Space Plasma and Energetic Charged particle TRansport on Unstructured Meshes: Documentation and Tutorial

Juan G Alonso Guzman[*]
Department of Space Science, University of Alabama in Huntsville
Vladimir A Florinski[†]
Department of Space Science, University of Alabama in Huntsville
Updated November 22, 2024

Abstract

This document serves as both a reference manual and a quick-start guide to the Space Plasma and Energetic Charged particle TRansport on Unstructured Meshes (SPECTRUM), an open-source suite of scientific numerical simulation codes, available on GitHub. Sections 1 and 2 provide a broad overview of the software and serve as a basic tutorial. Sections 3-9 delve more deeply into specific modules within the SPECTRUM framework, describing the relevant models for each in detail and providing notes on their recommended usage.

If you find any mistakes or have any constructive feedback, don't hesitate to contact us so we can improve this document. Thank you to ShareLaTeX for the template.

Contents

[*] jgg0008@uah.edu
[†] vaf0001@uah.edu

1 Introduction

This section describes the scope (i.e. applicability) and general structure of the SPECTRUM suite of codes. Both the file structure and software architecture are discussed. Finally, we give a (non-exhaustive) summary of SPECTRUM's current capabilities and outline the general procedure for using it.

1.1 Scope of the Software

SPECTRUM is designed to efficiently perform **test-particle** simulations, classical or relativistic, in virtually any astrophysical environment. By test-particle, we mean any particle whose trajectory is influenced by the force fields surrounding it (e.g. magnetic field generated by the Sun), but itself has a negligible effect on said fields, so that its equations of motion need not be solved self-consistently. In other words, the particle's movement can be accurately be decoupled from the equations specifying the background fields. Thus, for the purposes of studying charged particle transport, SPECTRUM assumes that the fields can be prescribed, and do not depend on the outcome of the particle simulation.

SPECTRUM is also designed with two main purposes in mind. The first is to compute real, physical trajectories of charged particles by specifying their equations of motion subject to known background fields. The second is to solve partial differential equations (PDEs) describing the diffusive transport of distribution functions via the **method of stochastic characteristics**. In a nutshell, this is a Monte Carlo (MC) approach that consists in finding a suitable stochastic differential equation (SDE) whose solution has an expected value that solves the PDE in question when weighed appropriately to account for the initial/boundary conditions, sources/sinks, and linear growth terms. Instead of attacking the transport PDE directly, the method of stochastic characteristics resorts to repeatedly solving an associated SDE, and leveraging the Law of Large Numbers to average these results, thus yielding an approximate solution to the original transport equation. Note that fully deterministic transport equations (i.e. without diffusion terms) can also be solved using SPECTRUM. In that case, the approach simplifies to the traditional method of characteristics, often called **Liouville mapping**.

Test-particle simulations can be categorized according to the relative direction of the passage of time within a particular run. Quite explicitly, simulations in which the particles traverse the domain with time flowing in the natural (forward) direction are termed **forward-in-time**, while simulations in which the particles experience the fields evolving in the reverse (backward) direction are called **backward-in-time**. In a purely deterministic setting, the equations of motion for the particles are time reversible, and thus these two modes of execution are equivalent, meaning that a particle starting at $\mathbf{r}_1$ and traveling forward-in-time to $\mathbf{r}_2$ should retrace its steps, to within the accuracy of the integrator being used, back to $\mathbf{r}_1$ if evolved backward-in-time from $\mathbf{r}_2$.

However, in a stochastic system, these two directions of integration calculate fundamentally different quantities. In this case, the forward-in-time mode computes possible trajectories for a particle subject to diffusive motion, while the paths generated during backward-in-time integration do not have a direct physical interpretation, but rather correspond to the stochastic characteristics of an underlying SDE, and are thus referred to as **pseudo-particles** or **pseudo-trajectories**.

1.2 File Structure

The basic file structure is quite simple. If the root directory is located at `$SPECTRUM`, the essential components are

- the `$SPECTRUM/common` folder,
- the `$SPECTRUM/src` folder,
- the `$SPECTRUM/configure.ac` file, and
- the `$SPECTRUM/Makefile.am` file.

The `common` and `src` folders contain the C++ (and a bit of Fortran) source code that makes up SPECTRUM. In general, `common` contains the more abstract methods and classes along with most physical and mathematical constants, while `src` contains more physics based and application specific code. The `configure.ac` and `Makefile.am` files handle the compilation of the code using Automake, which is the subject of Subsection 1.4.

→ Section 9
→ Sections 3-8

Additionally, the latest official version of the software should come with

- a `$SPECTRUM/benchmarks` folder,
- a `$SPECTRUM/tests` folder, and
- a `$SPECTRUM/runs` folder.

The `benchmarks` folder contains a collection of tests and benchmark codes that can be executed to ensure proper compilation and can also serve as template/tutorial programs. The `tests` and `runs` folders are meant to contain the codes built by the user to setup and run simulations, which we call **simulation files**. They each contain simulation programs meant to serve as a examples for the user.

Rather than using the default `tests` and `runs` subfolders, the user can create their own subdirectories to organize projects. In order to incorporate a new folder, call it `project1`, into the automatic compilation scheme, follow these steps:

1. Create the directory `$SPECTRUM/project1`.
2. Create the file `$SPECTRUM/project1/Makefile.am` and format it appropriately. You can also copy and paste the contents of an existing Makefile in a default subdirectory.
3. Add `project1` to the list of `SUBDIRS` in `$SPECTRUM/Makefile.am`.
4. Add `project1/Makefile` to the `AC_CONFIG_FILES` list of Makefiles to generate in `$SPECTRUM/configure.ac`.

→ Subsection 1.4

1.3 Software Architechture

As mentioned previously in Subsection 1.1, SPECTRUM calculates characteristic paths for PDEs or SDEs modeling the transport of distribution functions, which in certain conditions, can be equated with straightforward computation of particle trajectories. To achieve this task, SPECTRUM spawns "Trajectory Integrator" (TI) objects, which, as their name implies, calculate trajectories for a specified transport type, each drawing starting conditions according to some initial distribution and terminating when upon crossing an expiring (temporal) or absorbing (spatial) boundary, and recording data throughout its journey every time a binning boundary is encountered.

→ Section 3

Each TI object posseses several attributes that aid in the trajectory computation, the most important of which are

- a "Background" (BG) object, which produces the necessary fields at each point of trajectory,
- an array of "Partial Distribution" (PDis) objects, which collects and bins the pertinent quantities,

→ Section 4
→ Section 5

→ Section 6

- two “Initial Condition” objects (one for space, another for momentum), used for choosing the starting point/velocity of each trajectory,

→ Section 7

- an array of “Boundary Condition” objects (in time, space, or momentum), used for triggering binning events and delimiting the simulation domain by terminating trajectories, and

→ Section 8

- a “Diffusion” object, which, if employed by the chosen mode of transport, provides the coefficients needed for the stochastic contribution to each step.

Any of these objects can be declared, set up, and used as the central focus of a simulation. For example, a single TI object, along with its necessary members may be invoked to study the trajectory of a particle in some field or the BG object can be initialized to print values for visualization purposes (see the files in `$SPECTRUM/benchmarks` for examples). However, in a full simulation run, all of this elements work together as part of a bigger unit, which we call a Simulation object. The TI object is itself a member of a “Simulation Worker” (SimW) object, capable of sequentially integrating many trajectories and gathering cumulative distributions in the PDis objects, which are automatically shared between the SimW object and its TI member through smart pointers.

Many SimW processes can work simultaneously and independently in a parallel run. To coordinate this effort, a single “Simulation Master” (SimM) asynchronously assigns batches of trajectories to the SimW processes. As each SimW finishes their assigned batch, they send the results of their PDis to the SimM, which are collected in an array of “Final Distributions” (FDis), and wait to receive further instructions by posting a blocking request for a new batch of trajectories to perform. Once a total number of trajectories specified by the user, and only known by the SimM process, have been assigned, the SimM signals any SimW process waiting for work to exit the program. After all SimW processes have exited the program, the SimM outputs the data in the FDis objects are requested by the user, and the simulation concludes. The communication is performed using the `MPI_Send`, `MPI_Recv`, and `MPI_Irecv` functions.

In certain applications, the fields are previously computed numerically through a magnetohydrodynamic (MHD) solver, and is therefore only prescribed directly at discrete locations within the domain, and the field at every point in between these locations must be interpolated. The datasets that define these fields can be very large, and thus it is unfeasable for every SimW process to hold a copy of the data in its local memory. When this is the case, “Simulation Server” (SimS) processes are in charge of managing the background data and providing it to the SimW processes in an efficient manner. If there background data is shared in memory, the SimS processes simply create the necessary arrays and shared them with all the SimW workers. If the background data is to remain distributed, the SimS processes hold copies of the entire background domain and feed the SimW workers with local background data necessary for trajectory tracing.

→ Subsection 4.1

The SimW, SimS, and SimM proper class names in the code, in order of inheritance (i.e. from parent to child), are `SimulationWorker`, `SimulationBoss`, and `SimulationMaster` (for more information, see `$SPECTRUM/src/simulation.hh`). So as to achieve a respectable scaling within a high performance computing (HPC) cluster, there can be multiple SimS processes in each physical node. Note that when background servicing is not needed, the SimS processes act as regular SimW processes. Furthermore, in a serial run, there is only one process, technically defined as SimM process, and it performs all the work (trajectory integration, binning, output, etc). Currently, serial runs can only function with backgrounds that do not require servicing, and if background servicing is required, a minimum of 3 processes (1 SimW, 1 SimM, and 1 SimS) are required for execution, meaning that in this case the SimM process cannot double as a SimW process.

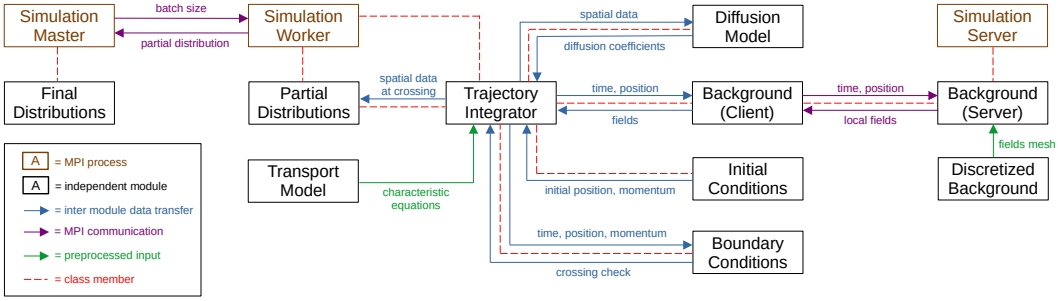

Figure 1: Schematic of SPECTRUM. See text for details.

Figure 1 depicts the sofware architecture of SPECTRUM described in the text. The black rectangles denote independent modules that work together to create a simulation. Each application-specific module is programmed as part of a hierarchical class structure and is derived from an abstract class. For example, `TrajectoryLorentz` derives from `TrajectoryBase`, which itself derived from the `Params` class. This practice prioritizes compartmentalization, reduces code complexity, and benefits future development. Furthermore, the relative independence of these components means that users can easily swap modules and set up entirely different simulations with a few simple changes to their simulation file.

→ Subsection 1.4

The green labels represent pre-processed inputs that will ensure the code is compiled to optimize execution for a particular application. Some modules are linked by composition through pointers (red-dashed lines), which makes run-time data transfer between them (blue arrows and labels) extremely efficient. The yellow-brown rectangles show the MPI roles involved in the parallelization scheme, as described in this subsection. MPI communication (purple arrows) occurs directly between the `SimulationMaster` and the `SimulationWorker` classes, but the data transfer between the `SimulationBoss` and `SimulationWorker` classes is handled by their attributes.

1.4 Compilation Procedure

In this subsection we explain the general approach to compile and run a user made simulation file. For an example on how to compile an already prepared simulation file, skip to Subsection 1.5. Writing simulation files for specific applications is the subject of Section 2. The following packages are **required** for successful execution and compilation:

- the GNU Automake tool,
- a C++ compiler, specifically the `c++17` version of `g++`,
- an MPI library, specifically either Open MPI or MPICH, and
- the GNU Scientific Library (GSL) of mathematical tools.

The following packages are **optional**, but encouraged, as they enable the use of certain features:

- a Fortran compiler, e.g. `gfortran` (necessary for coupling with BATL grid formatted background fields),
- the LLNL Silo library (necessary for outputing silo files), and

- the Slurm workload manager (necessary for scheduling jobs on HPC systems).

! → Before attempting to compile a program, you must know the following with certainty:

- The location and name of the program you want to compile and execute (e.g. `$SPECTRUM/project1/main_file1.cc`).
- The dependencies of the simulation file, namely all of the source files within `$SPECTRUM/src` and `$SPECTRUM/common` (and possibly other locations) necessary to properly define and use the virtual objects created within the simulation.
- The configuration parameters for the desired simulation.

The list of accepted parameters for step 3 of the compilation process are:

`mpi` : Which MPI library to use for compilation and execution. This parameters is **always required**.
Possible values: `openmpi` and `mpich`.

`execution` : Whether to run the code in serial or parallel execution. This parameters is **always required**.
Possible values: `SERIAL` and `PARALLEL`.

`trajectory` : Which trajectory mode to use in the simulation. This determines the transport equation being solved via characteristics. This parameters is **always required**.
→ Section 3 Possible values: `FIELDLINE`, `LORENTZ`, `GUIDING`, `GUIDING_SCATT`, `GUIDING_DIFF`, `GUIDING_DIFF_SCATT`, `FOCUSED`, and `PARKER`.

`time_flow` : Which time flow direction to use in the simulation. This parameters is **always required**.
→ Subsection 1.1 Possible values: `FORWARD` and `BACKWARD`.

`rkmethod` : Which integration method to use for the deterministic portions of the trajectories. This parameters is **always required**. Note that only explicit methods are currently functional. See the file `$SPECTRUM/common/rk_config.hh` to ascertain which numbers correspond to which methods.
Possible values: a number between `0` (Euler) and `29` (Dormant-Prince).

`server` : Which background field server to use in the simulation. This parameters is **always required**. A value of `SELF` means the data is not provided by a server and is instead computed locally by each `SimulationWorker` process.
→ Subsection 4.1 Possible values: `SELF`, `CARTESIAN`, and `BATL`.

`server_interp_order` : The order of the interpolation method to use when interpolating background field data on a grid. This parameter is **only required when** `server` is not `SELF`. A value of `-1` means that the field is pre-interpolated by the `SimulationBoss` process rather than served in blocks for local interpolation by the `SimulationWorker` processes.
Possible values: `-1`, `0`, and `1`.

`server_num_gcs` : Number of ghost cells around each block of field data defined on a grid. This parameter is **only required when** `server` is not `SELF`.
Possible values: `0` and `1`.

If you want to compile and run the user made simulation file `main_file1.cc` located in `$SPECTRUM/project1`, follow these steps:

1. In a terminal window, navigate to the `$SPECTRUM` directory.
2. If compiling from scratch or after a `make distclean`, execute

```
autoreconf
automake --add-missing
```

! → The first time you execute `autoreconf`, the commands will exit with an error status complaining about missing files, but the subsequent `automake` command will install them. These commands generate a configuration file in the main directory and Makefiles in the all the pre-specified subfolders (by default `$SPECTRUM/tests` and `$SPECTRUM/runs`). This process is automated and has been successfully tested on multiple machines, running different Linux operating systems, but, of course, it could still fail depending on the user's settings. Please ensure you are running the most recent version of all the relevant packages and be prepared to troubleshoot any unforeseen issues. For example, the `$SPECTRUM/configure.ac` might need to be slightly adjusted or a directory might have to be appended to the search path of `autoreconf`.

3. To configure the code for a specific application, you must execute the `./configure` command. The general structure of this command is

```
./configure CXXFLAGS=$cxx_flags \
      --with-parameter1=value1 \
      --with-parameter2=value2 \
      {...}
      --with-parameterN=valueN
```

where the `{...}` stands for an arbitrary list of specified parameters.

The `cxx_flags` variable are flags automatically passed to `g++` when compiling the code. For example, one could specify that `cxx_flags="-Ofast"` to compile the code using the "fast" optimization option. The user can also insert `CFFLAGS=$cf_flags` to automatically pass the flags `$cf_flags` to the Fortran compiler when necessary.

4. Next, navigate to `$SPECTRUM/project1` and ensure that `Makefile.am` in that folder contains the following:

```
AM_CXXFLAGS = $(MPI_CFLAGS)
AM_LDFLAGS = $(MPI_LIBS)

bin_PROGRAMS = main_file1

SPBL_COMMON_DIR = ../common
SPBL_SOURCE_DIR = ../src

main_file1_SOURCES = main_file1.cc \
   $(SPBL_SOURCE_DIR)/src_dependecy1.hh \
   $(SPBL_SOURCE_DIR)/src_dependecy1.cc \
   $(SPBL_SOURCE_DIR)/src_dependecy2.hh \
   $(SPBL_SOURCE_DIR)/src_dependecy2.cc \
   {...}
   $(SPBL_COMMON_DIR)/common_dependency1.hh \
   $(SPBL_COMMON_DIR)/common_dependency1.cc \
   $(SPBL_COMMON_DIR)/common_dependency2.hh \
   $(SPBL_COMMON_DIR)/common_dependency2.cc \
   {...}

main_file1_LDADD = $(MPI_LIBS) $(GSL_LIBS)
```

In the above template, `{...}` stands for additional dependencies. The dependencies can be listed in any order, but note that the last entry in the list should not end with a backslash. Moreover, many simulation files can be

ready for compilation within this same structure, so long as they are separated by whitespace within the `bin_PROGRAMS` line (a backslash allows continuation on the next line), i.e.

```
bin_PROGRAMS = main_file1 main_file2 \
               main_file3 main_file4 \
               {...}
```

and their dependencies listed separately but in the same format, i.e.

```
main_file1_SOURCES = main_file1.cc \
   $(SPBL_SOURCE_DIR)/src_dependecy11.hh \
   $(SPBL_SOURCE_DIR)/src_dependecy11.cc \
   $(SPBL_SOURCE_DIR)/src_dependecy12.hh \
   $(SPBL_SOURCE_DIR)/src_dependecy12.cc \
   {...}
   $(SPBL_COMMON_DIR)/common_dependency11.hh \
   $(SPBL_COMMON_DIR)/common_dependency11.cc \
   $(SPBL_COMMON_DIR)/common_dependency12.hh \
   $(SPBL_COMMON_DIR)/common_dependency12.cc \
   {...}

main_file1_LDADD = $(MPI_LIBS) $(GSL_LIBS)

main_file2_SOURCES = main_file2.cc \
   $(SPBL_SOURCE_DIR)/src_dependecy21.hh \
   $(SPBL_SOURCE_DIR)/src_dependecy21.cc \
   $(SPBL_SOURCE_DIR)/src_dependecy22.hh \
   $(SPBL_SOURCE_DIR)/src_dependecy22.cc \
   {...}
   $(SPBL_COMMON_DIR)/common_dependency21.hh \
   $(SPBL_COMMON_DIR)/common_dependency21.cc \
   $(SPBL_COMMON_DIR)/common_dependency22.hh \
   $(SPBL_COMMON_DIR)/common_dependency22.cc \
   {...}

main_file2_LDADD = $(MPI_LIBS) $(GSL_LIBS)
```

See `$SPECTRUM/benchmarks/Makefile.am` for an example.

→ Subsection **??** These templates are not adequate for compiling a code that uses a BATL grid formatted background.

5. To compile the code, enter

   ```
   make main_file1
   ```

   in the terminal.

6. Once the code is compiled, to run it in serial mode, type

   ```
   ./main_file1 $in1 $in2 {...}
   ```

   where `$in1`, `$in2`, etc are the inputs expected by the program and `{...}` stands for an arbitrary list of remaining inputs. To run the code in parallel, type

   ```
   mpirun -np $num_proc main_file1 $in1 $in2 {...}
   ```

   where `num_proc` is the number of processors that you want to use (at least 2).

! → Note that in certain operating systems you may need to "load" the MPI module

```
module load mpi
```

before attempting to run or even configure/compile the code.

1.5 Example Compilation

As an illustrative example, we will now provide precise commands to compile and run `$SPECTRUM/runs/gcr_modulation_example.cc` following the instructions in this subsection. This program performs a simple galactic cosmic ray (GCR) modulation example simulation. For context, this simulation solves the Parker Transport Equation, by means of stochastic characteristics (i.e. backward Parker trajectories), in a simplified Parker spiral solar wind background. The specific diffusion model, initial/boundary conditions, and binning distribution used will be explored as an example in Subsection 2.1.

→ Subsection 3.6

→ Subsection 4.5

1. Assuming the environment variable `SPECTRUM` is defined, type.

   ```
   cd $SPECTRUM
   ```

2. Type

   ```
   autoreconf
   automake --add-missing
   ```

3. Type

   ```
   ./configure CXXFLAGS="-Ofast" \
      --with-mpi=openmpi \
      --with-execution=PARALLEL \
      --with-trajectory=PARKER \
      --with-time_flow=BACKWARD \
      --with-rkmethod=29 \
      --with-server=SELF
   ```

   Note that this means you are compiling the code with the "fast" optimization option, you are using the OpenMPI library, you intend to run the code in parallel, you want to use the Parker transport model, this is a backward-in-time simulation, the deterministic portions of the trajectories will be integrated using the Dormant-Prince algorithm, and the background field will be computed independently by each `SimulationWorker` process.

4. Type

   ```
   cd runs
   ```

   In this prepared example, `Makefile.am` already contains the necessary depencies. Please review this file to familiarize yourself with the format.

5. Type

   ```
   make gcr_modulation_example
   ```

6. Assuming you have at least 6 cores available in your computer,

   ```
   mpirun -np 6 gcr_modulation_example 100000 1000
   ```

   If you don't have that many cores, just change the number `6` to your available number of cores, which must be at least 2. Once the code executes, the following files should appear in the same directory:

   - `gcr_modulation_example_distro_0.out`
     This file contains the raw distribution data (and metadata) in binary format.

- `modulated_gcr_spectrum.dat`
  This file contains the same distribution data in ascii format.

To post-process this data into a visualization software ready format, type

```
make postprocess_gcr_modulation_results
./postprocess_gcr_modulation_results
```

The file `modulated_gcr_spectrum_pp.dat` should appear in the same directory. When visualized, the results should look something like those in Figure 2. Note that your modulated spectra will appear slightly different than the one shown, due to the randomness of the stochastic trajectories.

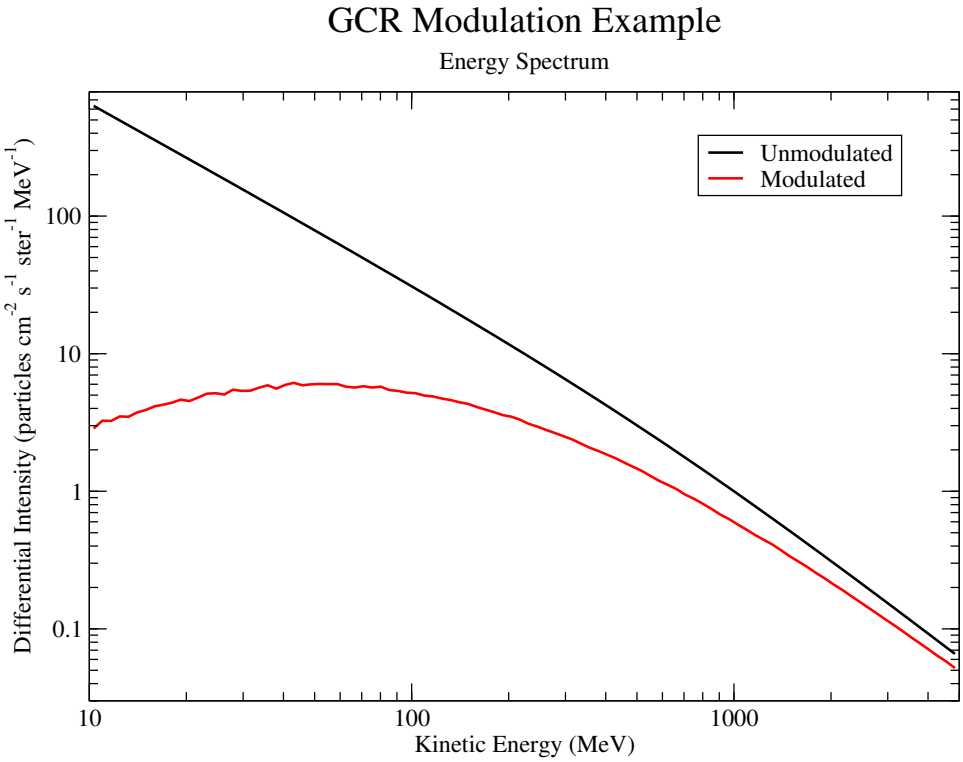

Figure 2: Results from a simple GCR modulation simulation.

1.6 Current Capabilities

SPECTRUM has a flexible and modularized code infrastructure which enables users to easily setup and run test-particle simulations. A number of physical models have already been incorporated into the code structure. Users might find Section **??** relevant if the latest official release of SPECTRUM does not possess the desired physics. Compiling an exhaustive list of SPECTRUM's current, officially supported, capabilities would be an onerous and futile task, given that the software is being actively developed. If desired, the user can automatically generate a complete list of classes, methods, and attributes with Doxygen. However, for the purpose of offering some help to neophytes, we list below some of the more utilized features and encourage users to delve into the source code and read the code comments for more information.

Specie : Alpha particle, electron, proton.
See `common/physics.hh` file.

Transport models : Focused, guiding center (with or without diffusion), Lorentz, Parker.
See `src/trajectory_*.hh` files.

Analytic backgrounds : Magnetic dipole, MHD flow around a Rankine half-body, MHD shock (with various degrees of smoothness), Parker Spiral (solar wind), superposition of waves, uniform (i.e. constant).
See `src/background_*.hh` files.

Discretized backgrounds : Block Adaptive Tree Library (BATL) grid format from SWMF, uniform Cartesian grid format.
See `src/background_*.hh` files.

Pitch-angle diffusion models : Constant (isotropic), quasi-linear, weakly nonlinear.
See `src/diffusion_other.hh` file.

Spatial diffusion models : Constant (parallel and/or perpendicular), power-law of various quantities, such as kinetic energy, magnetic field, or radial distance from the Sun.
See `src/diffusion_other.hh` file.

Binning distributions : First and second order position cumulatives (temporal), simple or bent power-law spectrum (energy), Uniform (temporal, spatial, or pitch-angle).
See `src/distribution_other.hh` file.

Initial spatial distributions : List of positions, uniform 0D (single position), uniform 1D (line or circle), uniform 2D (sphere, cylinder, or Rankine half-body), and uniform 3D (cube).
See `src/initial_space.hh` file.

Initial momentum distributions : Bi-Maxwellian, list of velocities, uniform 0D (single velocity, beam), uniform 1D (ring), uniform 2D (shell), uniform 3D (thick shell). See `src/initial_momentum.hh` file.

Temporal boundary conditions : Fixed (passing or expiring), and recurrent.
See `src/boundary_time.hh` file.

Spatial boundary conditions : Cylinder (absorbing), plane (passing, reflecting, or absorbing), Ranking half-body (absorbing), grid-defined region (absorbing), sphere (reflecting or absorbing),
See `src/boundary_space.hh` file.

Momentum boundary conditions : Magnetic mirroring.
See `src/boundary_momentum.hh` file.

Deterministic integration methods : Dormand-Prince (5th order, adaptive), Euler (1st order), midpoint (2nd order), Runge-Kutta (3rd and 4th order).[1]
See `common/rk_config.hh` file.

Stochastic integration methods : Euler-Maruyama (1st order), Milstein (1st order), Runge-Kutta (2nd order).[2]
See `common/rk_config.hh` file.

[1] All explicit.
[2] The order stated is in the weak sense.

2 Writing Simulation Files

In this section, we describe how to adequately write a simulation file using the existing SPECTRUM architecture. As mentioned in Subsection 1.3, any of the SPECTRUM modules can be independently set up and operated for any computational purpose. Nonetheless, the primary purpose of SPECTRUM is efficiently performing MC test-particle simulations, which requires the coordination of multiple modules. Therefore, the focus of this section is explaining how to properly write and run "full" simulations. We also describe how to write programs to visualize the output or functioning of other key modules, such as the Trajectory Integrator or Background objects.

2.1 Full Simulation Files

Instructions on how to compile and execute a correctly written simulation file were provided in Subsection 1.4. Here we expound on how to actually draft a main simulation program. As an outline, a typical simulation program contains the following:

1. several `#include` statements for dependencies or general C++ libraries,
2. creation of a Simulation object,
3. assignment of specie to simulate,
4. construction of the principal simulation modules, in no particular order,
   - Background object,
   - three Initial Condition objects (time, space, and momentum),
   - as many Boundary Condition objects as desired, with at least one terminal/absorbing,[3]
   - a Diffusion object, if necessary, and
   - at least one Distribution object,[4]
5. assignment of total number of trajectories to simulate,
6. command to run simulation, and
7. output of simulation results.

