

The cell refinement is based on two criteria:

1. The amplitude of ϕ.
2. The local curvature $|\nabla\phi|\,\Delta x/\phi$.

We refine if the curvature is above some threshold ϵ_1 *and* the amplitude is above some threshold ϵ_2. These can be adjusted through

```
# Cell tagging controls
# ---------------------
AdvectionDiffusion.refine_curv     = 0.1     ## Refine if curvature exceeds this
AdvectionDiffusion.refine_magn     = 1E-2    ## Only tag if magnitude eceeds this
AdvectionDiffusion.buffer          = 0       ## Grow tagged cells
```

By default, every cell that fails to meet the above two criteria is coarsened.

6.1.7 Setting up a new problem

To set up a new problem, using the Python setup tools in `$DISCHARGE_HOME/Physics/AdvectionDiffusion` is the simplest way. A full description is available in the `README.md` contained in the folder:

```
# Physics/AdvectionDiffusion
This physics module solves for an advection-diffusion process of a single scalar quantity. This module contains files for␣
↪setting up the initial conditions
and advected species, basic integrators, and a cell tagger for refining grid cells.

The source files consist of the following:

* **CD_AdvectionDiffusionSpecies.H** Implementation of CdrSpecies, for setting up initial conditions and turning on/off␣
↪advection and diffusion.
* **CD_AdvectionDiffusionTagger.H/cpp**  Implementation of CellTagger, for flagging cells for refinement and coarsening.
* **CD_AdvectionDiffusionStepper.H/cpp** Implementation of TimeStepper, for advancing the advection-diffusion equation.

# Setting up a new problem
To set up a new problem, use the Python script. For example:

```shell
./setup.py -base_dir=MyApplications -app_name=MyAdvectionDiffusion -geometry=CoaxialCable
```

The application will be installed to $DISCHARGE_HOME/MyApplications/MyAdvectionDiffusion.
The user will need to modify the geometry and set the initial conditions through the inputs file.
```

To see available setup options, use

```
./setup.py --help
```

6.1.8 Solver configuration

The `AdvectionDiffusionStepper` and `AdvectionDiffusionTagger` classes come with user-configurable input options that can be adjusted at runtime. The configuration options for `AdvectionDiffusionStepper` are given below:

```
# ================================================================================================
# AdvectionDiffusionStepper class options
# ================================================================================================
AdvectionDiffusion.verbosity      = -1      ## Verbosity
AdvectionDiffusion.diffusion      = true    ## Turn on/off diffusion
AdvectionDiffusion.advection      = true    ## Turn on/off advection
AdvectionDiffusion.integrator     = imex    ## 'heun' or 'imex'

# Default velocity, diffusion, and initial data
# ---------------------------------------------
AdvectionDiffusion.blob_amplitude     = 1.0      ## Blob amplitude
AdvectionDiffusion.blob_radius        = 0.1      ## Blob radius
AdvectionDiffusion.blob_center        = 0 0 0    ## Blob center
AdvectionDiffusion.omega              = 1.0      ## Rotation velocity
AdvectionDiffusion.diffco             = 1.0      ## Diffusion coefficient

# Time step settings
# ------------------
AdvectionDiffusion.cfl        = 0.5      ## CFL number
AdvectionDiffusion.min_dt     = 0.0      ## Smallest acceptable time step
AdvectionDiffusion.max_dt     = 1.E99    ## Largest acceptable time step

# Cell tagging controls
# ---------------------
AdvectionDiffusion.refine_curv     = 0.1     ## Refine if curvature exceeds this
AdvectionDiffusion.refine_magn     = 1E-2    ## Only tag if magnitude eceeds this
AdvectionDiffusion.buffer          = 0       ## Grow tagged cells
```

6.1.9 Example programs

Some example programs for this module are given in

- `$DISCHARGE_HOME/Exec/Examples/AdvectionDiffusion/DiagonalFlowNoEB`
- `$DISCHARGE_HOME/Exec/Examples/AdvectionDiffusion/PipeFlow`

6.1.10 Verification

Spatial and temporal convergence tests for this module (and thus also the underlying solver implementation) are given in

- `$DISCHARGE_HOME/Exec/Convergence/AdvectionDiffusion/C1`
- `$DISCHARGE_HOME/Exec/Convergence/AdvectionDiffusion/C2`

C1: Spatial convergence

A spatial convergence test is given in `$DISCHARGE_HOME/Exec/Convergence/AdvectionDiffusion/C1`. The problem solves for an advected and diffused scalar in a rotational velocity in the presence of an EB:

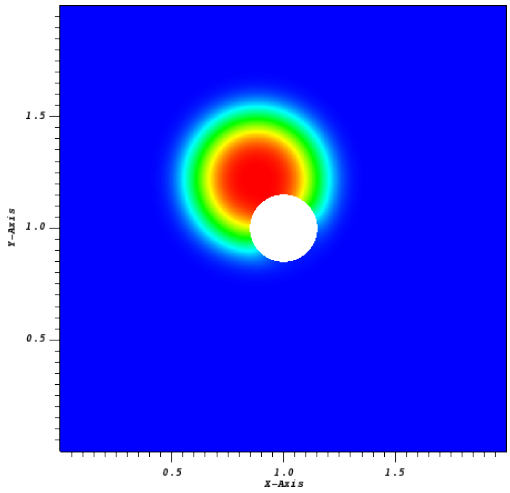

Fig. 6.1.1: Final state on a 512^2 uniform grid.

To compute the convergence rate we compute two solutions with grid spacings Δx and $\Delta x/2$, and estimate the solution error using the approach in *Spatial convergence*. Figure Fig. 6.1.2 shows the computed convergence rates with various choice of slope limiters. We find 2nd order convergence in all three norms for sufficiently fine grid when using slope limiters, and first order convergence when limiters are turned off. The reduced convergence rates at coarser grids occur due to 1) insufficient resolution of the initial density profile and 2) under-resolution of the geometry.

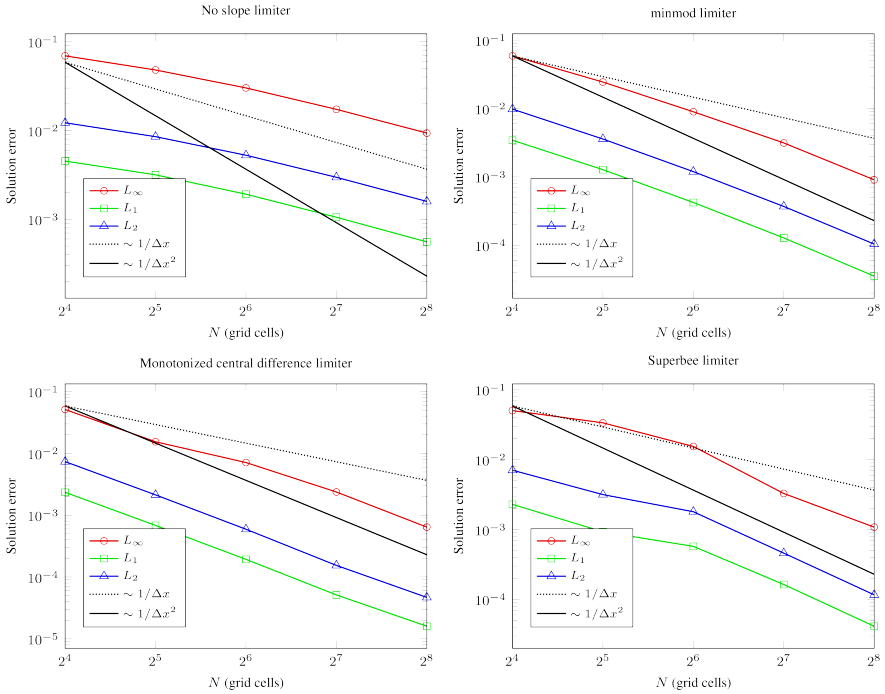

Fig. 6.1.2: Spatial convergence rates with various limiters.

