PlasmaC Documentation

Release 19.2

Robert Marskar

Aug 09, 2020

PLASMAC INTRODUCTION

This is an alpha release of PlasmaC. Development is still in progress, and various bugs may or may not be present. User interfaces can and will change.

PlasmaC is a modular and scalable computer code for Cartesian two- and three-dimensional simulations of low-temperature plasmas in complex geometries.

- Electrostatics with support for electrodes and dielectrics
- Radiative transport as a diffusion or Monte Carlo process
- Scalar advection-diffusion-reaction transport in complex geometries, with support for fluctuating hydrodynamics
- Îto particle models for microscopic drift-diffusion-reaction processes
- Parallel I/O with HDF5
- Sensible and simple-to-use physics interfaces
- Various time integration schemes
- Dual-grid simulations with separate fluid and particle grids.

Numerical solvers are designed to run on their own, but are best used through physics interfaces.

For scalability, *PlasmaC* is built on top of Chombo, and therefore additionally features

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

Our goal is that users will be able to use *PlasmaC* without modifying the underlying solvers. There are interfaces for describing the plasma physics, setting up boundary conditions, ensuring mesh refinement, and so on. As *PlasmaC* evolves, so will these interfaces. We aim for (but cannot guarantee) backward compatibility such that existing *PlasmaC* models can be run on future versions. This documentation is the user documentation *PlasmaC*. There is a separate Doxygen API that can be compiled together with the source code.

CHAPTER

ONE

INTRODUCTION

1.1 Getting started

This chapter discusses how you may obtain `PlasmaC` and compile it.

1.1.1 Obtaining *PlasmaC*

PlasmaC is obtained by cloning the following repository:

```
git clone ssh://git@git.code.sintef.no/~robertm/plasmac
```

1.1.2 Prerequisites

From the ground up, `PlasmaC` is built on top of the Chombo framework. To compile *PlasmaC*, you must first install the following:

- A Fortran compiler, usually gfortran or Intel Fortran
- A C++ compiler, usually g++ or Intel C++
- An MPI installation
- A parallel HDF5 installation
- LAPACK and BLAS

Usually, laptops and desktops already have appropriate Fortran and C++ compilers installed, as well as a version of MPI. On clusters, HDF5 is (usually) preinstalled, and in this case, it will be sufficient to modify the *Chombo* build files in order to compile *PlasmaC*. Following some changes that we've made to the way *Chombo* generates its embedded boundaries, *Chombo* is released together with *PlasmaC*.

If you already have HDF5 installed, you may skip directly to *Setting up your environment*.

1.1.2.1 Installing HDF5

If you do not have HDF5 installed, you may do the following:

1. Compile and install zlib, which is a compression library used by HDF5. zlib can be installed by

```
sudo apt-get install zlib1g-dev
```

2. Download HDF5 (version 1.8 or newer) and install it for parallel execution

```
./configure --prefix=/usr/local/hdf5_parallel --enable-production --enable-fortran --enable-parallel
make
make install
```

   This will install HDF5 in /usr/local/hdf5_parallel. You may need to install both parallel and serial versions of HDF5.

1.1.2.2 Setting up your environment

In *Chombo*,the system information is supplied through a file known as Make.defs.local, which resides in the *Chombo* library itself. This file contains a number of build settings, such as dimension, compilers, paths to HDF5 and so on. The configuration file is /lib/mk/Make.defs.local in your *Chombo* folder. The build parameters can also be controlled through the command line.

Here are the configuration variables that have been used used on the fram supercomputer

```
DIM           = 3
DEBUG         = FALSE
OPT           = HIGH
#PRECISION    =
#PROFILE      =
CXX           = icpc
FC            = ifort
MPI           = TRUE
## Note: don't set the MPICXX variable if you don't have MPI installed
MPICXX        = mpicxx
#OBJMODEL     =
#XTRACONFIG   =
## Optional features
#USE_64       =
#USE_COMPLEX  =
USE_EB        = TRUE
#USE_CCSE     =
USE_HDF       = TRUE
HDFINCFLAGS   = -I/cluster/software/HDF5/1.10.1-intel-2017a/include
HDFLIBFLAGS   = -L/cluster/software/HDF5/1.10.1-intel-2017a/lib -lhdf5 -lz
## Note: don't set the HDFMPI* variables if you don't have parallel HDF installed
HDFMPIINCFLAGS = -I/cluster/software/HDF5/1.10.1-intel-2017a/include
HDFMPILIBFLAGS = -L/cluster/software/HDF5/1.10.1-intel-2017a/lib -lhdf5 -lz
USE_MF        = TRUE
#USE_MT       =
#USE_SETVAL   =
#CH_AR        =
#CH_CPP       =
#DOXYGEN      =
#LD           =
#PERL         =
#RANLIB       =
#cppdbgflags  =
#cppoptflags  =
#cxxcppflags  =
#cxxdbgflags  =
cxxoptflags   = -Ofast -xCORE-AVX2 -march=native
#cxxprofflags =
#fcppflags    =
#fdbgflags    =
foptflags     = -Ofast -xCORE-AVX2
#fprofflags   =
flibflags     = -lblas -llapack
#lddbgflags   =
```

(continues on next page)

(continued from previous page)

```
#ldoptflags     =
#ldprofflags    =
syslibflags     = -ldl -lm -lz
```

We also recommend that you create environment variables that hold the path to your `PlasmaC` version. For example,

```
PLASMAC_HOME=/home/foo/plasmac
```

This environment variables is used in the `PlasmaC` makefile system so that our makefiles can find `PlasmaC`.

We recommend that you take care of your `Make.defs.local` for all your computers. If you want, you may place your `Make.defs.local` for all your computers into `/src/local` and push to the main repository.

1.1.3 Compiling *PlasmaC*

In *PlasmaC*, each problem is compiled as a mini-application into a subfolder. Mini-apps are usually set up through a Python pre-compilation script that generates the required source code, makefiles, and simulation parameters. There is no separate build for the `PlasmaC` source code and your own application files and you will *not* be able to install `PlasmaC` as a separate library.

Once an application has been set up, the makefile system tracks the necessary *Chombo* and `PlasmaC` source files. Compiling is done in the subfolder that houses your mini-app:

```
make -s -j8 DIM=2 OPT=HIGH <application_name>
```

We generally recommend that you compile with `OPT=HIGH` for performance reasons.

1.1.4 Visualization

PlasmaC writes output files to HDF5. Users can decide what data to output, as well as restrict plot depth to a certain grid levels level. There are also options for including ghost cells in the output files.

Our favorite tool for visualization is VisIt, which can be freely downloaded. Our experience is that client-server visualization is beneficial for visualization of three-dimensional simulation data. For information on how to set up host profiles for VisIt, please contact your local guru or refer to the VisIt documentation.

1.1.5 My first compilation

Before moving on with more complex descriptions of *PlasmaC*, we will try to compile a test problem which simply advects a scalar. The application we will use is a part of the regression testing in *PlasmaC*.

To run this application, navigate to `/regression/advection_diffusion` and compile with

```
make -s -j4 DIM=2 main
```

where `-j4` is the number of cores used for the compilation. If you want to compile this example in 3D, you should put DIM=3. If the application compiles successfully, you will see a file called `main2d.<bunch_of_options>.ex`. If you see this file, you will be able to compile all of *PlasmaC*. If you don't, you won't be able to compile any of it. Before moving on further, please make sure that your model compiles.

Once we have compiled our application, we are ready to run it. The example that we will run is a very simple setup of scalar advection and diffusion of a rotating flow, where the base code is provided in `/physics/advection_diffusion`. To run the example, you can do

```
mpirun -np 4 main2d.<bunch_of_options>.ex regression2d.inputs
```

Output files should now appear in `/regression/advection_diffusion/plt`.

1.1.6 Troubleshooting

If the prerequisites are in place, compilation of `PlasmaC` 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 *PlasmaC*. For that reason we refer you to the *Chombo* user guide for troubleshooting.

1.1.7 Using this documentation

This documentation was built using *reStructuredText* with *Sphinx*. If you want to build a PDF version of this documentation, please navigate to `plasmac/doc/sphinx` and execute

```
make latexpdf
```

A PDF version of this documentation named `PlasmaC.pdf` will appear in `plasma/doc/sphinx/build/latex`.

1.2 The `PlasmaC` code

The `PlasmaC` code is a loosely coupled code targeted at solving plasma problems. The code uses an embedded boundary (EB) adaptive mesh refinement (AMR) formalism where the grids frequently change and are adapted to the solution as simulations progress. By design, the `PlasmaC` does not subcycle and all the grids are advanced using the same time step. `PlasmaC` also supports the concept of a *Realm*, which in short means that `PlasmaC` supports using one set of grids for Eulerian solvers and a different set of grids for Lagrangian solvers.

The core functionality is centered around a set of solvers, for example a Poisson solver and a convection-diffusion-reaction solver, and then using the built-in solver functionality to advance the equations of motion. This is done through a class `time_stepper`, which is an abstract class that advances the equations of motion within the `PlasmaC` framework. The `time_stepper` can instantiate an arbitrary number of solvers, and allows developers to use a fairly high-level description of their problem. For example, all *solvers* have functions like `write_plot_data(...)` that the user may use within the `time_stepper` output routines.

Although many abstractions are in place so that user can describe a new set of physics, or write entirely new solvers into `PlasmaC` and still use the EBAMR formalism, `PlasmaC` also provide some physics modules for describing various types of problems. These modules reside in `/physics` and they are intended to both be problem-solving physics modules, and as well acting like benchmarks, regression tests, and examples for extension to new types of physics modules in the future.

1.2.1 Main functionality

The main functionality in `PlasmaC` is centered around the concept of a *Solver*.

In this section we summarize how components in `PlasmaC` are connected, such that users may understand more readily how the code is designed.

There are four major components *PlasmaC*:

1. A computational geometry which describes a level-set geometry consisting of electrodes and possibly also dielectrics. This functionality is encapsulated by *computational_geometry*.

2. An AMR mesh which contains the grids, grid generation routines, and functionality for handling data coarsening and refinement. This functionality is encapsulated by *amr_mesh*. The `amr_mesh` class acts as a centralized repository for grid generation, performing AMR operations like filling ghost cells, allocating data over the AMR hierarchy and so on. `amr_mesh` is a standalone class - it has no view over the rest of `PlasmaC`.
3. A time stepper which advances the equations of motion (whatever they are). This class has been made abstract with a public interface that is used by the `driver` class (see below). In order to actually use `PlasmaC` for anything the user must either write his own derived *time_stepper* class, or use one of the pre-defined physics modules. The `time_stepper` is purposefully quite general so that the whole `PlasmaC` framework can be set up to solve completely new sets of equations without affecting the rest of the framework.
4. A cell tagger which flags cells for refinement and coarsening. This functionality is encapsulated by the *cell_tagger* class and it, too, is abstract.

Instantiations of the above four classes are fed into the *driver* class which contains calling functions for generation the geometry, having the time integrator to perform and time step, performing, I/O, setting checkpoint/restart and so on. The reason for the above division of labor is that we have wanted to segregate responsibilities in order to increase flexibility. For that reason, the computational geometry does not have any view of the actual AMR grids; it only contains the level-set functions and some meta-information (such as the permittivity of a dielectric). Likewise, the *amr_mesh* class only acts a centralized repository of useful functions for AMR simulations. These functions include algorithms for generating AMR grids, allocating data across AMR, and synchronizing AMR levels (e.g. interpolating ghost cells).

All the physics is encapsulated by the *time_stepper* class. This class will have direct ownership of all the solvers and the functions required to advance them over a time step. Instantiations of the class will also contain the routines for setting up a simulation, e.g. instantiating solvers, setting up boundary conditions. Typically, implementation new physics consists of writing a new `time_stepper` class that allocates the relevant solvers, and then implement the time integration algorithms that advances them. The folder `/physics` contains implementation of a few different physics modules. Since problems within a physics module tend to be conceptually similar, all of these modules also have a Python setup script so that users can quickly set up new types of problems within the same module.

The *driver* class is only responsible for *running* a simulation, and it uses `time_stepper` to do so. The *driver* class will call for regrids at certain intervals, call the *time_stepper* for writing plot and checkpoint data, and also call for the *time_stepper* to advance the equations of motion through a function `advance(...)`. In order to understand how `PlasmaC` runs a simulation, it will be useful to first understand how *driver* works.

1.2.1.1 Solvers

Various solvers are implemented in `PlasmaC`, see *Supported Solvers*. All solvers are designed to run through the `time_stepper` class. Therefore, in order to run only a single solver (e.g. advection-diffusion or Poisson), one must have a `time_stepper` implementation that allocates the appropriate solver, sets it up, and runs it. Currently, there are separate physics modules for each type of solver such that users may see how they are set up and run. These are located in `/physics/`.

The solvers may be abstract or non-abstract. All solvers that are *not* abstract are supplemented by an options file that contain all the possible run-time configurations that can be made to the solver. Such options can include multigrid parameters, how to handle particle deposition with refinement boundaries, slope limiters, etc. For example, all numerical solvers have independent adjustment of output. The input parameters for each solver class is included in a separate file named `<solver>.options` that resides in the same folder as the solver. For example, the input parameters for the default Poisson solver defined in `/src/poisson/poisson_multifluid_gmg.H` is contained in a file `/src/poisson/poisson_multifluid_gmg.options`.