Along with general instructions, we will use the GCR modulation example in Subsection 1.5 to illustrate. Note that the specific code presented here might vary slightly from the most recent version in the official repository, but the general structure will be the same.

2.1.1 Including Dependencies

The following header files should always be included:

- `$SPECTRUM/src/simulation.hh`
- `$SPECTRUM/src/distribution_other.hh`
- `$SPECTRUM/src/background_*.hh`
- `$SPECTRUM/src/initial_time.hh`
- `$SPECTRUM/src/initial_space.hh`
- `$SPECTRUM/src/initial_momentum.hh`

[3] Otherwise trajectories will never end.
[4] Otherwise the results will not be recorded.

→ Subsection 1.3

The first contains the class declarations for the SimW, SimS, and SimM objects. The second contains all the derived distribution class declarations. The third contains the background class declaration pertinent to the simulation. Note that * denotes missing text that depends on which background is employed in the simulation. The fourth, fifth, and sixth contain the derived temporal, spatial, and momentum initial condition class declarations. In addition, the following header files might need to be included if their contents are used in the simulation:

- `$SPECTRUM/src/diffusion_other.hh`
- `$SPECTRUM/src/boundary_time.hh`
- `$SPECTRUM/src/boundary_space.hh`
- `$SPECTRUM/src/boundary_momentum.hh`

The first contains all the derived diffusion class declarations. The second, third, and forth contain the derived temporal, spatial, and momentum boundary condition class declarations.

In our example program, this portion is

```
#include "src/simulation.hh"
#include "src/distribution_other.hh"
#include "src/background_solarwind.hh"
#include "src/diffusion_other.hh"
#include "src/boundary_time.hh"
#include "src/boundary_space.hh"
#include "src/initial_time.hh"
#include "src/initial_space.hh"
#include "src/initial_momentum.hh"
#include <iostream>
#include <iomanip>
```

→ Subsection 4.5

Other than what has already been explained, the Parker Spiral background is being used here as indicated by the `_solarwind`, and the `iostream` and `iomanip` standard libraries are imported for miscellaneous purposes.

One more item to remark is the use of the `Spectrum` namespace via

```
using namespace Spectrum;
```

This is optional but will save the programmer more than few keystrokes since `Spectrum::` will not preceed every SPECTRUM function or global variable.

2.1.2 Creating a Simulation Object

A pointer to a Simulation object must be explicitly declared and created near the beginning of the program. In our example program, this is done by

```
  std::unique_ptr<SimulationWorker> simulation;
  simulation = CreateSimulation(argc, argv);
```

Here, `argc` and `argv` are the argument (integer) count and string array passed to the program through the main function

```
int main(int argc, char** argv)
```

The pointer is declared as a `SimulationWorker*` type, but C++ allows base class pointers to reference derived classes (polymorphism), and `SimulationMaster` is a child of `SimulationBoss`, which is itself a child of `SimulationWorker`. The

→ Subsection 9.1

`CreateSimulation` function uses the `MPI_Config` structure to internally as-

sign appropriate roles to each process based on the parameters specified in the `$SPECTRUM/configure.h` file and the way in which the code is executed (i.e. with or without `mpirun`).

2.1.3 Assigning a Specie

A specie should be formally assigned within the simulation object to ensure the calculations use the correct values for mass and charge. Species are indexed as non-negative integers in `$SPECTRUM/common/physics.hh`, conveniently defined as an `enum` structure for the user's benefit. In our example code, this is done by

```
int specie = Specie::proton;
simulation->SetSpecie(specie);
```

! → Note that many of the SPECTRUM functions default to using protons when a specie is unspecified, but this treatment is not guaranteed across all modules, so an explicit specie index assignment is strongly encouraged.

2.1.4 Constructing Simulation Modules

Constructing simulation modules is done in the following steps:

→ Subsection 9.2

1. If not already existing, define a DataContainer object.
2. Empty (`Clear`) the DataContainer object.
3. Load (`Insert`) the parameters relevant for the module construction to the DataContainer object.
4. Construct (`AddModuleName`) the desired module with the loaded DataContainer object and attach it to the Simulation object.

! → The parameters must be loaded in the exact order in which the `SetupBackground` function expects to receive (`Read`) them.

In our example code, the DataContainer object was already declared at the beginning of the program

```
DataContainer container;
```

The Background object construction is done following the above template by

```
container.Clear();

// Initial time
double t0 = 0.0;
container.Insert(t0);

// Origin
container.Insert(gv_zeros);

// Velocity
double umag = 4.0e7 / unit_velocity_fluid;
GeoVector u0(umag, 0.0, 0.0);
container.Insert(u0);

// Magnetic field
double RS = 6.957e10 / unit_length_fluid;
double r_ref = 3.0 * RS;
double BmagE = 5.0e-5 / unit_magnetic_fluid;
double Bmag_ref = BmagE * Sqr((GSL_CONST_CGSM_ASTRONOMICAL_UNIT /
    unit_length_fluid) / r_ref);
```

```
   GeoVector B0(Bmag_ref, 0.0, 0.0);
   container.Insert(B0);

// Effective "mesh" resolution
   double dmax = GSL_CONST_CGSM_ASTRONOMICAL_UNIT / unit_length_fluid;
   container.Insert(dmax);

// solar rotation vector
   double w0 = twopi / (25.0 * 24.0 * 3600.0) / unit_frequency_fluid;
   GeoVector Omega(0.0, 0.0, w0);
   container.Insert(Omega);

// Reference equatorial distance
   container.Insert(r_ref);

// dmax fraction for distances closer to the Sun
   double dmax_fraction = 0.1;
   container.Insert(dmax_fraction);

   simulation->AddBackground(BackgroundSolarWind(), container);
```

This defines a Parker Spiral background centered at the origin, with a radial solar wind speed of 400 km s^{-1}, a magnetic field with an equatorial strength of 5 nT at 1 au from the Sun (origin), and a solar rotation frequency of 25 days. See Subsection 4.5 for the meaning of the other parameters.

The phase-space Initial Condition objects for the pseudo-trajectories are built by

```
   container.Clear();

// Initial time
   double init_t = 0.0;
   container.Insert(init_t);

   simulation->AddInitial(InitialTimeFixed(), container);

   container.Clear();

// Initial position
   GeoVector init_pos(1.0 * GSL_CONST_CGSM_ASTRONOMICAL_UNIT /
       unit_length_fluid, 0.0, 0.0);
   container.Insert(init_pos);

   simulation->AddInitial(InitialSpaceFixed(), container);

   container.Clear();

// Lower bound for momentum
   double momentum1 = Mom(10.0 * SPC_CONST_CGSM_MEGA_ELECTRON_VOLT /
       unit_energy_particle, specie);
   container.Insert(momentum1);

// Upper bound for momentum
   double momentum2 = Mom(5000.0 * SPC_CONST_CGSM_MEGA_ELECTRON_VOLT /
       unit_energy_particle, specie);
   container.Insert(momentum2);

// Log bias
   bool log_bias = true;
   container.Insert(log_bias);
```

```
simulation->AddInitial(InitialMomentumThickShell(), container);
```

The initial time is fixed at 0. The initial position is fixed at 1 au. The initial momentum distribution is a logarithmically biased "thick shell", which produces random, uniformly distributed pitch and gyrophase angles and a logarithmically distributed momentum mangitude between 10 MeV and 5 GeV.

Two spatial and one temporal Boundary Condition objects are constructed through

```
container.Clear();

// Max crossings
int max_crossings_Sun = 1;
container.Insert(max_crossings_Sun);

// Action
std::vector<int> actions_Sun;
actions_Sun.push_back(-1);
container.Insert(actions_Sun);

// Origin
container.Insert(gv_zeros);

// Radius
double inner_boundary = 0.05 * GSL_CONST_CGSM_ASTRONOMICAL_UNIT /
    unit_length_fluid;
container.Insert(inner_boundary);

simulation->AddBoundary(BoundarySphereAbsorb(), container);

container.Clear();

// Max crossings
int max_crossings_outer = 1;
container.Insert(max_crossings_outer);

// Action
std::vector<int> actions_outer;
actions_outer.push_back(0);
container.Insert(actions_outer);

// Origin
container.Insert(gv_zeros);

// Radius
double outer_boundary = 80.0 * GSL_CONST_CGSM_ASTRONOMICAL_UNIT /
    unit_length_fluid;
container.Insert(outer_boundary);

simulation->AddBoundary(BoundarySphereAbsorb(), container);

container.Clear();

// Not needed because this class sets the value to -1
int max_crossings_time = 1;
container.Insert(max_crossings_time);

// Action
std::vector<int> actions_time;
actions_time.push_back(-1);
container.Insert(actions_time);
```

```
// Max duration of the trajectory
   double maxtime = 60.0 * 60.0 * 24.0 * 365.0 / unit_time_fluid;
   container.Insert(maxtime);

   simulation->AddBoundary(BoundaryTimeExpire(), container);
```

This defines a non-binning, absorbing (terminal), inner spherical boundary at 0.05 au, a binning, absorbing (terminal), outer spherical boundary at 80 au, and non-binning, expiring (terminal) time boundary of approximately 1 year. No momentum boundary conditions are employed in this simulation. Since this simulation has a single distribution (described below), only the first element of each actions vector needs to be set.

A Diffusion object is added to the simulation via

```
   container.Clear();

// Parallel mean free path
   double lam0 = 0.1 * GSL_CONST_CGSM_ASTRONOMICAL_UNIT / unit_length_fluid;
   container.Insert(lam0);

// Rigidity normalization factor
   double R0 = 1.0e9 / unit_rigidity_particle;
   container.Insert(R0);

// Magnetic field normalization factor
   container.Insert(BmagE);

// Power law slope for rigidity
   double pow_law_R = 0.5;
   container.Insert(pow_law_R);

// Power law slope for magnetic field
   double pow_law_B = -1.0;
   container.Insert(pow_law_B);

// Ratio of kappa_perp to kappa_para
   double kap_rat = 0.05;
   container.Insert(kap_rat);

// Pass ownership of "diffusion" to simulation
   simulation->AddDiffusion(DiffusionRigidityMagneticFieldPowerLaw(),
       container);
```

This specifies anisotropic spatial diffusion coefficients that are power-laws of both magnetic field magnitude (with index -1) and particle rigidity (with index 0.5). The ratio of perpendicular to parallel (with respect to the local magnetic field) diffusion coefficients is 0.05.

One Distribution object is constructed using

```
   container.Clear();

// Number of bins
   MultiIndex n_bins1(100, 0, 0);
   container.Insert(n_bins1);

// Smallest value
   GeoVector minval1(EnrKin(momentum1), 0.0, 0.0);
   container.Insert(minval1);
```

```
// Largest value
   GeoVector maxval1(EnrKin(momentum2), 0.0, 0.0);
   container.Insert(maxval1);

// Linear or logarithmic bins
   MultiIndex log_bins1(1, 0, 0);
   container.Insert(log_bins1);

// Add outlying events to the end bins
   MultiIndex bin_outside1(0, 0, 0);
   container.Insert(bin_outside1);

// Physical units of the distro variable
   double unit_distro1 = 1.0 / (Sqr(unit_length_fluid) * unit_time_fluid *
       fourpi * unit_energy_particle);
   container.Insert(unit_distro1);

// Physical units of the bin variable
   GeoVector unit_val1 = {unit_energy_particle, 1.0, 1.0};
   container.Insert(unit_val1);

// Don't keep records
   bool keep_records1 = false;
   container.Insert(keep_records1);

// Normalization for the "hot" boundary
   double J0 = 1.0 / unit_distro1;
   container.Insert(J0);

// Characteristic energy
   double T0 = 1.0 * SPC_CONST_CGSM_GIGA_ELECTRON_VOLT /
       unit_energy_particle;
   container.Insert(T0);

// Spectral power law
   double pow_law_T = -1.8;
   container.Insert(pow_law_T);

// Constant value for the "cold" condition
   double val_cold1 = 0.0;
   container.Insert(val_cold1);

   simulation->AddDistribution(DistributionSpectrumKineticEnergyPowerLaw(),
       container);
```

This binning distribution is a power law in kinetic energy (with index -1.8). It is composed of 100 energy bins, logarithmically spaced between 10 MeV and 5 GeV. See Subsection 5.1 and 5.8 a description of the other parameters. This particular distribution can yield a power-law in kinetic energy of the differential density, the differential intensity, or the particle distribution function. Although this simulation will work for any variation, `DISTRO_KINETIC_ENERGY_POWER_LAW_TYPE` should be set to `0` in `$SPECTRUM/src/distribution_other.hh` for the post-processing program to yield accurate results.

! →

2.1.5 Assigning Number of Trajectories

The total number of trajectories to simulate, along with the number of trajectories per batch, should be communicated to the SimM, which will in turn coordinate the SimW processes. In our example code, this is done with

→ Subsection 1.3

```
int n_traj;
int batch_size;

batch_size = n_traj = 1;
if(argc > 1) n_traj = atoi(argv[1]);
if(argc > 2) batch_size = atoi(argv[2]);

simulation->SetTasks(n_traj, batch_size);
```

The user gives these numbers directly at runtime when executing the program, but they default to 1 if the user does not specify them. A third input to the `SetTasks` function controls the maximum allowed number of trajectories to be integrated by any SimW process. If this parameter is not provided, it is effectively infinity. Note that although every MPI process executes this command, only the SimM's version of the function actually records the numbers.

2.1.6 Running Simulation

After all of the setup commands described previously, it is finally time to formally run the simulation. As shown in the example program, this is achieved with the function

```
simulation->MainLoop();
```

Each kind of Simulation object has their own implementation of the `MainLoop` function, since they perform different tasks.

2.1.7 Outputting Simulation Results

Once the simulation finishes running, and also at certain checkpoints during execution, binary files of the cumulative distributions collected by the SimM will be saved to disk. By default, these will appear in the same directory as the simulation file and will be named `distribution_*.out`, where `*` is the index of each distribution, detemined by the order in which Distribution objects are constructed. This default name and location can be changed by the user with the `DistroFileName` function, as illustrated in the example code

```
std::string simulation_files_prefix = "gcr_modulation_example_distro_";
simulation->DistroFileName(simulation_files_prefix);
```

These binary files contain all of the data (and metadata) of the (up to 3-dimensional) distributions, which may be more information than desired. They can be read and manipulated using during post-processing by a separate script, but it is often easier to collapse 1 or 2 dimensions and output the resulting marginal distribution in ascii format. The example program, this functionality is utilized through the method

```
simulation->PrintDistro1D(0, 0, "modulated_gcr_spectrum.dat", true);
```

This statement should print a 1D marginal distribution of the first (and only) Distribution object collapsed to the first (and only) dimension, with the specified filename and it "physical" units. A similar function, `PrintDistro2D`, also exists and can be useful for higher dimensional distributions. Figure 2 shows the expected output from this simulation, after some post-processing.

→ Subsection 5.1

2.2 Trajectory Simulation Files

Sometimes it is useful to perform a simulation for a single trajectory, perhaps as part of testing or for pedagogical applications. Setting up a simulation for a single

particle is very simple, and in many ways similar to programming "full" simulations (i.e. with many particles), which we described in the previous subsection. As an outline, a typical trajectory simulation program contains the following:

→ Subsection 2.1

1. several `#include` statements for dependencies or general C++ libraries,
2. creation of a Trajectory object,
3. linkage of a random number generator,
4. assignment of specie to simulate,
5. construction of the principal simulation modules, in no particular order,
   - Background object,
   - three Initial Condition objects (time, space, and momentum),
   - as many Boundary Condition objects as desired, with at least one terminal/absorbing, and
   - a Diffusion object, if necessary.
6. command to integrate the trajectory, and
7. output of simulation results.

We will use `$SPECTRUM/benchmarks/main_test_dipole_periods.cc` as an example. This code simulates the periodic motion of a charged particle in a dipole field, meant to idealize Earth's magnetic field, due to gyration, adiabatic invariants, and magnetic field drifts. Note that the specific code presented here might vary slightly from the most recent version in the official repository, but the general structure will be the same.

2.2.1 Including Dependencies

The following header files should always be included:

- `$SPECTRUM/src/traj_config.hh`
- `$SPECTRUM/src/background_*.hh`
- `$SPECTRUM/src/initial_time.hh`
- `$SPECTRUM/src/initial_space.hh`
- `$SPECTRUM/src/initial_momentum.hh`

The first imports the appropriate trajectory header file based on the `TRAJ_TYPE` macro which was set during the configure stage. The second contains the background class declaration pertinent to the simulation. Note that `*` denotes missing text that depends on which background is employed in the simulation. The third, fourth, and fifth contain the derived temporal, spatial, and momentum initial condition class declarations. In addition, the following header files might need to be included if their contents are used in the simulation:

→ Subsection 1.4

- `$SPECTRUM/src/diffusion_other.hh`
- `$SPECTRUM/src/boundary_time.hh`
- `$SPECTRUM/src/boundary_space.hh`
- `$SPECTRUM/src/boundary_momentum.hh`

The first contains all the derived diffusion class declarations. The second, third, and forth contain the derived temporal, spatial, and momentum boundary condition class declarations.

In our example program, this portion is

```
#include "src/traj_config.hh"
#include "src/background_dipole.hh"
#include "src/initial_time.hh"
#include "src/initial_space.hh"
#include "src/initial_momentum.hh"
#include "src/boundary_time.hh"
#include "src/boundary_space.hh"
#include "src/boundary_momentum.hh"
#include <iostream>
#include <iomanip>
```

→ Subsection 4.4

Other than what has already been explained, the dipole background is being used here as indicated by the `_dipole`, and the `iostream` and `iomanip` standard libraries are imported for miscellaneous purposes.

As done in the example of the previous subsection, we employ the `Spectrum` namespace via

```
using namespace Spectrum;
```

for convenience.

2.2.2 Creating a Trajectory Object

A pointer to a Trajectory object must be explicitly declared and created near the beginning, but within the `main` function, of the program.

```
std::unique_ptr<TrajectoryBase> trajectory =
    std::make_unique<TrajectoryType>();
```

The `TrajectoryType` macro is automatically defined in the corresponding trajectory header file, included through `$SPECTRUM/src/traj_config.hh`.

2.2.3 Linking a Random Number Generator

Although not all trajectory modules will require it, a random number generator should be linked to the trajectory object for proper functioning. This is accomplished by the following simple statements

```
std::shared_ptr<RNG> rng = std::make_shared<RNG>(time(NULL));
trajectory->ConnectRNG(rng);
```

Note that this is handled automatically during a "full" simulation, hence this step was not described in Subsection 2.1.

2.2.4 Assigning a Specie

As previously mentioned, a specie should be formally assigned within the simulation object to ensure the calculations use the correct values for mass and charge. Again, species are indexed as non-negative integers in `$SPECTRUM/common/physics.hh`, conveniently defined as an `enum` structure for the user's benefit. In our example code, this is done by

```
int specie = Specie::proton;
trajectory->SetSpecie(specie);
```

2.2.5 Constructing Simulation Modules

The general module construction procedure for trajectory simulations is identical to the one described for "full" simulations, with the technical exception that modules are attached to the Trajectory object, since there is no Simulation object in such applications. A DataContainer object must be declared before constructing the first simulation module.

```
DataContainer container;
```

In our example code, the Background object construction is done as follows

```
   container.Clear();

// Initial time
   double t0 = 0.0;
   container.Insert(t0);

// Origin
   container.Insert(gv_zeros);

// Velocity
   container.Insert(gv_zeros);

// Magnetic field
   double Bmag = 0.311 / unit_magnetic_fluid;
   GeoVector B0(0.0, 0.0, Bmag);
   container.Insert(B0);

// Effective "mesh" resolution
   double RE = 6.37e8 / unit_length_fluid;
   double dmax_fraction = 0.1;
   double dmax = dmax_fraction * RE;
   container.Insert(dmax);

// Reference equatorial distance
   container.Insert(RE);

// dmax fraction for distances closer to the dipole
   container.Insert(dmax_fraction);

   trajectory->AddBackground(BackgroundDipole(), container);
```

This defines a dipole background centered at the origin, with a dipole moment along the **z**-axis and a strength of 0.311 G at a distance of 6.37×10^8 cm (Earth's radius) on the equator. See Subsection 4.4 for the meaning of the other parameters.

The phase-space Initial Condition objects for the trajectory are built by

```
   container.Clear();

// Initial time
   double init_t = 0.0;
   container.Insert(init_t);

   trajectory->AddInitial(InitialTimeFixed(), container);

   container.Clear();

   double L = 3.0;
   GeoVector start_pos(L*RE, 0.0, 0.0);
```

```
   container.Insert(start_pos);

   trajectory->AddInitial(InitialSpaceFixed(), container);

   container.Clear();

// Initial momentum
   double MeV_kinetic_energy = 1.0;
   container.Insert(Mom(MeV_kinetic_energy *
       SPC_CONST_CGSM_MEGA_ELECTRON_VOLT / unit_energy_particle, specie));

   double theta_eq = DegToRad(30.0);
   container.Insert(theta_eq);

   trajectory->AddInitial(InitialMomentumRing(), container);
```

The initial time is fixed at 0. The initial position is fixed is on the equator at a distance of 3 Earth radii, specifically along the **x**-axis. The initial momentum is drawn from a ring distribution, with magnitude 1 MeV and pitch-angle of 30°.

One temporal, three spatial, and one momentum Boundary Condition objects are constructed through

```
   container.Clear();

// Max crossings
   int max_crossings_time = 1;
   container.Insert(max_crossings_time);

// Action
   std::vector<int> actions; // empty vector because there are no
       distributions
   container.Insert(actions);

// Duration of the trajectory
   double drift_period = 3600.0 * 1.05 / MeV_kinetic_energy / L / (1.0 +
       0.43 * sin(theta_eq)) / unit_time_fluid;
   double bounce_period = 2.41 * L * (1.0 - 0.43 * sin(theta_eq)) /
       sqrt(MeV_kinetic_energy) / unit_time_fluid;
   double maxtime = 10.0 * drift_period;
   container.Insert(maxtime);

   trajectory->AddBoundary(BoundaryTimeExpire(), container);

   container.Clear();

// Max crossings
   int max_crossings_Earth = 1;
   container.Insert(max_crossings_Earth);

// Action
   container.Insert(actions);

// Origin
   container.Insert(gv_zeros);

// Radius
   container.Insert(RE);

   trajectory->AddBoundary(BoundarySphereAbsorb(), container);
```

```
   container.Clear();

// Max crossings
   int max_crossings = -1;
   container.Insert(max_crossings);

// Action
   container.Insert(actions);

// Origin
   container.Insert(gv_zeros);

// Normal
   GeoVector normal_drift(1.0,0.0,0.0);
   container.Insert(normal_drift);

   trajectory->AddBoundary(BoundaryPlanePass(), container);

   container.Clear();

// Max crossings
   container.Insert(max_crossings);

// Action
   container.Insert(actions);

// Origin
   container.Insert(gv_zeros);

// Normal
   GeoVector normal_bounce(0.0,0.0,1.0);
   container.Insert(normal_bounce);

   trajectory->AddBoundary(BoundaryPlanePass(), container);

   container.Clear();

// Max crossings
   container.Insert(max_crossings);

// Action
   container.Insert(actions);

   trajectory->AddBoundary(BoundaryMirror(), container);
```

This defines an expiring (terminal) time boundary of 10 expected drift periods. In addition, an absorbing (terminal), inner spherical boundary representing Earth's surface and two passive planar boundaries (equatorial and meridional) are defined to estimate the bounce and orbital drift periods. Lastly, a passive momentum boundary condition to count the number of mirrorings is also implemented. Since this simulation does not have any Distribution Objects (described below), the actions vectors do not need to be prefilled with specific values.

2.2.6 Integrating Trajectory

To simulate the desired trajectory, it must first be explicitly initialized with the command

```
   trajectory->SetStart();
```

This member function will use the intial condition modules previously loaded to draw the starting time, position, and momentum for the particle. To integrate the trajectory until a terminal boundary is encountered, simply call the function

```
trajectory->Integrate();
```

2.2.7 Outputting Simulation Results

Once the trajectory finishes integrating, the user can opt to output information about it. For example, the following command will print a simple status interpreting the event that terminated the trajectory.

```
trajectory->InterpretStatus();
```

For a detailed analysis, more information can be written to the terminal or separate files.

```
std::string trajectory_file = "output_data/main_test_dipole_drifts_" +
    trajectory->GetName() + ".lines";
std::cout << std::endl;
std::cout << "DIPOLE FIELD DRIFT PERIODS" << std::endl;
std::cout << "======================================================="
    << std::endl;
std::cout << "Trajectory type: " << trajectory->GetName() << std::endl;
std::cout << "Time elapsed (simulated) = " << trajectory->ElapsedTime()
    * unit_time_fluid << " s" << std::endl;
std::cout << "drift period (theory)    = " << drift_period *
    unit_time_fluid << " s" << std::endl;
std::cout << "drift period (simulation) = " << 2.0 *
    trajectory->ElapsedTime() * unit_time_fluid /
    trajectory->Crossings(1,1) << " s" << std::endl;
std::cout << "bounce period (theory)   = " << bounce_period *
    unit_time_fluid << " s" << std::endl;
std::cout << "bounce period (simulation 1) = " << 2.0 *
    trajectory->ElapsedTime() * unit_time_fluid /
    trajectory->Crossings(1,2) << " s" << std::endl;
std::cout << "bounce period (simulation 2) = " << 2.0 *
    trajectory->ElapsedTime() * unit_time_fluid /
    trajectory->Mirrorings() << " s" << std::endl;
std::cout << "======================================================="
    << std::endl;
std::cout << "Trajectory outputed to " << trajectory_file << std::endl;
std::cout << std::endl;

trajectory->PrintCSV(trajectory_file,false);
```

The function `ElapsedTime()` will output the total simulated time of the trajectory, from start to encountering a terminal boundary, while `Crossings(output, bnd)` will output the number of crossings for the temporal (`output = 0`), spatial (`output = 1`), or momentum boundary (`output = 2`) with index `bnd`. The function `Mirrorings()` outputs the number of mirroring events specifically. Finally, the `PrintCSV(trajectory_file, false)` will output the trajectory points to a file (`trajectory_file`) as comma separated values in either physical (`true`) or code (`false`) units.

→ Subsection 3.4

Figure 3 shows the resulting trajectory for approximately one full orbital period when using the guiding center module (blue). This transport module does not resolve the gyration of the particles around the magnetic field lines, but it does capture the mirroring/bouncing near the poles as well as the orbital drift motion (about the dipole moment) as a result of the gradient and curvature of the magnetic

field. For comparison, Figure 4 shows the same graph focused near the first few mirroring locations and accompanied by the result of the same simulation using the
→ Subsection 3.3
Lorentz trajectory module (red), which does follow the full gyrating motion of the particle. We emphasize that the same code was used to obtain the results from both transport models, with only minor edits to the initial conditions in order to make both trajectories almost equivalent, which highlights the excellent modularity within the SPECTRUM framework.

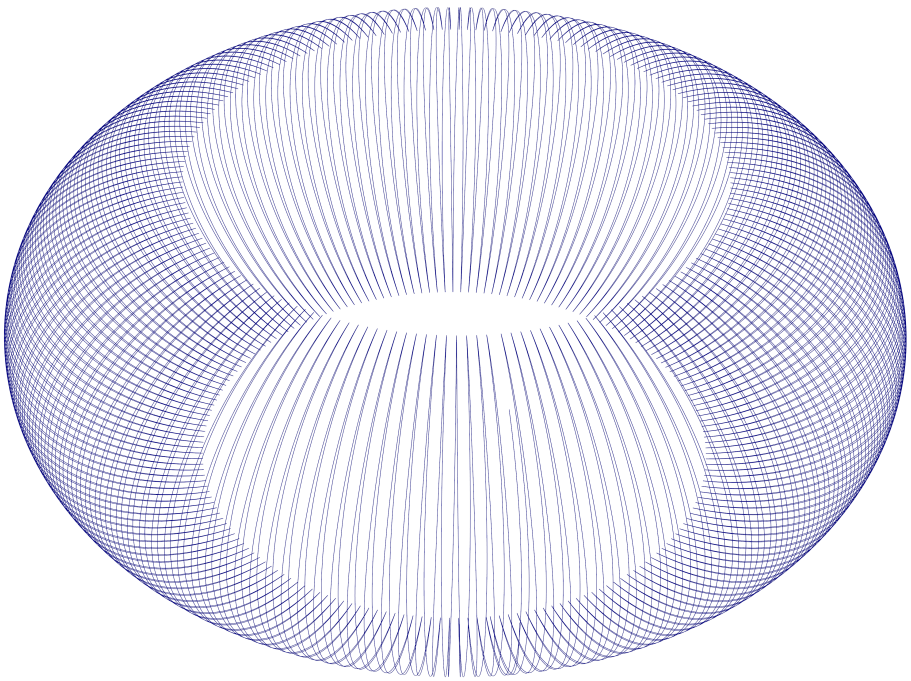

Figure 3: Guiding center trajectory (blue) of particle trapped in a magnetic dipole during approximately one full orbital drift period.