C2: Temporal convergence

A temporal convergence test is given in `$DISCHARGE_HOME/Exec/Convergence/AdvectionDiffusion/C2`. To compute the temporal convergence rate we compute two solutions using time steps Δt and $\Delta t/2$, and estimate the solution error using the approach in *Temporal convergence*. Figure Fig. 6.1.3 shows the computed convergence rates for the second order Runge-Kutta and the IMEX discretization. As expected, we find 2nd order convergence for Heun's method and first order convergence for the IMEX discretization.

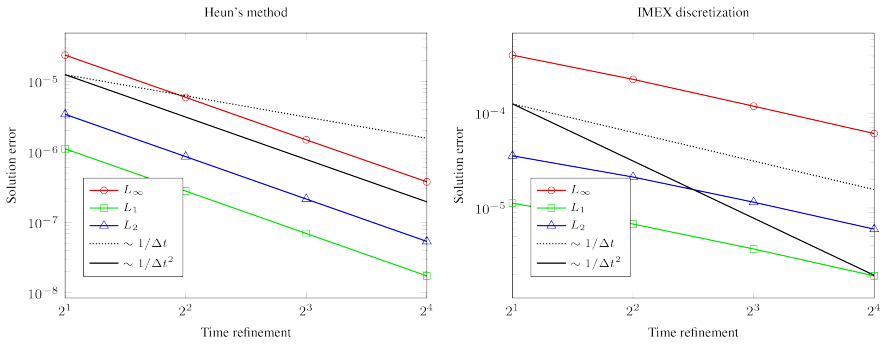

Fig. 6.1.3: Temporal convergence rates.

6.2 Brownian walker

The Brownian walker model runs a single microscropic drift-diffusion using the *ItoSolver*, where the underlying transport kernel is

$$d\mathbf{X} = \mathbf{V}dt + \sqrt{2Ddt}\mathbf{W}$$

where $\mathbf{X}$ is the spatial position of a particle $\mathbf{V}$ is the drift velocity and D is the diffusion coefficient *in the continuum limit*.

Tip: Source code is located in `$DISCHARGE_HOME/Physics/BrownianWalker`.

The model consists of the following implementation files:

- `CD_BrownianWalkerStepper.H/cpp` which implements *TimeStepper*.
- `CD_BrownianWalkerSpecies.H/cpp` which implements initial conditions through *ItoSpecies*.

6.2.1 Initial data

The initial data is randomly generated by sampling from an exponential distribution with specified radius and center. The user can specify the number of particles, and control this sampling through the respective input options, see *Solver configuration*.

6.2.2 Transport specification

The default velocity field for this class is

$$\begin{aligned} v_x &= -r\omega \sin\theta, \\ v_y &= r\omega \cos\theta, \\ v_z &= 0, \end{aligned}$$

where $r = \sqrt{x^2 + y^2}$, $\tan\theta = \frac{x}{y}$. I.e. the flow field is a circulation around the Cartesian grid origin. The diffusion coefficient is simply set to a constant value.

To adjust the velocity field or diffusion coefficients, adjust the configuration options given in *Solver configuration*.

6.2.3 Time stepping

Time stepping in this module is performed using a simple Euler-Maruyama scheme, i.e.,

$$\mathbf{X}^{k+1} = \mathbf{X}^k + \mathbf{V}\Delta t + \sqrt{2D\Delta t}\mathbf{W}$$

It is possible to adjust the time step size by setting the appropriate option (`BrownianWalker.cfl`). The time step is then limited by a particle-like CFL time step condition on the advective part (i.e., on $\mathbf{V}$).

6.2.4 Superparticle handling

Superparticle handling in this module occurs via *ItoSolver*, so the user needs to specify the superparticle algorithm through the *ItoSolver* configuration options. However, the *number* of superparticles is adjusted through `BrownianWalkerStepper`, and is set through `BrownianWalker.ppc`. Setting this value to anything less than 1 will turn off super-particle handling.

6.2.5 Solver configuration

The `BrownianWalkerStepper` class comes with user-configurable input options that can be adjusted at runtime. The configuration options for `BrownianWalkerStepper` are given below:

```
# ================================================================================================
# BrownianWalkerStepper class options.
# ================================================================================================
BrownianWalker.verbosity          = -1        ## Verbosity
BrownianWalker.realm              = primal    ## Realm
BrownianWalker.diffusion          = true      ## Turn on/off diffusion
BrownianWalker.advection          = true      ## Turn on/off advection
BrownianWalker.blob_amplitude     = 1.0       ## Blob amplitude
BrownianWalker.blob_radius        = 0.1       ## Blob radius
BrownianWalker.blob_center        = 0 0       ## Blob center
BrownianWalker.seed               = 0         ## RNG seed
BrownianWalker.num_particles      = 100       ## Number of initial particles
BrownianWalker.cfl                = 1.0       ## CFL-like number.
BrownianWalker.ppc                = -1        ## Particles per cell. <= 0 turns off superparticles
BrownianWalker.load_balance       = true      ## Turn on/off particle load balancing
BrownianWalker.which_balance      = mesh      ## Switch for load balancing method. Either 'mesh' or 'particle'.

# Velocity, diffusion, and CFL
# ----------------------------
BrownianWalker.mobility     = 1.0    ## Mobility coefficient
BrownianWalker.diffco       = 1.0    ## Diffusion coefficient
BrownianWalker.omega        = 1.0    ## Rotation velocity for advection field

# Cell tagging stuff
# ------------------
BrownianWalker.refine_curv     = 0.1     ## Refine if curvature exceeds this
BrownianWalker.refine_magn     = 1E-2    ## Only tag if magnitude eceeds this
BrownianWalker.buffer          = 0       ## Grow tagged cells
```

6.2.6 Setting up a new problem

To set up a new problem, using the Python setup tools in `$DISCHARGE_HOME/Physics/BrownianWalker` is the simplest way. A full description is available in the `README.md` contained in the folder:

```
# Physics/BrownianWalker
This physics module solves for advection-diffusion process of a single scalar quantity. This module contains files for setting␣
↪up the initial conditions
and species, integrators, and a cell tagger for refining grid cells. See https://chombo-discharge.github.io/
↪BrownianWalkerModel.html for implementation details.

The source files consist of the following:

* **CD_BrownianWalkerSpecies.H/cpp** Implementation of ItoSpecies, for setting up initial conditions and turning on/off␣
↪advection and diffusion.
* **CD_BrownianWalkerTagger.H/cpp**  Implementation of CellTagger, for flagging cells for refinement and coarsening.
* **CD_BrownianWalkerStepper.H/cpp** Implementation of TimeStepper, for advancing the advection-diffusion equation.

## Setting up a new application
To set up a new problem, use the Python script. For example:

```shell
./setup.py -base_dir=myApplications -app_name=myBrownianWalker -dim=2 -geometry=CoaxialCable
```

The application will be installed to $DISCHARGE_HOME/myApplications/myBrownianWalker.
The user will need to modify the geometry and set the initial conditions through the inputs file.
```

(continues on next page)

(continued from previous page)

```
## Modifying the application
The application is simply set up to advect and diffuse a scalar in a rotating flow.
Users are free to modify this application, e.g. adding new initial conditions and flow fields.
```

To see available setup options, use

```
./setup.py --help
```

6.3 Electrostatics model

The electrostatics model solves

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

subject to the constraints and boundary conditions given in *FieldSolver*.