1.2.1.2 Simulation inputs

PlasmaC 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 physics models in PlasmaC are therefore equipped with Python setup tools that collect all such options in a single file. Generally, these input parameters are fetched from the options file of each class 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
```

1.2.1.3 Simulation outputs

Mesh data from PlasmaC simulations is by default written to HDF5 files. Users that wish to write or output other types of data must supply code themselves.

In addition to plot files, MPI ranks can output information to separate files so that the simulation progress can be tracked. See *Controlling PlasmaC* for details. This is also useful for debugging purposes.

1.2.2 Spatial discretization

PlasmaC uses structured adaptive mesh refinement (SAMR provided by Chombo [ACG+04]. SAMR exists in two separate categories, patch-based and tree-based AMR. Patch-based AMR is the more general type and contain tree-based grids as a subset; they can use refinement factors other than 2, as well as accomodate anisotropic resolutions and non-cubic patches. In patch-based AMR the domain is subdivided into a collection of hierarchically nested overlapping patches (or boxes). Each patch is a rectangular block of cells which, in space, exists on a subdomain of the union of patches with a coarser resolution. Patch-based grids generally do not have unique parent-children relations: A fine-level patch may have multiple coarse-level parents. An obvious advantage of a patch-based approach is that entire Cartesian blocks are sent into solvers, and that the patches are not restricted to squares or cubes that align with the coarse-grid boundary. A notable disadvantage is that additional logic is required when updating a coarse grid level from the overlapping region of a finer level. Tree-based AMR use quadtree or octree data structures that describe a hierarchy of unique parent-children relations throughout the AMR levels: Each child has exactly one parent, whereas each parent has multiple children (4 in 2D, 8 in 3D). In PlasmaC and Chombo, computations occur over a set of levels with different resolutions, where the resolution refinement between levels can be a factor 2 or 4. On each level, the mesh is described by a set of disjoint patches (rectangular box in space), where the patches are distributed among MPI processes.

Embedded boundary applications are supported by additionally describing the mesh with a graph near cut-cells. This allows us to combine the efficiency of patch-based AMR with complex geometries. However, there is significant overhead with the embedded boundary approach and, furthermore, arbitrarily complex geometries are not possible.

1.2.2.1 Mesh generation

PlasmaC offers two algorithm for AMR grid generation. Both algorithms work by taking a set of flagged cells on each grid level and generating new boxes that cover the flags. The first algorithm that we support is the classical Berger-Rigoustous grid algorithm that ships with Chombo, see the figure below. The classical Berger-Rigoustous algorithm is serial-like in the sense that is collects the flagged cells onto each MPI rank and then generates the boxes. The algorithm is typically not used at large scale because of its memory consumption.

As an alternative, we also support 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 similar to octrees, but it is more general since it also supports refinement factors other than 2, and is not restricted to domain extensions that are an integer factor of 2 (e.g. 2^{10} cells in each direction).

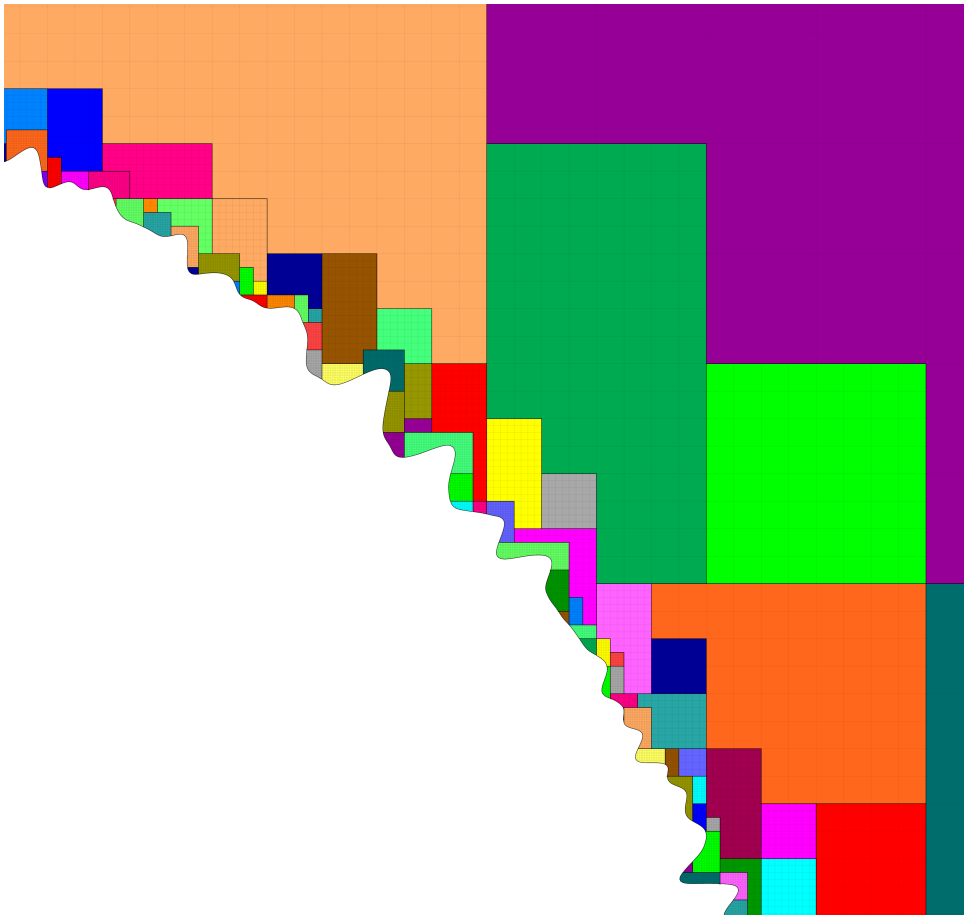


Fig. 1.2.1: Patch-based refinement (factor 4 between levels) of a complex surface. Each color shows a patch, which is a rectangular computational unit.

1.2.2.2 Geometry generation

Geometry generation for `PlasmaC` follows that of Chombo. In Chombo, the geometries are generated from an implicit function $f(\mathbf{x}) = 0$ that describes the level-set surface.

In *Chombo*, geometry generation is done by first constructing a set of boxes that covers the finest AMR level. If the function intersects one of these boxes, the box will allocate a *graph* that describes the connectivity of the volume-of-fluid indices in the entire box. The geometric data in the box is allocated sparsely so that memory consumption due to EB information storage is typically not very high. In general, there should be no graphs in boxes that are all-covered or all-regular.

When EB information is first generated across the AMR hierarchy, one begins by computing the information on the finest grid level. From there, coarser levels are generated through *coarsening* of the fine-information data. The default load-balancing for geometry generation in *Chombo* is an even division of the grid level among the ranks. This is a reasonable approach for porous media where the cut-cells distribute evenly through the computational domain. However, most geometries consists of a small 2D surface in 3D space and the default Chombo approach wastes a lot of time looking for cut-cells where they don't exist.

To achieve scalable geometry generation, we have changed how *Chombo* generates the geometry generation on the various levels. Our new approach first generates a map on a *coarse* level which is specified by the user. On the specified level the domain is broken up into equal-sized chunks and cut-cell boxes are located. Uncut and cut boxes are load balanced among the various ranks. We then proceed towards the next finer level where the cut-cell boxes are

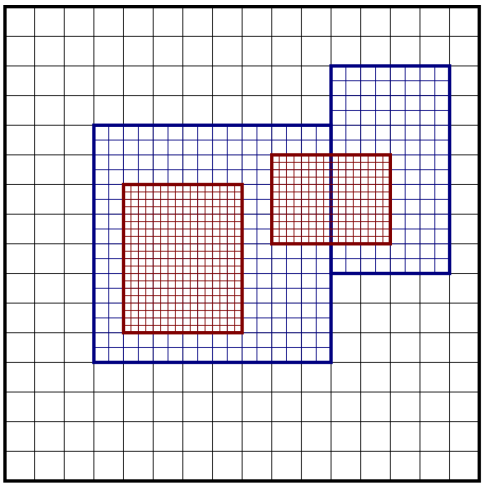


Fig. 1.2.2: Classical cartoon of patch-based refinement. Bold lines indicate entire grid blocks.

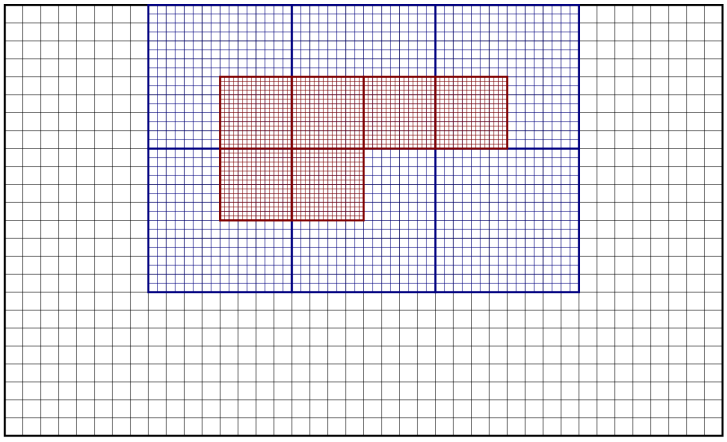


Fig. 1.2.3: Classical cartoon of tiled patch-based refinement. Bold lines indicate entire grid blocks.

identified by a refinement of the box distribution on the previous level. Boxes that resulted from a refinement of the coarse level cut boxes are again broken up into equal-sized chunks, whereas the uncut boxes are not. This is again followed by load-balancing of the cut boxes, and this process is repeated recursively down to the finest AMR level. In essence, the geometry generation is load balanced based on where the cut cells are going to be. For the user, he will be able to switch between the *Chombo* and `PlasmaC` approaches to geometry generation load balancing by flipping a flag in an input script.

1.3 Chombo basics

To fully understand this documentation the user must 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.

1.3.1 Real

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

1.3.2 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 `PlasmaC` 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(vx,vy)` in 2D and `RealVect(vx, vy, vz)` in 3D.

1.3.3 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.

1.3.4 Box

The `Box` object describes a Box in Cartesian space. The boxes are indexed by the low and high corners, both of which are `IntVect``s. 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.

1.3.5 EBCellFAB and FArrayBox

The `EBCellFAB` object is an array for holding cell-centered data in an embedded boundary context. The `EBCellFAB` is defined over a `Box` and the cut-cell geometry in the same `Box`. For cut-cells, there is a chance that the boundaries intersect the grid in such a way that a cell has more than one degree of freedom. Data therefore needs to live on more complex data structures than simple arrays.

The `EBCellFAB` has two data structures: An `FArrayBox` that holds the data on the cell centers, and is supported by a graph that additionally holds data in cells that are multiply cut. The `FArrayBox` is essentially a Fortran array that can be passed to Fortran for performance reasons. Doing arithmetic with `EBCellFAB` usually requires one to iterate over all the cell in the `FArrayBox`, and then to iterate over the *irregular cells* (i.e. cut-cells) later. The `VofIterator` can iterate over any number cells, but is typically used only to iterate over the cut-cells in a `Box`. Typically, code for doing anything with the `EBCellFAB` looks like this:

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

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

1.3.6 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`.

1.3.7 RefCountedPtr

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

In modern C++ speak, `RefCountedPtr<T>` can be thought of as a bare-bones `std::shared_ptr` (without the move semantics and so on).

1.3.8 DisjointBoxLayout

The `DisjointBoxLayout` describes a grid on an AMR level where all the boxes are *disjoint*, i.e. they don't overlap. `DisjointBoxLayout` is built upon a collection of boxes and the MPI rank ownership of those boxes. The constructor is

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

DisjointBoxLayout dbl(boxes, ranks);
```

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

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

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

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

Other data iterators exist that iterate over the *whole* grid:

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

This is typically used if one wants to do some global operation, e.g. count the number of cells in the grid or somesuch. If you try to use `LayoutIterator` to retrieve data that was allocated locally, you will get a memory corruption.

1.3.9 LevelData

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

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

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

where `[DataIndex]` is an indexing operator for `LevelData`.

`LevelData<T>` also includes the concept of ghost cells and exchange operations. Specifying ghost cells is primarily controlled in input scripts for simulations.

1.3.10 EBISLayout and EBISBox

The `EBISLayout` holds the geometric information over one `DisjointBoxLayout` level. Typically, the `EBISLayout` is used to obtain information about the cut-cells. `EBISLayout` also has an indexing operator that can be used to extract a pointer to the geometric information in only one of the boxes on the level. This indexing operator is

```
EBISLayout ebisl(...);
for (DataIterator dit = dbl.dataIterator(); dit.ok(); ++dit){
   const DataIndex& d = dit();

   EBISBox& ebisbox = ebisl[d];
}
```

where `EBISBox` contains the geometric information over only one `Box`.

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

```
const int comp = 0;

LevelData<EBCellFAB> myData(...);
EBISLayout ebisl(...);

for (DataIterator dit = dbl.dataIterator(); dit.ok(); ++dit){
   const DataIndex& d = dit();
   const Box bx       = dbl.get(d);

   EBCellFAB& ebcell = myData[d];
   EBISBox& ebisbox  = ebisl[d];

   const IntVectSet& ivs = ebisbox.getIrregIVS(box);
   const EBGraph&        = ebisbox.getEBGraph();

   for (VoFIterator vofit(ivs, ebgraph); vofit.ok(); ++vofit){
      const VolIndex& vof = vofit();

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

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

CHAPTER

TWO

UNDERSTANDING PLASMAC

2.1 Realm

The `realm` class is a class for centralizing EBAMR-related grids and operators for a specific AMR grid. For example, a `realm` consists of a set of grids (i.e. a `Vector<DisjointBoxLayout>`) as well as *operators*, e.g. functionality for filling ghost cells or averaging down a solution from a fine level to a coarse level.

The terminology *dual grid* is used when more than one `realm` is used in a simulation. With dual grid 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 particle solvers are involved since users can quickly generate an Eulerian set of grids and a set of grids for the Lagrangian particles, and the grids can be load balanced separately.

In general, users will not interact with `realm` directly. Every `realm` is owned by `amr_mesh`, and the user will only interact with realms through the public `amr_mesh` interface.

Internally, an instantiation of `realm` contains the grids and the geometric information (e.g. EBISLayout), as well as any operators that the user has seen fit to *register*. If a solver needs an operator for, say, ghost cell interpolation, the solver needs to *register* that operator through the `amr_mesh` public interface. Operator registration is a run-time procedure. Once an operator has been registered, `realm` will define those operators during e.g. regrids. Run-time abortions with error messages are issued if an AMR operator is called for, but has not been registered.

2.2 Understanding mesh data

Mesh datastructures in `PlasmaC` are derived from a class `EBAMRData<T>` which holds a typename `T` at every box at every level in the AMR hierarchy. Internally, the data is stored as a `Vector<RefCountedPtr<LevelData<T> > >` and the user may fetch the data through `EBAMRData<T>::get_data()`. Here, the `Vector` holds a set of data on each AMR level; the data is allocated with a smart pointer called `RefCountedPtr` which points to a `LevelData` template structure, see *Chombo basics*.

The reason for having class encapsulation of mesh data is due to *Realm*, so that we can only keep track on which realm the mesh data is defined. Users will not have to interact with `EBAMRData<T>` directly, but will do so primarily through `amr_mesh`. `amr_mesh` has functionality for defining most `EBAMRData<T>` types on one of the realms, and `EBAMRData<T>` itself it typically not used anywhere elsewhere within `PlasmaC`.

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

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

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

2.3 Understanding particle data

2.3.1 particle_container

The `particle_container<T>` is a container class that holds particles on an AMR hierarchy, inclusive of information on how to distribute them to grids and remap them. Under the hood, it uses the Chombo structure `ParticleData<T>` which holds the particles on each level, and also the mapping structure `ParticleValidRegion` which dictates on which patch the particles belong. To obtain the particles on one AMR level, the user will call something like

```
particle_container<T> particleContainer(...);

ParticleData<T>& levelParticles = particleContainer[lvl];
```

The indexing operator `[int]` yields the particles on one level.

2.3.2 Allocating particles

`amr_mesh` has a very simple function for allocating a `particle_container`:

```
template <typename T> void allocate(particle_container<T>& a_container, const int a_buffer);
```

which will allocate a `particle_container` with a buffer zone on refined grid levels. This buffer zone adjusts where on the fine levels the particles can live.

2.3.3 ParticleData

On each grid level, particles live in templated data holders

```
template <class T>
ParticleData<T>
```

where `T` is the particle type. Although `ParticleData<T>` can be thought of as a `LevelData<T>`, it actually inherits `LayoutData<ListBox<T> >`. However, `ParticleData<T>` still has routines for communicating particles with MPI.

On each patch, the particles are stored in a `ListBox<T>`. This class essentially consists of the domain box for the patch which holds the particles, and the particles themselves which are stored in a list `List<T>`. The particles are retrieved from a `ListBox<T>` object with a `listItems()` member function. Here is an example:

```
// Assume PD is our particle data and DBL is our grids
for (DataIterator dit = DBL.dataIterator(); dit.ok(); ++dit){
   ListBox<T>& lb = PD[dit()];
   List<T>& particles = lb.listItems();

   // ... do something with the particles
}
```

There are routines for allocating a particle data holder in `amr_mesh`, which simply looks like this:

```
template <typename T >
void allocate(Vector<RefCountedPtr<ParticleData<T> > >& a_particles);
```

This will simply create the `ParticleData<T>` object on all the AMR levels (without any particles).

2.3.4 Iterating over particles

In order to iterate over particles, you will use an iterator without random access:

```
// Assume PD is our particle data and DBL is our grids
for (DataIterator dit = DBL.dataIterator(); dit.ok(); ++dit){
   ListBox<T>& lb = PD[dit()];
   List<T>& particles = lb.listItems();

   ListIterator<T> lit(particles);
   for (lit.rewind(); lit; ++lit){
      T& p = particles[lit];

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

2.3.5 ParticleValidRegion

The `ParticleValidRegion` (PVR) allows particles to be transferred to coarser grid levels if they are within a specified number of grid cells from the refinement boundary. A compelling reason to do this is that if the particle lives on the refinement boundary, its deposition cloud will hang over the refinement boundary and into the ghost cells. So, it is useful to keep the particles on the grid in such a way that the deposition and interpolation kernels are entirel contained within the grid. Another reason is that it might be useful to keep the deposition kernel on a specific AMR level for a number of time steps to reduce the number of times the particles must be moved across AMR levels.