2.3 Background Simulation Files

In some cases, you might want to visualize the background utilized by SPECTRUM during a simulation. This can be accomplished by writing a simple background simulations, which contains the following steps:

1. several `#include` statements for dependencies or general C++ libraries,
2. creation of a Background object,
3. creation of a SpatialData object,
4. definition of the mask attribute,
5. construction of the Background module, and
6. evaluation/output of background at desired locations.

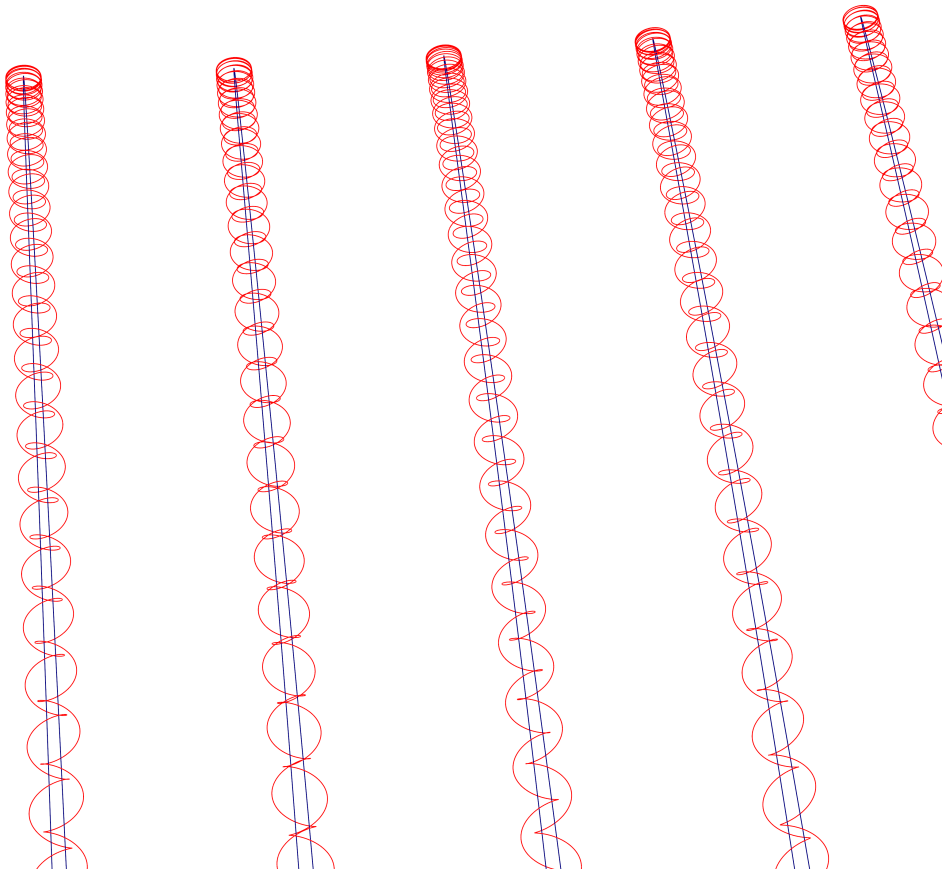

Figure 4: Same as Figure 3 but zoomed in near the first few northern mirroring locations. The red trajectory, which actually resolves the gyration, was computed using the Lorentz transport module for comparison.

We will use `$SPECTRUM/benchmarks/main_test_dipole_visualization.cc` as an example. This code generates the components of a dipole magnetic field, meant to idealize Earth's, on a cross-section containing its symmetry axis. Note that the specific code presented here might vary slightly from the most recent version in the official repository, but the general structure will be the same.

2.3.1 Including Dependencies

Only one header file is required:

- `$SPECTRUM/src/background_*.hh`

It contains the background class declaration pertinent to the simulation. Note that `*` denotes missing text that depends on which background is employed in the simulation. In our example program, this portion is

```
#include "src/background_dipole.hh"
#include <fstream>
#include <iostream>
#include <iomanip>
```

→ Subsection 4.4

The dipole background is being used here as indicated by the `_dipole`, and the `fstream`, `iostream` and `iomanip` standard libraries are imported for miscellaneous purposes.

As done in the previous examples, we employ the `Spectrum` namespace via

```
using namespace Spectrum;
```

for convenience.

2.3.2 Creating a Background Object

A Background object must be explicitly declared near the beginning, but within the `main` function, of the program.

```
  BackgroundDipole background;
```

The specific type of background must match the one defined in the header file included previously.

2.3.3 Creating a SpatialData Object

A SpatialData object must also be explicitly declared near the beginning, but within the `main` function, of the program.

```
  SpatialData spdata;
  double t = 0.0;
  int i,j,k;
  GeoVector pos, mom = gv_ones;
```

This object, along with the other variables declared in the portion of code, will be used to interface with the Background object.

2.3.4 Defining the Mask Attribute

The `_mask` attribute of the SpatialData object must should be set prior to any attempts to evaluate the field.

```
  spdata._mask = BACKGROUND_ALL;
```

This particular choice instructs the SpatialData object to collect the information for all background fields (flow velocity, magnetic field, and electric field), but none of its time or space derivatives. See Subsection 9.8 for details.

2.3.5 Constructing Background Module

Construcing a background module has been featured in previous examples, and it works the same way here, but there is one crucial difference in the very last step. Instead of "adding" or attaching the module to a larger class (e.g. a Trajectory or Simulation object), this module exists on its own, and must be "setup" manually. In practice, this is only a technicality, and doing this is no more complicated than using a different function called `SetupObject` instead of `AddBackground`.

→ Subsection 4.2

A DataContainer object must be declared before constructing the first simulation module.

```
DataContainer container;
```

In our example code, the Background object construction is done as follows

```
// Initial time
   double t0 = 0.0;
   container.Insert(t0);

// Origin
   container.Insert(gv_zeros);

// Velocity
   container.Insert(gv_zeros);

// Magnetic field
   double Bmag = 0.311 / unit_magnetic_fluid;
   GeoVector B0(0.0, 0.0, Bmag);
   container.Insert(B0);

// Largest absolute step size
   double RE = 6.37e8 / unit_length_fluid;
   double dmax_fraction = 0.1;
   double dmax0 = dmax_fraction * RE;
   container.Insert(dmax0);

// Reference distance
   container.Insert(RE);

// Relative step size
   container.Insert(dmax_fraction);

   background.SetupObject(container);
```

This defines a dipole background centered at the origin, with a dipole moment along the **z**-axis and a strength of 0.311 G at a distance of 6.37×10^8 cm (Earth's radius) on the equator. See Subsection 4.4 for the meaning of the other parameters.

2.3.6 Evaluating/Outputting the Background

Evaluating the background and writing the information to a file can be performed in countless ways, though it is often done in a loop. In our example, we use the following code

```
   std::ofstream outfile_x;
   std::ofstream outfile_y;
```

```
   std::ofstream outfile_z;
   std::string background_file =
       "output_data/main_test_dipole_visualization_B";
   outfile_x.open(background_file + "x.dat");
   outfile_y.open(background_file + "y.dat");
   outfile_z.open(background_file + "z.dat");
// Output field
   int N = 1000;
   int M = 100;
   double corner = 3.0 * RE;
   double dx = 2.0 * corner / (double)(N - 1);
   double dz = 2.0 * corner / (double)(N - 1);
   pos[1] = 0.0;
   outfile_x << std::setprecision(8);
   outfile_y << std::setprecision(8);
   outfile_z << std::setprecision(8);
   for(i = 0; i < N; i++) {
      pos[0] = -corner + i * dx;
      for(j = 0; j < N; j++) {
         pos[2] = -corner + j * dz;
         background.GetFields(t, pos, mom, spdata);
         outfile_x << std::setw(16) << spdata.Bvec[0] * unit_magnetic_fluid;
         outfile_y << std::setw(16) << spdata.Bvec[1] * unit_magnetic_fluid;
         outfile_z << std::setw(16) << spdata.Bvec[2] * unit_magnetic_fluid;
      };
      outfile_x << std::endl;
      outfile_y << std::endl;
      outfile_z << std::endl;
      if(i % M == 0) std::cerr << "i = " << i << std::endl;
   };
   std::cerr << "i = " << i << std::endl;
   outfile_x.close();
   outfile_y.close();
   outfile_z.close();

   std::cout << std::endl;
   std::cout << "DIPOLE FIELD VISUALIZATION (y=0)" << std::endl;
   std::cout << "=========================================================="
       << std::endl;
   std::cout << "Dipole axis: " << UnitVec(B0) << std::endl;
   std::cout << "Resolution: " << M << " x " << M << std::endl;
   std::cout << "Domain size (km): " << 2.0*corner*unit_length_fluid/1.0e5
       << " x " << 2.0*corner*unit_length_fluid/1.0e5 << std::endl;
   std::cout << "Domain center: " << gv_zeros << std::endl;
   std::cout << "=========================================================="
       << std::endl;
   std::cout << "Background outputed to " << background_file << "*.dat" <<
       std::endl;
   std::cout << std::endl;
```

Files are opened prior to the nested loops, and the components of the magnetic field are written to the appropriate files throughout the iterations. Note that we are "manually" sampling the field in a grid through the `GetFields` function, but if the Silo library is installed, a variety of auxiliary functions are included through `$SPECTRUM/src/background_base_visual.cc` so the user can output 2D slices (in space) or 3D meshes of the background at particular times with just a few lines of codes. Figure 5 shows the absolute value of the x and z components of the magnetic field outputed from this code (the y component is zero).

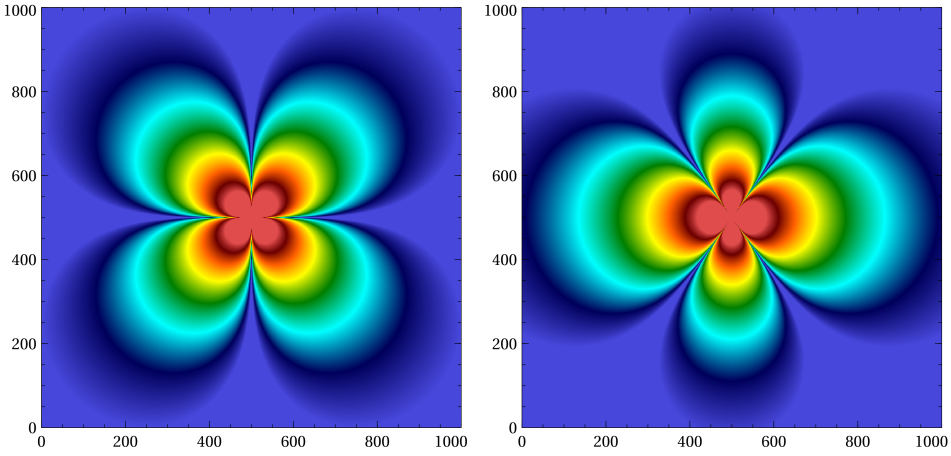

Figure 5: $|B_x|$ and $|B_z|$ in the plane $y = 0$ for the dipole described in the text. The colormaps are in logarithmic scale between 0.01 and 5 G. The axes numbers represent the index of the for loop variables corresponding to each evaluation.

3 Trajectory Modules

In this section, we describe the trajectory (transport) modules currently supported within the existing SPECTRUM architecture.

3.1 Base Trajectory

The `TrajectoryBase` class is not meant to be used directly, but rather contains and defines the general functionality shared by all the application specific trajectory modules, described in the following subsections. It is inherited from the `Params` class. In general, trajectory computation occurs in 3 steps, explained below.

→ Subsection 9.7

1. Initialize trajectory: Various trajectory integration parameters are reset, the position and momentum arrays are emptied, the status flags are lowered, and new initial conditions are drawn from the Initial Condition objects. These tasks all happen inside the public function

```
void SetStart(void);
```

2. Integrate trajectory: The trajectory is integrated, collecting all the pertinent data along the way, until a terminal, absorbing, or injecting boundary is reached. The trajectory integration procedure is encapsulated in the public function

```
void Integrate(void);
```

This class method successively calls the private function

```
void Advance(void);
```

at each step of the trajectory, which itself calls other functions depending on the type of transport simulated. The archetype for the stepping function is the private method

```
void RKAdvance(void);
```

which is in fact used directly for many transport types. In this function, trajectories are moved by one step using the Runge-Kutta (multi-stage) approach specified by the user during compilation. The core methods utilized in this process are

→ Subsection 1.4

```
bool RKSlopes(void);
bool RKStep(void);
void HandleBoundaries(void);
```

though other functions are important too.

3. Output trajectory results: This step is optional. Information about the integrated trajectory can be outputed by using any of the following public functions:

```
double GetBmagMin(void) const;
double GetBmagMax(void) const;
GeoVector GetPosition(double t_in) const;
GeoVector GetVelocity(double t_in) const;
double GetEnergy(double t_in) const;
double GetDistance(double t_in) const;
int Segments(void) const;
int Reflections(void) const;
int Mirrorings(void) const;
double ElapsedTime(void) const;
```

```
int Crossings(unsigned int output, unsigned int bnd) const;
void PrintTrajectory(const std::string traj_name, bool phys_units,
    unsigned int output, unsigned int stride = 1, double dt_out =
    0.0) const;
void PrintCSV(const std::string traj_name, bool phys_units, unsigned
    int stride = 1) const;
void InterpretStatus(void) const;
void PrintInfo(void) const;
```

Consult `$SPECTRUM/src/trajectory_base.cc` for details about what each of these functions do.

The file `$SPECTRUM/src/trajectory_base.hh` defines the following pre-processor macros and global constants:

`RECORD_TRAJECTORY` : This macro controls whether every step of the trajectory (time, position, and momentum) is stored in local arrays or if only the initial and current phase-space location are kept in memory.
Possible values: Commented or uncommented.

`RECORD_BMAG_EXTREMA` : This macro controls whether or not the highest and lowest value of the magnetic field magnitude encountered through the trajectory is stored.
Possible values: Commented or uncommented.

`TRAJ_ADV_SAFETY_LEVEL` : This macro controls which safety checks are implemented during the integration procedure. At the lowest setting, no safety checks are carried out. One level above this, the time-step `dt` is checked to ensure it does not become too small. At the highest level, along with the time-step check, the total number of steps taken so far and number of time-step adaptations performed each step are checked to ensure they don't become too large.
Possible values: `0`, `1`, and `2`.

`max_trajectory_steps` : The maximum number of steps per trajectory allowed. This is enforced only when `TRAJ_ADV_SAFETY_LEVEL` is set to `2`.
Possible values: Any positive integer.

`max_time_adaptations` : The maximum number of time-step adaptations allowed. This is enforced only when `TRAJ_ADV_SAFETY_LEVEL` is set to `2`.
Possible values: Any positive integer.

As a child of `Params` it possesses the member attributes `_t`, `_pos`, and `_mom`, which hold the current time, position, and momentum, respectively. The last two are GeoVector objects. From a Trajectory object's perspective, the value of the time variable can increase (forward-time) or decrease (backward-time), position is expressed in Cartesian coordinates, and the format of momentum depends on the transport being considered. This class also inherits `specie`, which it uses to convert between momentum and velocity (`_vel`) and establish its charge attribute (`q`). The arrays `traj_t`, `traj_pos`, and `traj_mom` record the phase-space trajectory information for post-processing.

→ Subsection 9.5

The timestep (`dt`) for all Trajectory objects constrained by a CFL condition on spatial advection. Some classes impose further constraints on the timestep from other processes such as spatial diffusion or momentum advection. These are physical constraints and they use a maximum spatial displacement variable, $\Delta r_{\max}$ (`dmax`), provided by the Background object and stored in the SpatialData object (`_spdata`) as the basis for the CFL condition. Additionally, the timestep is numerically restricted by the integration procedure when an adaptive integration method is used.

→ Subsection 9.8

A Trajectory object has a direct link to other important objects in the simulation via the following pointers:

→ Section 5
- `distributions`: pointer to an array of Distribution objects,

→ Section 4
- `background`: pointer to a Background object,

→ Section 8
- `diffusion`: pointer to a Diffusion object,

→ Section 7
- `bcond_t`, `bcond_s`, `bcond_m`: pointers to arrays of temporal, spatial, and momentum Boundary Condition objects, respectively, and

→ Section 6
- `icond_t`, `icond_s`, `icond_m`: pointers to temporal, spatial, and momentum Initial Condition objects, respectively.

Most other attributes of `TrajectoryBase` are transient variables that help transfer data between routines. One worth highlighting is the SpatialData object `spdata0`, which stores the initial background quantities. This information can be useful for certain distribution objects when performing binning operations.

3.2 Fieldline Trajectory

The `TrajectoryFieldline` class simply follows the fieldline for the desired background quantity,

$$\frac{d\mathbf{r}}{dt} = \mathbf{X}, \tag{1}$$

where $\mathbf{X}$ can be $\mathbf{u}$, $\mathbf{B}$, or $\mathbf{E}$. The timestep is determined by a simple CFL condition,

$$\Delta t = \mathrm{CFL}\frac{\Delta r_{\max}}{X}. \tag{2}$$

The file `$SPECTRUM/src/trajectory_fieldline.hh` defines the following global constants:

`cfl_adv_tf` : This is the CFL condition for the stepping procedure.
Possible values: Any double between `0.0` and `1.0`.

`which_field_to_follow` : This indicates which field to follow.
Possible values: `BACKGROUND_U`, `BACKGROUND_B`, and `BACKGROUND_E`.

The effective "momentum" of the fieldline tracer is held in `_mom[2]`, with the two other vector elements remaining unused, while the direction is obtained directly from the field being tracked.

3.3 Newton-Lorentz Trajectory

The `TrajectoryLorentz` class tracks particles using the **Newton-Lorentz equations of motion** for a charged particle,

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}, \tag{3}$$

$$\frac{d\mathbf{p}}{dt} = q\left(\mathbf{E} + \frac{\mathbf{v}\times\mathbf{B}}{c}\right), \tag{4}$$

where t is time, $\mathbf{r}$ is the particle's position, $\mathbf{p}$ is the particle's momentum, $\mathbf{v}$ is the particle's velocity, q is the particle's charge, c is the speed of light, and $\mathbf{E}, \mathbf{B}$ are the electric and magnetic field at $\mathbf{r}$. Equations (3)-(4) are the characteristics for the most general description for the evolution of a plasma species, called the **Vlasov-Boltzmann Transport Equation**,

$$\frac{\partial f}{\partial t} + \mathbf{v}\cdot\nabla f + q\left(\mathbf{E} + \frac{\mathbf{v}\times\mathbf{B}}{c}\right)\nabla_{\mathbf{p}} f = 0, \tag{5}$$

where $f(t, \mathbf{r}, \mathbf{p})$ is the charged particle phase-space distribution function. The timestep is constrained by an advection CFL condition as well as a minimum resolution of the gyro-orbit,

$$\Delta t = \min\left\{ \mathrm{CFL}\frac{\Delta r_{\max}}{v}, \frac{2\pi}{N\Omega} \right\}, \tag{6}$$

where N is the desired number of steps per gyro-orbit and Ω is the gyro-frequency.

The file `$SPECTRUM/src/trajectory_lorentz.hh` defines the following global constants:

`cfl_adv_tl` : This is the CFL condition for the stepping procedure.
Possible values: Any double between `0.0` and `1.0`.

`steps_per_orbit` : This is the minimum number of steps to take within each gyro-orbit. It is used in the computation of `dt_physical`.
Possible values: Any positive integer.

`mirror_thresh_lorentz` : This determines how many consecutive steps are taken after a suspected mirroring event before it is officially registered as such.
Possible values: Any positive integer.

All three components of `_mom` are utilized, and momentum is expressed in Cartesian coordinates.

3.4 Guiding Center Trajectory

The `TrajectoryGuiding` class tracks particles using the **Hamiltonian guiding center equations of motion** for a charged particle,

$$\frac{d\mathbf{r}_g}{dt} = \frac{1}{\mathbf{B}^* \cdot \mathbf{b}} \left(\frac{p_\parallel}{\gamma m}\mathbf{B}^* + c\mathbf{E}^* \times \mathbf{b} \right), \tag{7}$$

$$\frac{dp_\parallel}{dt} = \frac{q\mathbf{E}^* \cdot \mathbf{B}^*}{\mathbf{B}^* \cdot \mathbf{b}}, \tag{8}$$

$$\frac{dp_\perp}{dt} = \frac{p_\perp}{2B}\frac{\mathrm{d}B}{\mathrm{d}t}, \tag{9}$$

where t is time, $\mathbf{r}_g$ is the particle's guiding center, $p_\parallel$ is the particle's parallel component of momentum, $p_\perp$ is the particle's perpendicular component of momentum,[5] γ is the relativistic Lorentz factor, m is the particle's rest mass, q is the particle's charge, $\mathbf{E}$ and $\mathbf{B}$ represent the electric and magnetic fields at $\mathbf{r}_g$, B is the magnitude of $\mathbf{B}$, $\mathbf{b} = \mathbf{B}/B$, and

$$\frac{\mathrm{d}B}{\mathrm{d}t} = \frac{\partial B}{\partial t} + \frac{1}{\mathbf{B}^* \cdot \mathbf{b}} \left(\frac{p_\parallel}{\gamma m}\mathbf{B}^* + c\mathbf{E}^* \times \mathbf{b} \right) \cdot \nabla_g B.$$

The modified electromagnetic fields are

$$\mathbf{E}^* = \mathbf{E} - \frac{1}{q}\left(p_\parallel \frac{\partial \mathbf{b}}{\partial t} + \frac{p_\perp^2}{2\gamma m B}\nabla_g B \right), \tag{10}$$

$$\mathbf{B}^* = \mathbf{B} + \frac{p_\parallel c}{q}\nabla_g \times \mathbf{b}. \tag{11}$$

Equations (7)-(9) are the characteristics for the **Guiding Center Transport Equation**, applicable the limit of a weakly non-uniform, slowly varying electro-

[5] The momentum space coordinate axes are aligned with the magnetic field at $\mathbf{r}_g$ in the observer/rest frame.

magnetic background,[6]

$$\frac{\partial f}{\partial t} + \frac{1}{\mathbf{B}^* \cdot \mathbf{b}} \left(\frac{p_\parallel}{\gamma m} \mathbf{B}^* + c\mathbf{E}^* \times \mathbf{b} \right) \cdot \nabla_g f + \left(\frac{q\mathbf{E}^* \cdot \mathbf{B}^*}{\mathbf{B}^* \cdot \mathbf{b}} \right) \frac{\partial f}{\partial p_\parallel} + \left(\frac{p_\perp}{2B} \frac{\mathrm{d}B}{\mathrm{d}t} \right) \frac{\partial f}{\partial p_\perp} = 0, \tag{12}$$

where $f(t, \mathbf{r}_g, p_\parallel, p_\perp)$ is the particle distribution function. Equation (12) does not have any diffusion terms, and hence its characteristics are purely deterministic. Both perpendicular diffusion and pitch-angle scattering can be incorporated into this model. The timestep is determined by a simple CFL condition on spatial advection,

$$\Delta t = \mathrm{CFL}_a \frac{\Delta r_{\max}}{v_a + \varepsilon v}, \tag{13}$$

where v_a is the advection speed from (7), v is the particle's speed (computed relativistically from $p_\parallel$ and $p_\perp$, and ε is a small, positive number present to prevent division by zero.

The file `$SPECTRUM/src/trajectory_guiding.hh` defines the following pre-processor macros and global constants:

`PPERP_METHOD` : This macro controls whether $p_\perp$ is computed by directly enforcing the conservation of magnetic moment or according to (9), which does not guarantee conservation of magnetic moment when non-adiabatic terms (e.g. diffusion) are included.
Possible values: `0` and `1`.

`cfl_adv_tg` : This is the CFL condition for the advection portion of the stepping procedure.
Possible values: Any double between `0.0` and `1.0`.

`drift_safety_tg` : This is the fraction of the total velocity magnitude that is added to the denominator of the physical time-step calculation to prevent an overflow/underflow error when $p_\parallel$ is very small.
Possible values: Any double between `0.0` and `1.0`.

`mirror_thresh_guiding` : This determines how many consecutive steps are taken after a suspected mirroring event before it is officially registered as such.
Possible values: Any positive integer.

The perpendicular component of momentum is held in `_mom[0]`, while the parallel component in `_mom[2]`, and `_mom[1]` is unused.

3.4.1 Enabling Pitch-angle Scattering

→ Subsection 3.4

The `TrajectoryGuidingScatt` class is derived from `TrajectoryGuiding` and adds pitch-angle scattering effects to the guiding center equations of motion. In order to include the effect of pitch-angle scattering, the term

$$\frac{\partial}{\partial \mu} \left(D_{\mu\mu} \frac{\partial f}{\partial \mu} \right) = \frac{\partial D_{\mu\mu}}{\partial \mu} \frac{\partial f}{\partial \mu} + D_{\mu\mu} \frac{\partial^2 f}{\partial \mu^2}, \tag{14}$$

where μ represents the pitch-angle cosine of the particle and $D_{\mu\mu}$ is the pitch-angle scattering coefficient, is added to the right-hand side of (12). Thus, the stochastic equation

$$d\mu = \frac{\partial D_{\mu\mu}}{\partial \mu} dt + \sqrt{2 D_{\mu\mu}} dW_t, \tag{15}$$

where W_t is a 1D Wiener process, is simultaneously solved and the result combined with the solutions of (8)-(9) at each step in an elastic fashion (viz. no energy change).

[6] Specifically, this means $r_L \ll L$ and $\Omega_c^{-1} \ll T$, where r_L and Ω_c are the particle's Larmor radius and cyclotron frequency, respectively, while L and T are the characteristic length and timescales of the fields variations.

The constraints $0 \leq \mu \leq 1$ are enforced explicitly when numerically solving (15). The timestep is constrained by the spatial advection step as well as CFL conditions on pitch-angle advection and diffusion,

$$\Delta t = \min \left\{ \Delta t_a, \mathrm{CFL}_s \frac{\Delta \mu_{\max}}{|\partial D_{\mu\mu}/\partial \mu|}, \mathrm{CFL}_s \frac{(\Delta \mu_{\max})^2}{D_{\mu\mu}} \right\}, \tag{16}$$

where Δt_a is given by (13) and $\Delta \mu_{\max}$ is the maximum change in pitch-angle per step.

The file `$SPECTRUM/src/trajectory_guiding_scatt.hh` defines the following preprocessor macros and global constants:

`SPLIT_SCATT` : This macro controls whether or not to split the stochastic scattering step before and after the deterministic step.
Possible values: Commented or uncommented.

`CONST_DMUMAX` : This macro controls whether to use a maximum $\Delta\theta$ or $\Delta\mu$ ($\mu = \cos\theta$) as the resolution constraint for pitch-angle scattering.
Possible values: `0` and `1`.

`STOCHASTIC_METHOD_MU` : This macro controls the stochastic method used for pitch-angle scattering.
Possible values: `0`, `1`, and `2`.

`alpha` : This is the fraction of the stochastic scattering step to take before the deterministic step, when `SPLIT_SCATT` is uncommented.
Possible values: Any double between `0.0` and `1.0`.

`dmumax` : This is the maximum change in μ allowed during pitch-angle scattering when `CONST_DMUMAX` is set to `1`.
Possible values: Any double between `0.0` and `0.1`.

`dthetamax` : This is the maximum change in θ allowed during pitch-angle scattering when `CONST_DMUMAX` is set to `0`.
Possible values: Any double between `0.0` and `2.0 * M_PI / 90.0`.

`cfl_pa_gs` : This is the CFL condition for the pitch-angle scattering portion of the stepping procedure.
Possible values: Any double between `0.0` and `1.0`.

3.4.2 Enabling Perpendicular Diffusion

→ Subsection 3.4

The `TrajectoryGuidingDiff` class is derived from `TrajectoryGuiding` and adds perpendicular diffusion effects to the guiding center equations of motion. In order to include the effect of perpendicular diffusion, the term

$$\nabla_\perp \cdot (D_\perp \nabla_\perp f) = (\nabla_\perp D_\perp) \cdot \nabla_\perp f + D_\perp \nabla_\perp^2 f, \tag{17}$$

where $D_\perp$ is the spatial diffusion coefficient in the direction perpendicular to $\mathbf{B}$, is added to the right-hand side of (12). Thus, the stochastic equation

$$d\mathbf{r}_g = (\nabla_\perp D_\perp)\, dt + \sqrt{2D_\perp} d\mathbf{W}_t, \tag{18}$$

where $\mathbf{W}_t$ is a 2D Wiener process, is simultaneously solved and the result added to (7) at each step. The timestep is constrained by the advection step as well as a CFL condition on spatial diffusion,

$$\Delta t = \min \left\{ \Delta t_a, \mathrm{CFL}_d \frac{(\Delta r_{\max})^2}{D_\perp} \right\}, \tag{19}$$

where Δt_a is given by (13) but with v_a modified with $|\nabla_\perp D_\perp|$.