There is currently no *CellTagger* implementation for this module, so AMR support is restricted to refinement based on geometric criteria. The full implementation for this model consists only of the following class:

- `FieldStepper`, which implements *TimeStepper*.

Tip: The source for this module is found in `$DISCHARGE_HOME/Physics/Electrostatics`. The C++ API is found here.

6.3.1 Time stepping

`FieldStepper` is a class with only stationary solves. This is enforced by making the `TimeStepper::advance` method throw an error, while the field solve itself is done in the initialization procedure in `TimeStepper::postInitialize`.

Warning: To run the solver, one must set `Driver.max_steps = 0`.

6.3.2 Setting the space charge

Default behavior

By default, the initial space charge for this problem is given by a Gaussian

$$\rho(\mathbf{x}) = \rho_0 \exp\left(-\frac{|\mathbf{x}-\mathbf{x}_0|^2}{2R^2}\right),$$

where ρ_0 is an amplitude, $\mathbf{x}_0$ is the center and R is the Gaussian radius. These are set through respective configuration options, see *Solver configuration*.

For a more general way of specifying the space charge, `FieldStepper` has a public member function

```
/*!
  @brief Set space charge
  @param[in] a_rho Space charge distribution
*/
void
setRho(const std::function<Real(const RealVect& a_pos)>& a_rho, const phase::which_phase a_phase) noexcept;
```

6.3.3 Setting the surface charge

By default, the initial surface charge is set to a constant, see *Solver configuration*. For a more general way of specifying the surface charge, `FieldStepper` has a public member function

```
/*!
  @brief Set surface charge
  @param[in] a_sigma Surface charge distribution
*/
void
setSigma(const std::function<Real(const RealVect& a_pos)>& a_sigma) noexcept;
```

6.3.4 Solver configuration

The `FieldStepper` has relatively few configuration options, which are mostly limited to setting the space and surface charge The configuration options for `FieldStepper` are given below:

```
# ================================================================================================
# FIELD_STEPPER CLASS OPTIONS
# ================================================================================================
FieldStepper.verbosity        = -1        ## Verbosity
FieldStepper.realm            = primal    ## Realm where solver lives.
FieldStepper.load_balance     = false     ## Load balance or not.
FieldStepper.box_sorting      = morton    ## If you load balance you can redo the box sorting.
FieldStepper.init_sigma       = 0.0       ## Surface charge density
FieldStepper.init_rho         = 0.0       ## Space charge density (value)
FieldStepper.rho_center       = 0 0 0     ## Space charge blob center
FieldStepper.rho_radius       = 1.0       ## Space charge blob radius
```

Important: Setup of boundary conditions is *not* a job for `FieldStepper`, but occurs through the respective *ComputationalGeometry* and *FieldSolver*.

6.3.5 Setting up a new problem

To set up a new problem, using the Python setup tools in `$DISCHARGE_HOME/Physics/Electrostatics` is the simplest way. A full description is available in the `README.md` contained in the folder:

```
# Physics/Electrostatics
This physics module solves the Poisson equation. It does not feature mesh refinement, but this is possible to implement by␣
↪adding a CellTagger to the module.
See https://chombo-discharge.github.io/ElectrostaticsModel.html for implementation details.

The source files consist of the following:

* **CD_FieldStepper.H/cpp** Implementation of TimeStepper, for solving the Poisson equation.

## Setting up a new application
To set up a new problem, use the Python script. For example:

```shell
./setup.py -base_dir=MyApplications -app_name=MyElectrostatics -geometry=CoaxialCable
```

The application will be installed to $DISCHARGE_HOME/MyApplications/MyElectrostatics.
The user will need to modify the geometry and set the initial conditions through the inputs file.

Modifying the application
Users are free to modify this application, e.g. adding support for mesh refinement or setting up more complex boundary␣
↪conditions.
```

To see available setup options, use

```
./setup.py --help
```

6.3.6 Example programs

Some example programs for this module are given in

- `$DISCHARGE_HOME/Exec/Examples/Electrostatics/Armadillo`
- `$DISCHARGE_HOME/Exec/Examples/Electrostatics/Mechshaft`
- `$DISCHARGE_HOME/Exec/Examples/Electrostatics/ProfiledSurface`

6.3.7 Verification

Spatial convergence tests for this module (and consequently the underlying numerical discretization) are given in

- `$DISCHARGE_HOME/Exec/Convergence/Electrostatics/C1`
- `$DISCHARGE_HOME/Exec/Convergence/Electrostatics/C2`
- `$DISCHARGE_HOME/Exec/Convergence/Electrostatics/C3`

All tests operate by computing solutions on grids with resolutions Δx and $\Delta x/2$ and then computing the solution error using the approach outlined in *Spatial convergence*.

C1: Coaxial cable

`$DISCHARGE_HOME/Exec/Convergence/Electrostatics/C1` is a spatial convergence test in a coaxial cable geometry. Figure Fig. 6.3.1 shows the field distribution. Note that there is a dielectric embedded between the two cylindrical shells.

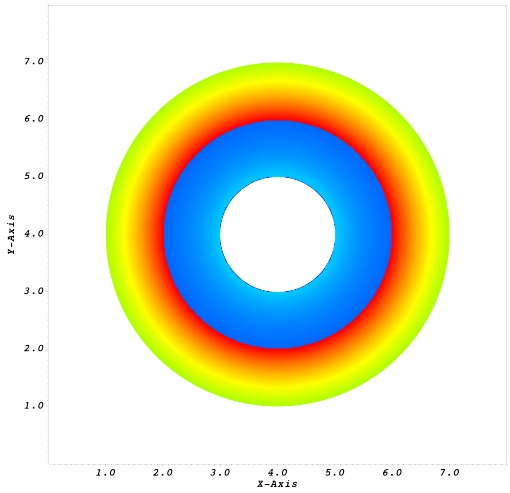

Fig. 6.3.1: Field distribution for a coaxial cable geometry on a 512^2 grid.

The computed convergence rates are given in Fig. 6.3.2. We find second order convergence in all three norms.

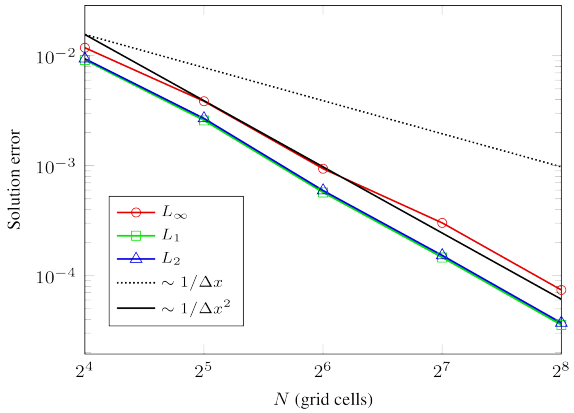

Fig. 6.3.2: Spatial convergence rates for two-dimensional coaxial cable geometry.