To allocate a PVR, Allocation of a PVR is done from `amr_mesh` (alternatively, call the constructor yourself) as follows:

```
void allocate(EBAMRPVR& a_pvr, const int a_buffer); // buffer is the number of cells from the refinement␣
↪boundary
```

Here, `EBAMRPVR` is simply a typedef'ed `Vector<RefCountedPtr<ParticleValidRegion> >`.

2.3.6 Remapping particles

Particle remapping to the correct MPI ranks must be done if a particle leaves a grid patch and enters a different one, or leaves over a refinement boundary. The figure below shows some typical cases.

`particle_container` has a function that will remap particles over the *whole* AMR hierarchy, including all three cases outlined above. In addition, the class has one particle for remapping only one one level;

```
void remap();                        // Remap everything
void levelRemap(const int a_level); // Remap only one level
```

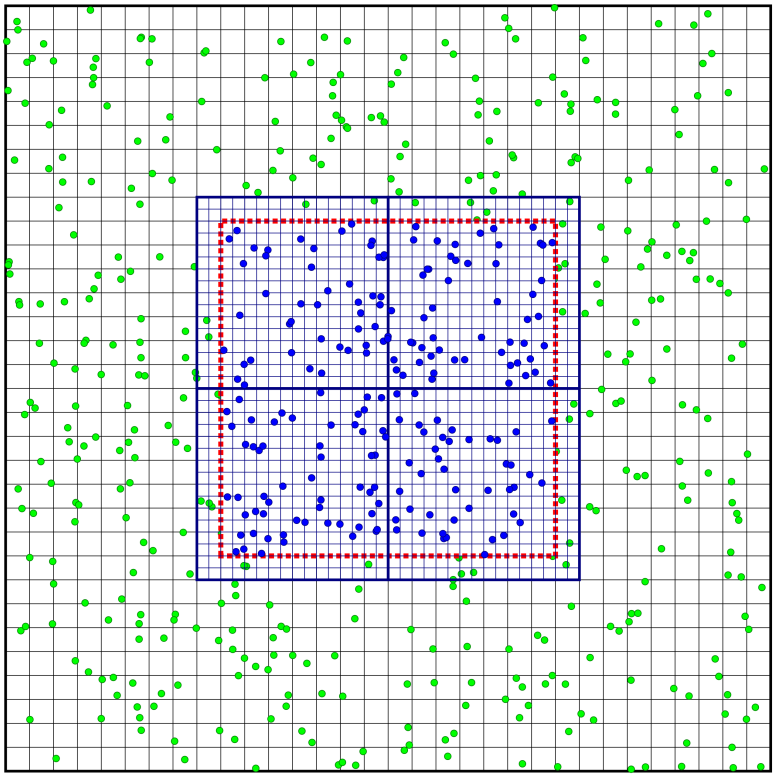


Fig. 2.3.1: The ParticleValidRegion allows particles whose position fall into a fine grid patch to be moved to a coarser level if they are within a specified distance from the refinement boundary. In this case, the green particles that overlap with the fine-level grid are placed in a `ParticleData<T>` holder on the coarse grid level.

2.3.6.1 Outcasts

After particles have moved, `ParticleData<T>` has a method for locally gathering particles that are no longer in the correct grid patch, and another method for distributing the particles to the correct grids. This is done as follows

```
// Assume PD is a ParticleData<T> object
PD.gatherOutcast();
PD.remapOutcast();
```

We remark that this concerns remapping on one single level. This means that:

1. Some particles may remain in the outcast list
2. The remapping does not respect the PVR on each level.

This is the actual calls in `levelRemap(const int a_level)`.

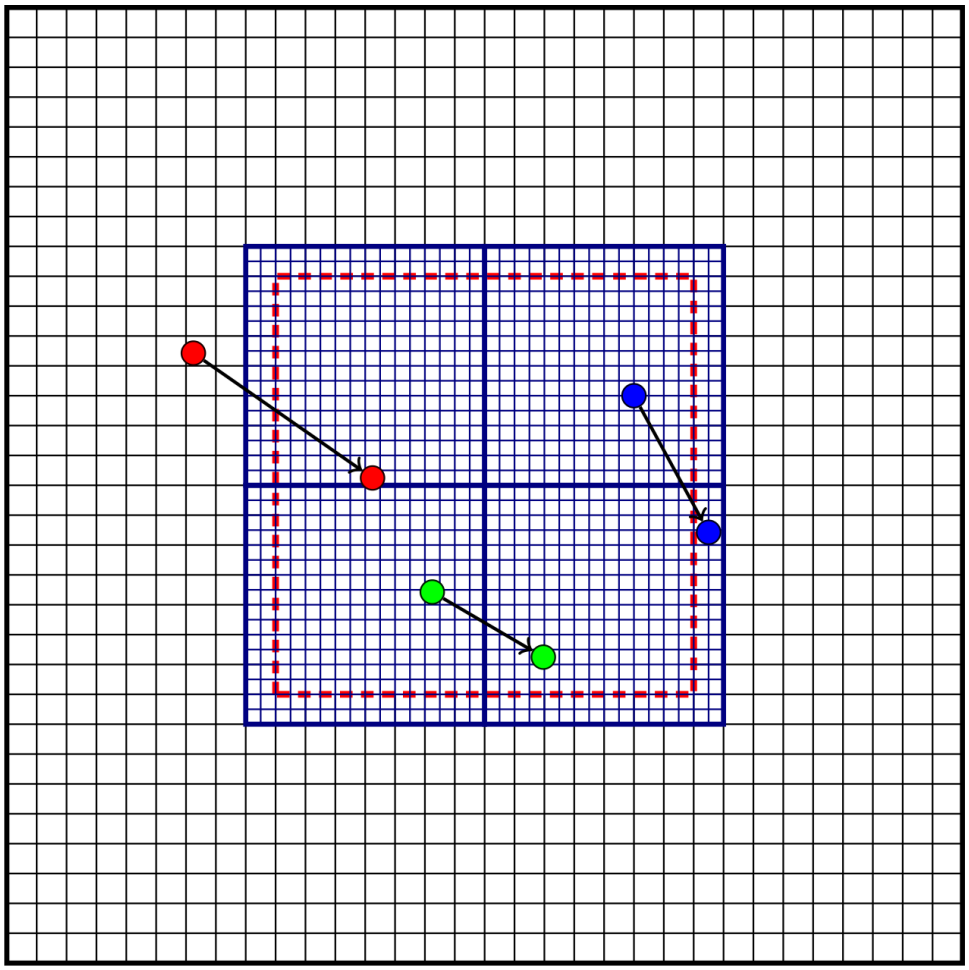


Fig. 2.3.2: Three cases of particle remapping. Here, the green particle stays on the fine level but needs to change MPI ownership. The red particle moves from the coarse level and to the fine level and needs to change both ownership and will also be deposited on the fine AMR level. The blue particle has moved to a different patch on the fine level but falls outside the fine level PVR must therefore be transferred to the coarse level.

2.4 Controlling `PlasmaC`

In this chapter we show how to run a `PlasmaC` simulation and control its behavior through input scripts or command line options.

2.4.1 Basic compiling and running

To run simulations, the user must first compile his application. Once the application has been defined, the user may compile is by

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

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

Next, applications are run by

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

where <input_file> is your input file. On clusters, this is a little bit different and usually requires passing the above command through a batch system. Note that if you define a parameter multiple times in the input file, the last definition is canon.

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.

2.4.2 Controlling output

PlasmaC 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, PlasmaC does it's best at creating it. In addition, it will create four more directories

- `output_directory/plt` contains all plot files.
- `output_directory/chk` contains all checkpoint files, which are used for restarting.
- `output_directory/mpi` contains information about individual MPI ranks.
- `output_directory/geo` contains geometric files that are written by PlasmaC (if you enable `driver.write_ebis`).

The files in `output_directory/geo` do *not* represent your geometry in the form of level sets. Instead, the files that are placed here are HDF5 representations of your embedded boundary graph, which can be *read* by PlasmaC if you enable `driver.read_ebis`. This is a shortcut that allows faster geometry generation when you restart simulations, but geometry generation is typically so fast that it is never used.

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

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

```
cdr_gdnv.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. Which variables are available for output changes for one class to the next. If you only want to plot the density, then you should put `cdr_gdnv.plt_vars = phi`. An empty entry like `cdr_gdnv.plt_vars =` will lead to run-time errors, so if you do not want a class to provide plot data you may put `cdr_gdnv.plt_vars = -1`.

2.4.3 Controlling processor output

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, PlasmaC prints information to these files as simulations advance (for example by displaying information of the current time step, or convergence rates for multigrid solvers). While it is possible to monitor the evolution of PlasmaC through each MPI, most of these files contain redundant information. To turn off the number of files that will be written, Chombo can read an environment variable `CH_OUTPUT_INTERVAL`. For example, if you only want the master MPI rank to write `pout.0`, you would do

```
export CH_OUTPUT_INTERVAL=999999999
```

You can, of course, put the definition in your `.bashrc` file (for Bourne shell). Note that 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.

2.4.4 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. If you set `driver.restart=0`, you will get a fresh simulation. Specifying anything but an integer is an error. When a simulation is restarted, PlasmaC will look for a checkpoint file with the `driver.output_names` variable and the specified restart step. If this file is not found, restarting will not work and `PlasmaC` 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.

2.4.5 Visualization

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

2.5 Visualization

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

CHAPTER

THREE

PLASMAC DESIGN

3.1 driver

The `driver` class is the class that runs *PlasmaC* simulations and is defined in `/src/driver/driver.cpp(H)`. The constructor for this class is

```
driver(const RefCountedPtr<computational_geometry>& a_compgeom,
       const RefCountedPtr<time_stepper>&           a_timestepper,
       const RefCountedPtr<amr_mesh>&               a_amr,
       const RefCountedPtr<cell_tagger>&            a_celltagger = RefCountedPtr<cell_tagger>(NULL),
       const RefCountedPtr<geo_coarsener>&          a_geocoarsen = RefCountedPtr<geo_coarsener>(NULL));
```

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

The usage of the `driver` class is primarily object construction with dependency injection of the geometry, the physics (i.e. `time_stepper`), the `amr_mesh` instance, and possibly a cell tagger. The driver class will automatically retrieve run-time options from the input script during object creation. Usually, only a single routine is used:

```
void setup_and_run();
```

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

3.1.1 How fresh simulations are set up

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

1. Ask `computational_geometry` to generate the cut-cell moments.
2. Collect all the cut-cells and ask `amr_mesh` to set up an initial grid where all the cut-cells are refined. It is possible to restrict the maximum level that can be generated from the geometric tags, or remove some of the cut-cell refinement flags through the auxiliary class `geo_coarsener`.
3. Ask the `time_stepper` to set up all the relevant solvers and fill them with initial data.
4. Perform the number of initial regrids that the user asks for.

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

3.1.2 How simulations are restarted

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

1. Ask `computational_geometry` 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.
3. Ask the `time_stepper` to perform a "post-checkpoint" step to initialize any remaining data so that a time step can be taken. This functionality has been included because not all data in every solver needs to be checkpointed. For example, an electric field solver only needs to write the electric potential to the checkpoint file because the electric field is simply obtained by taking the gradient.
4. Perform the number of initial regrids that the user asks for.

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

3.1.3 How simulations are run

The algorithm for running a simulation is very simple; the `driver` class simply calls `time_stepper` for computing a reasonable time step for advancing the equations, and then it asks `time_stepper` to actually perform the advance. Regrids, plot files, and checkpoint files are written at certain step intervals. In essence, the algorithm looks like this:

```
driver::run(...){

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

      time_stepper->computeTimeStep()
      time_stepper->advanceAllEquationsOneStep()

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

      KeepRunningTheSimulation = true or false
   }
}
```

None of the current physics modules use subcycling in time, but this *is* possible by having an implementation class of `time_stepper` that subcycles. The biggest caveat is that the recursive type of regridding that is performed by subcycled algorithms is not yet supported. It is possible to modify `driver` such that this is supported, but this has not been a priority.

3.1.4 How regrids are performed

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

1. Ask `cell_tagger` to generate tags for grid refinement and coarsening.
2. The `time_stepper` class stores data that is subject to regrids. How this happens depends on the solver that is run. For grid-based solvers, e.g. CDR solvers, the scalar ϕ is copied into a scratch space. The reason for this backup is that during the regrid ϕ will be allocated on the *new* AMR grids, but we must still have access to the previously defined data in order to interpolate to the new grids.

3. If necessary, `time_stepper` can deallocate unecessary storage. Implementing a deallocation function for `time_stepper`-derived classes is not a requirement, but can in certain cases be useful, for example when using the Berger-Rigoutsous algorithm at large scale.
4. The `amr_mesh` class generates the new grids and defines new AMR operators.
5. The `time_stepper` class regrids its solvers and internal data.

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

```
driver::regrid(){

   // Tag cells
   cell_tagger->tagCellsForRefinement()

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

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

   // Regrid physics and all solvers
   time_stepper->regrid()
}
```

The full code is defined in `driver::regrid()` in file `/src/driver/driver.cpp`.

3.1.5 Class options

Various class options are available for adjusting the behavior of the `driver` class.