The file `$SPECTRUM/src/trajectory_guiding_diff.hh` defines the following preprocessor macros and global constants:

`STOCHASTIC_METHOD_PERP` : This macro controls the stochastic method used for perpendicular diffusion.
Possible values: `0`, `1`, and `2`.

`cfl_dif_gd` : This is the CFL condition for the perpendicular diffusion portion of the stepping procedure.
Possible values: Any double between `0.0` and `1.0`.

3.4.3 Enabling Pitch-angle Scattering and Perpendicular Diffusion

→ Subsection 3.4.2
→ Subsection 3.4.1

The `TrajectoryGuidingDiffScatt` class is derived from `TrajectoryGuidingDiff` and `TrajectoryGuidingScatt` and adds perpendicular diffusion and pitch-angle scattering effects to the guiding center equations of motion.

3.5 Focused Transport Trajectory

The `TrajectoryFocused` class tracks particles using the **focused transport equations of motion** for a charged particle without pitch-angle diffusion,

$$\frac{d\mathbf{r}}{dt} = \mathbf{u} + v\mu\mathbf{b}, \tag{20}$$

$$\frac{dp}{dt} = \frac{p}{2}\left[(1-3\mu^2)\mathbf{bb}:\nabla\mathbf{u} - (1-\mu^2)\nabla\cdot\mathbf{u} - \frac{2\mu}{v}\mathbf{b}\cdot\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right], \tag{21}$$

$$\frac{d\mu}{dt} = \frac{1-\mu^2}{2}\left(v\nabla\cdot\mathbf{b} - 3\mu\mathbf{bb}:\nabla\mathbf{u} + \mu\nabla\cdot\mathbf{u} - \frac{2}{v}\mathbf{b}\cdot\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right), \tag{22}$$

where t is time, $\mathbf{r}$ is the particle's position, p is the particle's momentum magnitude, μ is the particle's pitch-angle cosine, v is the particle speed, $\mathbf{u}$ is the background plasma flow velocity at $\mathbf{r}$, $\mathbf{b}$ is the unit vector in the direction of the magnetic field at $\mathbf{r}$, $D_{\mu\mu}$ is the pitch-angle scattering coefficient, and

$$\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \frac{\partial\mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}.$$

Note that v and p are measured in the plasma flow frame, while μ and $\mathbf{b}$ are taken at the particle's current position. Equations (20)-(22) are the characteristics for the **Focused Transport Equation**, valid under the assumption of gyrotropy in the plasma flow frame (where the electric field is zero),

$$\frac{\partial f}{\partial t} + (\mathbf{u}+v\mu\mathbf{b})\cdot\nabla f + \frac{p}{2}\left[(1-3\mu^2)\mathbf{bb}:\nabla\mathbf{u} - (1-\mu^2)\nabla\cdot\mathbf{u} - \frac{2\mu}{v}\mathbf{b}\cdot\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right]\frac{\partial f}{\partial p} + \frac{1-\mu^2}{2}\left[v\nabla\cdot\mathbf{b} - 3\mu\mathbf{bb}:\nabla\mathbf{u} + \mu\nabla\cdot\mathbf{u} - \frac{2}{v}\mathbf{b}\cdot\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right]\frac{\partial f}{\partial\mu} = 0, \tag{23}$$

where $f(t,\mathbf{r},p,\mu)$ is the gyrophase averaged distribution function.

To include the effect of pitch-angle scattering, the term

$$\frac{\partial}{\partial\mu}\left(D_{\mu\mu}\frac{\partial f}{\partial\mu}\right) = \frac{\partial D_{\mu\mu}}{\partial\mu}\frac{\partial f}{\partial\mu} + D_{\mu\mu}\frac{\partial^2 f}{\partial\mu^2}, \tag{24}$$

is added to the right-hand side of (23). Thus, the stochastic equation

$$d\mu = \frac{\partial D_{\mu\mu}}{\partial\mu}dt + \sqrt{2D_{\mu\mu}}dW_t, \tag{25}$$

where W_t is a 1D Wiener process, is simultaneously solved and the result added to (22) at each step. The constraints $0 \le \mu \le 1$ are enforced explicitly when numerically solving (25). The effect of magnetic field drifts is not present in (23). However, it can be incorporated by adding the following drift term to (20),

$$\mathbf{v}_D = \frac{pvc}{qB^2}\left\{\frac{1-\mu^2}{2}\left[\mathbf{b}\times\nabla B + \mathbf{b}\left(\mathbf{b}\cdot\nabla\times\mathbf{B}\right)\right] + \mu^2\mathbf{b}\times\left[\left(\mathbf{b}\cdot\nabla\right)\mathbf{B}\right]\right\}, \tag{26}$$

where $\mathbf{B}$ represents the magnetic field at $\mathbf{r}$, and B is the magnitude of $\mathbf{B}$. When pitch-angle scattering is disabled, the timestep is determined by a simple CFL condition on spatial advection,

$$\Delta t = \mathrm{CFL}_a \frac{\Delta r_{\max}}{|\mathbf{u} + v\mu\mathbf{b} + \mathbf{v}_D| + \varepsilon v}, \tag{27}$$

where ε is a small, positive number present to prevent division by zero.

The file `$SPECTRUM/src/trajectory_focused.hh` defines the following pre-processor macros and global constants:

`PPERP_METHOD` : This macro controls whether μ is computed by directly enforcing the conservation of magnetic moment or according to (22), which does not guarantee conservation of magnetic moment when non-adiabatic terms (e.g. diffusion) are included.
Possible values: `0` and `1`.

`TRAJ_FOCUSED_USE_B_DRIFTS` : This macro controls whether or not to include the magnetic field drifts in the trajectory computation.
Possible values: Commented or uncommented.

`cfl_adv_tf` : This is the CFL condition for the advection portion of the stepping procedure.
Possible values: Any double between `0.0` and `1.0`.

`drift_safety_tf` : This is the fraction of v that is added to the denominator of the physical time-step calculation to prevent an overflow/underflow error when μ is very small.
Possible values: Any double between `0.0` and `1.0`.

`mirror_thresh_focused` : This determines how many consecutive steps are taken after a suspected mirroring event before it is officially registered as such.
Possible values: Any positive integer.

The magnitude of momentum is held in `_mom[0]`, while the pitch-angle cosine is in `_mom[2]`, and `_mom[1]` is unused.

3.6 Parker Trajectory

The `TrajectoryParker` class tracks pseudo-particles using the deterministic characteristic equations

$$\frac{d\mathbf{r}}{dt} = \mathbf{u} + \mathbf{v}_D + \mathbf{K}, \tag{28}$$

$$\frac{dp}{dt} = -p\frac{\nabla \cdot \mathbf{u}}{3}, \tag{29}$$

where t is time, $\mathbf{r}$ is the particle's position, p is the particle's momentum magnitude, $\mathbf{u}$ is the background plasma flow velocity at $\mathbf{r}$, and $\mathbf{v}_D$ incorporates the gradient and curvature magnetic field drifts in a compact form,

$$\mathbf{v}_D = \frac{pvc}{3q}\nabla \times \left(\frac{\mathbf{B}}{B^2}\right), \tag{30}$$

where v is the particle speed, q is the particle's charge, c is the speed of light, $\mathbf{B}$ is magnetic field at $\mathbf{r}$, and B is the magnitude of $\mathbf{B}$. The term $\mathbf{K}$ comes from solving the stochastic equation

$$d\mathbf{r} = (\nabla \cdot \kappa)\, dt + \sigma d\mathbf{W}_t, \tag{31}$$

where κ is the (symmetric) diffusion tensor at $\mathbf{r}$, $\mathbf{W}_t$ is a 3D Wiener process, and $\sigma\sigma^T = 2\kappa$. This equation is simultaneously solved and the result added to (28) at each step.

It is worth highlighting that (28)-(31) do not describe single particle motion. Instead, these are the characteristics for the **Parker Transport Equation**, valid for nearly isotropic description of plasma,

$$\frac{\partial f}{\partial t} + (\mathbf{u} + \mathbf{v}_D) \cdot \nabla f - \nabla \cdot (\kappa \cdot \nabla f) - p \frac{\nabla \cdot \mathbf{u}}{3} \frac{\partial f}{\partial p} = 0, \tag{32}$$

where $f(t, \mathbf{r}, p)$ is the pitch-angle averaged distribution function. The effect of spatial diffusion comes included in this type of transport by default. The timestep is constrained by the spatial advection step as well as CFL conditions on spatial diffusion and momentum advection,

$$\Delta t = \min \left\{ \mathrm{CFL}_a \frac{\Delta r_{\max}}{|\mathbf{u} + \mathbf{v}_D \pm \nabla \kappa|}, \mathrm{CFL}_d \frac{(\Delta r_{\max})^2}{|\kappa|_{\max}}, \mathrm{CFL}_p \frac{3 \Delta \ln p_{\max}}{|\nabla \cdot \mathbf{u}|} \right\}, \tag{33}$$

where $\pm$ depends on whether the simulation is forward or backward in time, $\Delta \ln p_{\max}$ is the maximum change in the natural logarithm of momentum, and $|\kappa|_{\max}$ is largest component of κ in absolute value.

The file `$SPECTRUM/src/trajectory_parker.hh` defines the following pre-processor macros and global constants:

`TRAJ_PARKER_STOCHASTIC_METHOD_DIFF` : This macro controls the stochastic method used for spatial diffusion.
Possible values: `0`, `1`, and `2`.

`TRAJ_FOCUSED_USE_B_DRIFTS` : This macro controls whether or not to include the magnetic field drifts in the trajectory computation.
Possible values: Commented or uncommented.

`TRAJ_PARKER_DIVK_METHOD` : This macro controls whether to compute the divergence of the diffusion tensor using central FD or by leveraging the already computed gradient quantities in `_spdata`.
Possible values: `0` and `1`.

`cfl_adv_tp` : This is the CFL condition for the advection portion of the stepping procedure.
Possible values: Any double between `0.0` and `1.0`.

`cfl_dif_tp` : This is the CFL condition for the spatial diffusion portion of the stepping procedure.
Possible values: Any double between `0.0` and `1.0`.

`cfl_acc_tp` : This is the CFL condition for the momentum acceleration portion of the stepping procedure.
Possible values: Any double between `0.0` and `1.0`.

`dlnpmax` : Maximum momentum change per step as a fraction of current momentum.
Possible values: Any positive double.

The magnitude of momentum is held in `_mom[0]`, with the two other vector elements remaining unused.

4 Background Modules

In this section, we describe the field background modules currently supported within the existing SPECTRUM architecture. These are broadly categorized into **analytic** and **discretized**. In this context, "Unstructured Meshes" can be viewed as the most demanding and sophisticated mode of operation with respect to the background employed to propagate trajectories.

4.1 Analytic vs Discretized Backgrounds

Analytic backgrounds are defined everywhere on the desired domain through a function and it can be independently computed by each SimW process. Discretized backgrounds are defined on a **mesh**, composed of **cells**, which could be structured or unstructured, and the field must be interpolated from its grid values. The background can be accessed by the SimW processes in two ways, through **distributed** or **shared memory**, but in both cases the SimS processes perform some initial tasks to set up the field data in memory. When the field data is distributed, the SimS processes hold the entire field data and pass local information to the SimW processes as these request it. This communication specifically happens between a `server_front`[7] and a `server_back` objects (part of SimW and SimS, respectively). When the field data is shared, the SimS processes pass array pointers to the SimW processes so these can access any cell directly.

Groups of adjacent cells are automatically assembled into **blocks**, so that when a SimW requests the necessary cell data to calculate the fields at location **r**, it receives an entire block of data containing **r**. This enables the SimW to potentially advance multiple steps within the region covered by the block of data, thus reducing the MPI communication overhead. Furthermore, each `server_front` object keeps a cache line of recently sampled blocks to improve efficiency.

4.2 Base Background

The `BackgroundBase` class is not meant to be used directly, but rather contains and defines the general functionality shared by all the application specific background modules, described in the following subsections. It is inherited from the `Params` class.

→ Subsection 9.7

The following inputs are required to build any background class:

`t0` : Reference time for all time-dependent computations.
Possible values: Any double.

`r0` : Origin of the coordinate system in which the background is defined.
Possible values: Any GeoVector.

`u0` : Reference vector for bulk plasma flow.
Possible values: Any GeoVector.

`B0` : Reference vector for magnetic field.
Possible values: Any GeoVector.

`dmax0` : Default value of maximum distance travelled by a trajectory per step.
Possible values: Any positive double.

Derived background classes will typically require additional parameters for initialization. These parameters are passed through a DataContainer object in the public function

→ Subsection 9.2

[7] This object is defined in the `BackgroundDiscretized` submember of the `TrajectoryType` member of the `SimulationWorker` object.

```
void SetupObject(const DataContainer& cont_in);
```

which calls the protected

```
virtual void SetupBackground(bool construct);
```

A Simulation or Trajectory Integrator (TI) object would call the function

```
void AddBackground(const BackgroundBase& background_in, const
    DataContainer& container_in, const std::string& fname_pattern_in = "");
```

to construct a background module.

The file `$SPECTRUM/src/background_base.hh` defines the following pre-processor macros and global constants:

`BACKGROUND_NUM_GRAD_EVALS` : This macro controls the number of local coordinate systems in which to evaluate and average numerical derivatives.
Possible values: Any positive integer.

`incr_dmax_ratio` : This is the fraction of `_spdata.dmax` to use when shifting the position for numerical derivatives.
Possible values: Any double between `0.0` and `1.0`.

`local_rot_ang` : This is the rotation angle of local (x, y) plane in numerical derivative evaluation/averaging.
Possible values: Any double.

`sin_lra` : This is a recomputed sine of `local_rot_ang`.
Possible values: Automatically determined by `local_rot_ang`.

`cos_lra` : This is a precomputed cosine of `local_rot_ang`.
Possible values: Automatically determined by `local_rot_ang`.

The Trajectory Integrator object will obtain the necessary background quantities through use of the public function

```
void GetFields(double t_in, const GeoVector& pos_in, SpatialData& spdata);
```

This function calls the protected methods

```
virtual void EvaluateDmax(void);
virtual void EvaluateBackground(void);
virtual void EvaluateBackgroundDerivatives(void);
```

→ Subsection **??** The fields variables are locally stored in SpatialData objects `_spdata`, while `_spdata_tmp` is used to compute numerical derivatives though the functions

```
void DirectionalDerivative(int xyz);
void NumericalDerivatives(void);
```

→ Subsection 3.1 Background objects are linked to TI objects through the latter's `background` pointer.

4.3 Uniform Background

The `BackgroundUniform` class simply prescribes uniform fields,

$$\mathbf{u} = \mathbf{u}_0, \tag{34}$$

$$\mathbf{B} = \mathbf{B}_0, \tag{35}$$

$$\mathbf{E} = -\frac{\mathbf{u}_0 \times \mathbf{B}_0}{c}. \tag{36}$$

Naturally, all spatial and temporal derivatives of these fields are zero. The maximum spatial displacement per step is always `dmax0`.

No additional parameters need to be provided beyond those required for `BackgroundBase`.

4.4 Dipole Background

The `BackgroundDipole` class implements the magnetic field of a magnetic dipole,

$$\mathbf{u} = 0, \tag{37}$$

$$\mathbf{B} = \frac{3(\mathbf{m}\cdot\mathbf{r})\mathbf{r} - r^2\mathbf{m}}{r^5}, \tag{38}$$

$$\mathbf{E} = 0, \tag{39}$$

where $\mathbf{m}$ is the magnetic moment of the dipole and $\mathbf{r}$ is the position relative to $\mathbf{r}_0$. The only non-zero derivatives are the gradient of the magnetic field and its magnitude, which is

$$\nabla\mathbf{B} = 3\frac{r^2\left(\mathbf{mr}+\mathbf{rm}\right) + \mathbf{m}\cdot\mathbf{r}\left(r^2\mathrm{I} - 5\mathbf{rr}\right)}{r^7}, \tag{40}$$

$$\nabla B = \nabla\mathbf{B}\cdot\mathbf{b}, \tag{41}$$

where I is the identity matrix. The maximum spatial displacement per step is computed with the formula

$$\Delta r_{\max} = \min\left\{\Delta r_{\max,0}, \delta r\right\}, \tag{42}$$

where δ is a user-specified constant. Figure 6 shows a 2D cross-section of the magnetic field magnitude obtained with this model, along with some magnetic fieldlines.

On top of the inputs already required by `BackgroundBase`, the following parameters should be provided when constructing this module:

`r_ref` : Reference distance on the equatorial plane with respect to the dipole's symmetry axis used to define the magnetic moment vector using the formula $\mathbf{m} = r_{\mathrm{ref}}^3\mathbf{B}_0$.
Possible values: Any positive double.

`dmax_fraction` : Fraction of distance from the origin to use as a threshold for the calculation of `_spdata.dmax`. Possible values: Any positive double.

The file `$SPECTRUM/src/background_dipole.hh` defines the following pre-processor macro:

`DIPOLE_DERIVATIVE_METHOD` : This macro controls whether to compute the derivatives for this background analytically or numerically.
Possible values: `0` and `1`.

4.5 Parker Spiral Background

The `BackgroundSolarWind` class implements a simple, analytic solar wind background following a Parker Spiral, with the options of some modifications. The simplest form of this class uses the following fields, expressed in spherical coordinates:

$$u_r = u_0, \quad u_\theta = 0, \quad u_\phi = 0, \tag{43}$$

$$B_r = B_0\left(\frac{r_{\mathrm{ref}}}{r}\right)^2, \quad B_\theta = 0, \quad B_\phi = -B_r\frac{(r-r_{\mathrm{ref}})\Omega}{u_r}\sin\theta, \tag{44}$$

$$\mathbf{E} = -\frac{\mathbf{u}\times\mathbf{B}}{c}, \tag{45}$$

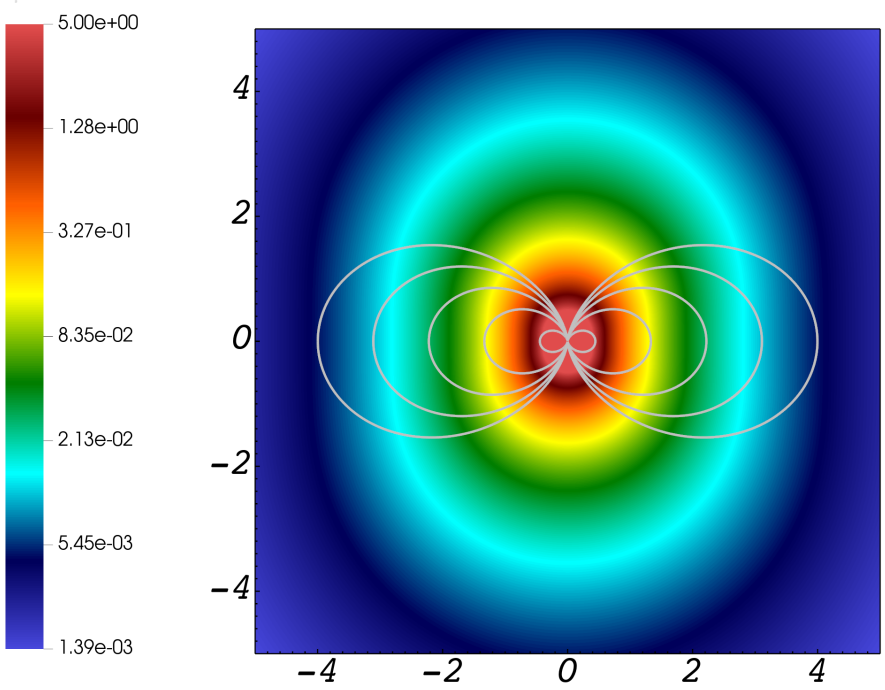

Figure 6: 2D cross-section magnetic field magnitude along the **xz**-plane. The far-away field is parallel to **z** and the field was defined to have a strength of 0.31 G at 1 distance unit (Earth radius) on the equator. The colorplot is saturated at 5 G for visualization purposes. The gray curves show the magnetic field streamlines.

where $r_{\rm ref}$ is a reference distance near the Sun at which the magnetic field is assumed to be purely radial, $\mathbf{\Omega}$ is the angular velocity vector of the Sun's rotation, and $\mathbf{r}$ is the position relative to $\mathbf{r}_0$. The code computes $\mathbf{u}$ and $\mathbf{B}$ in spherical coordinates with a **z**-axis parallel to $\mathbf{\Omega}$ and converts them to the global Cartesian system. Note that only the first component of `u0` and `B0` are used to determine u_0 and B_0, respectively. The derivatives for this background are computed numerically. Figure 7 shows a few Parker spiral fieldlines.

The maximum spatial displacement per step is computed with the formula

$$\Delta r_{\rm max} = \min\left\{\Delta r_{\rm max,0}, \delta r\right\}, \tag{46}$$

where δ is a user-specified constant. The first component of the `_spdata.region` is used as an indicator variable to distinguish the heliosphere, with a value of $+1$, from the local interstellar medium (LISM), with a value of -1. The second component of the `_spdata.region` is used as an indicator variable to distinguish regions containing a mostly unipolar magnetic field, with a value of -1, and sectored magnetic field, with a value of $+1$.

If a heliospheric current sheet (HCS) is enabled, the orientation of the magnetic field below the current sheet is reversed. The current sheet can be flat at the solar rotation equator or wavy. Assuming a constant profile in the solar wind speed, the wavy current sheet is defined with the Jokipii and Thomas model. The polar angle locating a static current sheet is determined by,

$$\theta = \frac{\pi}{2} + \alpha \sin\left(\phi + \frac{r\Omega}{u_0}\right), \tag{47}$$

where α is the (HCS) tilt angle. If the wavy current sheet evolves in time, the above

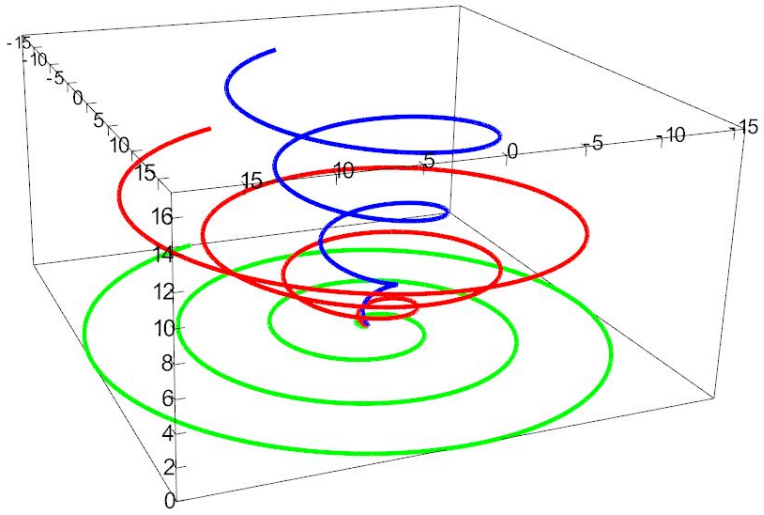

Figure 7: Sample Parker spiral fieldlines.

formula is enhanced to

$$\theta = \frac{\pi}{2} + \alpha\left(\frac{r}{u_0} - t\right)\sin\left[\phi + \left(\frac{r}{u_0} - t\right)\Omega\right]. \tag{48}$$

where t is the time relative to t_0 and the tilt angle is varied according to

$$\alpha(s) = \alpha_0 + \Delta\alpha\cos\left[C\left(\frac{4\pi}{T_s}s\right)\right], \tag{49}$$

where α_0 is the average tilt, $\Delta\alpha$ is the tilt variation over the solar cycle, T_s is the period of the solar cycle, and C is a periodic function over 2π used to stretch the minima and compress the maxima defined by the cubic polynomial

$$C(x) = x\left[(1-\epsilon)\frac{x}{\pi}\left(\frac{x}{\pi} - 3\right) + 3 - 2\epsilon\right], \qquad 0 \le x \le 2\pi, \tag{50}$$

and $0 \le \epsilon \le 1$ controls the amount of stretching. The solar cycle dependance is also included by flipping the polarity of the magnetic field emanating from the Sun every T_s years and propagating this change at a speed u_0 from the solar surface to the rest of the heliosphere.

The standard Parker Spiral magnetic field given by (44) can be modified according to different theories. The main purpose of these modifications are to strengthen of the field in the polar regions. The Smith and Bieber correction makes

$$B_r = B_0\left(\frac{r_{\text{ref}}}{r}\right)^2, \quad B_\theta = 0, \quad B_\phi = -B_r\frac{\Omega}{u_r}\left[(r - r_{\text{ref}})\sin\theta + \delta_{\text{SB}}r\right], \tag{51}$$

where δ_{SB} is a differential rotation factor. The Fisk correction makes

$$B_r = B_0\left(\frac{r_{\text{ref}}}{r}\right)^2, \tag{52}$$

$$B_\theta = B_r\frac{(r - r_{\text{ref}})\omega_\theta}{u_r}\sin\left(\phi + \frac{(r - r_{\text{ref}})\Omega}{u_r}\right), \tag{53}$$

$$B_\phi = B_r\frac{(r - r_{\text{ref}})}{u_r}\left[\omega_\theta\cos\theta\cos\left(\phi + \frac{(r - r_{\text{ref}})\Omega}{u_r}\right) + (\omega_\phi - \Omega)\sin\theta\right], \tag{54}$$

where ω_θ and ω_ϕ are the polar and azimuthal differential rotation angular frequencies, respectively.

The solar wind in this model is always radial. However, it may have a non constant latitudinal profile, distinguishing the slow solar wind near the equator from the fast solar wind in the polar regions. In such cases, u_0 is taken to be the value of the slow wind, and a parameter $F > 1$ is defined as the ratio of fast to slow wind. If the transition happens linearly, the transition in the northern hemisphere is dictated by

$$u_r = \begin{cases} Fu_0, & \theta < \theta_t - \Delta\theta, \\ \left[1 + (F-1)\dfrac{\theta_t + \Delta\theta - \theta}{2\Delta\theta}\right] u_0, & \theta_t - \Delta\theta \le \theta \le \theta_t + \Delta\theta, \\ u_0, & \theta > \theta_t + \Delta\theta, \end{cases} \tag{55}$$

where θ_t is the midpoint of the latitude transition region and $\Delta\theta$ is the (half) width of the latitude transition region. If the transition happens smoothly, the transition in the northern hemisphere is dictated by

$$u_r = \frac{1}{2}\left[F + 1 - (F-1)\tanh\left(\frac{\theta - \theta_t}{\Delta\theta}\right)\right], \tag{56}$$

where $\Delta\theta$ becomes the "effective" (half) width. The southern hemisphere is symmetric to the north in both cases.

On top of the inputs already required by `BackgroundBase`, the following parameters should be provided when constructing this module:

`Omega` : Angular velocity vector of the Sun's rotation. If this is the zero vector, the global **z**-axis is used as the default directly of Ω.
Possible values: Any GeoVector.

`r_ref` : Reference distance near the Sun at which the magnetic field is assumed to be purely radial.
Possible values: Any positive double.

`dmax_fraction` : Fraction of distance from the origin to use as a threshold for the calculation of `_spdata.dmax`. Possible values: Any positive double.

The file `$SPECTRUM/src/background_solarwind.hh` defines the following preprocessor macros and global constants:

`SOLARWIND_DERIVATIVE_METHOD` : This macro controls whether to compute the derivatives for this background analytically or numerically.
Possible values: `0` and `1`.

`SOLARWIND_CURRENT_SHEET` : This macro controls which type of heliospheric current sheet model to use, if any.
Possible values: `0`, `1`, `2`, and `3`.

`SOLARWIND_SECTORED_REGION` : This macro controls what regions of space to classify as unipolar or sectored.
Possible values: `0` and `1`.

`SOLARWIND_POLAR_CORRECTION` : This macro controls which correction to the standard Parker Spiral model to use, if any.
Possible values: `0`, `1`, and `2`.

`SOLARWIND_SPEED_LATITUDE_PROFILE` : This macro controls the latitudinal profile of the solar wind speed.
Possible values: `0`, `1`, and `2`.

`hp_rad_sw` : This represents an artificial limit past which to classify space as outside of the heliosphere (i.e. in the LISM).
Possible values: Any positive double.

`tilt_ang_sw` : This is the magnetic axis tilt angle relative to the solar rotation axis. It only affects the fields computations when `SOLARWIND_CURRENT_SHEET` is `2` or `3`.
Possible values: Any positive double.

`dtilt_ang_sw` : This is the amplitude of variation to magnetic axis tilt angle. It is only used when `SOLARWIND_CURRENT_SHEET` is `3`.
Possible values: Any positive double.

`W0_sw` : This is the angular frequency of the solar cycle. It is only used when `SOLARWIND_CURRENT_SHEET` is `3`.
Possible values: Any positive double.

`stilt_ang_sw` : This is a factor to thin peaks and widen troughs of the tilt angle cycle. It is only used when `SOLARWIND_CURRENT_SHEET` is `3`.
Possible values: Any double between `0.0` and `1.0`.

`delta_omega_sw` : This is the differential rotation factor used in corrections to the Parker Spiral. It is only used when `SOLARWIND_POLAR_CORRECTION` is not `0`.
Possible values: Any positive double.