C2: Profiled surface

`$DISCHARGE_HOME/Exec/Convergence/Electrostatics/C2` is a 2D spatial convergence test for an electrode and a dielectric slab with surface profiles. Figure Fig. 6.3.3 shows the field distribution.

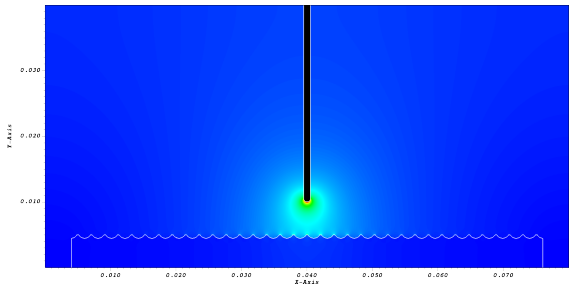

Fig. 6.3.3: Field distribution for a profiled surface geometry on a 2048^2 grid.

The computed convergence rates are given in Fig. 6.3.4. We find second order convergence in all three norms.

C3: Dielectric shaft

`$DISCHARGE_HOME/Exec/Convergence/Electrostatics/C3` is a 3D spatial convergence test for a dielectric shaft perpendicular to the background field. Figure Fig. 6.3.5 shows the field distribution for a 256^3 grid.

The computed convergence rates are given in Fig. 6.3.6. We find second order convergence in L_1 and L_2 on all grids, and find second order convergence in the max-norm on sufficiently fine grids. On coarser grids, the reduced convergence rate in the max-norm is probably due to under-resolution of the geometry.

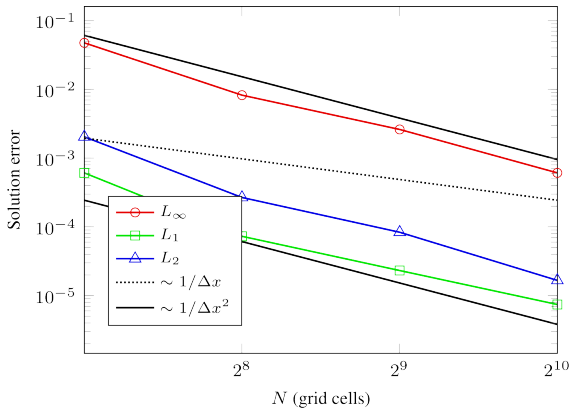

Fig. 6.3.4: Spatial convergence rates for two-dimensional dielectric surface profile.

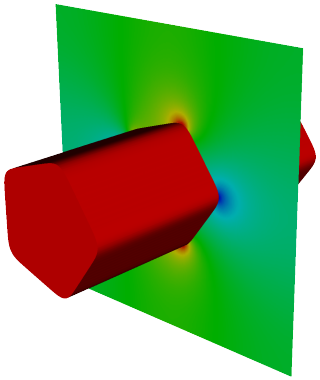

Fig. 6.3.5: Field distribution for a profiled surface geometry on a 256^3 grid.

6.4 Geometry

6.4.1 Functionality

The `GeometryStepper` class is a solver-less class used for supporting the development of new geometries, which in general derive from *ComputationalGeometry*. `GeometryStepper` implements *TimeStepper* without any true functionality. E.g., calling the `advance` method simply returns an infinitely large time step, and `GeometryStepper` provides no I/O functionality or cell refinemention functionality.

Important: Using `GeometryStepper` is the simplest way of creating new geometries since all other functionality other than geometry generation is essentially turned off.

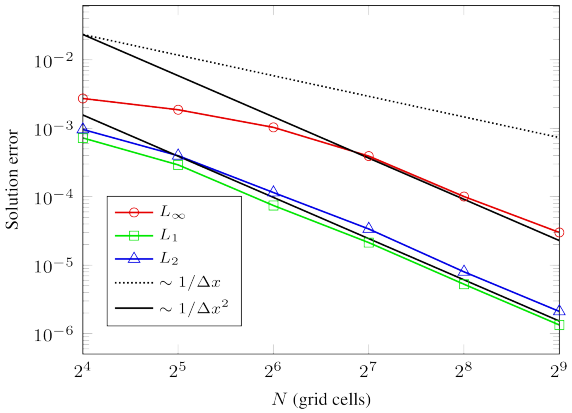

Fig. 6.3.6: Spatial convergence rates.

6.4.2 Setting up a new problem

To set up a new problem, using the Python setup tools in `$DISCHARGE_HOME/Physics/Geometry` is the simplest way. A full description is available in the `README.md` contained in the folder:

```
# Physics/Geometry
This physics module only sets up a geometry -- it does include any solvers whatsoever and so all TimeStepper routines are␣
↪empty.
This is typically used when developing/testing a new geometry.
See https://chombo-discharge.github.io/Geometry.html for implementation details.

The source files consist of the following:

* **CD_GeometryStepper.H/cpp** Implementation of TimeStepper -- does not provide ANY solver functionality and can only␣
↪instantiate a geometry.

## Setting up a new application
To set up a new problem, use the Python script. For example:

```shell
./setup.py -base_dir=MyApplications -app_name=myGeometry -dim=2 -geometry=CoaxialCable
```

The application will be installed to $DISCHARGE_HOME/MyApplications/myGeometry.

Modifying the application
Users are free to modify this application.
```

To see available setup options, use

```
./setup.py --help
```

6.5 Mesh ODE

The `MeshODEStepper` module is designed to illustrate the usage of a *Mesh ODE solver*. This model has no functionality for cell-refinement based on the solutions, so refinement is limited to the geometric refinement methods supplied by *Driver*.

The main class implements *TimeStepper* and is templated as follows:

```
/*!
  @brief Implementation of TimeStepper for solving an ODE on a mesh. N is the number of variables.
*/
template <size_t N>
class MeshODEStepper : public TimeStepper
```

Here, N is the number of variables that will be stored on the mesh.

6.5.1 Example problem

The example problem set up by MeshODEStepper is

$$\partial_t \phi(t) = \cos\left(2\pi f t\right),$$

where f is a user-supplied frequency. The user can also set the initial value for ϕ through the configuration options for MeshODEStepper<N>, see *Solver configuration*

6.5.2 Time advancement

Integration over a time step is done using either a second (Heun's method) or fourth order Runge-Kutta method (classical RK4). The user can choose between the methods through the configuration options, see *Solver configuration*.

6.5.3 Setting up a new problem

To set up a new problem, using the Python setup tools in $DISCHARGE_HOME/Physics/MeshODEStepper is the simplest way. A full description is available in the README.md contained in the folder:

```
# Physics/MeshODE
This physics module solves for an ODE problem of a single scalar quantity. This module contains files for setting up problems.

The source files consist of the following:

* **CD_MeshODEStepper.H** Implementation of TimeStepper, for advancing the equations of motion.

# Setting up a new problem
To set up a new problem, use the Python script. For example:

```shell
./setup.py -base_dir=MyApplications -app_name=MyProblem -geometry=CoaxialCable
```

The application will be installed to $DISCHARGE_HOME/MyApplications/MyProblem.
The user will need to modify the geometry and set the initial conditions through the inputs file.
```

To see available setup options, use

```
./setup.py --help
```

Note: The Python setup tools will set up a single scalar (i.e., N=1).