```
# ================================================================================================
# DRIVER OPTIONS
# ================================================================================================
driver.verbosity                          = 2             # Engine verbosity
driver.geometry_generation                = plasmac       # Grid generation method, 'plasmac' or 'chombo'
driver.geometry_scan_level                = 0             # Geometry scan level for plasmac geometry generator
driver.recursive_regrid                   = false         # Recursive regrids
driver.plot_interval                      = 10            # Plot interval
driver.regrid_interval                    = 10            # Regrid interval
driver.checkpoint_interval                = 10            # Checkpoint interval
driver.initial_regrids                    = 0             # Number of initial regrids
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.ebis_memory_load_balance           = false         # Use memory as loads for EBIS generation
driver.write_memory                       = false         # Write MPI memory report
driver.write_ebis                         = false         # Write geometry to an HDF5 file
driver.read_ebis                          = false         # Read EBIS when restarting a simulation
driver.output_directory                   = ./            # Output directory
driver.grow_tags                          = 0             # Grow tagged by this in every direction
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                           = 0             # 'tags', 'mpi_rank'
driver.restart                            = 0             # Restart step (less or equal to 0 implies fresh␣
↪simulation)
driver.allow_coarsening                   = true          # Allows removal of grid levels according to cell_tagger
driver.refine_geometry                    = -1            # Refine geometry, -1 => Refine all the way down
driver.refine_electrodes                  = -1            # Refine electrode surfaces. -1 => equal to refine_
↪geometry
driver.refine_dielectrics                 = -1            # Refine dielectric surfaces. -1 => equal to refine_
↪geometry
driver.refine_electrode_gas_interface     = -1            # Refine electrode-gas interfaces. -1 => ----"-----
driver.refine_dielectric_gas_interface    = -1            # Refine dielectric-gas interfaces. -1 => ----"-----
driver.refine_solid_gas_interface         = -1            # Refine solid-gas interfaces. -1 => ----"-----
driver.refine_solid_solid_interface       = -1            # Refine solid-solid interfaces. -1 => ----"-----
```

3.2 amr_mesh

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

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

3.2.1 Main functionality

There are two main functionalities in `amr_mesh`:

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

In practice, users will not interact directly with this functionality, it is called by `driver` at the appropriate stages of regrids.

The AMR operators are, for example, coarsening operators, interpolation operators, ghost cell interpolators etc. To save some regrid time, we don't always build every AMR operator that we might ever need, but have solvers *register* the ones that they specifically need.

3.2.2 Registering operators

To register an operator, one must call a public member function of `amr_mesh`:

```
void register_operator(const std::string a_operator, const phase::which_phase a_phase);
```

where the string identifier is the name of the operator and `a_phase` is the phase on which the operator will live. Currently, the following operators are supported:

1. *eb_gradient* for allocating stencils for computing the cell-centered gradient.
2. *eb_irreg_interp* for allocating interpolation stencils in irregular cells, e.g. interpolation to centroids or boundary centroids.
3. *eb_coar_ave* for conservative coarsening of data.
4. *eb_quad_cfi* for quadratic filling of ghost cells.
5. *eb_fill_patch* for linear interpolation of ghost cells.
6. *eb_pwl_interp* for piecewise linear interpolation during e.g. regrids.
7. *eb_flux_reg* for holding fluxes on refinement boundaries.
8. *eb_redist* for holding redistribution objects (for e.g. particle deposition or divergence computatoins).
9. *eb_copier* for adding the contents in ghost cells to the valid region on the same AMR level.
10. *eb_ghostcloud* for adding the contents in ghost cells on the refinement boundary to the coarser AMR level.
11. *eb_non_cons_div* for computing stencils for doing the non-conservative divergence.

3.2.3 Class options

```
# ================================================================================================
# AMR_MESH OPTIONS
# ================================================================================================
amr_mesh.lo_corner       = -1 -1 -1    # Low corner of problem domain
amr_mesh.hi_corner       =  1  1  1    # High corner of problem domain
amr_mesh.verbosity       = -1          # Controls verbosity.
amr_mesh.coarsest_domain = 128 128 128 # Number of cells on coarsest domain
amr_mesh.max_amr_depth   = 0           # Maximum amr depth
amr_mesh.max_sim_depth   = -1          # Maximum simulation depth
amr_mesh.refine_all_lvl  = 0           # Refine everything down to this level.
amr_mesh.fill_ratio      = 1.0         # Fill ratio for grid generation
amr_mesh.irreg_growth    = 2           # How much to grow irregular tagged cells
amr_mesh.buffer_size     = 2           # Number of cells between grid levels
amr_mesh.grid_algorithm  = br          # Berger-Rigoustous 'br' or 'tiled' for the tiled algorithm
amr_mesh.blocking_factor = 16          # Default blocking factor (16 in 3D)
amr_mesh.max_box_size    = 16          # Maximum allowed box size
amr_mesh.max_ebis_box    = 16          # Maximum allowed box size
amr_mesh.ref_rat         = 2 2 2 2 2 2 # Refinement ratios
amr_mesh.num_ghost       = 3           # Number of ghost cells. Default is 3
amr_mesh.eb_ghost        = 4           # Set number of of ghost cells for EB stuff
amr_mesh.centroid_sten   = linear      # Centroid interp stencils. 'pwl', 'linear', 'taylor, 'lsq'
amr_mesh.eb_sten         = pwl         # EB interp stencils. 'pwl', 'linear', 'taylor, 'lsq'
amr_mesh.redist_radius   = 1           # Redistribution radius for hyperbolic conservation laws
amr_mesh.ghost_interp    = pwl         # Ghost cell interpolation type. Valid options are 'pwl' or 'quad'
amr_mesh.ebcf            = true        # If you have EBCF crossing, this must be true.
```

We now discuss the various `amr_mesh` class options.

- `amr_mesh.verbosity` controls the verbosity of this class. `amr_mesh` can potentially do a lot of output, so it is best to leave this to the default value (-1) unless you are debugging.
- `amr_mesh.coarsest_domain` is the partitioning of the *coarsest* grid level that discretizes your problem domain. The entries in this option must all be integers of `amr_mesh.max_box_size`.
- `amr_mesh.blocking_factor` sets the minimum box size that can be generated by the mesh generation algorithm. We remark that if you are doing particle deposition, `amr_mesh.blocking_factor` and `amr_mesh.max_box_size` MUST be equal.
- `amr_mesh.max_box_size` sets the maximum box size that can be generated by the mesh generation algorithm.
- `amr_mesh.max_ebis_box` sets the maximum box size that will be used in the geometry generation step. A smaller box will consume less memory, but geometry generation runtime will be longer.
- `amr_mesh.max_amr_depth` defines the largest possible number of grids that can be used.
- `amr_mesh.max_sim_depth` defines the maximum simulation depth for the simulation. This options exists because you may want to run one part of a simulation using a coarser resolution than `amr_mesh.max_amr_depth`.
- `amr_mesh.refine_all_lvl` is deprecated class option.
- `amr_mesh.mg_coarsen` is a "pre-coarsening" method for multigrid solvers. The deeper multigrids levels are there to facilitate convergence, and it often helps to use fairly large box sizes on some of these levels before aggregating boxes.
- `amr_mesh.fill_ratio` is the fill ratio for the mesh refinement algorithm. This value must be between 0 and 1; a smaller value will result in larger grids. A higher value results in more compact grids, but possibly with more boxes.
- `amr_mesh.irreg_growth` controls how much irregular tags (e.g. boundary tags) are grown before being passed into the mesh refinement algorithm.
- `amr_mesh.buffer_size` is the minimum number of cells between grid levels.

- `amr_mesh.ref_rat` is the refinement factor between levels. Values 2 and 4 are supported, and you may use mixed refinement ratios. The length of this vector must be at least equal to the number of refinement levels.
- `amr_mesh.num_ghost` indicates how many ghost cells to use for all data holders. The typical value is 3 for EB-applications, but non-EB applications might get away with fewer ghost cells.
- `amr_mesh.eb_ghost` controls how ghost cells are used for the EB generation.
- `amr_mesh.centroid_sten` controls which stencil is used for interpolating data to irregular cell centroids. Currently available options are 'pwl' (piecewise linear), 'linear' (bi/trilinear), 'taylor' (higher order Taylor expansion), and 'lsq' which is a least squares fit. Only 'pwl' is guaranteed to have positive weights in the stencil.
- `amr_mesh.eb_sten` controls which stencil is used for interpolation/extrapolation to embedded boundary centroids. We cannot guarantee that the stencils have only positive weights.
- `amr_mesh.redist_radius` is the redistribution radius for hyperbolic redistribution.
- `amr_mesh.ghost_interp` defines the ghost cell interpolation type. Algorithms that require very specific ghost cell interpolation schemes (advection, for example) use their own interpolation method that is outside user control. The available options are 'pwl' (piecewise linear) and 'quad' (quadratic).
- `amr_mesh.load_balance` tells `amr_mesh` how to load balance the grids.
- `amr_mesh.ebcf` allows `amr_mesh` to turn on certain optimizations when there are **not** crossing between embedded boundaries and grid refinement boundaries. If such crossings exist, and you set this flag to false, *PlasmaC will* compute incorrect answers.

3.3 computational_geometry

computational_geometry is the class that implements that geometry. In *PlasmaC*, we use level-set functions for description of surfaces. *computational_geometry* is not an abstract class; if you pass in an instance of *computational_geometry* (rather than a casted instance), you will get a regular geometry without any boundaries. A new *computational_geometry* class requires that you set the following class members:

```
Real m_eps0;
Vector<electrode> m_electrodes;
Vector<dielectric> m_dielectrics;
```

Here, `m_eps0` is the gas permittivity, `m_electrodes` are the electrodes for the geometry and `m_dielectrics` are the dielectrics for the geometry.

3.3.1 electrode

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

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

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

3.3.2 dielectric

The *dielectric* class describes a electrode, this class is lightweight and consists only of a tuple that holds a level-set function and the permittivity. The constructors are

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

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

3.3.3 How `computational_geometry` works

When geometries are created, the `computational_geometry` class will first create the level-set functions that describe two possible material phases (gas and solid) and then compute the cut cell moments.

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

```
const Vector<dielectric>& get_dielectrics() const;
const Vector<electrode>& get_electrodes() const;

const RefCountedPtr<BaseIF>& get_gas_if() const;
const RefCountedPtr<BaseIF>& get_sol_if() const;
```

3.4 time_stepper

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

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

```
// Setup routines
virtual void setup_solvers() = 0;
virtual void initial_data() = 0;
virtual void post_checkpoint_setup() = 0;

// IO routines
virtual void write_checkpoint_data(HDF5Handle& a_handle, const int a_lvl) const = 0;
virtual void read_checkpoint_data(HDF5Handle& a_handle, const int a_lvl) = 0;
virtual int get_num_plot_vars() const = 0;
virtual void write_plot_data(EBAMRCellData& a_output, Vector<std::string>& a_plotvar_names, int& a_icomp)␣
↪const = 0;

// Advance routines
virtual void compute_dt(Real& a_dt, time_code::which_code& a_timecode) = 0;
virtual Real advance(const Real a_dt) = 0;
virtual void synchronize_solver_times(const int a_step, const Real a_time, const Real a_dt) = 0;
virtual void print_step_report() = 0;

// New regrid routines
virtual void register_operators() = 0;
virtual void pre_regrid(const int a_lmin, const int a_old_finest_level) = 0;
virtual void deallocate() = 0;
virtual void regrid(const int a_lmin, const int a_old_finest_level, const int a_new_finest_level) = 0;
```

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

Depending on the physics that is resolved, writing `time_stepper` can be a small or a big task. For example, the code used for advection-diffusion in `/physics/advection_diffusion/` is only a few hundred lines where most of the code is simply calling member functions from *Convection-Diffusion-Reaction*.

3.5 cell_tagger

The `cell_tagger` class is responsible for flagging grid cells for refinement or coarsening. In `PlasmaC`, refinement flags live in a data holder called `EBAMRTags` inside of `driver`. This data is typedef'ed as

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

The `LayoutData<T>` structure can be thought of as a `LevelData<T>` without possibilities for communication. `cell_tagger` is an abstract class that the user *must* overwrite - it is not possible to

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

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

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

3.5.1 Restrict tagging

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

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

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

3.5.2 Adding a buffer

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

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

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

CHAPTER

FOUR

SUPPORTED SOLVERS

4.1 Solver

4.2 Convection-Diffusion-Reaction

Here, we discuss the discretization of the equation

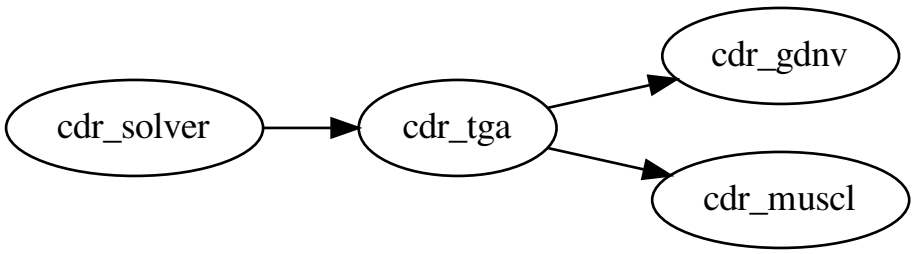
$$\frac{\partial \phi}{\partial t} + \nabla \cdot \left(\mathbf{v}\phi - D\nabla\phi + \sqrt{2D\phi}\mathbf{Z} \right) = S.$$

We assume that ϕ is discretized by cell-centered averages (note that cell centers may lie inside solid boundaries), and use finite volume methods to construct fluxes in a cut-cells and regular cells.

Here, $\mathbf{v}$ indicates a drift velocity, D is the diffusion coefficient, and the term $\sqrt{2D\phi}\mathbf{Z}$ is a stochastic diffusion flux. S is the source term.

4.2.1 cdr_solver

The `cdr_solver` class contains the interface for solving advection-diffusion-reaction problems. The class is abstract and resides in `/src/cdr_solver/cdr_solver.H(cpp)` together with specific implementations. By design `cdr_solver` does not contain any specific advective and diffusive discretization, and these are supposed to be added through inheritance. For example, `cdr_tga` inherits from `cdr_solver` and adds a second order diffusive discretization together with multigrid code for performing implicit diffusion. Below that, the classes `cdr_gdnv` and `cdr_muscl` inherit everything from `cdr_tga` and also adds in the advective discretization. Thus, adding new advection code is done by inheriting from `cdr_tga` and implementing new advection schemes. This is much more lightweight than rewriting all of `cdr_solver` (which is several thousand lines of code).



Currently, we mostly use the implementation given in `/src/cdr_solver/cdr_gdnv.H(cpp)` which contains a second order accurate discretization with slope limiters which was distributed by the Chombo team. The alternative implementation in `/src/cdr_muscl.H(cpp)` contains a MUSCL implementation with van Leer slope limiting (i.e. much the same as the Chombo code).

4.2.2 Using cdr_solver

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

1. Fill the solver with relevant data (e.g. velocities, diffusion coefficients, source terms etc.).
2. Call public member functions for computing advective or diffusive derivates, or perform implicit diffusion advances.

There are on time integration algorithms built into the `cdr_solver`, and the user will have to supply these through `time_stepper`. It is up to the developer to ensure that the solver is filled with appropriate data before calling the public member functions. This would typically look something like this:

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

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

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

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

More complete code is given in the physics module for advection-diffusion problems in `/physics/advection_diffusion/advection_diffusion_stepper`. This code is also part of a regression test found in `/regression/advection_diffusion`.

4.2.3 Setting up the solver

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