`polar_offset_sw` : This is the polar correction angle in the Fisk correction to the Parker Spiral. It is only used when `SOLARWIND_POLAR_CORRECTION` is `2`.
Possible values: Any positive double.

`dwt_sw` : This is the ratio of polar differential rotation to angular frequency of rotation in the Fisk correction to the Parker Spiral. It is only used when `SOLARWIND_POLAR_CORRECTION` is `2`.
Possible values: Automatically determined by both `delta_omega_sw` and `polar_offset_sw`.

`dwp_sw` : This is the ratio of azimuthal differential rotation to angular frequency of rotation in the Fisk correction to the Parker Spiral. It is only used when `SOLARWIND_POLAR_CORRECTION` is `2`.
Possible values: Automatically determined by both `delta_omega_sw` and `polar_offset_sw`.

`fast_slow_ratio_sw` : This is the ratio of fast solar wind spped to slow solar wind speed. It is only used when `SOLARWIND_SPEED_LATITUDE_PROFILE` is not `0`.
Possible values: Any positive double.

`fast_slow_lat_sw` : This is the polar angle marking the transition between fast and slow speeds. It is only used when `SOLARWIND_SPEED_LATITUDE_PROFILE` is not `0`.
Possible values: Any positive double.

`fast_slow_dlat_sw` : This is a factor that controls the size of the transition region between and slow speeds. It is only used when `SOLARWIND_SPEED_LATITUDE_PROFILE` is not `0`.
Possible values: Any positive double.

4.6 Parker Spiral with a Termination Shock Background

→ Subsection 4.5

The `BackgroundSolarWindTermShock` class is derived from `BackgroundSolarWind` and enhances the latter by adding a spherical termination shock (TS) and modifying the behavior of the fields beyond it. This model separates the heliosphere between the supersonic solar wind, delimited by the TS, and the heliosheath (HS), a region of slower plasma flow which lies beyond it.

This class modifies the solar wind speed from its parent class to be

$$u_r = \begin{cases} u_0, & r < r_{\mathrm{TS}}, \\ \left[1 + \left(\frac{1}{s} - 1\right) \frac{r - r_{\mathrm{TS}}}{w_{\mathrm{TS}}}\right] u_0, & r_{\mathrm{TS}} \le r \le r_{\mathrm{TS}} + w_{\mathrm{TS}}, \\ \frac{1}{s} \left(\frac{r_{\mathrm{TS}} + w_{\mathrm{TS}}}{r}\right)^n u_0, & r > r_{\mathrm{TS}} + w_{\mathrm{TS}}, \end{cases} \tag{57}$$

where r_{TS}, w_{TS}, and s are the location, width, and strength of the TS, respectively, and the (integer) exponent n determines the rate at which the plasma flow slows down with radial distance. Any formulas for the magnetic field involving u_r, including corrections, utilize the updated value in the HS. This will result in the non-radial components of the HS magnetic field increasing by the same factor by which the flow is reduced, so the motional electric field remains unchanged. Figure 8 shows a 2D cross-section of the magnetic field magnitude obtained from this model.

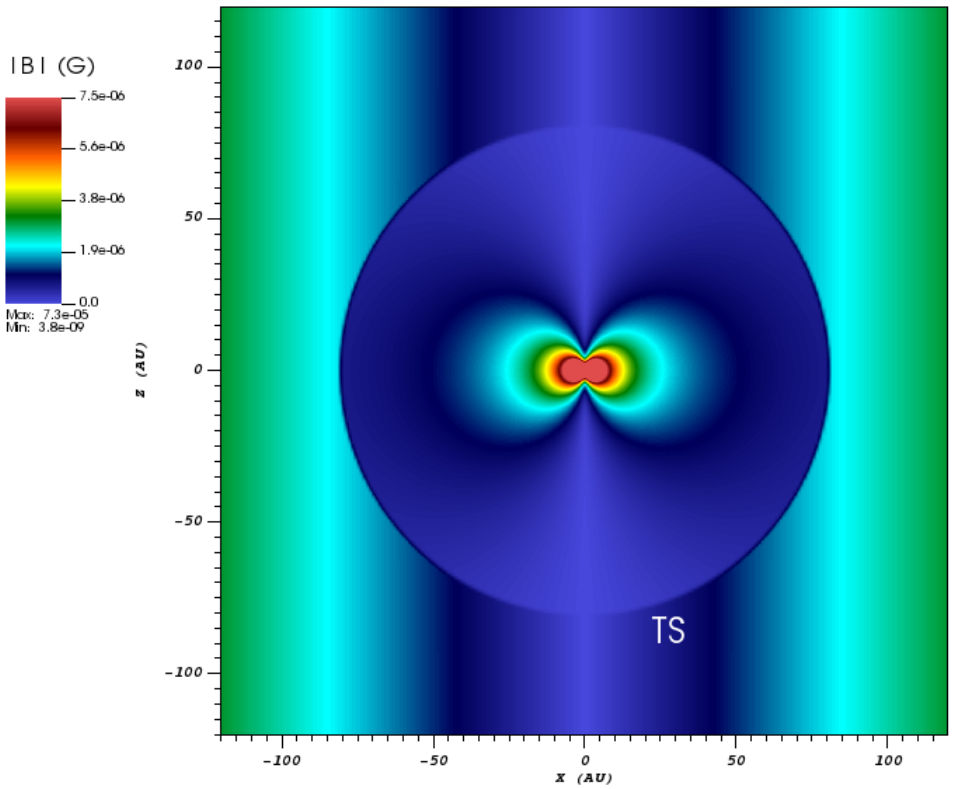

Figure 8: 2D cross-section magnetic field magnitude of a discretized Parker Spiral with a termination shock (HP). The scale was saturated beyond 7.5 μG to aid visualization.

The maximum spatial displacement per step is set to $\Delta r_{\mathrm{frac}} w_{\mathrm{TS}}$ inside the shock transition region, i.e. $r_{\mathrm{TS}} \le r \le r_{\mathrm{TS}} + w_{\mathrm{TS}}$. It is also modified within $\Delta r_{\mathrm{max},0}$ of the shock (on both sides) to be linearly interpolated from $\Delta r_{\mathrm{max},0}$ to the value within the shock. This ensures that Δr_{max} is never large enough for pseudo-particles to "miss" the shock by traveling from one side to the other in a single step, forcing them to sample the shock transition region, for which $\nabla \cdot \mathbf{u} < 0$, which is important for acceleration effects.

The only other change in the HS is the shape of HCS. The updated formula for a

static current sheet is

$$\theta = \frac{\pi}{2} + \alpha \sin\left[\phi + t_{\rm lag}(r)\Omega\right], \tag{58}$$

while for a time-dependant current sheet

$$\theta = \frac{\pi}{2} + \alpha(t_{\rm lag}(r) - t) \sin\left[\phi + (t_{\rm lag}(r) - t)\,\Omega\right], \tag{59}$$

where

$$t_{\rm lag} = \begin{cases} \dfrac{r}{u_0}, & r \le r_{\rm TS}, \\ \dfrac{r_{\rm TS}}{u_0}\left(1 + \dfrac{s}{n+1}\dfrac{r^{n+1} - r_{\rm TS}^{n+1}}{r_{\rm TS}^{n+1}}\right), & r > r_{\rm TS}, \end{cases} \tag{60}$$

On top of the inputs already required by `BackgroundBase` and `BackgroundSolarWind`, the following parameters should be provided when constructing this module:

`r_TS` : Radial location of the TS.
Possible values: Any positive double.

`w_TS` : Width of TS.
Possible values: Any positive double.

`s_TS` : Strength of TS. Possible values: Any positive double.

The file `$SPECTRUM/src/background_solarwind_termshock.hh` defines the following pre-processor macro:

`SOLARWIND_TERMSHOCK_SPEED_EXPONENT` : This macro controls the rate of solar wind speed decrease with radial distance.
Possible values: `0`, `1`, and `2`.

4.7 CKF LISM Background

The `BackgroundVLISMBochum` class implements the analytic model of Röken, Kleimann, and Fichtner for the local interstellar medium (LISM). This model assumes a simplified, axisymmetric shape for the heliopause (HP), the boundary between the heliosphere and the LISM, called a Rankine half-body. A Rankine half-body is a quartic surface of revolution given by

$$s^4 + s^2 z^2 - 4 z_0^2 s^2 - 4 z_0^2 z^2 + 4 z_0^4 = 0, \tag{61}$$

where z_0 is the distance from the Sun (origin) to the stagnation point (tip) on the HP and $s^2 = x^2 + y^2$.

In cylindrical coordinates, the flow velocity is given by

$$u_s = \frac{z_0^2 s}{r^3} u_0, \quad u_\varphi = 0, \quad u_z = \left(\frac{z_0^2 s}{r^3} - 1\right) u_0, \tag{62}$$

where $r^2 = s^2 + z^2$. The magnetic field has a more intricate expression. It is the sum of two components, one longitudinal to the velocity and another transverse to it,

$$\mathbf{B} = \mathbf{B}^l + \mathbf{B}^t \tag{63}$$

The longitudinal component is straightforward,

$$B_s^l = \frac{z_0^2 s}{r^3} B_0^l, \quad B_\varphi^l = 0, \quad B_z^l = \left(\frac{z_0^2 s}{r^3} - 1\right) B_0^l. \tag{64}$$

The transverse component is more complex,

$$B_s^t = \left[\frac{z_0^3 s}{ar^3}\mathcal{T} + \frac{a}{s}\left(1 + \frac{z_0^2 z}{r^3}\right)\right] B_{s,0}^t, \tag{65}$$

$$B_\varphi^t = \frac{s}{a} B_{\varphi,0}^t, \tag{66}$$

$$B_z^t = \left[\frac{z_0}{a}\left(\frac{z_0^2 z}{r^3} - 1\right)\mathcal{T} + \frac{a z_0^2 z^2}{r^3 s^2}\right] B_{s,0}^t, \tag{67}$$

where

$$\mathcal{T} = \left(2\eta - \frac{1}{\eta}\right) E\left[\sin^{-1}(\lambda\eta), \eta^{-1}\right] - 2\left(\eta - \frac{1}{\eta}\right) F\left[\sin^{-1}(\lambda\eta), \eta^{-1}\right], \tag{68}$$

F and E are incomplete elliptic integrals of the first and second kind,

$$\lambda = \sqrt{1 - \frac{a^2}{s}}, \quad \eta = \sqrt{1 + \frac{a^2}{4z_0^2}}, \tag{69}$$

and

$$a = \sqrt{s^2 - 2z_0^2\left(1 - \frac{z}{r}\right)} \tag{70}$$

is the impact parameter, i.e. the distance from the flow streamline going through (s, φ, z) to the **z**-axis far upstream.

Technically, the above expressions for the magnetic field diverge near the HP. To overcome this problem at each point, we first identify the isochrone surface containing that point, which is the surface of constant flow travel time from the plane perpendicular to **u** far away from the HP. Aligning the **z**-axis with this far away flow, the isochrone surface will asymptotically become perpendicular to **z** as $s \to \infty$. The isochrone label, ζ, is the z value that the isochrone surface approaches in this limit, and is given by the expression

$$\begin{aligned}\zeta = z &- 2z_0\lambda\sqrt{\frac{1 - \lambda^2\eta^2}{1 - \lambda^2}} \\ &- z_0\left(2\eta - \frac{1}{\eta}\right) F\left[\sin^{-1}(\lambda\eta), \eta^{-1}\right] + 2z_0\eta z_0 E\left[\sin^{-1}(\lambda\eta), \eta^{-1}\right],\end{aligned} \tag{71}$$

The z value of the intersection between the isochrone surface and the **z**-axis, z_a, is implicitly given by the equation

$$\frac{z_a}{z_0} = \frac{\zeta}{z_0} - \frac{1}{2}\ln\frac{z_a - z_0}{z_a + z_0}, \tag{72}$$

Now, to fix the divergent field, $\mathbf{B}^t$ for any point on an isochrone with $z_a < z_t$, where $z_t > z_0$ is a user-defined threshold, is multiplied by the following amplification factor

$$R = \sqrt{\frac{\left(1 - \frac{z_0^2}{z_a^2}\right)\left(1 - \frac{z_0^2}{\alpha^2 z_a^2}\right)}{\left(1 - \frac{z_0^2}{\alpha_2 z_t^2}\right)\left(1 - \frac{z_0^2}{z_t^2}\right)}}, \tag{73}$$

where

$$\alpha = \sqrt{\frac{K^2(z_t^2 - z_0^2) - z_0^2}{K^2(z_t^2 - z_0^2) - z_t^2}}, \tag{74}$$

and K is the desired value of $B^t(s = 0, z = z_0)/B^t(z \to \infty)$. Figure 9 depicts the most relevant geometrical shapes in this procedure.

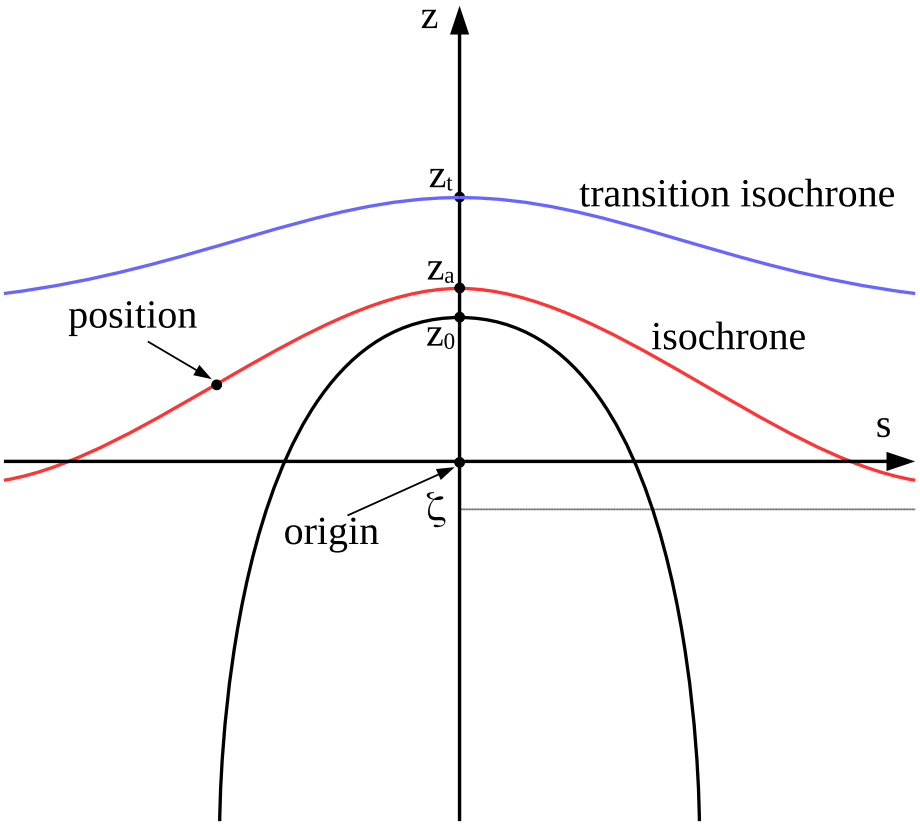

Figure 9: A 2D projection of a Rankine half-body, representing the HP, along with a particular isochrone (red) placed within the HP and the transition isochrone (blue).

The code computes **u** and **B** in cylindrical coordinates relative to $\mathbf{r}_0$ with a **z**-axis (the HP symmetry axis) parallel to $\mathbf{u}_0$ and converts them to the global Cartesian system. $B^t_{s,0}$ and $B^t_{\varphi,0}$ are computed using the first and second components of `B0`, while B^l_0 is the third component of `B0`. The derivatives for this background are computed numerically and the maximum spatial displacement per step is constant. Figure 10 shows a 2D cross-section of the magnetic field magnitude obtained with this model, along with some magnetic fieldlines.

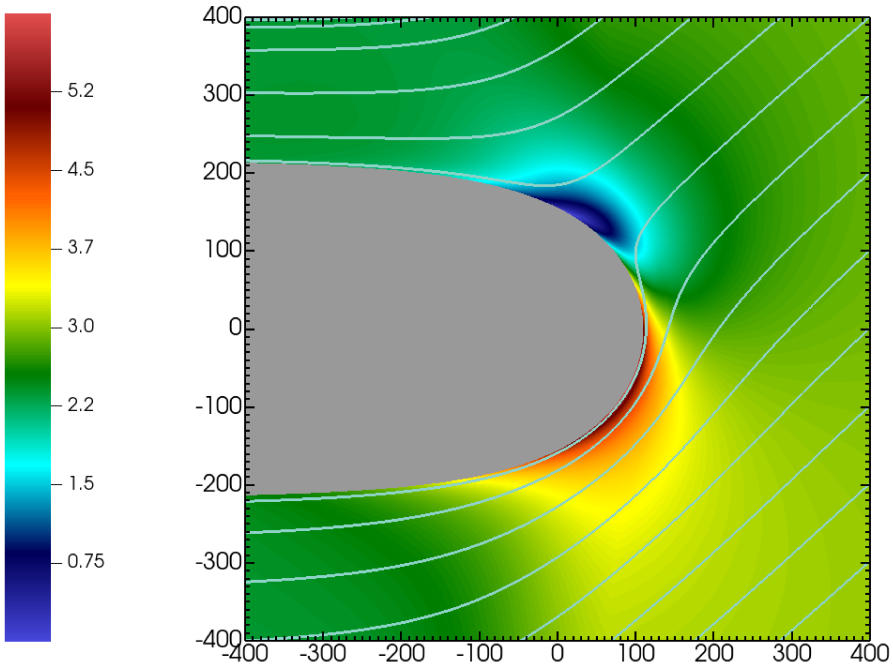

Figure 10: 2D cross-section magnetic field magnitude along the **xz**-plane. The far-away field has a strength of 3 μG and an angle of 40° with respect to the flow direction (negative **z**-axis). The gray curves show the magnetic field streamlines, draping around the HP surface (Rankine half-body). Note the magnetic field stretching (blue) and compression (red) regions.

On top of the inputs already required by `BackgroundBase`, the following parameter should be provided when constructing this module:

`z_nose` : Distance between the Sun and the stagnation point (tip) of the HP.
Possible values: Any positive double.

The file `$SPECTRUM/src/background_vlism_bochum.hh` defines the following preprocessor macros and global constants:

`MOD_TYPE` : This macro controls what type of amplification factor to use.
Possible values: `0`, `1`, `2`, and `3`.

`MOD_RPOS` : This macro controls whether to scale the field relative to $s = 0$ or $s = \infty$.
Possible values: `0` and `1`.

`ztr` : This is the threshold z value for the isochrone intersection with the **z**-axis to apply the amplification factor. It is only used when `MOD_TYPE` is not `0`.
Possible values: Any double.

`scB` : Desired ratio of the transverse magnetic field at the stagnation point to the transverse magnetic field far away from the HP. It is only used when `MOD_TYPE` is `3`.
Possible values: Any double.

4.8 MHD Shock Background

The `BackgroundShock` class implements an infinitely thin, and therefore discontinuous, planar, moving shock. The flow and magnetic are both described by the equation

$$\mathbf{F} = \begin{cases} \mathbf{F}_0, & \mathbf{r} > \mathbf{r}_0 + vt\mathbf{n}, \\ \mathbf{F}_1, & \mathbf{r} < \mathbf{r}_0 + vt\mathbf{n}, \end{cases} \tag{75}$$

where $\mathbf{F}_0$ and $\mathbf{F}_1$ represent the upstream and downstream values, respectively, and $\mathbf{n}$ and v are the shock normal and speed, respectively. Meanwhile, the electric field is given by

$$\mathbf{E} = -\frac{\mathbf{u} \times \mathbf{B}}{c}. \tag{76}$$

The derivatives are zero everywhere, unless numerically computed, at which point they will be non-zero only near the shock front. The maximum spatial displacement per step is constant.

The parameters provided to `BackgroundBase` represent the upstream quantities. On top of the inputs already required by `BackgroundBase`, the following parameters should be provided when constructing this module:

`r0_shock` : Initial position of the shock.
Possible values: Any GeoVector.

`n_shock` : Shock normal.
Possible values: Any GeoVector.

`v_shock` : Shock speed.
Possible values: Any double.

`u1` : Downstream plasma flow.
Possible values: Any GeoVector.

`B1` : Downstream magnetic field.
Possible values: Any GeoVector.

4.9 Smooth MHD Shock Background

→ Subsection 4.8

The `BackgroundSmoothShock` class is derived from `BackgroundShock` and adds a finite transition layer between the up and downstream regions. The flow and magnetic field are updated to use the equation

$$\mathbf{F} = \phi(s)\mathbf{F}_0 + [1 - \phi(s)]\,\mathbf{F}_1, \quad s = \frac{\mathbf{r} - \mathbf{r}_0 + vt\mathbf{n}}{w}, \tag{77}$$

where ϕ is a function that transitions from 0 to 1 in the interval $[-\frac{1}{2}, \frac{1}{2}]$. The derivatives for the flow and magnetic field are

$$\nabla\mathbf{F} = \frac{1}{w}\phi'(s)\,(\mathbf{n}\mathbf{F}_0 - \mathbf{n}\mathbf{F}_1)\,, \tag{78}$$

$$\frac{d\mathbf{F}}{dt} = \frac{v}{w}\phi'(s)\,(\mathbf{F}_1 - \mathbf{F}_0)\,, \tag{79}$$

while the electric field derivatives are

$$\nabla \mathbf{E} = -\frac{\nabla \mathbf{u} \times \mathbf{B} + \mathbf{u} \times \nabla \mathbf{B}}{c}, \tag{80}$$

$$\frac{d\mathbf{E}}{dt} = -\frac{1}{c}\left(\frac{d\mathbf{u}}{dt} \times \mathbf{B} + \mathbf{u} \times \frac{d\mathbf{B}}{dt}\right). \tag{81}$$

The maximum spatial displacement per step is constant.

The possible transition functions in the code are polynomials of order 1 (continous, non-differentiable)

$$\phi(s) = \begin{cases} 0, & s < -\frac{1}{2}, \\ s + \frac{1}{2}, & -\frac{1}{2} \leq s \leq \frac{1}{2}, \\ 1, & s > \frac{1}{2}, \end{cases} \tag{82}$$

order 3 (once differentiable),

$$\phi(s) = \begin{cases} 0, & s < -\frac{1}{2} \\ \left(s + \frac{1}{2}\right)^2 \left[3 - 2\left(s + \frac{1}{2}\right)\right], & -\frac{1}{2} \leq s \leq \frac{1}{2} \\ 1, & s > \frac{1}{2} \end{cases} \tag{83}$$

order 5 (twice differentiable)

$$\phi(s) = \begin{cases} 0, & s < -\frac{1}{2}, \\ \left(s + \frac{1}{2}\right)^3 \left[10 - 15\left(s + \frac{1}{2}\right) + 6\left(s + \frac{1}{2}\right)^2\right], & -\frac{1}{2} \leq s \leq \frac{1}{2}, \\ 1, & s > \frac{1}{2}, \end{cases} \tag{84}$$

and order 7 (three times differentiable)

$$\phi(s) = \begin{cases} 0, & s < -\frac{1}{2}, \\ \left(s + \frac{1}{2}\right)^4 \left[35 - 84\left(s + \frac{1}{2}\right)\right. & \\ \qquad \left. +70\left(s + \frac{1}{2}\right)^2 - 20\left(s + \frac{1}{2}\right)^3\right], & -\frac{1}{2} \leq s \leq \frac{1}{2}, \\ 1, & s > \frac{1}{2}, \end{cases} \tag{85}$$

as well as a hyperbolic tangent (smooth)

$$\phi(s) = \frac{1}{2}\left[1 + \tanh(hs)\right] \tag{86}$$

where $h > 1$ is an empirical factor to make the jump between 0 and 1 more closely fit within the interval $[-\frac{1}{2}, \frac{1}{2}]$, since the hyperbolic tangent technically transitions over $[-\infty, \infty]$. Figure 11 shows the transition functions available in this module.

On top of the inputs already required by `BackgroundBase` and `BackgroundShock`, the following parameter should be provided when constructing this module:

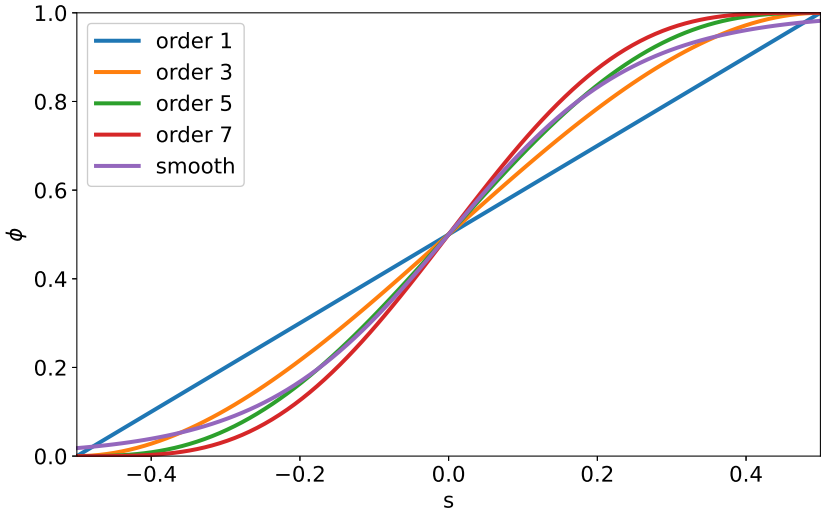

Figure 11: Transition functions available in this module. The smooth curve was drawn using $h = 4$.

`w_shock` : Shock width.
Possible values: Any double.

The file `$SPECTRUM/src/background_smooth_shock.hh` defines the following preprocessor macros and global constants:

`SMOOTH_SHOCK_ORDER` : This macro controls what transition function to use.
Possible values: Any integer.

`SMOOTHSHOCK_DERIVATIVE_METHOD` : This macro controls whether to compute the derivatives for this background analytically or numerically.
Possible values: `0` and `1`.

`tanh_width_factor` : This is the scaling factor for the hyperbolic tangent transition function. It is only used when `SMOOTH_SHOCK_ORDER` is not `0`, `1`, `2`, or `3`.
Possible values: Any positive double.

4.10 Superposition of Waves Background

The `BackgroundWaves` generates a static turbulent magnetic field as a superposition of linearly polarized waves, in addition to a uniform magnetic field. At this time the velocity and electric fields are set to zero. Four turbulent geometries are offered: Alfvén or slab (A), transverse 2D (T), longitudinal 2D (L), and isotropic (I). A set of waves comprising one turbulent geometry must be encoded in a TurbProp structure. The intended use is in simulations based on the Lorentz trajectory type that are commonly employed to test the particle diffusion theories and analytic models.

→ Subsection 9.9
→ Subsection 3.3

The setup routine creates two auxilliary coordinate frames for each wave mode. The first is the B-frame, whose **z**-axis is aligned with $\mathbf{B}_0$ (the other two axes are arbitrary). The second frame is the K-frame, where the **z**-axis is parallel to the wave-vector of the mode, the **y**-axis is along $\mathbf{B}_0 \times \mathbf{k}$ and the **x**-axis is normal to those two. In the K-frame only the x and y components of the fluctuating field $\delta\mathbf{B}$ are non-zero because $\mathbf{k} \cdot \delta\mathbf{B} = 0$. Therefore, for each mode n in that frame we may

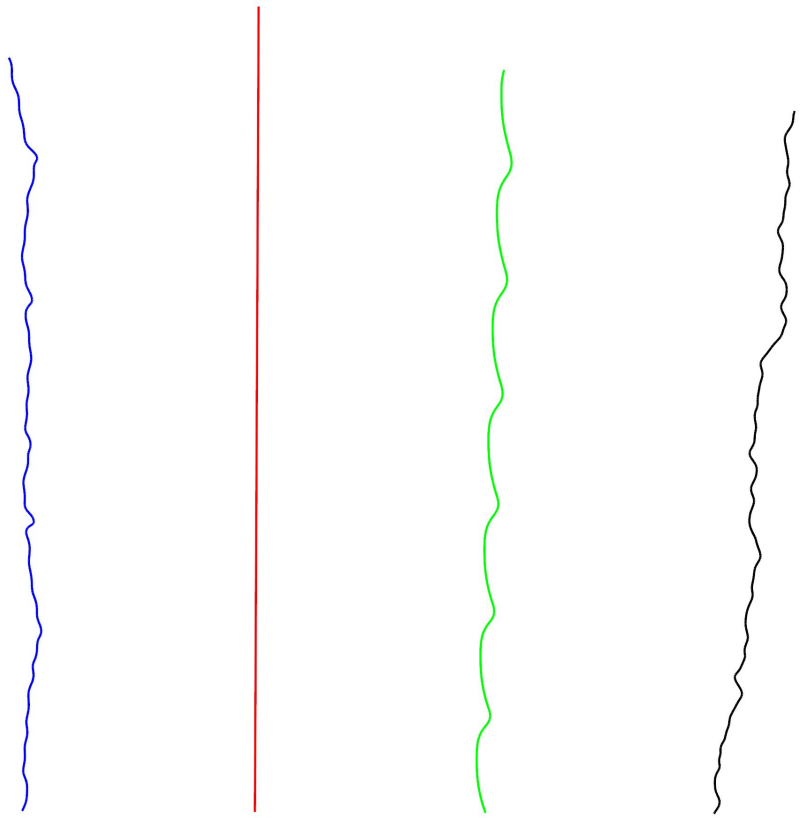

Figure 12: Sample fieldlines from random realizations by superposing 50 waves with only A modes (black), only T modes (green), only L modes (red), and only I modes (blue).

write

$$\delta B_{x,n} = A_n \cos\alpha_n \cos(k_n z + \omega_n t + \psi_n), \tag{87}$$
$$\delta B_{y,n} = -A_n \sin\alpha_n \sin(k_n z + \omega_n t + \psi_n), \tag{88}$$

where A_n is the amplitude, α_n is the polarization angle, k_n is the wavenumber, and ψ_n is the initial phase, randomly distributed between 0 and 2π. The term responsible for the wave propagation, $\omega_n t$, is not implemented at this time. In the future, additional functionality will be added to calculate $\omega_n(k_n)$ based on a specific dispersion relation (possibly, in derived classes).