6.5.4 Solver configuration

The MeshODEStepper class comes with configurable input options that can be adjusted at runtime, which are listed below

```
# ================================================================================================
# MeshODEStepper class options
# ================================================================================================
MeshODEStepper.verbosity       = -1      ## Verbosity level
MeshODEStepper.integration     = rk2     ## 'euler', 'rk2', or 'rk4'
MeshODEStepper.init_phi        = 0.0     ## Initial value for phi.
MeshODEStepper.frequency       = 10      ## Frequency
MeshODEStepper.dt              = 1E-3    ## Time step
```

6.6 Radiative transfer

The radiative transfer model runs a single solver using the *RtSolver* interface, where the underlying solver can derived from any subclass of *RtSolver*. This module is designed to show how to set up and run *RtSolver*, and the code was not written to solve any particular problem. Selecting between different types of solvers must be done at compile time.

Tip: The source code is located in `$DISCHARGE_HOME/Physics/RadiativeTransfer`.

The model consists of the following implementation files:

- `CD_RadiativeTransferStepper.H` which implements *TimeStepper*.
- `CD_RadiativeTransferSpecies.H` which implements the necessary transport conditions through *RtSpecies*.

Note: The current radiative transfer module does not incorporate solver-based adaptive mesh refinement, so refinement is restricted to refinement and coarsening through the *Driver* interface.

6.6.1 Basic problem

Currently, `RadiativeTransferStepper` simply instantiates a solver and advances the solution using a synthetic source given by

$$\eta\left(\mathbf{x}\right) = \eta_0 \exp\left[-\frac{\left(\mathbf{x}-\mathbf{x}_0\right)^2}{2R^2}\right],$$

where the source strength η_0, source radius R, and source center $\mathbf{x}_0$ are set by the user. Similarly, the user can set the photon absorption length, see *Solver configuration*.

6.6.2 Solver configuration

The `RadiativeTransferStepper` class comes with user-configurable input options that can be adjusted at runtime. These configuration options are given below.

```
# ================================================================================================
# RadiativeTransferStepper class options
# ================================================================================================
RadiativeTransferStepper.verbosity          = -1        ## Verbosity
RadiativeTransferStepper.realm              = primal    ## Realm
RadiativeTransferStepper.kappa              = 1.0       ## Inverse absorption coefficient
RadiativeTransferStepper.dt                 = 1.E-10    ## Time step
RadiativeTransferStepper.blob_amplitude     = 1.0       ## Blob amplitude
RadiativeTransferStepper.blob_radius        = 0.1       ## Blob radius
RadiativeTransferStepper.blob_center        = 0 0       ## Blob center
```

6.6.3 Setting up a new problem

To set up a new problem, using the Python setup tools in `$DISCHARGE_HOME/Physics/RadiativeTransfer` is the simplest way. A full description is available in the `README.md` contained in the folder:

```
# Physics/RtPhysics
This physics module solves for a radiative transfer problem. This module contains files for setting up the initial conditions␣
↪and integrators. It does not feature adaptive mesh refinement.
See https://chombo-discharge.github.io/RtPhysicsModel.html for implementation details.

The source files consist of the following:

* **CD_RtPhysicsSpecies.H/cpp** Implementation of RtSpecies, for setting up a radiative transfer equation.
* **CD_RtPhysicsStepper.H/cpp** Implementation of TimeStepper, for advancing the radiative transfer equation, either␣
↪stationary or transient.

## Setting up a new application
To set up a new problem, use the Python script. For example:

```shell
./setup.py -base_dir=myApplications -app_name=myRtPhysics -dim=2 -geometry=CoaxialCable -RtSolver=EddingtonSP1
```

Note that the user can choose between either using discrete or continuum models with the -RtSolver flag.
Acceptable options are EddingtonSP1 or McPhoto.

The application will be installed to $DISCHARGE_HOME/myApplications/myRtPhysics.
The user will need to modify the geometry and set the initial conditions through the inputs file.

Modifying the application
By default, this application specifies a Gaussian source for the photons.
Users are free to modify this application, e.g. adding other initial conditions.
```

To see available setup options, use

```
./setup.py --help
```

Tip: To set up a new problem using either a Monte-Carlo or diffusion-based radiative transfer solver, pass in the flag `-RtSolver` to set up the problem using the desired solver.

6.7 Tracer particle model

The tracer particle model was written in order to illustrate the usage of *TracerParticleSolver*. The module can trace particles passively in a fixed velocity fieldk.`).

Tip: The source code is located in `$DISCHARGE_HOME/Physics/TracerParticle`.

The model consists of the following implementation files:

- `CD_TracerParticleStepper.H`, which implements *TimeStepper* as a subclass `TracerParticleStepper`.

Note: The current implementation does not incorporate solver-based adaptive mesh refinement, so refinement is restricted to refinement and coarsening through the *Driver* interface.

6.7.1 Basic problem

Currently, `TracerParticleStepper` simply instantiates a *TracerParticleSolver* the particles in a fixed velocity field. There is currently no handling of particles that fall off the domain.

Initial conditions

The user can set the velocity field, which is either a velocity field that is diagonal to the Cartesian mesh, or a rotational flow. Likewise, the user can set the number of particles, which are sampled uniformly across the domain.

Integration algorithm

`TracerParticleStepper` implements the following algorithms for the particle advection: The explicit Euler method, Heun's method, and the classical fourth order Runge-Kutta method.

6.7.2 Solver configuration

The `TracerParticleStepper` class comes with user-configurable input options that can be adjusted at runtime. These configuration options are given below.

```
# ================================================================================================
# TracerParticleStepper class options
# ================================================================================================
TracerParticleStepper.initial_particles     = 10000    ## Number of uniformly distributed initial particles
TracerParticleStepper.integration           = euler    ## 'euler', 'rk2', or 'rk4'
TracerParticleStepper.verbosity             = -1       ## Verbosity level
TracerParticleStepper.cfl                   = 1.0      ## "CFL" number.
TracerParticleStepper.velocity_field        = 0        ## Velocity field to use.
                                                       # 0 => Diagonal translation
                                                       # 1 => Rotational flow
```

6.7.3 Setting up a new problem

To set up a new problem, using the Python setup tools in `$DISCHARGE_HOME/Physics/TracerParticle` is the simplest way. A full description is available in the `README.md` contained in the folder:

```
# Physics/TracerParticle
This physics module solves for an tracer particle solver.
This module contains files for setting up the initial conditions
and advected species, basic integrators, and a cell tagger for refining grid cells.
See https://chombo-discharge.github.io/TracerParticleModel.html for implementation details.

The source files consist of the following:

* **CD_TracerParticleStepper.H**  Implementation of TimeStepper, for advancing a tracer particle solver.

## Setting up a new application
To set up a new problem, use the Python script. For example:

```shell
./setup.py -base_dir=MyApplications -app_name=TracerParticle -dim=2 -geometry=CoaxialCable
```

The application will be installed to $DISCHARGE_HOME/MyApplications/TracerParticle.
The user will need to modify the geometry and set the initial conditions through the inputs file.

Modifying the application
The application is simply set up to track particles in various flow fields.
Users are free to modify this application, e.g. adding new initial conditions and flow fields.
```

To see available setup options, use

```
./setup.py --help
```

CHAPTER

SEVEN

UTILITIES

7.1 Data parsing

Routines for reading simple column data into *LookupTable1D* are available. This is typically used, e.g., when parsing transport coefficients or other types of data requires by a computer simulation.