```
// Assume m_solver and m_species are pointers to a cdr_solver and cdr_species
m_solver  = RefCountedPtr<cdr_solver>  (new my_cdr_solver());
m_species = RefCountedPtr<cdr_species> (new my_cdr_species());

// Solver setup
m_solver->set_verbosity(10);
m_solver->set_species(m_species);
m_solver->parse_options();
m_solver->set_phase(phase::gas);
m_solver->set_amr(m_amr);
m_solver->set_computational_geometry(m_compgeom);
m_solver->sanity_check();
m_solver->allocate_internals();
```

To see an example, the advection-diffusion code in `/physics/advection_diffusion/advection_diffusion_stepper` shows how to set up the solver.

4.2.4 Filling the solver

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

```
EBAMRCellData& get_state();         // Return  phi
EBAMRCellData& get_velo_cell();     // Get cell-centered velocity
EBAMRFluxData& get_diffco_face();   // Returns D
EBAMRCellData& get_source();        // Returns S
EBAMRIVData& get_ebflux();          // Returns flux at EB
EBAMRIFData& get_domainflux();      // Returns flux at domain boundaries
```

To set the drift velocities, the user will fill the *cell-centered* velocities. Interpolation to face-centered transport fluxes are done by `cdr_solver` when needed.

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

```
cdr_solver solver(...);

EBAMRCellData& velo_cell = solver.get_velo_cell();
```

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

4.2.5 Adjusting output

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

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

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

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

4.2.6 cdr_species

The `cdr_species` class is a supporting class that passes information and initial conditions into `cdr_solver` instances.

4.2.7 Discretization details

4.2.7.1 Computing explicit divergences

Computing explicit divergences for equations like

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

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

$$\phi^{k+1} = \phi^k - \frac{\Delta t}{\kappa \Delta x^{\mathrm{DIM}}} \int_V \nabla \cdot \mathbf{G} dV, \tag{4.2.1}$$

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

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

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

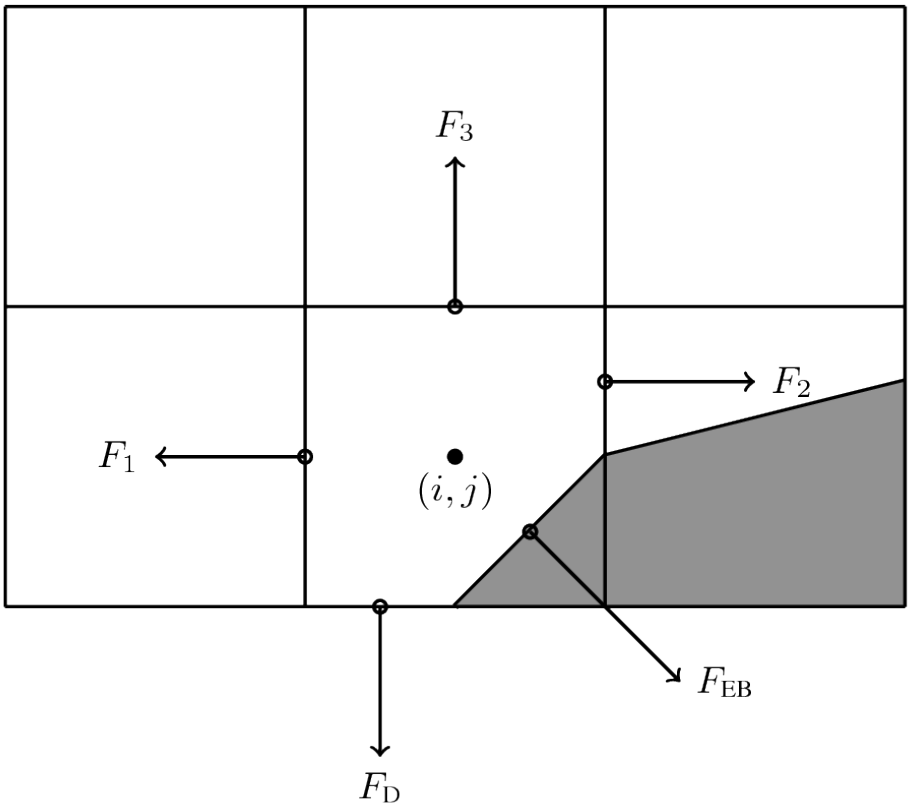


Fig. 4.2.1: Location of centroid fluxes for cut cells.

However, taking $[\nabla \cdot \mathbf{G}^k]$ to be this sum leads to a time step constraint proportional to κ, which can be arbitrarily small. This leads to an unacceptable time step constraint for Eq.4.2.1. We use the Chombo approach and expand the range of influence of the cut cells in order to stabilize the discretization and allow the use of a normal time step constraint. First, we compute the conservative divergence

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

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

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

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

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

and perform an intermediate update

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

The hybrid divergence update fails to conserve mass by an amount $\delta M_{\mathbf{i}} = \kappa_{\mathbf{i}}\left(1-\kappa_{\mathbf{i}}\right)\left(D_{\mathbf{i}}^{c} - D_{\mathbf{i}}^{nc}\right)$. 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

```
virtual void cdr_solver::compute_divG(EBAMRCellData& a_divG, EBAMRFluxData& a_G, const EBAMRIVData& a_ebG)
```

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

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

4.2.7.2 Maintaining non-negative densities

Although the redistribution functionality is conservative, the cut-cells represent boundaries that make the evolution non-monotone. In particular, the redistribution process does not guarantee monotonicity. In some cases, negative values of ϕ are non-physical and the lack of non-negativeness can lead to numerical issues.

The `cdr_solver` has an option to use mass-weighted redistribution in order to redistribute mass in the neighborhood of the cut cells. To turn this one, one must use e.g. `cdr_gdnv.redist_mass_weighted = true` in the input script. The default is false, in which case the redistribution uses volume-weighted redistribution.

As a last effort, we support another redistribution step in the cut cells that redistributes mass from regular cells and into the cut cells in order to maintain non-negative densities.

```
void redistribute_negative(EBAMRCellData& a_phi);
```

Note that this *will* give give an $\mathcal{O}(1)$ error in the solution and so it is not a very attractive solution. The alternative of maintaining non-negative densities through mass injection introduces the same error, but in addition to adding a $\mathcal{O}(1)$ error into the solution, this also has the side-effect of being non-conservative.

4.2.7.3 Explicit advection

Scalar advective updates follows the computation of the explicit divergence discussed in *Computing explicit divergences*. The face-centered fluxes $\mathbf{G} = \phi\mathbf{v}$ are computed by instantiation classes for the convection-diffusion-reaction solvers. These solvers may compute $\mathbf{G}$ in different ways. There is, for example, support for low-order upwind methods as well as Godunov methods. The function signature for explicit advection is

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

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

```
// Assume that data holders divF and phi are defined, and that 'solver' is
// a valid convection-diffusion reaction solver with defined velocities.
solver->compute_divF(divF, phi, 0.0); // Computes divF
data_ops:incr(phi, divF, -dt);        // makes phi -> phi - dt*divF
```

4.2.7.4 Explicit diffusion

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

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

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

```
// Assume that data holders divD and phi are defined, and that 'solver' is
// a valid convection-diffusion reaction solver with defined diffusion coefficients
solver->compute_divD(divD, phi); // Computes divD
data_ops:incr(phi, divD, dt);    // makes phi -> phi + dt*divD
```

4.2.7.5 Explicit advection-diffusion

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

```
void compute_divJ(EBAMRCellData& a_divJ, const EBAMRCellData& a_state, const Real a_extrap_dt)
```

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

```
// Assume that data holders divJ and phi are defined, and that 'solver' is
// a valid convection-diffusion reaction solver with defined velocities and
// diffusion coefficients
solver->compute_divJ(divJ, phi, 0.0); // Computes divF
data_ops: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 non-split (i.e. not using a Strang or Godunov splitting), the integrators will often call `compute_divJ` rather separately calling `compute_divF` and `compute_divD`. If you had a split-step Godunov method, the above procedure for a forward Euler method for both parts would be:

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

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

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

4.2.7.6 Implicit diffusion

Occasionally, the use of implicit diffusion is necessary. The convection-diffusion-reaction solvers support two basic diffusion solves: Backward Euler and the Twizel-Gumel-Arigu (TGA) methods (it should be straightforward for the user to change the backward Euler method into a Crank-Nicholson scheme). The function signatures for these are

```
void advance_euler(EBAMRCellData& phiNew, const EBAMRCellData& phiOld, const EBAMRCellData& src, const Real dt)
void advance_tga(  EBAMRCellData& phiNew, const EBAMRCellData& phiOld, const EBAMRCellData& src, const Real dt)

void advance_euler(EBAMRCellData& phiNew, const EBAMRCellData& phiOld, const Real dt)
void advance_tga(  EBAMRCellData& phiNew, const EBAMRCellData& phiOld, const Real dt)
```

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

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

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

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

4.2.7.7 Adding a stochastic flux

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

```
void GWN_diffusion_source(EBAMRCellData& a_ransource, const EBAMRCellData& a_cell_states);
```

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

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

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

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

4.3 Poisson

The code for the Poisson solver is given in `/src/poisson` *and* `/src/elliptic`. The first folder contains the interface and code for only the Poisson solver, whereas the `/src/elliptic` folder also contains generic code for elliptic problems.

The Poisson solver is named `poisson_solver` and currently only has one specific implementation: `poisson_multifluid_gmg` which uses a multifluid embedded boundary formulation together with geometric multigrid.

4.3.1 Setting up the solver

In order to set up the solver, one must provide the

Creating a solver is often done with smart pointer casts like so:

```
RefCountedPtr<poisson_solver> poisson = RefCountedPtr<poisson_solver> (new poisson_multifluid_gmg());
```

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

```
poisson->parse_options();              // Parse class options
poisson->set_amr(amr);                 // Set amr - we assume that `amr` is an object
poisson->set_computational_geometry(); // Set the computational geometry
poisson->allocate_internals();         // Allocate storage for potential etc.
poisson->set_potential(potential);     // Set the potential
```

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

```
Real potential(const Real a_time)
```

and allows setting a time-dependent potential for the solver.

The *PlasmaC* field solver has a lot of supporting functionality, but essentially relies on only one critical function: Solving for the potential. This is done by calling a class-specific function

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

where `phi` is the resulting potential that was computing with the space charge density `rho` and surface charge density `sigma`.

Currently, only one field solver is implemented and this solver uses a geometric multigrid method for solving for the potential. The solver supports three phases: electrodes, gas, and dielectric.

4.3.2 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 Dirichlet boundary condition is enforced by the conventional matching boundary condition that follows from Gauss` law.

4.3.3 Tuning multigrid performance

The Poisson equation is currently solved with a geometric multigrid method (GMG). Various switches are enabled that adjust the performance of GMG, and these are listed below

```
# ================================================================================================
# POISSON_MULTIFLUID_GMG_GMG CLASS OPTIONS (MULTIFLUID GMG SOLVER SETTINGS)
# ================================================================================================
poisson_multifluid_gmg.bc_x_low  = neumann           # BC type. "neumann", "dirichlet_ground", "dirichlet_live"
poisson_multifluid_gmg.bc_x_high = neumann           # BC type. "neumann", "dirichlet_ground", "dirichlet_live"
poisson_multifluid_gmg.bc_y_low  = dirichlet_ground  # BC type. "neumann", "dirichlet_ground", "dirichlet_live"
poisson_multifluid_gmg.bc_y_high = dirichlet_live    # BC type. "neumann", "dirichlet_ground", "dirichlet_live"
poisson_multifluid_gmg.bc_z_low  = neumann           # BC type. "neumann", "dirichlet_ground", "dirichlet_live"
poisson_multifluid_gmg.bc_z_high = neumann           # BC type. "neumann", "dirichlet_ground", "dirichlet_live"
poisson_multifluid_gmg.plt_vars  = phi rho E res     # Plot variables. Possible vars are 'phi', 'rho', 'E',
↪'res'

poisson_multifluid_gmg.auto_tune         = false     # Do some auto-tuning
poisson_multifluid_gmg.gmg_verbosity     = -1        # GMG verbosity
poisson_multifluid_gmg.gmg_pre_smooth    = 12        # Number of relaxations in downsweep
poisson_multifluid_gmg.gmg_post_smooth   = 12        # Number of relaxations in upsweep
poisson_multifluid_gmg.gmg_bott_smooth   = 12        # NUmber of relaxations before dropping to bottom solver
poisson_multifluid_gmg.gmg_min_iter      = 5         # Minimum number of iterations
poisson_multifluid_gmg.gmg_max_iter      = 32        # Maximum number of iterations
poisson_multifluid_gmg.gmg_tolerance     = 1.E-10    # Residue tolerance
poisson_multifluid_gmg.gmg_hang          = 0.2       # Solver hang
poisson_multifluid_gmg.gmg_bottom_drop   = 4         # Bottom drop
poisson_multifluid_gmg.gmg_bc_order      = 2         # Boundary condition order for multigrid
poisson_multifluid_gmg.gmg_bottom_solver = bicgstab  # Bottom solver type. 'simple', 'bicgstab', or 'gmres'
poisson_multifluid_gmg.gmg_bottom_relax  = 32        # Number of relaxations in bottom solve ('simple' solver␣
↪only)
poisson_multifluid_gmg.gmg_cycle         = vcycle    # Cycle type. Only 'vcycle' supported for now
poisson_multifluid_gmg.gmg_relax_type    = gsrb      # Relaxation type. 'jacobi', 'gauss_seidel', or 'gsrb'
```

4.3.4 Adjusting output

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

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

4.4 Radiative transfer

Radiative transfer is supported in the diffusion (i.e. Eddington or Helmholtz) approximation and with Monte Carlo sampling of discrete photons. The solvers share a common interface but since diffusion RTE is deterministic and discrete Monte Carlo photons are stochastic, not all temporal integration methods will support both. The diffusion approximation relies on solving an elliptic equation in the stationary case and a parabolic equation in the time-dependent case, while the Monte-Carlo approach solves solves for fully transient or ''stationary" transport.

4.4.1 Diffusion approximation

In the diffusion approximation, the radiative transport equation is

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

which is called the Eddington approximation. The radiative flux is $F = -\frac{c}{3\kappa}\nabla\Psi$. We do not currently support flux-limited diffusion radiative transfer. In the stationary case this yields a Helmholtz equation

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

which is solved by a geometric multigrid method. The default boundary conditions are of the Robin type. For fully transient radiative transport, we offer discretizations based on the backward Euler and TGA schemes.