The unormalized power spectral density (PSD) of each type of turbulence is a two-interval turbulence consisting of a flat energy range and a decaying inertial range with a smooth transition,

$$P(k) = \frac{k^d}{1 + (l_0 k)^{d+\gamma}} \tag{89}$$

where d is the dimensionality of the PSD, l_0 is the bendover length, and γ is the slope of the inertial range. The normalized amplitudes A_n are found by dividing $P(k_n)$ by the magnetic variance, $\langle \delta B^2 \rangle$, and taking a square root. Also, the energy range slope depends on the value of d, which ensures that the integral converges.

Wave modes are *logarithmically* spaced between a smallest, $k_{\min}$, and a largest wavenumbers, $k_{\max}$. Each wave mode has a specific propagation direction given by the angles θ_n and φ_n that give the polar coordinates of $\mathbf{k}_n$ in the B-frame. These are calculated as shown in the table below, which lists the parameters inherent to each geometry.

Geometry	d	$\cos\theta_n$	φ_n	α_n
A	0	1	0	$2\pi R_u$
T	1	0	$2\pi R_u$	$2\pi R_u$
L	1	0	$2\pi R_u$	0
I	2	$2R_u - 1$	$2\pi R_u$	$2\pi R_u$

R_u is a random number from a uniform distribution on $[0, 1]$.

The setup routine saves the triplet of unit vectors that form the basis of the simulation frame as seen from the K-frame (`basis`). The field evaluator first generates the field in the K-frame and then transforms it into the simulation frame by projection onto the basis. The spatial derivatives are also easily computed analytically.

The maximum spatial displacement per step is computed with the formula

$$\Delta r_{\max} = \min\left\{\Delta r_{\max,0}, \frac{2\pi}{\tilde{k}_{\max}}\right\}, \tag{90}$$

where

$$\tilde{k}_{\max} = \max\left\{k^A_{\max}, k^T_{\max}, k^L_{\max}, k^I_{\max}\right\}. \tag{91}$$

On top of the inputs already required by `BackgroundBase`, the following parameter should be provided when constructing this module:

`properties_A` : A `TurbProp` structure holding the turbulent properties of the Alfvénic fluctuations.
Possible values: A TurbProp object.

`properties_T` : A `TurbProp` structure holding the turbulent properties of the 2D transverse fluctuations.
Possible values: A TurbProp object.

`properties_L` : A `TurbProp` structure holding the turbulent properties of the 2D longitudinal fluctuations.
Possible values: A TurbProp object.

`properties_I` : A `TurbProp` structure holding the turbulent properties of the isotropic fluctuations.
Possible values: A TurbProp object.

Here is a typical sequence for loading the turbulent properties into a DataContainer object for generating a power spectrum:

```
DataContainer container;
TurbProp properties;
properties.kmin = M_2PI / (10.0 * GSL_CONST_CGSM_ASTRONOMICAL_UNIT /
    unit_length_fluid);
properties.kmax = M_2PI / (0.001 * GSL_CONST_CGSM_ASTRONOMICAL_UNIT /
    unit_length_fluid);
properties.l0 = 0.1 * GSL_CONST_CGSM_ASTRONOMICAL_UNIT / unit_length_fluid;
properties.n_waves = 200;
properties.variance = Sqr(0.1 * Bmag);
properties.slope = -5.0 / 3.0;
container.Insert(properties);
```

Note that all four geometries must be defined, or the code will crash. If some geometries are not needed, it is best to set their `n_waves` parameters to zero to prevent wasting CPU cycles.

4.11 Cylindrical Obstacle Background

The `BackgroundCylindricalObstacle` class implements a magnetic field around a cylindrical obstacle. Outside of the obstacle, which has a radius of a and an axis of symmetry $\mathbf{e}$ intersecting $\mathbf{r}_0$, the fields are

$$\mathbf{u} = 0, \tag{92}$$

$$\mathbf{B} = \mathbf{B}_0 - \frac{a^2}{r_\perp^2}\left[2\frac{(\mathbf{r}\cdot\mathbf{B}_0)\,\mathbf{r}}{r_\perp^2} - \mathbf{B}_0\right], \tag{93}$$

$$\mathbf{E} = 0, \tag{94}$$

where $\mathbf{r}$ is the position relative to $\mathbf{r}_0$ and $\mathbf{r}_\perp$ is the vector starting at $\mathbf{r}$ perpendicular to $\mathbf{e}$. The only non-zero derivatives are the gradient of the magnetic field and its magnitude, which have not yet been implemented analytically in the code. Inside of the obstacle, all fields and derivatives are zero.

Note that only the portion of `B0` perpendicular to the cylinder's axis is utilized when constructing this class. The maximum spatial displacement per step is computed with the formula

$$\Delta r_{\max} = \min\left\{\Delta r_{\max,0}, \delta r_\perp\right\}, \tag{95}$$

where δ is a user-specified constant and $\mathbf{r}_\perp$ is component of $\mathbf{r}$ perpendicular to $\mathbf{m}$. Figure 13 shows a 2D cross-section of the magnetic field magnitude obtained with this model, along with some magnetic fieldlines.

On top of the inputs already required by `BackgroundBase`, the following parameters should be provided when constructing this module:

`axis` : Symmetry axis of the cylindrical obstacle.
Possible values: Any GeoVector.

`r_cylinder` : Radius of the cylindrical obstacle.
Possible values: Any positive double.

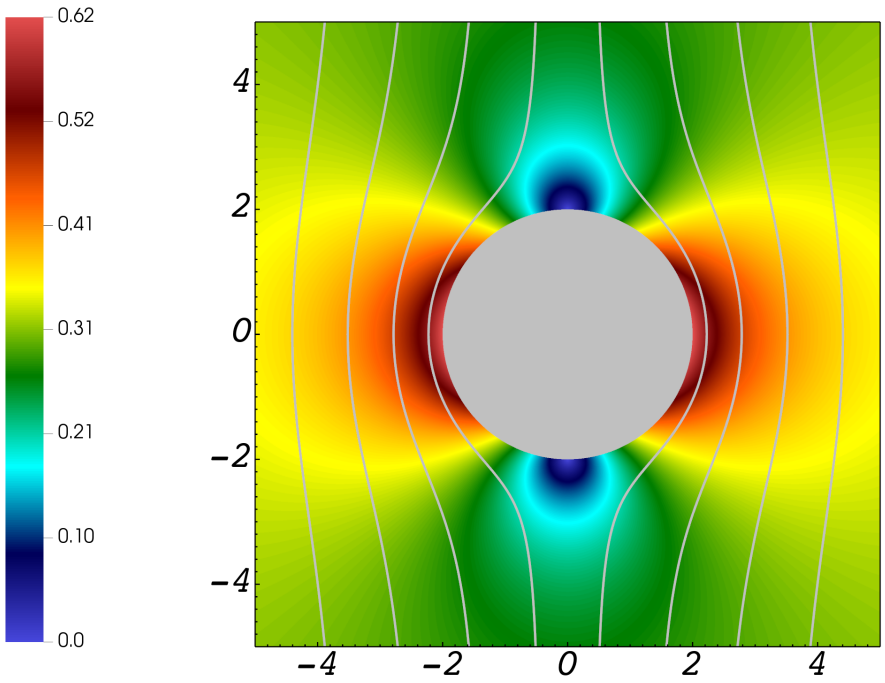

Figure 13: 2D cross-section magnetic field magnitude along the **xz**-plane. $\mathbf{r}_0$ is the origin, $\mathbf{B}_0$ is parallel to **z** with a strength of 0.31 G, and **e** is parallel to **y**. The gray curves show the magnetic field streamlines.

`dmax_fraction` : Fraction of distance from the origin to use as a threshold for the calculation of `_spdata.dmax`. Possible values: Any positive double.

The file `$SPECTRUM/src/background_cylindrical_obstacle.hh` defines the following pre-processor macro:

`CYLINDRICAL_OBSTACLE_DERIVATIVE_METHOD` : This macro controls whether to compute the derivatives for this background analytically or numerically.
Possible values: `0` and `1`.

4.12 Magnetized Cylinder Background

→ Subsection **??**

The `BackgroundMagnetizedCylinder` class is derived from `BackgroundCylindricalObstacle` and it implements the magnetic field of a uniformly magnetized cylinder, both outside and inside of the cylinder. The magnetic field everywhere is simply

$$\mathbf{B} = \mathbf{B}_0 - \mathbf{B}_c \tag{96}$$

where $\mathbf{B}_c$ is the magnetic field from a cylindrical obstacle, given by (93) outside of the cylinder (and zero inside). The only derivative that changes is

$$\nabla\mathbf{B} = -\nabla\mathbf{B}_c, \tag{97}$$

outiside of the cylinder, but it remains zero inside. Construction of this class does not need any additional inputs or parameters beyond those already required by `BackgroundCylindricalObstacle`. Figure 14 shows a 2D cross-section of the magnetic field magnitude obtained with this model, along with some magnetic fieldlines.

The file `$SPECTRUM/src/background_magnetized_cylinder.hh` defines the following pre-processor macro:

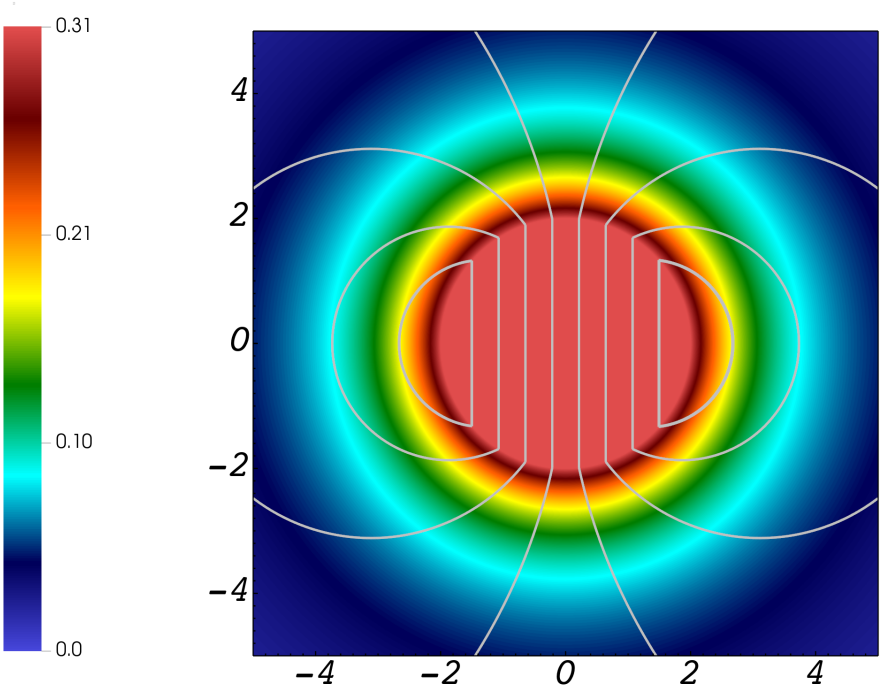

Figure 14: 2D cross-section magnetic field magnitude along the **xz**-plane. $\mathbf{r}_0$ is the origin, $\mathbf{B}_0$ is parallel to **z** with a strength of 0.31 G, and **e** is parallel to **y**. The gray curves show the magnetic field streamlines.

`MAGNETIZED_CYLINDER_DERIVATIVE_METHOD` : This macro controls whether to compute the derivatives for this background analytically or numerically.
Possible values: `0` and `1`.

4.13 Spherical Obstacle Background

The `BackgroundSphericalObstacle` class implements a magnetic field around a spherical obstacle. Outside of the obstacle, which has a radius of a and is centered at r_0, the fields are

$$\mathbf{u} = 0, \tag{98}$$

$$\mathbf{B} = \mathbf{B}_0 - \frac{3(\mathbf{m}\cdot\mathbf{r})\mathbf{r} - r^2\mathbf{m}}{r^5}, \tag{99}$$

$$\mathbf{E} = 0, \tag{100}$$

where $\mathbf{m} = a^3\mathbf{B}_0$ and $\mathbf{r}$ is the position relative to $\mathbf{r}_0$. The only non-zero derivatives are the gradient of the magnetic field and its magnitude, which is

$$\nabla\mathbf{B} = -3\frac{r^2\left(\mathbf{mr}+\mathbf{rm}\right) + \mathbf{m}\cdot\mathbf{r}\left(r^2\mathrm{I} - 5\mathbf{rr}\right)}{r^7}, \tag{101}$$

$$\nabla B = \nabla\mathbf{B}\cdot\mathbf{b}, \tag{102}$$

where I is the identity matrix. Inside of the obstacle, all fields and derivatives are zero.

$$\Delta r_{\max} = \min\left\{\Delta r_{\max,0}, \delta r\right\}, \tag{103}$$

where δ is a user-specified constant. Figure 15 shows a 2D cross-section of the magnetic field magnitude obtained with this model, along with some magnetic fieldlines.

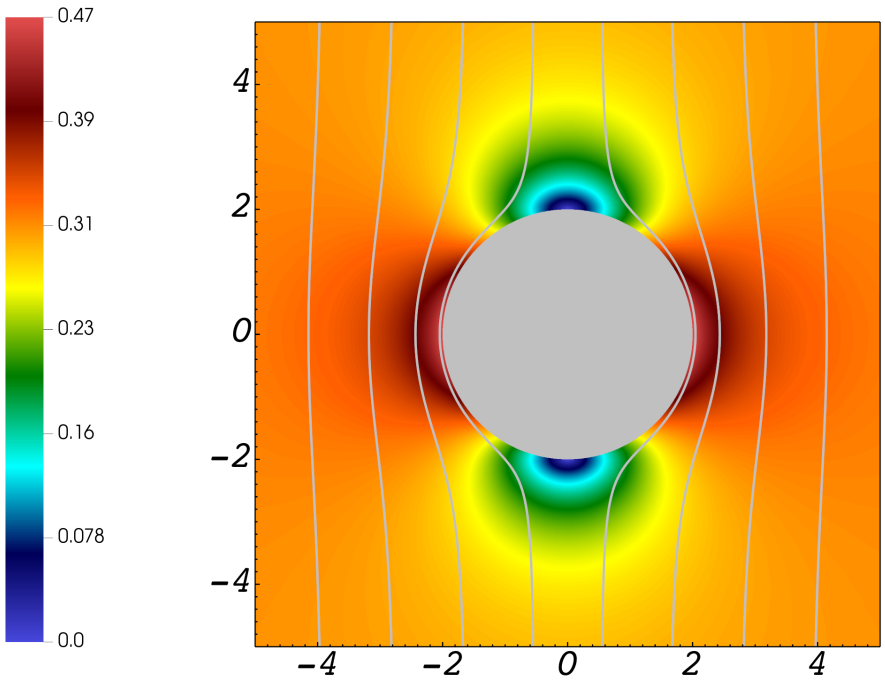

Figure 15: 2D cross-section magnetic field magnitude along the **xz**-plane. $\mathbf{r}_0$ is the origin and $\mathbf{B}_0$ is parallel to **z** with a strength of 0.31 G. The gray curves show the magnetic field streamlines.

On top of the inputs already required by `BackgroundBase`, the following parameters should be provided when constructing this module:

`r_sphere` : Radius of the spherical obstacle.
Possible values: Any positive double.

`dmax_fraction` : Fraction of distance from the origin to use as a threshold for the calculation of `_spdata.dmax`. Possible values: Any positive double.

The file `$SPECTRUM/src/background_spherical_obstacle.hh` defines the following pre-processor macro:

`SPHERICAL_OBSTACLE_DERIVATIVE_METHOD` : This macro controls whether to compute the derivatives for this background analytically or numerically.
Possible values: `0` and `1`.

4.14 Magnetized Sphere Background

→ Subsection 4.13

The `BackgroundMagnetizedSphere` class is derived from `BackgroundSphericalObstacle` and it implements the magnetic field of a uniformly magnetized sphere, both outside and inside of the sphere. The magnetic field everywhere is simply

$$\mathbf{B} = \mathbf{B}_0 - \mathbf{B}_s \tag{104}$$

where $\mathbf{B}_s$ is the magnetic field from a spherical obstacle, given by (99) outside of the sphere and zero inside. The only derivative that changes is

$$\nabla\mathbf{B} = -\nabla\mathbf{B}_s, \tag{105}$$

outiside of the sphere, but it remains zero inside. Construction of this class does not need any additional inputs or parameters beyond those already required by

`BackgroundSphericalObstacle`. Figure 16 shows a 2D cross-section of the magnetic field magnitude obtained with this model, along with some magnetic fieldlines.

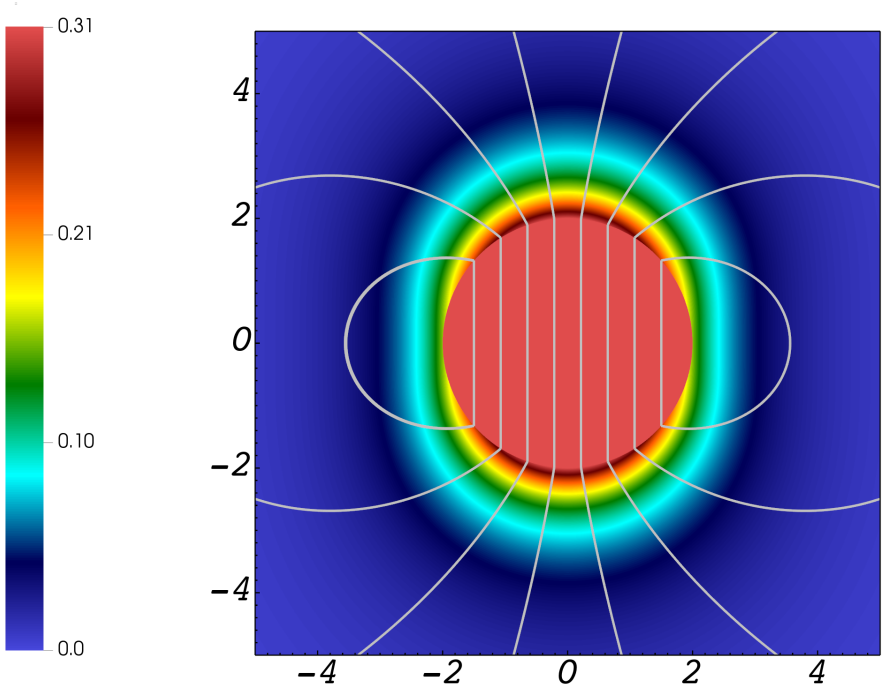

Figure 16: 2D cross-section magnetic field magnitude along the **xz**-plane. $\mathbf{r}_0$ is the origin and $\mathbf{B}_0$ is parallel to **z** with a strength of 0.31 G. The gray curves show the magnetic field streamlines.

The file `$SPECTRUM/src/background_magnetized_sphere.hh` defines the following pre-processor macro:

`MAGNETIZED_SPHERE_DERIVATIVE_METHOD` : This macro controls whether to compute the derivatives for this background analytically or numerically.
Possible values: `0` and `1`.

4.15 Server Background

The `BackgroundServer` class is a stub (common ancestor) for all discretized backgrounds stored on distributed memory. This class is not meant to be implemented directly. The fields and their derivatives for any Background object derived from `BackgroundServer` are calculated by interpolation according to the grid geometry and the interpolation settings specified during the configuration stage. Interpolation is locally performed by the SimW processes through the `ServerFront` object of `BackgroundServer`. The `ServerFront` objects use MPI to communicate with the `ServerBack` objects of the SimS processes. The latter feed blocks of data from a global mesh on the domain to the former so that pseudo-trajectories can hopefully advance several steps between MPI messages, yielding faster execution times.

Although this class does not need any additional parameters beyond those required by `BackgroundBase`, constructing any Background object derived from `BackgroundServer` involves a third argument when calling `AddBackground`, which is the name, without extension, of the file containing the discretized data in binary format (extension `.out`) and meta-data in ascii format (extension `.info`).[8] Keep in

[8] Both of these files should have the same name, disregarding the extension.

mind that `EvaluateBackground` will return an error if the fields are requested for a position where interpolation is not possible, such as outside of the specified bounds for the data's domain according to the meta-data file. It is the responsibility of the user to make sure that pseudo-particles are initialized within the discretized domain and absorbing or reflecting spatial boundary conditions are set such that they never exit the regions where interpolation is possible.

→ Subsection 7.3

The `\$SPECTRUM\src\server_base.hh` header file contains important macros for server communications.

`SERVER_VAR_INDEX_RHO` : This macro controls the index for the mass density variable, if provided by the server.
Possible values: Any non-negative integer.

`SERVER_VAR_INDEX_DEN` : This macro controls the index for the number density variable, if provided by the server.
Possible values: Any non-negative integer.

`SERVER_VAR_INDEX_MOM` : This macro controls the starting index for the bulk momentum vector, if provided by the server.
Possible values: Any non-negative integer.

`SERVER_VAR_INDEX_FLO` : This macro controls the starting index for the bulk flow vector, if provided by the server.
Possible values: Any non-negative integer.

`SERVER_VAR_INDEX_MAG` : This macro controls the starting index for the magnetic field vector, if provided by the server.
Possible values: Any non-negative integer.

`SERVER_VAR_INDEX_ELE` : This macro controls the starting index for the electric field vector, if provided by the server.
Possible values: Any non-negative integer.

`SERVER_VAR_INDEX_REG` : This macro controls the starting index for the region indicator variables, if provided by the server.
Possible values: Any non-negative integer.

`SERVER_NUM_INDEX_REG` : This macro controls the number of region indicator variables copied from the server.
Possible values: Any non-negative integer.

`SERVER_VAR_INDEX_PRE` : This macro controls the index for the thermal pressure variable, if provided by the server.
Possible values: Any non-negative integer.

Uncommenting any of these macros will disable the corresponding background quantity to speed up server communications. There are also important global constants defined in this same header file.

`unit_length_server` : Length unit for server data.
Possible values: Any positive double.

`unit_number_density_server` : Number density unit for server data.
Possible values: Any positive double.

`unit_velocity_server` : Velocity unit for server data.
Possible values: Any positive double.

`unit_magnetic_server` : Magnetic field unit for server data.
Possible values: Any positive double.

`unit_electric_server` : Electric field unit for server data.
Possible values: Any positive double.

`unit_pressure_server` : Pressure unit for server data.
Possible values: Any positive double.

The rest global constants are tags utilized for MPI communication that should not be edited.

4.16 Cartesian Background

→ Subsection 4.15

The `BackgroundCartesian` class is derived from `BackgroundServer` and implements a static Cartesian discretized background on distributed memory. The Cartesian mesh is uniform along each dimension, but the size and resolution of the domain along each dimension can be different. At the configuration stage, the user can instruct this class to perform zeroth (discontinuous) or first order (trilinear) interpolation on block data when ghost cells are disabled. Note that B and $\mathbf{B}$ are interpolated separately. This avoids computing artificially small magnetic field magnitudes due to interpolation between fields that reverse directions within one layer of cells, which can adversely affect drift and diffusion calculations. Figure 17 illustrates this by showing a 2D cross-section of the magnetic field magnitude obtained by discretizing the Parker Spiral with a termination shock background, plotted with first order interpolation. The field was modified using a Fisk polar correction, as in Equation (52), as well as a wavy heliospheric current sheet according to Equation (47).

→ Subsection 1.4

! →

→ Subsection 4.6

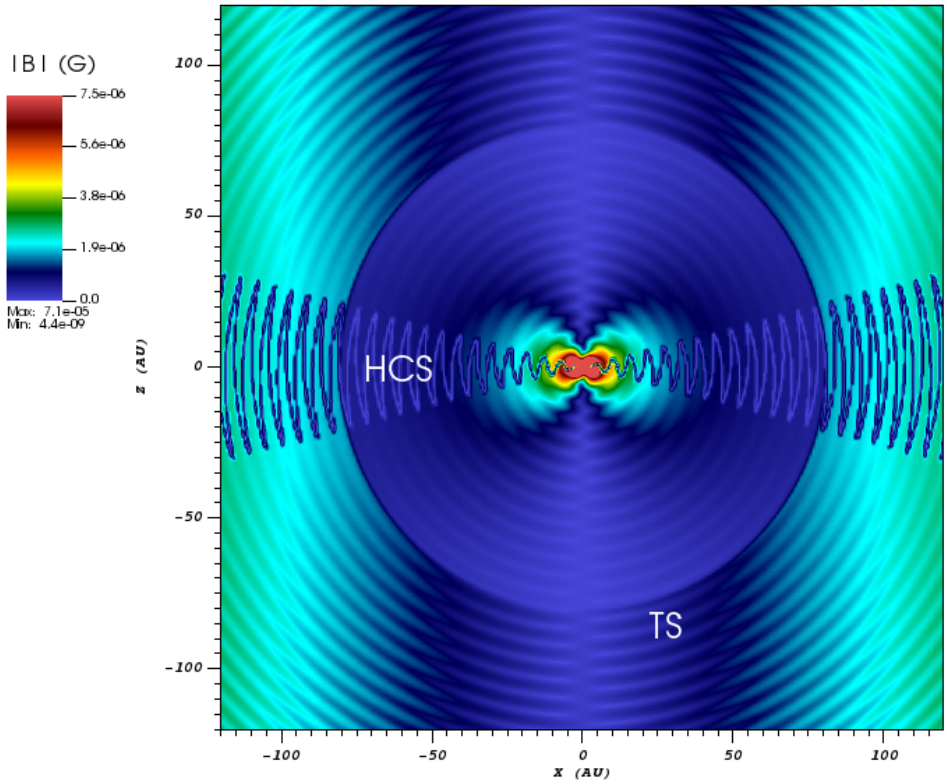

Figure 17: 2D cross-section magnetic field magnitude of a discretized Parker Spiral with a termination shock (TS), Fisk polar correction, and wavy heliospheric current sheet (HCS). Notice the extremely low magnitude at the surface where the field reverses sign.

The code can also be configured so that the server returns pre-interpolated first order accurate point data directly instead of blocks, though this is mainly for testing purposes, as it is much slower than sending block data and provides no precision

benefit. Construction of this class does not need any additional inputs or parameters beyond those already required by `BackgroundServer`.

The `ReaderCartesian` object is responsible for reading the file(s) containing the discretized fields. The format of the meta-data file should be as follows: The first line should contain 3 integers separated by spaces, the number of blocks per dimension. The second line should contain 3 doubles separated by spaces, the lower bounds for the domain in each dimension. The third line should contain 3 doubles separated by spaces, the upper bounds for the domain in each dimension. The fourth and last line should contain a single integer, the number of variables to be read within each cell. Note that each vector field counts as 3 separate variables, one for each component, so if, for example, the binary data file specifies the density (1), bulk flow (3), and the magnetic field (3), the last line of the meta-data file should indicate 7.

The `\$SPECTRUM\src\reader_cartesian.hh` header file contains an important global constant:

`block_size_cartesian` : MultiIndex object specifying the number of cells per dimension in a block.
Possible values: Any MultiIndex.

4.17 BATL Background

→ Subsection 4.16

The `BackgroundBATL` class is derived from `BackgroundCartesian` and implements an AMR Cartesian discretized background on distributed memory. The binary files containing the mesh data for this type of background should be in BATL format, like

→ Subsection 1.4

those from the output of a BATS-R-US simulations. At the configuration stage, the suer can instruct this class to perform zeroth (discontinuous) or first order (trilinear) interpolation on block data when a single layer of ghost cells is enabled. The code can also be configured so that the server retuns pre-interpolated second order accurate point data directly instead of blocks when ghost cells are disabled, though for many applications this is prohibitingly slower than sending block data. Construction of this class does not need any additional inputs or parameters beyond those already required by `BackgroundCartesian`. Figure 18 shows a 2D cross-section of the magnetic field magnitude obtained from an MHD model, plotted with first order interpolation.