Tip: The `DataParser` C++ API is found at https://chombo-discharge.github.io/chombo-discharge/doxygen/html/namespaceDataParser.html.

Currently, only two types of file reads are supported:

1. Read column data, where the user can specify the columns that are parsed into a lookup table.

   These files can be arranged in the form

   ```
   0.0 1.0
   1.0 2.0
   2.0 3.0
   ```

   It is also possible to restrict which rows that are read, by specifying string identifiers on the line preceding the data and on the line immediately after the data.

   Important: After parsing into a *LookupTable1D*, the user must regularize the table in order to use the interpolation functions. See *Regularize table*.

2. Read particle data, where the particle data is in a (x, y, z, w) file format, where w is the particle weight.

   The function signatures for reading this type of data are:

```
/*!
  @brief Simple file parser which reads a file and puts the data into two columns (a lookup table).
  @details This will read ASCII row/column data into a LookupTable1D (which is a simple x-y data structure). The user can specify which columns
  to use as the x and y coordinates in the LookupTable1D. If the user ask for a column that does not exist in the input file, the data will not be read
  into the LookupTable1D.
  @param[in] a_fileName    Input file name. Must be an ASCII file organized into rows and columns.
  @param[in] a_xColumn     Which column to use as the x-column.
  @param[in] a_yColumn     Which column to use as the y-column.
  @param[in] a_ignoreChars Characters indicating comments in the file. Lines starting with these characters are ignored.
  @note xColumn = 0 is the FIRST column, xColumn=1 is the SECOND column and so on.
*/
LookupTable1D<Real, 1>
simpleFileReadASCII(const std::string       a_fileName,
                    const int               a_xColumn     = 0,
                    const int               a_yColumn     = 1,
```

(continues on next page)

(continued from previous page)

```
                  const std::vector<char> a_ignoreChars = {'#', '/'});

/*!
  @brief Simple ASCII file parser which reads a file and puts the data into two columns (a lookup table)
  @details This version can read a partial file. It will ignore all lines until the a_startRead string is
  encountered. After that, it will read files as usual, skipping lines that begin with comments. Empty lines
  are NOT skipped, since an empty line typically identifies the end of a field from e.g. BOLSIG outputs. Instead
  to stop reading when an empty line is encountered, one can use a_stopRead="".
  @param[in] a_fileName    Input file name. Must be an ASCII file organized into rows and columns.
  @param[in] a_startRead   Identifier where we start parsing lines into the table
  @param[in] a_stopRead    Identifier where we stop parsing lines into the table
  @param[in] a_xColumn     Which column to use as the x-column.
  @param[in] a_yColumn     Which column to use as the y-column.
  @param[in] a_ignoreChars Characters indicating comments in the file. Lines starting with these characters are ignored.
 ↪
*/
LookupTable1D<Real, 1>
fractionalFileReadASCII(const std::string       a_fileName,
                        const std::string       a_startRead,
                        const std::string       a_stopRead,
                        const int               a_xColumn     = 0,
                        const int               a_yColumn     = 1,
                        const std::vector<char> a_ignoreChars = {'#', '/'});

/*!
  @brief Simple file parser which will read particles (position/weight) from an ASCII file.
  @details Particles should be arranged as rows, e.g. in the form
  x   y   z   w
  0.0 0.0 0.0 1
  @param[in] a_fileName    Input file name. Must be an ASCII file organized into rows and columns.
  @param[in] a_xColumn     Column containing the x-coordinate
  @param[in] a_yColumn     Column containing the y-coordinate
  @param[in] a_zColumn     Column containing the z-coordinate
  @param[in] a_wColumn     Column containing the particle weight
  @param[in] a_ignoreChars Characters indicating comments in the file. Lines starting with these characters are ignored.
 ↪
*/
List<PointParticle>
readPointParticlesASCII(const std::string       a_fileName,
                        const unsigned int      a_xColumn     = 0,
                        const unsigned int      a_yColumn     = 1,
                        const unsigned int      a_zColumn     = 2,
                        const unsigned int      a_wColumn     = 3,
                        const std::vector<char> a_ignoreChars = {'#', '/'});
```

7.2 LookupTable1D

`LookupTable1D` is a class for storing and interpolating structured data for lookup in one independent variable. It is used in order to easily retrieve input data that can be stored in table formats.

Important: LookupTable1D is used for data lookup *in one independent variable*. It does not support higher-dimensional data interpolation.

The class is templated as

```
template <typename T = Real, size_t N = 1, typename I = std::enable_if_t<std::is_floating_point<T>::value>>
class LookupTable1D
```

where the template parameter `N` indicates the number of dependent variables (`N=1` yields a compile-time error). Internally, the data is stored as an `std::vector<std::array<T, N + 1> >` where the vector entries are rows and the `std::array<T, N + 1>` are the column entries in that row.

Tip: The internal floating point representation defaults to `Real` and the number of dependent variables to 1.

For performance reasons, the LookupTable1D class is designed to only be used on regularly spaced data (either uniformly or logarithmically spaced). The user can still pass irregularly spaced data into LookupTable1D, and then regularize the data later (i.e., interpolate it onto a regularly spaced 1D grid). Usage of LookupTable1D will therefore consist of the following steps:

1. Add data rows into the table.
2. Swap columns or scale data if necessary.
3. Truncate data ranges if necessary, and specify what happens if the user tries to fetch data outside the valid range.
4. Regularize the table with a specified number of grid points and specified grid spacing.
5. *Retrieve data*, i.e., interpolate data from the table.

All of these steps are discussed below.

7.2.1 Inserting data

To add data to the table, use the member function

```
/*!
  @brief Add entry.
  @param[in] x Entry to add. For example addData(1,1,1,1). Number of elements in x must be N+1
*/
template <typename... Ts>
inline void
addData(const Ts&... x) noexcept;
```

where the parameter pack must have N+1 entries. For example, to add two rows of data to a table with two dependent variables:

```
LookupTable1D<Real, 2> myTable;

myTable.addData(4.0, 5.0, 6.0);
myTable.addData(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. While users will insert rows one by one; LookupTable1D has functions for sorting and regularizing the table later.

7.2.2 Data modification

Scaling

To scale data in a particular column, use

```
/*!
  @brief Utility function which scales one of the columns (either dependent or independent variable)
  @param[in] a_scale Scaling factor
*/
template <size_t K>
inline void
scale(const T& a_scale) noexcept;
```

where K is the column to be scaled.

Column swapping

To swap columns in the table, use

```
/*!
  @brief Utility function for swapping columns.
  @details This is done on the raw data -- if the user wants to swap columns in the resampled/structured data
  then he needs to call this function first and then resample the table.
  @param[in] a_columnOne Column to swap.
  @param[in] a_columnTwo Column to swap.
*/
inline void
swap(const size_t a_columnOne, const size_t a_columnTwo) noexcept;
```

For example, if the input data looked like

```
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
```

Warning: The swapping function only swaps *raw* data. Usually, one must later re-regularize the table, especially if the independent variable was swapped.

Range truncation

One can restrict the data range of the table by calling