4.4.2 Monte Carlo methods

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

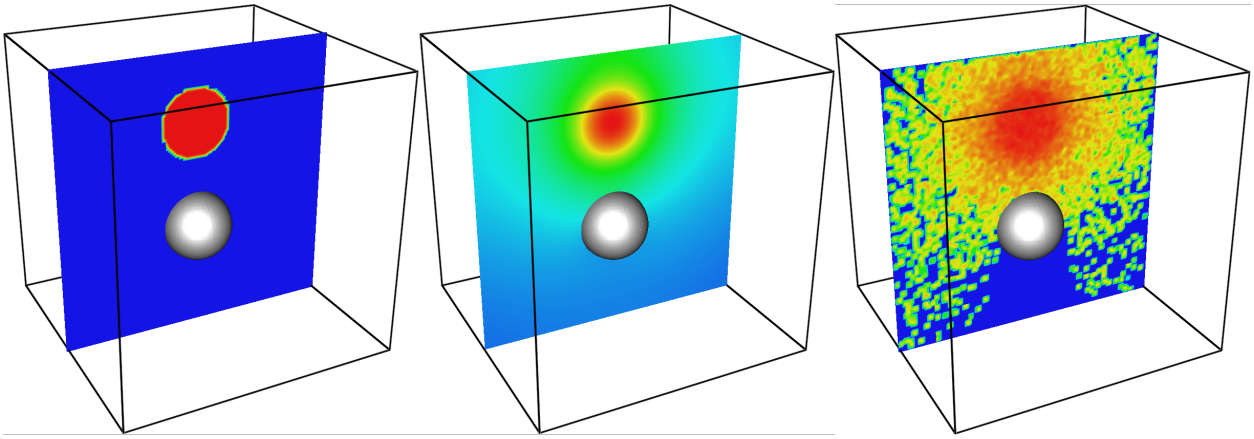


Fig. 4.4.1: Qualitative comparison between predictions made with a diffusion RTE solver and a Monte Carlo RTE solver. Left: Source term. Middle: Solution computed in the diffusion approximation with homogeneous Robin boundary conditions. Right: Solution computed with a Monte Carlo method.

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

4.4.2.1 photon particle

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

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

To obtain the fields, the user will call

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

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

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

4.4.2.2 Interaction with boundaries

4.4.2.3 Stationary Monte Carlo

The stationary Monte Carlo method proceeds as follows.

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

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

4.4.2.4 Transient Monte Carlo

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

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

4.4.3 Limitations

4.5 Surface charge solver

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

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

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

4.6 Îto diffusion

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

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

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

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

4.6.1 The Îto particle

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

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

To obtain the fields, the user will call

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

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

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

4.6.2 ito_species

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

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

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

4.6.2.1 Setting initial conditions

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

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

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

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

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

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

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

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

4.6.3 Computing time steps

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

4.6.3.1 Drift

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

For the drift, the signatures are

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

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

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

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

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

4.6.3.2 Diffusion

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

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

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

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

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

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

4.6.4 Remapping particles

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

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

4.6.5 Limitations

CHAPTER

FIVE

IMPLEMENTED MODELS

5.1 Poisson model

5.2 Advection diffusion model

5.3 Brownian walker model

5.4 Minimal plasma model

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

- `/physics/cdr_plasma/plasma_models` which contains various implementation of some plasma models that we have used.
- `/physics/cdr_plasma/time_steppers` contains various algorithms for advancing the equations in an EBAMR context.
- `/physics/cdr_plasma/cell_taggers` contains various algorithms for advancing the equations in an EBAMR context.
- `/physics/cdr_plasma/python` contains Python source files for quick setup of "mini-applications".

If users decide to develop new code for the minimal plasma model, e.g. new grid refinement routines, plasma models, integrators, or other types of improvements, they should be put in the folders above.

5.4.1 Equations of motion

In the minimal plasma model, we are solving

$$\nabla \cdot (\epsilon_r \nabla \Phi) = -\frac{\rho}{\epsilon_0}, \tag{5.4.1}$$

$$\frac{\partial \sigma}{\partial t} = F_\sigma, \tag{5.4.2}$$

$$\frac{\partial n}{\partial t} + \nabla \cdot \left(\mathbf{v} n - D\nabla n + \sqrt{2D\phi}\mathbf{Z} \right) = S, \tag{5.4.3}$$

where $\sqrt{2D\phi}\mathbf{Z}$ is a stochastic diffusion flux suitable for fluctuating hydrodynamics models. By default, this flux is turned off.

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

Radiative transport is also supported, which is done either in the diffusive approximation or by means of Monte Carlo methods. Diffusive RTE methods involve solving

$$\partial_t \Psi + \kappa \Psi - \nabla \cdot \left(\frac{1}{3\kappa} \nabla \Psi \right) = \frac{\eta}{c}, \tag{5.4.4}$$

where Ψ is the isotropic photon density, κ is an absorption length and η is an isotropic source term. The time dependent term can be turned off and the equations can be solved stationary. As an alternative, we also provide discrete photon methods that solve for the photoionization profile on a mesh by sampling discrete photons. Our discrete photon methods are capable of including far more physics; they can easily be adapted to e.g. scattering media and also provide much better qualitative features (like shadows, for example). They are, on the other hand, inherently stochastic which implies that some extra care must be taken when integrating the equations of motion.

The coupling that is (currently) available in *PlasmaC* is

$$\epsilon_r = \epsilon_r(\mathbf{x}), (\text{can additionally be discontinuous}), \tag{5.4.5}$$

$$\mathbf{v} = \mathbf{v}\left(t, \mathbf{x}, \mathbf{E}, n\right), \tag{5.4.6}$$

$$D = \mathbf{v}\left(t, \mathbf{x}, \mathbf{E}, n\right), \tag{5.4.7}$$

$$S = S(t, \mathbf{x}, \mathbf{E}, \nabla \mathbf{E}, n, \nabla n, \Psi), \tag{5.4.8}$$

$$\eta = \eta\left(t, \mathbf{x}, \mathbf{E}, n\right), \tag{5.4.9}$$

$$F = F(t, \mathbf{x}, \mathbf{E}, n), \tag{5.4.10}$$

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

PlasmaC works by embedding the equations above into an abstract C++ framework that the user must implement or reuse existing pieces of, and then compile into a *mini-application*.

5.4.2 cdr_plasma_physics

cdr_plasma_physics represents the microphysics in `/physics/cdr_plasma`. The entire class is an interface, whose implementations are used in the time integrators that advance the equations. As mentioned above, the time integrators are located in `/physics/cdr_plasma/time_steppers`.

There are no default input parameters for *cdr_plasma_physics*, as users must generally implement their own kinetics. A successful implementation of *cdr_plasma_physics* has the following:

- Instantiated *species*. These contain metadata for the transport solvers.
- Instantiated *photons*. These contain metadata for the radiative transport solvers.
- Implemented the core functionality that couple all solvers.

PlasmaC automatically allocates the specified number of convection-diffusion-reaction and radiative transport solvers. For information on how to interface into the CDR solvers, see *species*. Likewise, see *photons* for how to interface into the RTE solvers.

There are currently no support for existing file formats for describing reactions and so on. If you have a huge list of reactions that need to be implemented, it would probably pay off to write a code-generating interface between *cdr_plasma_physics* and your list of reactions.

Implementation of the core functionality is comparatively straightforward. In the constructor, the user should fetch his input parameters (if he has any) and **must** instantiate all species and photons in the internal containers. When the class is used by *PlasmaC* later on, all arguments in the core functions follow that ordering. The core functionality is given by the following functions:

```
virtual Vector<RealVect> compute_cdr_velocities(const Real&          a_time,
                                                const RealVect&      a_pos,
                                                const RealVect&      a_E,
                                                const Vector<Real>& a_cdr_densities) const = 0;

virtual Vector<Real> compute_cdr_diffusion_coefficients(const Real&          a_time,
                                                        const RealVect&      a_pos,
                                                        const RealVect&      a_E,
                                                        const Vector<Real>& a_cdr_densities) const = 0;

virtual Vector<Real> compute_cdr_source_terms(const Real               a_time,
                                              const RealVect&          a_pos,
                                              const RealVect&          a_E,
                                              const RealVect&          a_gradE,
                                              const Vector<Real>&      a_cdr_densities,
                                              const Vector<Real>&      a_rte_densities,
                                              const Vector<RealVect>& a_grad_cdr) const = 0;

virtual Vector<Real> compute_cdr_electrode_fluxes(const Real&          a_time,
                                                  const RealVect&      a_pos,
                                                  const RealVect&      a_normal,
                                                  const RealVect&      a_E,
                                                  const Vector<Real>& a_cdr_densities,
                                                  const Vector<Real>& a_cdr_velocities,
                                                  const Vector<Real>& a_cdr_gradients,
                                                  const Vector<Real>& a_rte_fluxes,
                                                  const Vector<Real>& a_extrap_cdr_fluxes) const = 0;

virtual Vector<Real> compute_cdr_dielectric_fluxes(const Real&          a_time,
                                                   const RealVect&      a_pos,
                                                   const RealVect&      a_normal,
                                                   const RealVect&      a_E,
                                                   const Vector<Real>& a_cdr_densities,
                                                   const Vector<Real>& a_cdr_velocities,
                                                   const Vector<Real>& a_cdr_gradients,
                                                   const Vector<Real>& a_rte_fluxes,
                                                   const Vector<Real>& a_extrap_cdr_fluxes) const = 0;

virtual Vector<Real> compute_cdr_domain_fluxes(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_cdr_densities,
                                               const Vector<Real>&   a_cdr_velocities,
                                               const Vector<Real>&   a_cdr_gradients,
                                               const Vector<Real>&   a_rte_fluxes,
                                               const Vector<Real>&   a_extrap_cdr_fluxes) const = 0;

virtual Vector<Real> compute_rte_source_terms(const Real&          a_time,
                                              const RealVect&      a_pos,
                                              const RealVect&      a_E,
                                              const Vector<Real>& a_cdr_densities) const = 0;

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

The above code blocks do exactly what their signatures indicate. It is up to the user to implement these. The length of the Vector holding the return values from these functions are expected to be equal to the number of CDR solvers, with the exception of *compute_rte_source_terms* which has the length given by the number of RTE solvers. Note that in all of the above, the ordering of the input vectors are expected to be the same as the ordering of the species vector of *cdr_plasma_physics*.

For example, if the user has defined only a single advected species, he may implement the constructor as

```
my_kinetics::my_kinetics(){
   m_num_species = 1;
   m_num_photons = 1;

   m_species.resize(m_num_species);
   m_photons.resize(m_num_photons);

   m_species[0] = RefCountedPtr<species> (new my_species());
   m_photons[0] = RefCountedPtr<photon> (new my_photon());
}
```

This constructor assumes that *my_species* has already been defined somewhere (for example, as a private class within *my_kinetics*).

5.4.2.1 Defining drift velocities

Next the user may implement the velocity computation function, which sets $\mathbf{v}$ in the CDR equations:

```
Vector<RealVect> compute_cdr_velocities(const Real&          a_time,
                                        const RealVect&      a_pos,
                                        const RealVect&      a_E,
                                        const Vector<Real>& a_cdr_densities) const {
   Vector<RealVect> velo(1);
   velo[0] = a_E;
   return velo;
}
```

This implementation is a full implementation of the velocity coupling of the CDR equations. In this case, the velocity of the advected component is equal to $\mathbf{E}$. For a full plasma simulation, there will also be mobilities involved, which the user is reponsible for obtaining.

5.4.2.2 Defining diffusion coefficients

In order to define diffusion coefficients, the user implements *compute_cdr_diffusion_coefficients*, which returns the diffusion coefficients for the diffused species. If a species (e.g. positive ions) is not diffusive, it does not matter what diffusion coefficient you set.

```
Vector<Real> compute_cdr_diffusion_coefficients(const Real&          a_time,
                                                const RealVect&      a_pos,
                                                const RealVect&      a_E,
                                                const Vector<Real>& a_cdr_densities) const {
   Vector<Real> diffco(2, 0.0);
   diffco[0] = 1.0;
   return diffco;
}
```

5.4.2.3 Defining chemistry terms

The function *compute_cdr_source_terms* is reponsible for computing S in the CDR equations. If we want, for example, $S_1 = kn_1n_2$, where k is some rate and n_1 and n_2 are densities of some species (e.g. electrons and positive ions),

```
Vector<Real> compute_cdr_source_terms(const Real              a_time,
                                      const RealVect&         a_pos,
                                      const RealVect&         a_E,
                                      const RealVect&         a_gradE,
                                      const Vector<Real>&     a_cdr_densities,
                                      const Vector<Real>&     a_rte_densities,
                                      const Vector<RealVect>& a_grad_cdr) const {
   Vector<Real> source(m_num_species, 0.0);
   source[1] = k*a_cdr_densities[0]*a_cdr_densities[1];
   return source;
}
```

In the above function, the user may also implement photoionization: The argument *Vector<Real> a_rte_densities* is the isotropic photon densities, i.e. the number of photons per unit volume.

5.4.2.4 Defining photon production terms

Reverse coupling between the CDR equations and the RTE equations occur through the *compute_rte_source_terms* function. The return value of this function is the mean number of photons produced per steradian. Often, such functions may be complicated. If we assume, for example, that the RTE source term is $\eta = n/\tau$, where τ is a spontaneous emission lifetime, then we can implement the coupling as

```
Vector<Real> compute_rte_source_terms(const Real&          a_time,
                                      const RealVect&      a_pos,
                                      const RealVect&      a_E,
                                      const Vector<Real>& a_cdr_densities) const {
   Vector<Real> source(1);
   source[0] = a_cdr_densities[0]/tau;
   return source;
}

Generally, one wants to ensure consistency in how one handles photon production and excited state relaxation.␣
↪For the above radiative transfer example, the user should also include a corresponding term in the function␣
↪that computes the chemistry source terms.
```

5.4.2.5 Setting transport boundary conditions

Boundary conditions are support through three functions that handle transport through three types of boundaries: domain boundaries, dielectric surfaces, and electrode surfaces. The three functions have (almost) the same signature:

```
Vector<Real> compute_cdr_electrode_fluxes(const Real&          a_time,
                                          const RealVect&      a_pos,
                                          const RealVect&      a_normal,
                                          const RealVect&      a_E,
                                          const Vector<Real>& a_cdr_densities,
                                          const Vector<Real>& a_cdr_velocities,
                                          const Vector<Real>& a_cdr_gradients,
                                          const Vector<Real>& a_rte_fluxes,
                                          const Vector<Real>& a_extrap_cdr_fluxes) const {
   Vector<Real> fluxes(m_num_species, 0.0);
   return fluxes;
}
```

This function expects you to return the transport fluxes at the boundaries - like those occuring in a finite volume context. For domain boundaries, this signature is slightly changed: The argument `a_normal` (which is the normal vector *into* the gas volume) is replaced by two arguments that describe the side and direction of the domain wall. The `a_normal` is the normal into the gas volume, which is opposite to the convention used in finite volume formulations. The arguments `a_cdr_velocities` are the drift velocities projected on the outward normal, and the same convention is used for `a_cdr_gradients` (which hold the spatial gradients) and `a_extrap_cdr_fluxes` which hold the extrapolated drift fluxes. If you want simple extrapolated boundary conditions you would set one of the fluxes equal to `a_extrap_cdr_fluxes`. A small caveat: `a_extrap_cdr_fluxes` are the extrapolated *drift* fluxes; if you also want the diffusive flux you can use the gradient argument and recompute the diffusion coefficient.

5.4.2.6 Setting initial surface charge

Finally, the final function specifies the initial surface charge in the domain. If there is no initial surface charge, then

```
Real initial_sigma(const Real      a_time,
                   const RealVect& a_pos) const {
   return 0.0
}
```

5.4.2.7 species

The *species* is a lightweight class used to provide information into convection-diffusion-reaction solvers. This class is mostly used within *cdr_plasma_physics* in order to provide information on how to instantiate CDR solvers. *species* is abstract so that the user must implement

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

This function specifies the initial data of the species that is advected. For example, the following implementation sets the initial CDR density value to one:

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

In addition to this, the user *must* provide information on the charge of the species, and whether or not it is mobile or diffusive. In addition, he should set the name of the species so that it can be identified in output files. In *PlasmaC*, this is done by setting the following four values in the constructor

```
std::string m_name; // Solver name
int m_charge;       // Charge (in units of the elementary charge)
bool m_diffusive;   // Diffusive species or not
bool m_mobile;      // Mobile species or not
```

Usually, these are set through the constructor. The `m_charge` unit is in units of the elementary charge. For example, the following is a full implementation of an electron species:

```
class electron : public species {
  electron() {
     m_name    = "electrons";
     m_charge = -1;
     m_diffusive = true;
     m_mobile = true;
  }

  ~electron(){}

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

The members `m_mobile` and `m_diffusive` are used for optimization in *PlasmaC*: If the user specifies that a species is immobile, *PlasmaC* will skip the advection computation. Note that `m_diffusive` and `m_mobile` override the specifications in *cdr_plasma_physics*. If the user provides a non-zero velocity through *cdr_plasma_physics* function *compute_cdr_velocities*, and sets `m_mobile` to `false`, the species velocity will be zero. Of course, the user will often want to provide additional input information to his species, for example by specifying a seed for the initial conditions.