The file `\$SPECTRUM\src\spectrum_interface.f90` contains some routines to interface with the source files from BATL. Refer to their documentation to learn about the formatting for their data and meta-data files.

The `\$SPECTRUM\src\block_batl.hh` header file contains an important global constants:

`n_variables_batl` : Maximum number of variables to read from file.
Possible values: Any positive integer.

`block_size_batl` : MultiIndex object specifying the number of cells per dimension in a block.
Possible values: Any MultiIndex.

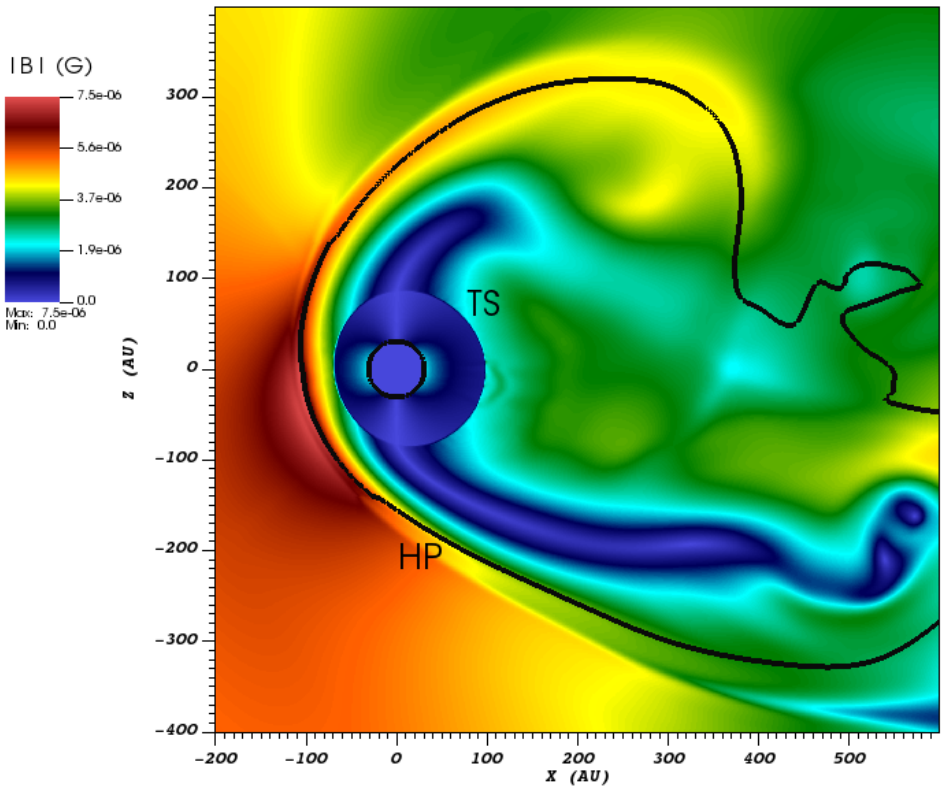

Figure 18: 2D cross-section magnetic field magnitude of a discretized heliosphere model. The heliopause boundary (HP) and was calculated as an isosurface of one of the `_spdata.region` components.

5 Distribution Modules

In this section, we describe the binning distributions modules currently supported within the existing SPECTRUM architecture.

5.1 Base Distribution

The `DistributionBase` class is not meant to be used directly, but rather contains and defines the general functionality shared by all the application specific boundary condition modules, described in the following subsections. It is inherited from the `Params` class.

→ Subsection 9.7

The parameters common to all (application) Distribution objects are listed in Subsection 5.2. These parameters are passed through a DataContainer object in the public function

→ Subsection 9.2

```
void SetupObject(const DataContainer& cont_in);
```

which calls the protected

```
virtual void SetupDistribution(bool construct);
```

A Simulation or Trajectory Integrator (TI) object would call the function

```
void AddDistribution(const BoundaryBase& boundary_in, const DataContainer&
    container_in);
```

to construct a distribution module.

Distribution objects are linked to TI objects through the latter's `distributions` pointer. The file `$SPECTRUM/src/distribution_base.hh` defines series of global constants beginning with `DISTRO_` that help sort the distribution modules. Distributions are dynamic objects that collect cumulative information about pseudo-trajectories during or at the end of their integration. Multiple distributions can be activated during a simulation, in order to maximize the userful output.

→ Subsection 3.1

Data collection is triggered when a pseudo-trajectory crosses a boundary and passes the initial and current pseudo-particle state, along with the indices from the action vector of the corresponding boundary condition module to the distribution modules via the public function

→ Section 7

```
void ProcessTrajectory(double t1, const GeoVector& pos1, const GeoVector&
    mom1, const SpatialData& spdata1, double t2, const GeoVector& pos2,
    const GeoVector& mom2, const SpatialData& spdata2, int action_in);
```

The indices instruct the distribution modules how to bin the relevant data by specifying which function within the `ActionTable` array to execute. The internal, protected functions involved in data binning are

```
virtual void EvaluateValue(void);
virtual void EvaluateWeight(int action_in);
virtual void AddEvent(void);
virtual void AddRecord(void);
```

! → Although any number of Distribution objects can be constructed, each is at most 3 dimensional, though most are 1 or 2 dimensional.[9] The attribute `dims`, treated as a bitset, keeps track of which dimensions are active. The restriction on the maximum number of dimensions inevitably results in a collapse/projection of higher dimensional information (e.g. 6D phase-space state). Also, the process of binning

[9] This refers to the dimensionality of the binned values.

averages data over all trajectories, erasing the individual footprint from each. For most applications, this is enough and, in fact, efficient, since only a few relatively small arrays need to be stored in memory to store the desired simulation output. However, it is sometimes helpful to have access more detailed, individual pseudo-trajectory information, prior to binning. To offer more flexibility to users in this aspect, a distribution module can keep raw (unbinned) data for output and custom post-processing. The public routines related to data output (in ascii format) are

```
virtual void Print1D(int ijk, const std::string& file_name, bool
    phys_units) const;
virtual void Print2D(int ijk1, int ijk2, const std::string& file_name, bool
    phys_units) const;
virtual void PrintRecords(const std::string& file_name, bool phys_units)
    const;
```

which are employed when a Simulation object calls

```
virtual void PrintDistro1D(int distro, int ijk, const std::string&
    file_name, bool phys_units) const;
virtual void PrintDistro2D(int distro, int ijk1, int ijk2, const
    std::string& file_name, bool phys_units) const;
virtual void PrintRecords(int distro, const std::string& file_name, bool
    phys_units) const;
```

A simulation regularly outputs all the data (including internal parameters) of its Distribution objects via the function

```
virtual void Dump(const std::string& file_name) const;
```

By default, simulations begin with empty distributions. However, it is often desirable to resume or extend a simulation, which entails editing its distribution. For this reason, previously computed distributions can be loaded through the function

```
virtual void Restore(const std::string& file_name);
```

To accomplish this, a Simulation object would call

```
virtual void RestoreDistro(int distro);
```

prior to running/continuing the simulation.

5.2 Templated Distribution

→ Subsection 9.7

The `DistributionTemplated` class is derived from, and should be considered an extention of, `DistributionBase`. It is not meant to be implemented directly. The binned values are 1, 2, or 3 dimensional and always encoded as double type, the binning weights can be of any type, such as doubles, GeoVectors, or even GeoMatrices.

→ Subsection 9.5
→ Subsection 9.6

The following inputs are required to build any distribution class:

`n_bins` : Number of bins per dimension. If this value is 0 or negative, the corresponding dimensions is ignored.
Possible values: Any MultiIndex.

`minval` : Minimum value for each dimension
Possible values: Any GeoVector.

`maxval` : Maximum value for each dimension
Possible values: Any GeoVector.

`log_bins` : Flag to indicate whether to bin linearly (0) or logarithmically (not 0) along each dimension.

Possible values: Any MultiIndex.

`bin_outside` : Flag to indicate whether to bin events (not 0) or not (0) that fall outside of the specified minimum and maximum values.
Possible values: Any MultiIndex.

`unit_distro` : Unit for binning weights.
Possible values: A variable of `distroClass` type.

`unit_val` : Unit for binned values. Possible values: Any GeoVector.

`keep_records` : Flag to indicate whether to or not to keep raw (unbinned) data. Possible values: `true` or `false`.

Derived distribution classes will typically require additional parameters for initialization.

5.3 Uniform Distribution

The `DistributionUniform` class a stub (common ancestor) for all uniform distributions. This class is not meant to be implemented directly. Uniform distributions assign a scalar (double type), constant, uniform hot, f_1, or cold, f_2, weight,

$$f(t, \mathbf{v}, \mathbf{p}) = \begin{cases} f_1, \\ f_2, \end{cases} \tag{106}$$

to the binned values of any trajectory that reaches the associated boundaries. When crossing the boundary, the first action assigns the hot value to the binning weight, while the second uses the cold value.

On top of the inputs already required by `DistributionTemplated`, the following parameters should be provided when constructing this module:

`val_hot` : Hot value.
Possible values: Any double.

`val_cold` : Cold value.
Possible values: Any double.

5.4 Uniform Time Distribution

→ Subsection 5.3

The `DistributionTimeUniform` class is derived from `DistributionUniform` and implements a uniform distribution in time.

Only the first dimension (time) of this distribution is active. On top of the inputs already required by `DistributionTemplated` and `DistributionUniform`, the following parameters should be provided when constructing this module:

`val_time` : Whether to use the initial time (0) or the time at the boundary crossing (not 0).
Possible values: Any integer.

5.5 Uniform Position Distribution

→ Subsection 5.3

The `DistributionPositionUniform` class is derived from `DistributionUniform` and implements a uniform distribution in position.

All three dimensions (position) of this distribution are active. On top of the inputs already required by `DistributionTemplated` and `DistributionUniform`, the following parameters should be provided when constructing this module:

`val_time` : Whether to use the initial position (0) or the position at the boundary crossing (not 0).
Possible values: Any integer.

`val_coord` : Whether to bin in Cartesian coordinates (0) or spherical coordinates (not 0), both in the global reference frame.
Possible values: Any integer.

5.6 Uniform Momentum Distribution

→ Subsection 5.3

The `DistributionMomentumUniform` class is derived from `DistributionUniform` and implements a uniform distribution in momentum.

All three dimensions (momentum) of this distribution are active. On top of the inputs already required by `DistributionTemplated` and `DistributionUniform`, the following parameters should be provided when constructing this module:

`val_time` : Whether to use the initial momentum (0) or the momentum at the boundary crossing (not 0).
Possible values: Any integer.

`val_coord` : Whether to bin in "native" coordinates[10] (0) or local spherical coordinates[11] (not 0).
Possible values: Any integer.

5.7 LISM Anisotropy Distribution

The `DistributionAnisotropyLISM` class implements a scalar (double type) distribution designed to model the high-energy galactic cosmic ray (GCR) anisotropy measured at Earth. Some of this anisotropy already exists in the local interstellar medium (LISM), while a non-negligible portion of arises from the interaction between these particles and the heliospheric electromagnetic environment. The intended use is in simulations based on the Lorentz trajectory type and with a backward-in-time integration direction.

→ Subsection 3.3

Due to the relative velocity between the LISM and Earth, $\mathbf{u}_{\mathrm{LISM}}$, a particle's momentum in the frame of the LISM, $\mathbf{p}$, is related to its momentum in the frame of an observer at Earth, $\mathbf{p}_0$ through the formula

$$\mathbf{p} = \mathbf{p}_0 - \gamma m_0 \mathbf{u}_{\mathrm{LISM}}, \tag{107}$$

where γ and m_0 are the Lorentz factor and rest mass of the particle, respectively. If the distribution function of high-energy GCRs both around Earth and in the LISM is assumed to be the same (decaying) power law of momentum,

$$f(\mathbf{p}) = f_0 \left(\frac{p}{p_0} \right)^{\alpha}. \tag{108}$$

where f_0 is a reference distribution value at Earth and $\alpha < 0$ is the power law index. This reference frame transformation leads to a measured anisotropy in the cosmic ray distribution function at Earth, and this is called the Compton-Getting effect. Since the LISM speed is much smaller than the observed particle speed, v_0, which is close to the speed of light, c, for high-energy particles, $u_{\mathrm{LISM}} \ll v_0 \approx c$, the result of the transformation (108) on the distribution function is approximately

$$f(\mathbf{p}) = f_0 \left(1 - \frac{\alpha \mathbf{u}_{\mathrm{LISM}} \cdot \mathbf{p}_0}{v_0 p_0} \right). \tag{109}$$

[10] This are the momentum coordinates intrinsically associated with a trajectory type.
[11] As always, local momentum spherical coordinates have a **z**-axis parallel to the magnetic field.

The anistropy should be apparent in the above formula.

Another source of anisotropy can be due to the interaction of GCRs with our heliospheric magnetic field, which can distort the "pristine" LISM pitch-angle, μ, dipole

$$f(\mathbf{p}) = f_1\mu, \tag{110}$$

and quadrupole anisotropies,

$$f(\mathbf{p}) = f_2\frac{1}{2}(3\mu - 1), \tag{111}$$

where f_1 and f_2 indicate the relative contributions to the overall anisotropy. Yet another source of anisotropy comes from the spatial gradient of GCR density in the LISM,

$$f(\mathbf{r}, \mathbf{p}) = \mathbf{G}_\perp \cdot \mathbf{r}_{\rm gc} \tag{112}$$

where $\mathbf{G}_\perp$ is the GCR density gradient perpendicular to the LISM magnetic field, $\mathbf{B}$, and

$$\mathbf{r}_{\rm gc} = \mathbf{r} + \frac{\mathbf{p}\times\mathbf{B}}{qB^2} \tag{113}$$

is the particle's guiding center (q is the particle charge).

When crossing the boundary, the first action assigns the approximate Compton-Getting factor to the binning weight, the second uses the precise Compton-Getting effect formula, the third computes the first Legendre anistropy, the fourth calculates the second Legendre anisotropy, and finally the fifth utilizes the gradient anisotropy. These correspond to Equations (109), (108), (110), (111), and (112), respectively.

Only the first two dimensions (momentum magnitude and pitch-angle cosine) of this distribution are active. On top of the inputs already required by `DistributionTemplated`, the following parameters should be provided when constructing this module:

`rot_matrix[3]` : An orthonormal basis for converting relative momentum prior to binning. Note that all 3 members of the basis must be inserted separately in the container.
Possible values: Any 3 orthonormal GeoVectors.

`U_LISM` : LISM plasma flow.
Possible values: Any GeoVector.

`mom_pow_law` : Slope of the momentum power law for the Compton-Getting effect.
Possible values: Any double.

`grad_perp_dens` : Gradient of density perpendicular to the LISM magnetic field.
Possible values: Any GeoVector.

5.8 Kinetic Energy Power Law Distribution

The `DistributionSpectrumKineticEnergyPowerLaw` class implements a scalar (double type), uninterrupted power law spectral distribution in the particle's kinetic energy, T,

$$f(p) = f_0(p)\left(\frac{T}{T_0}\right)^\alpha, \tag{114}$$

where T_0 is a reference kinetic energy and α the power law index. The factor $f_0(p)$, which can depend on particle momentum, p, or equivalently particle speed, v, controls whether the power law describes the differential density, differential intensity, or the distribution function. These would be, respectively

$$f_0(p) = \frac{J_0 v}{p^2}, \tag{115}$$

where J_0 is a differential intensity value,

$$f_0(p) = \frac{J_0}{p^2}, \tag{116}$$

$$f_0(p) = f_0, \tag{117}$$

where f_0 is a reference distribution value. When crossing the boundary, the first action assigns the spectrum value to the binning weight, while the second uses a constant, uniform cold value.

Only the first dimension (kinetic energy) of this distribution is active. On top of the inputs already required by `DistributionTemplated`, the following parameters should be provided when constructing this module:

`J0` : Reference multiplicative constant for the spectrum, in any quantity.
Possible values: Any positive double.

`T0` : Reference kinetic energy.
Possible values: Any positive double.

`pow_law` : Power law index.
Possible values: Any double.

`val_cold` : Cold value.
Possible values: Any double.

The file `$SPECTRUM/src/distribution_other.hh` defines the following pre-processor macro:

`DISTRO_KINETIC_ENERGY_POWER_LAW_TYPE` : This macro controls whether to use a power law in differential density (0), differential intensity (not 0), or distribution function (2).
Possible values: `0`, `1`, and `2`.

5.9 Kinetic Energy Bent Power Law Distribution

→ Subsection 5.8

The `DistributionSpectrumKineticEnergyBentPowerLaw` class is derived from `DistributionSpectrumKineticEnergyPowerLaw` and modifies its power law to be bent beyond some bendover kinetic energy, T_b,

$$f(p) = f_0(p) \frac{\left(\frac{T}{T_0}\right)^{\alpha}}{\left[1 + \left(\frac{T}{T_b}\right)^{(\beta-\alpha)/d}\right]^d}, \tag{118}$$

where β the power law index after the bend and d a smoothness factor for the bend. The action indices and the meaning of $f_0(p)$ are the same as the the parent class.

Only the first dimension (kinetic energy) of this distribution is active. On top of the inputs already required by `DistributionTemplated` and `DistributionSpectrumKineticEnergyPow` the following parameters should be provided when constructing this module:

`T_b` : Bendover kinetic energy.
Possible values: Any positive double.

`pow_law_b` : Power law index after the bend.
Possible values: Any double.

`bend_smoothness` : Bend smoothness factor.
Possible values: Any double between 0 and 1.

5.10 Displacement First Order Distribution

→ Subsection 9.5

The `DistributionPositionCumulativeOrder1` class implements a vector (GeoVector type) distribution to track the average displacement in pseudo-trajectories with time. When crossing the boundary, there is only one action which assigns the weight vector $\mathbf{r} - \mathbf{r}_0$, where $\mathbf{r}$ is the position at the boundary crossing and $\mathbf{r}_0$ is the initial position. Only the first dimension (time) of this distribution is active. No additional parameters need to be provided beyond those required for `DistributionTemplated`.

5.11 Displacement Second Order Distribution

→ Subsection 9.6

The `DistributionPositionCumulativeOrder2` class implements a matrix (GeoMatrix type) distribution to track the average displacement in pseudo-trajectories with time. When crossing the boundary, there is only one action which assigns the weight (dyadic product) matrix $(\mathbf{r} - \mathbf{r}_0)(\mathbf{r} - \mathbf{r}_0)$, where $\mathbf{r}$ is the position at the boundary crossing and $\mathbf{r}_0$ is the initial position. Only the first dimension (time) of this distribution is active. No additional parameters need to be provided beyond those required for `DistributionTemplated`.

5.12 Loss Cone Distribution

→ Subsection 9.5

The `DistributionLossCone` class implements a vector (GeoVector type) distribution to track the loss cone pitch-angle along a pseudo-trajectory. Particle's can be magnetically confined if they are surrounded by regions of stronger magnetic field. At any point on a deterministic (no scattering) path, it is possible to know whether a particle will remain trapped (i.e. indefinately mirror) within or eventually escape a magnetic environment by comparing its pitch-angle, θ, to the pitch-angle threshold, θ_c, given by the formula

$$\sin^2 \theta_c = \frac{B}{B_{\max}}, \tag{119}$$

where B is the current magnetic field magnitude and $B_{\max}$ the maximum magnetic field magnitude along the path. The particle will escape if $\theta \leq \theta_c$ and remain trapped otherwise. When crossing the boundary, there is only one action which assigns the following components of the weight vector: the minimum magnetic field encountered during the pseudo-trajectory in the first component, the maximum magnetic field encountered during the pseudo-trajectory in the second component, and the threshold mirroring pitch-angle given by (119) in the third component.

All three dimensions (position) of this distribution are active. On top of the inputs already required by `DistributionTemplated`, the following parameters should be provided when constructing this module:

`val_time` : Whether to use the initial magnetic field magnitude (0) or the magnetic field magnitude at the boundary crossing (not 0).
Possible values: Any integer.

`val_coord` : Whether to bin in Cartesian coordinates (0) or spherical coordinates (not 0), both in the global reference frame.
Possible values: Any integer.

6 Initial Condition Modules

In this section, we describe the initial condition modules currently supported within the existing SPECTRUM architecture.

6.1 Base Initial Conditions

The `InitialBase` class is not meant to be used directly, but rather contains and defines the general functionality shared by all the application specific initial condition modules, described in the following subsections. It is inherited from the `Params` class.

→ Subsection 9.7

Derived initial condition classes will typically require some parameters for initialization. These parameters are passed through a DataContainer object in the public function

→ Subsection 9.2

```
void SetupObject(const DataContainer& cont_in);
```

which calls the protected

```
virtual void SetupInitial(bool construct);
```

A Simulation or Trajectory Integrator (TI) object would call the function

```
void AddInitial(const InitialBase& background_in, const DataContainer&
    container_in);
```

to construct an inital condition module.

The Trajectory Integrator object will obtain the necessary initial condition quantities through use of the public functions

```
double GetTimeSample(void);
GeoVector GetPosSample(void);
GeoVector GetMomSample(const GeoVector& axis_in);
```

These function call the protected method

```
virtual void EvaluateInitial(void);
```

→ Subsection 3.1

Initial Condition objects are linked to TI objects through the latter's `icond_t`, `icond_s`, and `icond_m` pointers. The file `$SPECTRUM/src/initial_base.hh` defines series of global constants beginning with `INITIAL_` that help sort the initial conditions modules. Initial conditions should be thought of as probability distributions from which initial values are generated. The GeoVector `axis` stores a preferred direction with which to generate initial values. For initial momentum conditions, this is typically the magnetic field direction, **b**. Note that not all initial condition modules, particularly in momentum, are compatible with every trajectory module.

! →

6.2 Table Initial Condition

The `InitialTable` templated class is a stub (common ancestor) for all initial condition modules that generate initial quantities by reading them from a file in memory. This class is not meant to be implemented directly.

Derived classes all function similarly. During the setup stage, a file containing the initial values is read and kept in memory. When an initial value is requested, the list is either sequentially or randomly iterated, depending on a user-specified parameters (`random`). If the list is exhausted during sequential iteration, the values will repeat in the same order, starting from the top of the list.

The format of this file should be as follows: The first line of this file will determine what type of quantities are stored: `S` for scalars, `RTP` for vectors in spherical coordinates, and `XYZ` for vectors in Cartesian coordinates. The second line specifies the number of initial quantities listed in the file, N. The remaining N lines are the initial values.

The following inputs are required to build this class:

`init_file_name` : Name of file containing list of initial quantities.
Possible values: Any string.

`scale` : Scale used in initial quantities list.
Possible values: Any positive double.

`random` : Whether to iterate sequentially or randomly through the initial quantities list.
Possible values: `true` or `false`.

6.3 Temporal Initial Conditions

Temporal initial conditions set the value for the initial time of a pseudo-particle. They are important to help synchronize a trajectory with a time-dependant background. They can also be used to implement delta distribution source terms.

6.3.1 Fixed Time Initial Condition

The `InitialTimeFixed` class assigns a fixed initial time,

$$t = t_0. \tag{120}$$

The following input is required to build this class:

`inittime` : Fixed initial time.
Possible values: Any double.

6.3.2 Time Interval Initial Condition

The `InitialTimeInterval` class assigns an initial time within a specified interval,

$$t_1 \leq t \leq t_2, \tag{121}$$

where $t_1 < t_2$ delimit the range of the interval. The initial times can be evenly spaced by a desired increment or random, with a uniform distribution, within the interval.

The following inputs are required to build this class:

`starttime` : Start time for interval.
Possible values: Any double.

`endtime` : End time for interval.
Possible values: Any double.

`n_intervals` : Number of equally sized subintervals used to determine the increment. If this parameter is 0 or less, times will be randomly generated within the interval.
Possible values: Any integer.

6.3.3 Time Table Initial Condition

→ Subsection 6.2

The `InitialTimeTable` class is derived from `InitialTable` and sets the initial time for a trajectory from values saved on a file. No additional parameters need to be provided beyond those required for `InitialTable`.

6.4 Spatial Initial Conditions

Spatial initial conditions set the vector for the starting position of a pseudo-particle.

6.4.1 Fixed Position Initial Condition

The `InitialSpaceFixed` class assigns a fixed initial position,

$$\mathbf{r} = \mathbf{r}_0. \tag{122}$$

The following input is required to build this class:

`initpos` : Fixed initial position.
Possible values: Any GeoVector.

6.4.2 Spatial Segment Initial Condition

The `InitialSpaceLine` class assigns initial positions along a specified segment,

$$\mathbf{r} = (1-s)\mathbf{r}_1 + s\mathbf{r}_2, \quad 0 \leq s \leq 1, \tag{123}$$

where $\mathbf{r}_1$ and $\mathbf{r_2}$ are the segment endpoints. The initial positions can be evently spaced by a desired increment or random, with a uniform distribution, within the segment.

The following inputs are required to build this class:

`startpos` : Start position for segment.
Possible values: Any GeoVector.

`endpos` : End position for segment.
Possible values: Any GeoVector.

`n_intervals` : Number of equally sized subintervals used to determine the increment. If this parameter is 0 or less, times will be randomly, uniformly generated within the segment.
Possible values: Any integer.

6.4.3 Spatial Circle Initial Condition

The `InitialSpaceCircle` class assigns initial positions along a specified circle,

$$\mathbf{r} = \mathbf{c} + R_{\mathbf{n}}(\mathbf{d}, \theta), \quad 0 \leq \theta \leq 2\pi \tag{124}$$

where $\mathbf{c}$ is the center of the circle, $\mathbf{d}$ any vector along the radius of the circle, and $R_{\mathbf{n}}(\mathbf{d}, \theta)$ the counterclockwise rotation of vector $\mathbf{d}$ by angle θ along the axis $\mathbf{n}$, which is perpendicular to the plane containing the circle. The distribution of initial positions is uniform in θ

The following inputs are required to build this class:

`origin` : Center of the circle.
Possible values: Any GeoVector.

`normal` : Any vector perpendicular to the plane containing the circle.
Possible values: Any GeoVector.

`radius` : Radius of the circle.
Possible values: Any positive double.

6.4.4 Spatial Box Initial Condition

→ Subsection 6.4.2

The `InitialSpaceBox` class is derived from `InitialSpaceLine` and it assigns initial positions within a box (rectangular prism). The positions are randomly generated in a box with opposite corners $\mathbf{r}_1$ and $\mathbf{r}_2$ according to a uniform distribution. No additional parameters need to be provided beyond those required for `InitialSpaceLine`.

6.4.5 Spatial Sphere Initial Condition

The `InitialSpaceSphere` class randomly assigns initial positions on the surface of a sphere,

$$\mathbf{r} = \mathbf{c} + R_{\mathbf{n}}(\theta, \phi), \quad 0 \leq \theta \leq \pi,\, 0 \leq \phi \leq 2\pi \tag{125}$$

where $\mathbf{c}$ and R are the center of the sphere and radius of the sphere, respectively, and $\mathbf{n}(\theta, \phi)$ is a unit vector determined by the spherical angles θ and ϕ in the global reference frame. The distribution of initial positions is uniform in $\cos\theta$ and ϕ.

The following inputs are required to build this class:

`origin` : Center of the sphere.
Possible values: Any GeoVector.

`radius` : Radius of the sphere.
Possible values: Any positive double.

6.4.6 Spatial Spherical Sector Initial Condition

→ Subsection 6.4.5

The `InitialSpaceSphereSector` class is derived from `InitialSpaceSphere` and it randomly assigns initial positions on the surface of a spherical sector,

$$\mathbf{r} = \mathbf{c} + R_{\mathbf{n}}(\theta, \phi), \quad \theta_1 \leq \theta \leq \theta_2,\, \phi_1 \leq \phi \leq \phi_2, \tag{126}$$

where θ_1, θ_2, ϕ_1, and ϕ_2 define the sector in spherical coordinates using the global reference frame. The distribution of initial positions is uniform in $\cos\theta$ and ϕ.

On top of the inputs already required by `InitialSpaceSphere`, the following parameters should be provided when constructing this module:

`theta1` : Lower limit for polar angle of sector.
Possible values: Any double.

`theta2` : Upper limit for polar angle of sector.
Possible values: Any double.

`phi1` : Lower limit for azimuthal angle of sector.
Possible values: Any double.

`phi2` : Upper limit for azimuthal angle of sector.
Possible values: Any double.

6.4.7 Rankine Half-body Initial Condition

→ Subsection 4.7

The `InitialSpaceRankine` class assigns initial positions on a Rankine half-body surface with an axis of symmetry along the **z**-axis. The position is uniformly distributed along the azimuthal angle and the cosine of the polar angle.

A Rankine half-body is an unbounded surface. Therefore, a limit but be set along the "tail" (away from the "nose"). That limit was implemented in the form of a maximum distance from the origin beyond which no initial positions are generated, $r_{\max}$. This is then converted into a maximum polar angle cosine with the formula

$$\cos\theta_{\max} = \frac{2z_0^2}{r_{\max}^2} - 1 \tag{127}$$

where z_0 is the nose distance of the Rankine half-body.

The following inputs are required to build this class:

`origin` : Origin of the Rankine half-body.
Possible values: Any GeoVector.