```
/*!
  @brief Utility function for truncating raw data along one of the variables (either dependent or independent).
  @details This will discard (from the raw data) all data that fall outside the input interval. This is done on the raw data --
↪ the user will
  need to call prepareTable if the result should propagate into the resampled/structured data.
  @param[in] a_min Minimum value represented.
  @param[in] a_max Maximum value represented.
*/
inline void
truncate(const T& a_min, const T& a_max, const size_t a_column) noexcept;
```

where `a_min` and `a_max` are the permissible ranges for data in the input column (`a_column`). Data outside these ranges is discarded from the table. This applies regardless of whether or not `a_column` indicates the independent or dependent variables.

7.2.3 Regularize table

When regularizing the table, the raw-data (which can be irregularly spaced) is interpolated onto a regular grid. The user must specify:

1. The independent variable in which one will later interpolate.
2. Number of grid points in the regular grid.
3. How to space the grid points.

A `LookupTable1D` is regularized through

```
/*!
  @brief Turn the raw data into uniform data for fast lookup.
  @param[in] a_independentVariable The independent variable (i.e., column in the input data).
  @param[in] a_numPoints Number of points in the input table.
  @param[in] a_spacing Table spacing
*/
inline void
prepareTable(const size_t& a_independentVariable, const size_t& a_numPoints, const LookupTable::Spacing& a_spacing);
```

Here, `a_independentVariable` is the independent variable and `a_numPoints` is the number of grid points in the regularized table. Two different spacings are supported: `LookupTable::Spacing::Uniform` and `LookupTable::Spacing::Exponential`.

Uniform spacing

With uniform spacing, grid points in the table are spaced as

$$x_i = x_{\min} + \frac{i}{N-1}\left(x_{\max} - x_{\min}\right), \quad i \in [0, N-1]$$

where $x_{\min}$ and $x_{\max}$ are the minimum and maximum values in the independent variable, and N is the number of grid points.

Exponential spacing

If grid points are exponentially spaced then the spacing follows a power law:

$$x_i = x_{\min}\left(\frac{x_{\max}}{x_{\min}}\right)^{\frac{i}{N-1}}, \quad i \in [0, N-1].$$

> **Warning:** One must have $x_{\min} > 0$ when using exponentially spaced points.

An example code for regularizing a table is given below:

```
LookupTable1D<Real, 2> myTable;

myTable.prepareTable(0, 100, LookupTable::Spacing::Exponential);
```

7.2.4 Data interpolation

To interpolate data from the table, one can fetch either a specific value in a row, or the entire row. In any case, the values that are returned are linearly interpolated between grid points (in the independent variable). The function signatures are

```
/*!
  @brief Interpolation function for specific dependent variable K
  @param[in] a_x Independent variable x
*/
template <size_t K>
inline T
interpolate(const T& a_x) const;

/*!
  @brief Interpolate whole table.
  @param[in] a_x Independent variable x
*/
inline std::array<T, N + 1>
interpolate(const T& x) const;
```

In the above, the template parameter `K` is the column to retrieve and `a_x` is the value of the independent variable.

Important: `LookupTable1D` 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

```
LookupTable1D<Real, 2> myTable,

const T val = myTable.interpolate<2>(2.0);
```

which will return a value of 4.5 (linearly interpolated).

7.2.5 Out-of-range strategy

Extrapolation outside the valid data range is determined by a user-specified strategy. I.e., when calling the `interpolate` function with an argument that exceeds the bounds of the raw or regular data, the range strategy is either:

- Return the value at the endpoint of the table.
- Extrapolate from the endpoint.

To set the range strategy one can use

```
/*!
  @brief Set the out-of-range strategy on the low end
  @param[in] a_strategy Out-of-range strategy on low end
*/
inline void
setRangeStrategyLo(const LookupTable::OutOfRangeStrategy& a_strategy) noexcept;

/*!
  @brief Set the out-of-range strategy on the high end
  @param[in] a_strategy Out-of-range strategy on the high end
*/
inline void
setRangeStrategyHi(const LookupTable::OutOfRangeStrategy& a_strategy) noexcept;
```

where `a_strategy` must be either of

- `LookupTable::OutOfRangeStrategy::Constant`.
- `LookupTable::OutOfRangeStrategy::Interpolate`.

The default behavior is `LookupTable::OutOfRangeStrategy::Constant`.

7.2.6 Viewing tables

For debugging purposes, `LookupTable1D` can write the internal data to an output stream or a file through various member functions:

```
/*!
  @brief Dump raw table data to file
  @param[in] a_file File name
*/
inline void
writeRawData(const std::string& a_file) const noexcept;

/*!
```

(continues on next page)

(continued from previous page)

```
  @brief Dump structured table data to file
  @param[in] a_file File name
*/
inline void
writeStructuredData(const std::string& a_file) const noexcept;

/*!
  @brief Dump raw table data to output stream.
  @param[in] a_ostream Output stream
*/
inline void
outputRawData(std::ostream& a_ostream = std::cout) const noexcept;

/*!
  @brief Dump structured table data to file.
  @param[in] a_ostream Output stream
*/
inline void
outputStructuredData(std::ostream& a_ostream = std::cout) const noexcept;
```

These functions will print the table (either raw or regularized) to an output stream or file, and the user can later plot the data in an external plotting tool.

7.3 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 supplied by `std::random`.

To use the `Random` class, simply include `<CD_Random.H>`, e.g.,

```
#include <CD_Random.H>
```

See the Random API for further details.

7.3.1 Drawing random numbers

General approach

The general routine for drawing a random number is

```
/*!
  @brief For getting a random number from a user-supplied distribution. T must be a distribution for which we can call T(s_rng)
  @param[in] a_distribution Distribution. Must have object of type
*/
template <typename T>
inline static Real
get(T& a_distribution);
```

Here, the template parameter `T` is some distribution that follows the appropriate C++ template constraints of `<random>`. For example:

```
std::uniform_real_distribution<Real> dist(0.0, 100.0);

const Real randomNumber = Random::get(dist);
```

Pre-defined distributions

Pre-defined distributions exist for performing the following operations:

```
/*!
  @brief Get Poisson distributed number
  @param[in] a_mean Poisson mean value (inverse rate)
*/
template <typename T, typename = std::enable_if_t<std::is_integral<T>::value>>
inline static T
getPoisson(const Real a_mean);

/*!
  @brief Get Poisson distributed number.
  @param[in] a_N Number of trials
  @param[in] a_p The usual success probability in binomial distributions
*/
template <typename T, typename = std::enable_if_t<std::is_integral<T>::value>>
inline static T
getBinomial(const T a_N, const Real a_p) noexcept;

/*!
  @brief Get a uniform real number on the interval [0,1]
*/
inline static Real
getUniformReal01();

/*!
  @brief Get a uniform real number on the interval [-1,1]
*/
inline static Real
getUniformReal11();

/*!
  @brief Get a number from a normal distribution centered on zero and variance 1
*/
inline static Real
getNormal01();