5.4.2.8 photons

photons is the class that supplies extra information to the RTE solvers. In those solvers, the source term computation is handled by *cdr_plasma_physics*, so the *photons* class is very lightweight. The user must implement a single function which specifies the absorption coefficient at a point in space:

```
virtual const Real get_absorption_coeff(const RealVect& a_pos) const = 0;
```

In addition, the user should provide a name for the RTE solver so that it can be identified in the output files. This is done by setting a `m_name` attribute in the *photons* class.

The following is a full implementation of the *photons* class:

```
class my_photon : public photon {
  my_photon() {
     m_name = "my_photon";
  }

  ~my_photon(){}

  const Real get_absorption_coeff(const RealVect& a_pos) const {
     return 1.0;
  }
};
```

By default, there are no input parameters available for the *photons* class, but the user will often want to include these, for example by modifying the absorption coefficient. Note that you are allowed to use a spatially varying absorption coefficient.

For most users, this will mostly include implementing a new geometry or a new plasma-kinetic scheme. It is possible to generate entirely new physics interfaces, too. Our goal is that the user does not need to worry about temporal or spatial discretization of these equations, but rather focus on the actual setup of the geometry and physics.

5.4.3 Temporal discretization

In this chapter we discuss the supported temporal integrators for *PlasmaC*, and discuss their input parameters. These integrators differ in their level of efficiency and accuracy. Currently, none of the integrators can subcycle in time.

5.4.4 Time step limitations

Our time integrators have different time step limitations. Fully explicit codes are limited by the advective and diffusive CFL constraints and usually also the dielectric relaxation time. However, some of the *PlasmaC* integrators eliminate the dielectric relaxation time, and all integrators can handle diffusion either implicitly or explicitly. In some cases only the advective CFL constraint is the only time step restriction.

5.4.5 Deterministic integrators

5.4.5.1 godunov

The `godunov` temporal integrator is a rather unsophisticated, but very stable, temporal integrator. `godunov` uses an operator splitting between charge transport and plasma chemistry in the following way:

1. Advance $\partial_t \phi = -\nabla \cdot \left(\mathbf{v}\phi - D\nabla\phi + \sqrt{2D\phi}\mathbf{Z} \right)$ (and the surface charge solver) over a time step Δt.
2. Compute the electric field
3. Advance the plasma chemistry over the same time step using the field computed in 2). I.e. advance $\partial_t \phi = S$ over a time step Δt.
4. Move photons and deposit them on the mesh.

Various integration options for the transport and chemistry steps are available but are discussed elsewhere. Note that the `godunov` integrator uses a semi-implicit coupling between the plasma chemistry terms and the electric field, and therefore eliminates the so-called dielectric relaxation time. The formal order of convergence of the `godunov` integrator is 1, but the accuracy can be quite good depending on the transport and chemistry schemes that are chosen.

5.4.5.2 imex sdc

`imex_sdc` is a semi-implicit spectral deferred correction method for the *PlasmaC* equation set and is an adaptive high-order discretization with implicit diffusion. This method integrates the advection-diffusion-reaction equations in the following way.

Spectral deferred corrections

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.

We now discuss the semi-implicit SDC (SISDC) method. First, we apply the method of lines (MOL) such that

$$\frac{d\phi_{\mathbf{i}}}{dt} = \mathcal{F}_{\mathrm{AR}}\left(t,\phi_{\mathbf{i}}\right) + \mathcal{F}_{\mathrm{D}}\left(t,\phi_{\mathbf{i}};\mathbf{E}_{\mathbf{i}}\right), \tag{5.4.11}$$

$$\frac{d\sigma_{\mathbf{i}}}{dt} = \mathcal{F}_{\sigma}\left(t,\phi_{\mathbf{i}}\right), \tag{5.4.12}$$

where $\phi_{\mathbf{i}}$ denotes a cell-averaged variable, $\mathcal{F}_{\sigma}$ is as described in *Spatial discretization*, $\mathcal{F}_{\mathrm{AR}}\left(t,\phi_{\mathbf{i}}\right) = -D^{c}_{\mathbf{i}} + S_{\mathbf{i}}$ is the advection-reaction operator , and $\mathcal{F}_{\mathrm{D}}(t,\phi_{\mathbf{i}};\mathbf{E}_{\mathbf{i}}) = \frac{1}{\kappa_{\mathbf{i}}}\int_{V_{\mathbf{i}}}\left[\nabla\cdot\left(D\nabla\phi\right)\right]dV_{\mathbf{i}}$ is the diffusion operator. Note that the advective operator contains the hybrid divergence discussed in *Convection-Diffusion-Reaction* and $\mathcal{F}_{\mathrm{D}}$ is parametrically coupled to $\mathbf{E}$ through $D = D\left(\mathbf{E}\right)$ (we use a semi-colon to indicate this dependence). Strictly speaking, $\mathcal{F}_{\mathrm{AR}}$ is parametrically coupled in the same way through the mobilities and boundary conditions, and additionally coupled to Ψ through source terms so that the notation $\mathcal{F}_{\mathrm{AR}}\left(t,\phi_{\mathbf{i}};\mathbf{E}_{\mathbf{i}},\Psi_{\mathbf{i}}\right)$ would be appropriate. However, charge injection, advection, and chemistry will be integrated explicitly so this dependence is notationally suppressed. On the other hand, the diffusion part will be solved with the backward Euler method - which yields a Helmholtz equation - and so we need to maintain this dependence for now. Later, we will clarify how this dependence is resolved. The rationale for solving diffusion implicitly is due to the numerical time step constraint of explicit diffusion methods which scales as $\mathcal{O}\left(\Delta x^2\right)$, whereas advection scales more favorably at $\mathcal{O}\left(\Delta x\right)$. We have chosen to integrate the reactive terms explicitly. The reason is that the reactive terms can be non-local, i.e. they can depend on the electron gradient. This is for example the case for fluid models in the local energy approximation where the electron energy source term contains

terms that are proportional to the electron diffusion term $D_e \nabla \phi_e$. Implicit discretization of the reactive terms then yield a fully coupled system rather than systems coupled only within individual cells. Charge injection is also handled explicitly. This design choice is mandated by the fact that implicit charge injection through the diffusion terms couples every diffusive species, leading to a system of diffusion equations that are fully coupled through their boundary conditions. Although charge injection could reasonably be performed as a separate step, this leads to numerical instabilities for cut-cell methods since the injected charge must be normalized by the volume fraction of the cell (which can be arbitrarily small).

SISDC predictor

Next, we present the SISDC method. In what follows, we suppress the index $\mathbf{i}$ as it is not explicitly needed. Given an interval $[t_n, t_{n+1}]$ on which a solution is sought, SDC methods divide this interval into p subintervals $t_n = t_{n,0} < t_{n,1} < \ldots < t_{n,p} = t_{n+1}$. Our discussion, however, pertains only to the interval $[t_n, t_{n+1}]$ so we compress the notation to $t_m \equiv t_{n,m}$. We obtain an initial solution $\phi_m^0, m = 0, 1, \ldots, p$ as the semi-implicit advance

$$\phi_{m+1}^0 = \phi_m^0 + \Delta t_m \left[\mathcal{F}_{\text{AR}}\left(t_m, \phi_m^0\right) + \mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^0; \mathbf{E}_{m+1}^0\right)\right], \tag{5.4.13}$$

$$\sigma_{m+1}^0 = \sigma_m^0 + \Delta t_m F_\sigma\left(t_m, \phi_m^0\right). \tag{5.4.14}$$

This defines a Helmholtz problem for ϕ_{m+1}^0 through $\mathcal{F}_{\text{D}}$. Generally, the upper subscript denotes an SDC iteration where subscript 0 is the SISDC predictor, and we also have $\phi_0^0 = \phi(t_n)$ and $\sigma_0^0 = \sigma(t_n)$. This predictor treats advection and chemistry terms explicitly, and diffusion implicitly. Other types of semi-implicit or multi-implicit couplings are possible [BLM03][LM04][NBD+12]. SDC improves this solution by using deferred corrections: Given a numerical solution ϕ_{m+1}^k, we compute an error δ_{m+1}^k and obtain the next iterate $\phi_{m+1}^{k+1} = \phi_{m+1}^k + \delta_{m+1}^k$. Each iteration raises the discretization order by one [DGR00][Min03], to maximum order $p + 1$. Critical to the success of this approach is the precise evaluation of the numerical quadrature.

The parametric coupling of the electric field complicates things since the predictor contains $\mathbf{E}_{m+1}^0 = \mathbf{E}\left(\phi_{m+1}^0\right)$, implying that the Poisson equation and the diffusion advance require concurrent solves for the diffusion update. We simplify this system by using a weak coupling by first computing

$$\phi_{m+1}^{0,*} = \phi_m^0 + \Delta t_m \mathcal{F}_{\text{AR}}\left(t_m, \phi_m^0\right), \tag{5.4.15}$$

$$\sigma_{m+1}^0 = \sigma_m^0 + \Delta t_m F_\sigma\left(t_m, \phi_m^0\right), \tag{5.4.16}$$

Next, we will approximate $\mathbf{E}_{m+1}^0$ for use in the predictor. There are two choices for this coupling; one may either use $\mathbf{E}_m^0$ for computation of the diffusion coefficients, which we will refer to as the semi-implicit coupling, or one may use fixed-point iteration and compute $\mathbf{E}_{m+1}^{0,*} = \mathbf{E}\left(\phi_{m+1}^{0,*}, \sigma_{m+1}^0\right)$, followed by the diffusion advance

$$\phi_{m+1}^{0,\dagger} = \phi_{m+1}^{0,*} + \Delta t_m \mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^0; \mathbf{E}_{m+1}^*\right),$$

which we will refer to as the implicit coupling. This is e.g. the electric field coupling used in [Mar19]. This approximation can be improved by using more fixed-point iterations that computes $\mathbf{E}_{m+1}^{0,\dagger} = \mathbf{E}\left(\phi_{m+1}^{0,\dagger}, \sigma_{m+1}^0\right)$ and then re-solves the predictor equation with $\mathbf{E}_{m+1}^{0,\dagger}$ in place of $\mathbf{E}_{m+1}^{0,*}$. The process can then be repeated for increased accuracy. Regardless of which coupling is used, we have now calculated ϕ_{m+1}^0, σ_{m+1}^0, through which we obtain $\mathbf{E}_{m+1}^0 = \mathbf{E}\left(\phi_{m+1}^0, \sigma_{m+1}^0\right)$, and $\Psi_{m+1}^0 = \Psi\left(\mathbf{E}_{m+1}^0, \phi_{m+1}^0\right)$. Finally, we remark that the SISDC predictor is a sequentially advanced semi-implicit Euler method, which is locally second order accurate and globally first order accurate. Each step of the predictor can be thought of as a Godunov splitting between the advective-reactive and diffusive terms.

SISDC corrector

Next, the semi-implicit discretization of the correction equation is

$$\begin{aligned}\delta_{m+1}^k = \delta_m^k + \Delta t_m \left[\mathcal{F}_{\text{AR}}\left(t_m, \phi_m^k + \delta_m^k\right) - \mathcal{F}_{\text{AR}}\left(t_m, \phi_m^k\right)\right. \\ \left. + \mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^k + \delta_{m+1}^k; \mathbf{E}_{m+1}^k\right) - \mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^k; \mathbf{E}_{m+1}^k\right)\right] - \left(R_{m+1}^k - R_m^k\right).\end{aligned}$$

We furthermore define

$$\begin{aligned}R_{m+1}^k - R_m^k &= \int_{t_m}^{t_{m+1}} \left[\mathcal{F}_{\text{AR}}\left(\phi^k\right) + \mathcal{F}_{\text{D}}\left(\phi^k; \mathbf{E}^k\right)\right] d\tau - \phi_{m+1}^k + \phi_m^k \\ &\equiv I_m^{m+1}\left(\phi^k\right) - \phi_{m+1}^k + \phi_m^k.\end{aligned}$$

Evaluation of I_m^{m+1} yields p quadrature rules and we may write

$$I_m^{m+1}\left(\phi^k\right) = \sum_{l=0}^{p} q_m^l \left[\mathcal{F}_{\text{AR}}\left(t_l, \phi_l^k\right) + \mathcal{F}_{\text{D}}\left(t_l, \phi_l^k; \mathbf{E}_l^k\right)\right],$$

where the weights q_m^l are quadrature weights. The final update for ϕ_{m+1}^{k+1} is then

$$\begin{aligned}\phi_{m+1}^{k+1} = \phi_m^{k+1} + \Delta t_m \left[\mathcal{F}_{\text{AR}}\left(t_m, \phi_m^{k+1}\right) - \mathcal{F}_{\text{AR}}\left(t_m, \phi_m^k\right)\right. \\ \left. + \mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^{k+1}; \phi_{m+1}^{k+1}\right) - \mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^k; \mathbf{E}_{m+1}^k\right)\right] + I_m^{m+1}\left(\phi^k\right).\end{aligned}$$

With the exception of $\mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^{k+1}; \mathbf{E}_{m+1}^{k+1}\right)$, all quantities on the right-hand are known and the correction equation is reduced to a Helmholtz equation for ϕ_{m+1}^{k+1} with error $\delta_{m+1}^k = \phi_{m+1}^{k+1} - \phi_{m+1}^k$. An analogous equation is found for σ_{m+1}^{k+1}.