/*!
  @brief Get a random direction in space.
  @details Uses Marsaglia algorithm.
*/
inline static RealVect
getDirection();
```

7.3.2 Seeding the RNG

By default, the random number generator is seeded using the MPI rank. It is necessary to seed MPI ranks using different seeds to avoid them producing the same number sequences. If using MPI+OpenMP, additional steps are taken to ensure that each OpenMP thread obtains a unique random number generator.

`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.

7.4 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. These routines are often needed because the embedded boundary introduces grid pathologies which are difficult to meet with pure finite differencing, see, e.g., *Multigrid ghost cell interpolation*.

7.4.1 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:

$$\begin{pmatrix} 1 & (x_1 - x) & (y_1 - y) \\ 1 & (x_2 - x) & (y_1 - y) \\ \vdots & \ddots & \vdots \\ 1 & (x_N - x) & (y_N - y) \end{pmatrix} \begin{pmatrix} f \\ \partial_x f \\ \partial_y f \end{pmatrix} \left(\mathbf{x}\right) = \begin{pmatrix} f(\mathbf{x}_1) \\ f(\mathbf{x}_2) \\ \vdots \\ f(\mathbf{x}_N) \end{pmatrix}$$

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.

7.4.2 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. 7.4.1.

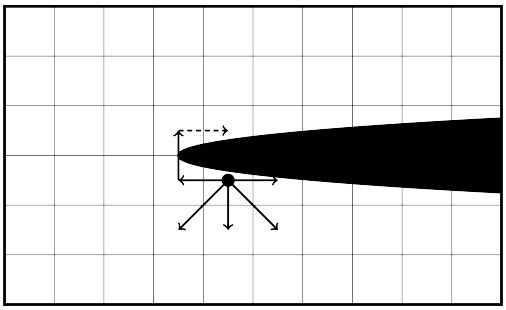

Fig. 7.4.1: Neighborhood algorithm, only reaching into grid cells that can be reached by a monotone path. The grid cell at the end of the dashed line is excluded (even though it is a neighbor to the starting grid cell) since the path circulates the embedded boundary.

7.4.3 Weighted equations

Weights can also be added to each equation, e.g. to ensure that close grid points are more important than remote ones:

$$\begin{pmatrix} w_1 & w_1(x_1-x) & w_1(y_1-y) \\ w_2 & w_2(x_2-x) & w_2(y_2-y) \\ \vdots & \ddots & \vdots \\ w_N & w_N(x_N-x) & w_N(y_N-y) \end{pmatrix} \begin{pmatrix} f \\ \partial_x f \\ \partial_y f \end{pmatrix} (\mathbf{x}) = \begin{pmatrix} w_1 f(\mathbf{x}_1) \\ w_2 f(\mathbf{x}_2) \\ \vdots \\ w_N f(\mathbf{x}_N) \end{pmatrix}$$

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

$$w_{\mathbf{i}} = \frac{1}{|\mathbf{x_i} - \mathbf{x}|^p}.$$

7.4.4 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

$$\begin{pmatrix} f \\ \partial_x f \\ \partial_y f \end{pmatrix} (\mathbf{x}) = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1N} \\ C_{21} & C_{22} & \ddots & C_{2N} \\ C_{31} & C_{32} & \dots & C_{3N} \end{pmatrix} \begin{pmatrix} f(\mathbf{x}_1) \\ f(\mathbf{x}_2) \\ \vdots \\ f(\mathbf{x}_N) \end{pmatrix}$$

7.4.5 Pruning equations

If some terms in the Taylor series are specified, one can prune equations from the systems. E.g. if $f\left(\mathbf{x}\right)$ happens to be known, the system of equations can be rewritten as

$$\begin{pmatrix} w_1(x_1-x) & w_1(y_1-y) \\ w_2(x_2-x) & w_2(y_2-y) \\ \vdots & \vdots \\ w_N(x_N-x) & w_N(y_N-y) \end{pmatrix} \begin{pmatrix} \partial_x f \\ \partial_y f \end{pmatrix} \left(\mathbf{x}\right) = \begin{pmatrix} w_1 f(\mathbf{x}_1) - w_1 f(\mathbf{x}) \\ w_2 f(\mathbf{x}_1) - w_2 f(\mathbf{x}) \\ \vdots \\ w_N f(\mathbf{x}_1) - w_N f(\mathbf{x}) \end{pmatrix}$$

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

$$\begin{pmatrix} \partial_x f \\ \partial_y f \end{pmatrix} \left(\mathbf{x}\right) = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1N} \\ C_{21} & C_{22} & \dots & C_{2N} \end{pmatrix} \begin{pmatrix} f(\mathbf{x}_1) - f(\mathbf{x}) \\ f(\mathbf{x}_2) - f(\mathbf{x}) \\ \vdots \\ f(\mathbf{x}_N) - f(\mathbf{x}) \end{pmatrix}.$$

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.

Tip: 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.

8.1.1 Pull requests

If you want to submit code to `chombo-discharge`, use the pull request system at https://github.com/chombo-discharge. When submitting a pull request, mark it as *draft* if you believe the pull request is not yet ready for merging. Note that when a pull request is submitted, you are asked to provide a brief summary of the changes that are made.

Important: Always squash your git commits into a single commit for the PR as a whole.

It will be beneficial to first compile the changes locally and run e.g. the test suite, see *Code testing* in debug mode. Also make sure to run e.g. *valgrind* to spot memory leaks. In serial, running valgrind can be done by calling valgrind with `--leak-check=yes`, e.g.:

```
valgrind --leak-check=full --track-origins=yes <my_executable> <my_inputs_file>
```

With MPI this looks like

```
mpirun -np 12 valgrind --leak-check=full --track-origins=yes <my_executable> <my_inputs_file>
```

8.1.2 Bug reports

`chombo-discharge` is probably not bug-free. If encountering unexpected behavior, do not hesitate to use the issue tracker at https://github.com/chombo-discharge/chombo-discharge/issues.

8.1.3 Continuous integration

chombo-discharge uses continuous integration (CI) with GitHub actions for:

- Running the test suite (see *Code testing*).
- Building HTML, PDF, and doxygen documentation.
- Ensuring correct code format.

When submitting pull request for review, the above tests will start. In general, all the above tests should pass before merging the pull request into the main branch. Note that all actions on GitHub run in debug mode (DEBUG=TRUE) for turning on assertions.

Upon merging with the main branch, the documentation is again rebuilt and deployed to GitHub pages (see GitHub pages). This ensures that the online documentation (HTML, PDF, and doxygen) is always up-to-date with the latest chombo-discharge release.

8.2 Code standard

When submitting new code to chombo-discharge, the following guidelines below show be followed.

8.2.1 C++ standard

We are currently at C++14.

8.2.2 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.

8.2.3 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.

8.2.4 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. In addition, include header guards identical to the filename, replacing dots by underscores. I.e. for a file CD_MyClass.H the header guard shall read

```
#ifndef CD_MyClass_H
#define CD_MyClass_H

#endif
```

8.2.5 File inclusions

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>`.

8.2.6 Example format

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
```

8.2.7 Code syntax

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 `//`.

8.2.8 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, one may 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
```

8.2.9 clang-format

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 {} +
```

BIBLIOGRAPHY

[R1] 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.

[R2] 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.

[R3] Yang Cao, Daniel T. Gillespie, and Linda R. Petzold. Efficient step size selection for the tau-leaping simulation method. *Journal of Chemical Physics*, 2006. doi:10.1063/1.2159468.

[R4] 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.

[R5] 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.

[R6] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81:2340–2361, 1977. doi:10.1021/j100540a008.

[R7] 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.

[R8] 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.

[R9] 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.

[R10] 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.