The correction step has the same coupling to the electric field as the prediction step in that $\mathbf{E}_{m+1}^{k+1}$ appears in the update equation for ϕ_{m+1}^{k+1}. As for the prediction, we use a weak coupling through which we first compute

$$\phi_{m+1}^{k+1,*} = \phi_m^{k+1} + \Delta t_m \left[\mathcal{F}_{\text{AR}}\left(t_m, \phi_m^{k+1}\right) - \mathcal{F}_{\text{AR}}\left(t_m, \phi_m^k\right)\right] + I_m^{m+1}\left(\phi^k\right), \tag{5.4.17}$$

$$\sigma_{m+1}^{k+1} = \sigma_m^{k+1} + \Delta t_m \left[F_\sigma\left(t_m, \phi_m^{k+1}\right) - F_\sigma\left(t_m, \phi_m^k\right)\right] + \Sigma_m^{m+1}\left(\phi^k\right). \tag{5.4.18}$$

The solution for σ_{m+1}^{k+1} is final since all charge is injected through the advection operator for ϕ. The term Σ_m^{m+1} contains the injected charge through $I_m^{m+1}\left(\phi^k\right)$, as was discussed in *Spatial discretization*. We then solve

$$\phi_{m+1}^{k+1} = \phi_{m+1}^{k+1,*} + \Delta t_m \left[\mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^{k+1}; \mathbf{E}_{m+1}^{k+1}\right) - \mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^k; \mathbf{E}_{m+1}^k\right)\right],$$

with some approximation for $\mathbf{E}_{m+1}^{k+1}$. As before, this coupling can be made either semi-implicitly or implicitly. The corrector equation defines a Helmholtz equation for ϕ_{m+1}^{k+1} using $\phi_{m+1}^{k+1,*}$ as the previous solution and $-\mathcal{F}_{\text{D}}\left(\phi_{m+1}^k; \mathbf{E}_{m+1}^k\right)$ as a source term.

Order, stability, and computational cost

For consistency with the literature, denote the SISDC method which uses P nodes (i.e. $P-1$ subintervals) and K total iterations (i.e. $K-1$ iterations of the correction equation) by $\texttt{SISDC}_P^K$. This method will have a global order of accuracy $\min(K, P)$ if the quadrature can be evaluated with appropriate accuracy. Order reductions may occur if the interpolating polynomial in the quadrature suffers from Runge's phenomenon. As we discuss below, uniformly spaced nodes have some computational advantage but is therefore also associated with some risk. Safer choices include Lobatto nodes or Chebyshev nodes (with inclusion of endpoints) to minimize the risk of order reductions. Implications on the choice of quadrature nodes can be found in [LM05].

For explicit advection, the deferred correction procedure integrates the correction equation sequentially and therefore does not allow each substep Δt_m to exceed the CFL-limited time step Δt_{cfl}, i.e. $\Delta t_m < \Delta t_{\text{cfl}} \forall m$. Since we have $\Delta t = \sum_m \Delta t_m$, uniform nodes maximize Δt subject to the CFL constraint. For example, an $\texttt{SISDC}_P^K$ method with uniformly spaced nodes has a maximum possible time step $\Delta t < (P-1)\Delta t_{\text{cfl}}$. For the same number of function evaluations, the allowed time step with for Lobatto or Chebyshev nodes is smaller. For $P \leq 3$, the uniform nodes, Lobatto nodes, and Chebyshev nodes coincide. Larger time steps are possible with uniform nodes for $P > 3$, which has some computational consequence. The table below summarizes the largest possible time steps for the $\texttt{SISDC}_P^K$ method with the various quadratures. Finally, note that $\Delta t_m < \Delta t_{\text{cfl}}$ does not guarantee stability since further restrictions may required for stability of the reaction terms.

P	Lobatto	Chebyshev	Uniform
2	Δt_{cfl}	Δt_{cfl}	Δt_{cfl}
3	$2\Delta t_{\text{cfl}}$	$2\Delta t_{\text{cfl}}$	$2\Delta t_{\text{cfl}}$
4	$2.26\Delta t_{\text{cfl}}$	$1.73\Delta t_{\text{cfl}}$	$3\Delta t_{\text{cfl}}$
5	$3.05\Delta t_{\text{cfl}}$	$2.82\Delta t_{\text{cfl}}$	$4\Delta t_{\text{cfl}}$
6	$3.50\Delta t_{\text{cfl}}$	$3.29\Delta t_{\text{cfl}}$	$5\Delta t_{\text{cfl}}$
7	$4.26\Delta t_{\text{cfl}}$	$4.36\Delta t_{\text{cfl}}$	$6\Delta t_{\text{cfl}}$

For the predictor step, it is necessary to evaluate $\mathcal{F}_{\text{AR}}\left(\phi_m^{k+1}\right)$ and thus update the Poisson and radiative transfer equations at each node. In addition, it is necessary to solve the diffusion equation at every node except $m = 0$ for every diffusive species, which may also require auxiliary updates of the electric field. The corrector step contains extra floating point operator due to the extra terms $\mathcal{F}_{\text{AR}}\left(t_m, \phi_m^k\right)$ and $\mathcal{F}_{\text{D}}\left(t_{m+1}, \phi_{m+1}^k\right)$ and the quadrature I_m^{m+1}. The computational cost of adding in these terms is small compared to the cost of an Euler update of the advection-reaction equation since one must also computate source terms, drift velocities, and boundary conditions in addition to construction of the hybrid divergence. In short, the computational cost of the predictor and corrector steps are about the same.

Next, we provide some remarks on the extra computational work involved for higher order methods. Broadly speaking, the total amount of floating point operations increases quadratically with the order. Each node requires evaluation of one advection-reaction operator, at least one electric field update, and one radiative transfer update. Likewise, each substep requires one diffusion solve. Thus, $\texttt{SISDC}_K^K$ requires K^2 advection-reaction evaluations, $(K-1)^2$ diffusion solves, $(K-1)^2$ radiative transfer updates, and at least K^2 electric field updates. In these estimates we have assumed that the diffusion solve couples semi-implicitly to the electric field, thus each corrector iteration requires one electric field update per node, giving a total cost K^2. Strictly speaking, the number of advection-reaction evaluations is slightly less since $\mathcal{F}_{\text{AR}}\left(t_0, \phi_0^k\right)$ does not require re-evaluation in the corrector, and $\mathcal{F}_{\text{AR}}\left(t_p, \phi_p^{K-1}\right)$ does not need to be computed for the final iteration since the lagged quadrature is not further needed. Nonetheless, the computational work is quadratically increasing, but this is partially compensated by allowance of larger time steps since the $\texttt{SISDC}_K^K$ has a stability limit of $(K-1)\Delta t_{\text{cfl}}$ rather than Δt_{cfl} for uniformly spaced nodes. For comparison with the predictor $\texttt{SISDC}_K^1$ which is a first order method, the work done for integration over $(K-1)\Delta t_{\text{cfl}}$ amounts to $K-1$ advection-reaction updates, $K-1$ diffusion updates, $K-1$ radiative transfer updates, and K electric field updates. If we take the electric field updates as a reasonable metric for the computational work, the efficiency of the K th order method over the first order method is about K for integration over the same time interval, i.e. it increases linearly rather than quadratically. However, this estimate is only valid if we do not take accuracy into account. In practice, the predictor does not provide the same accuracy as the corrector over the same integration interval. A fair comparison of the extra computational work involved would require that the accuracy of the two methods be the same after integration over a time $(K-1)\Delta t_{\text{cfl}}$, which will generally require more substeps for the first order method. While we do not further pursue this quantification in this paper, the pertinent point is that the extra computational work involved for tolerance-bound higher order discretizations increases sub-linearly rather than quadratically when compared to lower-order equivalents.

We have implemented the SISDC algorithm in the `sisdc` class in `/time_steppers/sisdc`. The following class options are available:

```
# ============================================================================================================
# IMEX_SDC CLASS OPTIONS
#
# This class uses semi-implicit spectral deferred corrections. Diffusion is handled implicitly,
# and advection-reaction is handled explicitly.
#
# The maximum possible global order of accuracy is (p+1) where p is the number of subintervals. Each
# correction raises the order by 1 (corr_iter=0 is the first order solution). To reach the maximum
# possible order, you should perform p correction iterations.
#
#
# ============================================================================================================
imex_sdc.verbosity        = -1      # Class verbosity
imex_sdc.solver_verbosity = -1      # Individual solver verbosities
imex_sdc.fast_rte         = 1       # Solve RTE every this time steps
imex_sdc.fast_poisson     = 1       # Solve Poisson every this time steps
imex_sdc.min_dt           = 0.      # Minimum permitted time step
imex_sdc.max_dt           = 1.E99   # Maximum permitted time step
imex_sdc.cfl              = 0.5     # CFL number
imex_sdc.relax_time       = 1.0     # Relaxation time constant
imex_sdc.source_comp      = interp  # Interpolated 'interp' or cell-average 'cell_ave' for source computations

# -----------------------------------
# Nodes, subintervals, and corrections
# -----------------------------------
imex_sdc.quad_nodes   = lobatto     # Nodes to be used for quadrature. 'lobatto', 'uniform', or 'chebyshev'
imex_sdc.subintervals = 1           # Number of subintervals. This will be the maximum possible order.
imex_sdc.corr_iter    = 1           # Number of iterations of the correction equation. Should be (subintervals-
↪1)
                            # for maximum order

# ------------------------------------------------------
# Diffusive coupling
#
# This defines the weak coupling used for the implicit diffusion advance.
# ------------------------------------------------------
imex_sdc.diffusive_coupling = weak  # Diffusion coupling, either 'weak' or 'strong'
imex_sdc.use_tga            = false # Use second order diffusion per substep ('true')
imex_sdc.num_diff_corr      = 0     # Number of corrections for 'strong'

# Adaptive time stepping
# ------------------------------------------------------
imex_sdc.print_report    = false    # Print report with error and stuff
imex_sdc.adaptive_dt     = false    # Use adaptive time stepping
imex_sdc.error_norm      = 2        # Error norm (0 = Linf)
imex_sdc.min_corr        = 0        # Minimum number of corrections
imex_sdc.max_retries     = 100      # Maximum number of tries (for step rejection)
imex_sdc.max_growth      = 1.2      # Maximum permissible time step growth
imex_sdc.decrease_safety = 0.9      # Fudge factor when we decrease the time step.
imex_sdc.min_cfl         = 0.1      # Minimum CFL
imex_sdc.max_cfl         = 0.75     # Maximum CFL
imex_sdc.max_error       = 1.E-3    # Error threshold
imex_sdc.error_index     = -1       # Error index. If -1, evaluate all CDR solvers
imex_sdc.safety          = 0.75     # Safety factor for time stepping

# ------------------------------------------------------
# "Asymptotic preserving". Development feature.
# ------------------------------------------------------
imex_sdc.use_AP        = false      # Turn on the "asymptotic preserving" feature.

# ------------------------------------------------------
# Debugging options
# ------------------------------------------------------
imex_sdc.consistent_E      = true   # Use consistent E-field computations (update in between RK stages)
imex_sdc.consistent_rte    = true   # Use consistent RTE updates
imex_sdc.compute_v         = true   # Compute v in between substeps.
imex_sdc.compute_S         = true   # Compute S in between substeps.
imex_sdc.compute_D         = true   # Compute S in between substeps.
imex_sdc.do_advec_src      = true   # Turn off code functionality. Only for debugging
imex_sdc.do_diffusion      = true   # Turn off code functionality. Only for debugging
imex_sdc.do_poisson        = true   # Turn off code functionality. Only for debugging
imex_sdc.do_rte            = true   # Turn off code functionality. Only for debugging
imex_sdc.profile_steps     = false  # Profile time steps with order, accuracy, cfl, etc.

# ---------------------
# Advection extrapolation. Currently does NOT work with subcycling (but I'm working on it...)
# ---------------------
imex_sdc.extrap_advect = true      # Time extrapolate with source and diffusion in advection step
```

5.4.6 Stochastic integrators

Deterministic CFD integrators are generally not suitable for stochastic ODEs. For example, the recursive nature of Heun's method or spectral deferred corrections hardly make sense for stochastic ODEs.

For fluctuating hydrodynamics we are preparing several temporal integrators. `godunov` is generally useful for FHD but we also provide an Euler-Maruyama integrator (`euler_maruyama`) which is first order accurate in time (although with an accurate advective integrator). Although `euler_maruyama` is functionally similar to `godunov`, it does not eliminate the dielectric relaxation time and is therefore less stable for some simulation cases.

5.4.6.1 euler_maruyama

euler_maruyama implements the Euler-Maruyama method. This method is based on an Euler method with explicit or implicit diffusion.

CHAPTER

SIX

TUTORIAL

6.1 Introduction

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

6.2 Setting up `time_stepper`

CHAPTER

SEVEN

REFERENCES

7.1 References

BIBLIOGRAPHY

[ACG+04] M Adams, P. Colella, D.T. Graves, J. N. Johnson, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B Van Straalen. Chombo Software Package for AMR Applications - Design Document. Technical Report, Lawrence Berkeley National Laboratory, 2004.

[BLM03] Anne Bourlioux, Anita T. Layton, and Michael L. Minion. High-order multi-implicit spectral deferred correction methods for problems of reactive flow. *Journal of Computational Physics*, 2003. doi:10.1016/S0021-9991(03)00251-1.

[DGR00] Alok Dutt, Leslie Greengard, and Vladimir Rokhlin. Spectral deferred correction methods for ordinary differential equations. *BIT Numerical Mathematics*, 2000. doi:10.1023/A:1022338906936.

[LM04] Anita T. Layton and Michael L. Minion. Conservative multi-implicit spectral deferred correction methods for reacting gas dynamics. *Journal of Computational Physics*, 2004. doi:10.1016/j.jcp.2003.09.010.

[LM05] Anita T. Layton and Michael L. Minion. Implications of the choice of quadrature nodes for Picard integral deferred corrections methods for ordinary differential equations. *BIT Numerical Mathematics*, 2005. doi:10.1007/s10543-005-0016-1.

[Mar19] Robert Marskar. An adaptive cartesian embedded boundary approach for fluid simulations of two- and three-dimensional low temperature plasma filaments in complex geometries. *Submitted to Journal of Computational Physics*, 2019.

[Min03] Michael L. Minion. Semi-implicit spectral deferred correction methods for ordinary differential equations. *Communications in Mathematical Sciences*, 2003. doi:10.4310/CMS.2003.v1.n3.a6.

[NBD+12] A. Nonaka, J. B. Bell, M. S. Day, C. Gilet, A. S. Almgren, and M. L. Minion. A deferred correction coupling strategy for low Mach number flow with complex chemistry. *Combustion Theory and Modelling*, 2012. arXiv:1512.06459, doi:10.1080/13647830.2012.701019.