`z_nose` : Nose distance of the Rankine half-body.
Possible values: Any positive double.

`radius` : Maximum distance from origin within which to generate initial positions.
Possible values: Any double.

6.4.8 Position Table Initial Condition

→ Subsection 6.2

The `InitialPositionTable` class is derived from `InitialTable` and sets the initial position for a trajectory from values saved on a file. No additional parameters need to be provided beyond those required for `InitialTable`.

6.5 Momentum Initial Conditions

Momentum initial conditions set the vector for the starting momentum of a pseudo-particle. Since some transport models collapse 1 or 2 momentum dimensions, not all initial conditions modules are applicable to all trajectory modules. In particular, the Parker and fieldline trajectories do not resolve gyro-phase or pitch-angle, so initial condition modules that specify the momentum direction are not defined for this trajectory type.

! →
→ Subsection 3.6
→ Subsection 3.2

6.5.1 Fixed Momentum Initial Condition

The `InitialMomentumFixed` class assigns a fixed initial momentum,

$$\mathbf{p} = \mathbf{p}_0. \tag{128}$$

This class is only defined for Lorentz and fieldline trajectory trajectory types.

! →
→ Subsection 3.3
→ Subsection 3.2

The file `$SPECTRUM/src/initial_momentum.hh` defines the following pre-processor macro:

`INITIAL_MOM_FIXED_COORD` : This macro controls whether the fixed momentum is specified in the global Cartesian or an initial momentum in local spherical coordinates.[12]
Possible values: `0` or `1`.

If `INITIAL_MOM_FIXED_COORD` is set to `0`, the following input is required to build this class:

`initpos` : Fixed initial position.
Possible values: Any GeoVector.

Otherwise, the following inputs are required to build this class:

`p0` : Initial momentum magnitude.
Possible values: Any positive double.

`theta0` : Initial momentum polar angle.
Possible values: Any double.

`p0` : Initial momentum azimuthal angle.
Possible values: Any double.

[12] In this local frame, the **z**-axis is determined by the direction of the magnetic field at each point.

6.5.2 Momentum Beam Initial Condition

The `InitialMomentumBeam` class implements a cold beam initial distribution, which has a fixed momentum magnitude and is directed along the local magnetic field,

$$\mathbf{p} = p_0 \mathbf{b}. \tag{129}$$

The following input is required to build this class:

`p0` : Initial momentum magnitude.
Possible values: Any positive double.

6.5.3 Momentum Ring Initial Condition

The `InitialMomentumRing` class implements a cold ring initial distribution, which has a fixed momentum magnitude and pitch-angle and a uniformly distributed gyro-phase with respect to the local magnetic field,

$$\mathbf{p} \cdot \mathbf{b} = p_0 \cos\theta_0. \tag{130}$$

The following inputs are required to build this class:

`p0` : Initial momentum magnitude.
Possible values: Any positive double.

`theta0` : Initial momentum polar angle.
Possible values: Any double.

6.5.4 Momentum Shell Initial Condition

The `InitialMomentumShell` class implements a cold, isotropic, non-drifting shell initial distribution, which has a fixed momentum magnitude and uniformly distributed pitch-angle and gyro-phase with respect to the local magnetic field,

$$p = p_0. \tag{131}$$

The following input is required to build this class:

`p0` : Initial momentum magnitude.
Possible values: Any positive double.

6.5.5 Momentum Thick Shell Initial Condition

The `InitialMomentumThickShell` class implements a warm, isotropic, non-drifting shell initial distribution, which has a momentum magnitude uniformly distributed within a specified interval and uniformly distributed pitch-angle and gyro-phase with respect to the local magnetic field,

$$p_1 \leq p \leq p_2, \tag{132}$$

where $p_1 < p_2$.

The following inputs are required to build this class:

`p1` : Lower limit for initial momentum magnitude.
Possible values: Any positive double.

`p2` : Upper limit for initial momentum magnitude.
Possible values: Any positive double.

`log_bias` : Whether to draw samples uniformly in p or $\ln p$.
Possible values: `true` or `false`.

6.5.6 Momentum Maxwellian Initial Condition

The `InitialMomentumMaxwell` class implements a non-relativistic drifting bi-Maxwellian initial distribution. The component of momentum parallel to local magnetic field follows a Gaussian distribution, while the perpendicular portion obeys a Rayleigh distribution.

The following inputs are required to build this class:

`p0` : Drift momentum in the parallel direction.
Possible values: Any double.

`dp_para` : Momentum spread in the parallel direction.
Possible values: Any positive double.

`dp_perp` : Momentum spread in the perpendicular direction.
Possible values: Any positive double.

6.5.7 Momentum Table Initial Condition

→ Subsection 6.2

The `InitialMomentumTable` class is derived from `InitialTable` and sets the initial momentum for a trajectory from values saved on a file. No additional parameters need to be provided beyond those required for `InitialTable`.

7 Boundary Condition Modules

In this section, we describe the boundary condition modules currently supported within the existing SPECTRUM architecture.

7.1 Base Boundary Conditions

The `BoundaryBase` class is not meant to be used directly, but rather contains and defines the general functionality shared by all the application specific boundary condition modules, described in the following subsections. It is inherited from the `Params` class.

→ Subsection 9.7

The following inputs are required to build any boundary class:

`max_crossings` : Maximum number of crossings allowed. If this value is negative, it is unlimited.
Possible values: Any integer.

`actions` : A vector of what action to trigger for each distribution in the simulation. A negative action denotes no action.
Possible values: Any `std::vector` of integers.

Derived boundary classes will typically require additional parameters for initialization. These parameters are passed through a DataContainer object in the public function

→ Subsection 9.2

```
void SetupObject(const DataContainer& cont_in);
```

which calls the protected

```
virtual void SetupBoundary(bool construct);
```

A Simulation or Trajectory Integrator (TI) object would call the function

```
void AddBoundary(const BoundaryBase& boundary_in, const DataContainer&
    container_in);
```

to construct a boundary module.

Boundary Condition objects are linked to TI objects through the latter's `bcond_t`, `bcond_s`, and `bcond_m` pointers. The file `$SPECTRUM/src/boundary_base.hh` defines series of global constants beginning with `BOUNDARY_` that help sort the boundary conditions modules. Boundary conditions should be thought of as hypersurfaces in time, physical space, or momentum space. The `_delta` double variable is used to gauge the pseudo-distance to a boundary[13] and has opposite signs on either side of the boundary. In other words, a boundary divides a domain in two regions, one where `_delta` is positive, and another where it is negative.

→ Subsection 3.1

Boundary crossing events, i.e. flips in the sign of `_delta`, are checked in every trajectory step and they may trigger binning operations for the distribution modules. This behavior is controlled by the `actions` array. This array is the same length as the TI's `distributions` array. When a boundary is crossed, that boundary's `actions` array is iterated and each component, if non-negative, is passed to the corresponding Distribution object to indicate how that crossing should be registered.

→ Section 5

Upon a boundary crossing, the trajectory's state may also be edited. Boundary Condition objects can be categorized in three types, depending on how they affect a pseudo-trajectory when they register a crossing event. Terminal boundaries will end a pseudo-trajectory immediately after crossing. Reflecting boundaries will (roughly) reflect a pseudo-particle's path upon crossing, changing its state (position

[13] This may not be the "shortest" distance but it does approach zero near the boundary.

and momentum) in the process based on the local geometry of the reflecting boundary. Finally, passive boundaries will not affect a pseudo-trajectory when crossed.

! → Terminal boundaries override the user-specified value of `max_crossings` to 1.

7.2 Temporal Boundary Conditions

Temporal boundary conditions establish boundary conditions in time for a pseudo-particle.

7.2.1 Single Time Boundary

The `BoundaryTime` class is a stub (common ancestor) for all temporal boundary condition objects on a single, fixed time boundary. This class is not meant to be implemented directly. For application purposes, the terminal `BoundaryTimeExpire` and passive `BoundaryTimePass` classes are derived from `BoundaryTime`.

On top of the inputs already required by `BoundaryBase`, the following parameter should be provided when constructing this module:

`timemark` : Fixed boundary time.
Possible values: Any double.

7.2.2 Recurrent Time Boundary

The `BoundaryTimeRecurrent` class is derived from `BoundaryTime` and it implements a temporal boundary condition of equally spaced, recurrent, passive time boundaries. The time boundaries are placed at $\{nt_* : n \in \mathbb{N}\}$, where t_* is the first boundary time. No additional parameters need to be provided beyond those required for `BoundaryBase` and `BoundaryTime`.

7.3 Spatial Boundary Conditions

Spatial boundary conditions establish boundary conditions in physical space for a pseudo-particle.

7.3.1 Plane Boundary

The `BoundaryPlane` class a stub (common ancestor) for all spatial boundary condition objects on a plane. This class is not meant to be implemented directly. For application purposes, the terminal `BoundaryPlaneAbsorb`, reflecting `BoundaryPlaneReflect`, and passive `BoundaryPlanePass` classes are derived from `BoundaryPlane`.

On top of the inputs already required by `BoundaryBase`, the following parameters should be provided when constructing this module:

`origin` : Any point on the plane.
Possible values: Any GeoVector.

`endtime` : Any vector perpendicular to the plane.
Possible values: Any GeoVector.

7.3.2 Box Boundary

The `BoundaryBox` class a stub (common ancestor) for all spatial boundary condition objects on a box (rectangular prism). This class is not meant to be implemented directly. For application purposes, the reflecting `BoundaryBoxReflect` class is derived from `BoundaryBox`.

On top of the inputs already required by `BoundaryBase`, the following parameters should be provided when constructing this module:

`corner1` : One corner of the box.
Possible values: Any GeoVector.

`normal1` : One edge of the box pointing away from `corner1`.
Possible values: Any GeoVector.

`normal2` : Another edge of the box, different from `normal1`, pointing away from `corner1`.
Possible values: Any GeoVector.

`normal3` : Another edge of the box, different from `normal1` and `normal2`, pointing away from `corner1`.
Possible values: Any GeoVector.

7.3.3 Sphere Boundary

The `BoundarySphere` class a stub (common ancestor) for all spatial boundary condition objects on a sphere. This class is not meant to be implemented directly. For application purposes, the terminal `BoundarySphereAbsorb` and reflecting `BoundarySphereReflect` classes are derived from `BoundarySphere`.

On top of the inputs already required by `BoundaryBase`, the following parameters should be provided when constructing this module:

`origin` : Center of the sphere.
Possible values: Any GeoVector.

`radius` : Radius of the sphere.
Possible values: Any positive double.

7.3.4 Rankine Half-Body Boundary

The `BoundaryRankine` class a stub (common ancestor) for all spatial boundary condition objects on a Rankine half-body surface with an arbitrary axis of symmetry. This class is not meant to be implemented directly. For application purposes, the `BoundaryRankineAbsorb` class is derived from `BoundaryRankine`.

→ Subsection 4.7

On top of the inputs already required by `BoundaryBase`, the following parameters should be provided when constructing this module:

`origin` : Origin of the Rankine half-body.
Possible values: Any GeoVector.

`axis` : Axis connecting the origin and the nose of the Rankine half-body.
Possible values: Any GeoVector.

`z_nose` : Nose distance of the Rankine half-body.
Possible values: Any positive double.

7.3.5 Cylinder Boundary

The `BoundaryCylinder` class a stub (common ancestor) for all spatial boundary condition objects on a cylinder. This class is not meant to be implemented directly. For application purposes, the `BoundaryCylinderAbsorb` class is derived from `BoundaryCylinder`.

On top of the inputs already required by `BoundaryBase`, the following parameters should be provided when constructing this module:

`origin` : Origin of the cylinder.
Possible values: Any GeoVector.

`axis` : Symmetry axis of the cylinder.
Possible values: Any GeoVector.

`radius` : Radius of the cylinder.
Possible values: Any positive double.

7.3.6 Region Boundary

→ Subsection 9.8

The `BoundaryRegion` class a stub (common ancestor) for all spatial boundary condition objects on arbitrary indicator variable given by the `_spdata.region` GeoVector. This class is not meant to be implemented directly. For application purposes, the `BoundaryRegionAbsorb` class is derived from `BoundaryRegion`. Figure 18 illustrates a boundary object of this type.

On top of the inputs already required by `BoundaryBase`, the following parameters should be provided when constructing this module:

`region_ind` : Index of the `_spdata.region` associated with the indicator variable.
Possible values: Any non-negative integer (between 0 and 2).

`region_val` : Threshold for the indicator variable.
Possible values: Any double.

7.4 Momentum Boundary Conditions

Momentum boundary conditions establish boundary conditions in momentum space for a pseudo-particle.

7.4.1 Injection Boundary

The `BoundaryMomentum` class a stub (common ancestor) for all momentum boundary condition objects on a single, fixed momentum magnitude. This class is not meant to be implemented directly. For application purposes, the `BoundaryMomentumInjection` class is derived from `BoundaryMomentum`.

On top of the inputs already required by `BoundaryBase`, the following parameter should be provided when constructing this module:

`momentum` : Fixed boundary momentum.
Possible values: Any double.

7.4.2 Mirror Boundary

! →
→ Subsection 3.6
→ Subsection 3.2

The `BoundaryMirror` class is a passive boundary that checks when a pseudo-trajectory has undergone magnetic mirroring, meaning it has reversed directions with respect to the local magnetic field. This class is not defined for the Parker and fieldline trajectory types, since these types do not resolve the momentum direction. No additional parameters need to be provided beyond those required for `BoundaryBase`.

8 Diffusion Modules

In this section, we describe the diffusion (and scattering) modules currently supported within the existing SPECTRUM architecture. These modules are only relevant for transport models that include the effects of spatial diffusion and/or pitch-angle diffusion.

8.1 Base Diffusion

The `DiffusionBase` class is not meant to be used directly, but rather contains and defines the general functionality shared by all the application specific diffusion modules, described in the following subsections. It is inherited from the `Params` class.

→ Subsection 9.7

Derived diffusion classes will typically require some parameters for initialization. These parameters are passed through a DataContainer object in the public function

→ Subsection 9.2

```
void SetupObject(const DataContainer& cont_in);
```

which calls the protected method

```
virtual void SetupDiffusion(bool construct);
```

A Simulation or Trajectory Integrator (TI) object would call the function

```
void AddDiffusion(const DiffusionBase& diffusion_in, const DataContainer&
    container_in);
```

to construct a diffusion module.

The Trajectory Integrator object will obtain the necessary diffusion quantities through use of the public functions

```
double GetComponent(int comp, double t_in, const GeoVector& pos_in, const
    GeoVector& mom_in, const SpatialData& spdata_in);
virtual double GetDirectionalDerivative(int xyz);
virtual double GetMuDerivative(void);
```

! → The momentum GeoVector `mom_in` must be in local spherical coordinates with the magnetic field along the `z`-axis. This function calls the protected method

```
virtual void EvaluateDiffusion(void);
```

→ Subsection 3.1

Diffusion objects are linked to TI objects through the latter's `diffusion` pointer. The GeoVector `Kappa` stores all the diffusion coefficients and the integer `comp_eval` controls which component is modified. The doubles `vmag`, `mu`, `st2`, and `Omega`, as well as the SpatialData object `_spdata` aid in the diffusion coefficient computations. In all of the formulas in this section use v is the particle speed, B the local magnetic field magnitude, μ the particle pitch-angle, and Ω the particle gyro-frequency. Note that any diffusion modules computing pitch-angle dependent coefficients should only be used with trajectory modules that resolve pitch-angle, such as guiding center or focused transport trajectory types. In particular, the Parker trajectories does not resolve pitch-angle, so only spatial diffusion modules are defined for this trajectory type.

! →

→ Subsection 3.4
→ Subsection 3.5
→ Subsection 3.6

Diffusion can take place in pitch-angle or in space. Pitch-angle scattering coefficient, $D_{\mu\mu}$, should always be thought of as pitch-angle dependent, even when constant. Spatial diffusion coefficients, which are always meant to be applied with respect to the local direction of the magnetic field, can be pitch-angle dependent or pitch-angle averaged. By convention, we use D to denote the former and κ for the latter, adding

a $\parallel$ or $\perp$ subscript to denote whether the coefficient is in the direction parallel or perpendicular to the local magnetic field. These coefficients are connection by

$$\kappa_{\parallel} = \frac{1}{2}\int_{-1}^{1} D_{\parallel}\, d\mu, \qquad \kappa_{\perp} = \frac{1}{2}\int_{-1}^{1} D_{\perp}\, d\mu. \tag{133}$$

Pitch-angle scattering also generates parallel spatial diffusion,

$$\kappa_{\parallel} = \frac{v^2}{8}\int_{-1}^{1} \frac{\left(1-\mu^2\right)^2}{D_{\mu\mu}(\mu)}\, d\mu. \tag{134}$$

8.2 Pitch-angle Scattering

The following diffusion modules compute pitch-angle coefficients in order to implement a variety of pitch-angle scattering models, which are locally stored in `Kappa[2]`.

8.2.1 Istropic Pitch-angle Scattering

The `DiffusionIsotropicConstant` class calculates a pitch-angle scattering coefficient consistent with an isotropic pitch-angle scattering process,

$$D_{\mu\mu} = D_0\left(1-\mu^2\right). \tag{135}$$

where D_0 is the desired (isotropic) pitch-angle scattering coefficient.

The following inputs are required to build this class:

`D0` : Isotropic pitch-angle scattering coefficient.
Possible values: Any positive double.

8.2.2 QLT Pitch-angle Scattering

The `DiffusionQLTConstant` class calculates a pitch-angle scattering coefficient consistent with Quasilinear Theory (QLT). In this model, particles resonate with Alfvénic turbulence with a spectral density of the form

$$P(k) = \langle(\delta B^A)^2\rangle \frac{\gamma-1}{2k_0}\left(\frac{k_0}{k}\right)^{\gamma} \tag{136}$$

where $\langle(\delta B^A)^2\rangle$ is the Alfvénic in fluctuations, γ the index of the decaying power law in wavenumber, and k_0 a characteristic wavenumber associated with the maximum turbulent lengthscale. It is assumed that the relative variance of Alfvénic turbulence is spatially constant, i.e. $\langle(\delta B^A)^2\rangle/B^2$ is constant in space. The pitch-angle scattering coefficient is

$$D_{\mu\mu} = \frac{\pi(\gamma-1)}{4}\frac{(1-\mu^2)\Omega^2}{k_0 v|\mu|}\frac{\langle(\delta B^A)^2\rangle}{B^2}\left(\frac{k_0 v\mu}{\Omega}\right)^{\gamma}. \tag{137}$$

The following inputs are required to build this class:

`A2A` : Relative variance of Alfvénic turbulence.
Possible values: Any positive double.

`l_max` : Maximum turbulent lengthscale.
Possible values: Any positive double.

`ps_index` : Spectral index of decaying power law.
Possible values: Any positive double.

8.2.3 WNLT Pitch-angle Scattering

→ Subsection 8.2.2

The `DiffusionWNLTConstant` class is derived from `DiffusionQLTConstant` and it calculates a pitch-angle scattering coefficient consistent with Weakly Nonlinear Theory (WNLT). In this model, particles resonate with the Alfvénic turbulence previously described in Subsection 8.2.2, as well as 2D incompressible (transverse) and compressible (longitudinal) turbulence. It is assumed that all turbulence types share the same spectral power law index and characteristic wavenumber and that the relative variances of all turbulence types are spatially constant, i.e. $\langle \delta B^2 \rangle / B^2$ is constant in space. The pitch-angle scattering coefficient is

$$D_{\mu\mu} = D^A_{\mu\mu} + D^T_{\mu\mu} \tag{138}$$

where $D^A_{\mu\mu}$ is given by Equation (137),

$$D^T_{\mu\mu} = \frac{D^T_{\mu\mu,1} D^T_{\mu\mu,2}}{\sqrt{{D^T_{\mu\mu,1}}^2 + {D^T_{\mu\mu,2}}^2}}, \tag{139}$$

$$D^T_{\mu\mu,1} = \frac{\left(1-\mu^2\right) D_\perp k_0^2}{2\left(1+\xi_1^2\right)} \frac{\langle (\delta B^T)^2 \rangle}{B^2} {}_2F_1\left(1, 1, \frac{\gamma+1}{2}; \frac{1}{1+\xi_1^2}\right), \tag{140}$$

$$D^T_{\mu\mu,2} = \frac{\left(1-\mu^2\right) D_\perp k_0^2}{2\left(1+\xi_2^2\right)} \frac{\gamma-1}{\gamma+1} \frac{\langle (\delta B^T)^2 \rangle}{B^2} {}_2F_1\left(1, 1, \frac{\gamma+5}{2}; \frac{1}{1+\xi_2^2}\right), \tag{141}$$

${}_2F_1$ is the Gaussian hypergeometric function,

$$\xi_1 = \frac{v k_0 \sqrt{1-\mu^2}}{\sqrt{2}\Omega}, \qquad \xi_2 = \frac{D_\perp k_0^2}{\Omega}, \tag{142}$$

and $D_\perp$ is given by Equation (143).

On top of the inputs already required by `DiffusionQLTConstant`, the following parameters should be provided when constructing this module:

`A2T` : Relative variance of transverse turbulence.
Possible values: Any positive double.

`A2L` : Relative variance of longitudinal turbulence.
Possible values: Any positive double.

The file `$SPECTRUM/src/diffusion_other.hh` defines the following pre-processor macro:

`USE_QLT_SCATT_WITH_WNLT_DIFF` : This macro controls whether the pitch-angle coefficient should be computed using the QLT term only or by also adding the WLNT contribution.
Possible values: Commented or uncommented.

8.2.4 WNLT VLISM Pitch-angle Scattering

→ Subsection 8.2.3
→ Subsection 4.7

The `DiffusionWNLTRampVLISM` class is derived from `DiffusionWNLTConstant` and it is only meant to be used with a CFK LISM background. This class keeps the WNLT coefficients unchanged far from the heliopause (HP), which has the shape of a Rankine half-body, but modifies the coefficients near the HP.

In this model, the maximum turbulent lengthscale, l, is spatially variable within the HP sheath, which is delimited by another Rankine half-body with the same axis of symmetry as the HP but a larger distance between the Sun (origin) and the stagnation point (tip), referred to as the nose distance. Note that every point in the HP sheath falls on some Rankine half-body with a nose distance, z, between the nose distance of the HP, $z_{\rm HP}$, and the nose distance delimiting the HP sheath, z_0. It varies linearly with z between l_0 on and away from the surface delimiting the

HP sheath and l_{HP} at the HP and the characteristic wavenumber is computed as $k_0 = 2\pi/l$. The relative variances are scaled so that the total turbulent power, i.e. the integral from k_0 to ∞ of $P(k)$, remains constant.

On top of the inputs already required by `DiffusionWNLTConstant`, the following parameters should be provided when constructing this module:

`l_max_HP` : Maximum turbulent lengthscale at the HP.
Possible values: Any positive double.

`z_nose` : Distance between the Sun and the stagnation point (tip) of the HP.
Possible values: Any positive double.

`z_sheath` : Distance between the Sun and the tip of the HP sheath.
Possible values: Any positive double.

8.3 Spatial Diffusion

The following diffusion modules compute spatial diffusion coefficients in order to implement a variety of spatial diffusion models. Perpendicular and parallel coefficients are locally stored in `Kappa[0]` and `Kappa[1]`, respectively.

8.3.1 WNLT Perpendicular Diffusion

The `DiffusionWNLTConstant` class, already discussed in Subsection 8.2.3, also calculates a perpendicular diffusion coefficient consistent with WNLT,

$$D_\perp = v \left[\frac{\gamma - 1}{\gamma + 1} \frac{\mu^2}{2k_0^2} \frac{\langle (\delta B^T)^2 \rangle}{B^2} + \frac{v^2 \left(1 - \mu^2\right)^2}{8\Omega^2} \frac{\langle (\delta B^L)^2 \rangle}{B^2} \right]^{1/2} . \tag{143}$$

8.3.2 WNLT VLISM Perpendicular Diffusion

The `DiffusionWNLTRampVLISM` class was already described in Subsection 8.2.4.

8.3.3 Uniform Parallel Diffusion

The `DiffusionParaConstant` class calculates a uniform parallel diffusion coefficient,

$$D_\parallel = D_0. \tag{144}$$

The following input is required to build this class:

`D0` : Uniform parallel diffusion coefficient.
Possible values: Any positive double.

8.3.4 Uniform Perpendicular Diffusion

The `DiffusionPerpConstant` class calculates a uniform perpendicular diffusion coefficient,

$$D_\perp = D_0. \tag{145}$$

The following input is required to build this class:

`D0` : Uniform perpendicular diffusion coefficient.
Possible values: Any positive double.

8.3.5 Uniform Isotropic Diffusion

The `DiffusionFullConstant` class calculates a uniform isotropic spatial diffusion coefficients,

$$D_{\parallel} = D_{\perp} = D_0. \tag{146}$$

The following input is required to build this class:

`D0` : Uniform isotropic diffusion coefficient.
Possible values: Any positive double.

8.3.6 Bulk Flow Power Law Diffusion

The `DiffusionFlowPowerLaw` class calculates anisotropic spatial diffusion coefficients that are power laws of the bulk flow magnitude

$$\kappa_{\parallel} = \kappa_0 \left(\frac{U}{u_0} \right)^{\alpha}, \qquad \kappa_{\perp} = \eta \kappa_{\parallel}, \tag{147}$$

where κ_0 is a reference diffusion coefficient, U is the bulk flow magnitude, u_0 is a reference bulk flow magnitude, α is the index of the power law, and η is the ratio of perpendicular to parallel diffusion.

The following inputs are required to build this class:

`kappa0` : Reference diffusion coefficient.
Possible values: Any positive double.

`U0` : Reference bulk flow magnitude.
Possible values: Any positive double.

`pow_law_U` : Index of bulk flow power law.
Possible values: Any double.

`kap_rat` : Ratio of perpendicular to parallel diffusion coefficients.
Possible values: Any positive double.

8.3.7 Momentum Power Law Diffusion

The `DiffusionMomentumPowerLaw` class calculates anisotropic spatial diffusion coefficients that are power laws of the particle's momentum magnitude

$$\kappa_{\parallel} = \kappa_0 \left(\frac{p}{p_0} \right)^{\alpha}, \qquad \kappa_{\perp} = \eta \kappa_{\parallel}, \tag{148}$$

where κ_0 is a reference diffusion coefficient, p is the particle's momentum magnitude, p_0 is a reference particle's momentum magnitude, α is the index of the power law, and η is the ratio of perpendicular to parallel diffusion.

The following inputs are required to build this class:

`kappa0` : Reference diffusion coefficient.
Possible values: Any positive double.

`p0` : Reference particle's momentum magnitude.
Possible values: Any positive double.

`pow_law_p` : Index of particle's momentum power law.
Possible values: Any double.

`kap_rat` : Ratio of perpendicular to parallel diffusion coefficients.
Possible values: Any positive double.

8.3.8 Rigidity and Magnetic Field Power Laws Diffusion

The `DiffusionRigidityMagneticFieldPowerLaw` class calculates anisotropic spatial diffusion coefficients that are power laws of the particle's rigidity and magnetic field magnitude,

$$\kappa_{\parallel} = \frac{\lambda_0 v}{3}\left(\frac{R}{R_0}\right)^{\alpha_1}\left(\frac{B}{B_0}\right)^{\alpha_2}, \qquad \kappa_{\perp} = \eta\kappa_{\parallel}, \tag{149}$$

where λ_0 is a reference parallel mean free path, R is the particle's rigidity, R_0 is a reference rigidity, α_1 is the index of the rigidity power law, B_0 is a reference mangetic field magnitude, α_2 is the index of the magnetic field power law, and η is the ratio of perpendicular to parallel diffusion.

The following inputs are required to build this class:

`lambda0` : Reference parallel mean free path.
Possible values: Any positive double.

`R0` : Reference rigidity.
Possible values: Any positive double.

`B0` : Reference magnetic field magnitude.
Possible values: Any positive double.

`pow_law_R` : Index of rigidity power law.
Possible values: Any double.

`pow_law_B` : Index of magnetic field power law.
Possible values: Any double.

`kap_rat` : Ratio of perpendicular to parallel diffusion coefficients.
Possible values: Any positive double.

8.3.9 Kinetic Energy and Radial Distance Power Laws Diffusion

The `DiffusionKineticEnergyRadialDistancePowerLaw` class calculates anisotropic spatial diffusion coefficients that are power laws of the particle's kinetic energy and radial distance from the Sun (origin),

$$\kappa_{\parallel} = \kappa_0\left(\frac{T}{T_0}\right)^{\alpha_1}\left(\frac{r}{r_0}\right)^{\alpha_2}, \qquad \kappa_{\perp} = \eta\kappa_{\parallel}, \tag{150}$$

where κ_0 is a reference diffusion coefficient, T is the particle's kinetic energy, T_0 is a reference kinetic energy, α_1 is the index of the kinetic energy power law, r is the radial distance from the Sun (origin), r_0 is a reference radial distance, α_2 is the index of the radial distance power law, and η is the ratio of perpendicular to parallel diffusion.

The following inputs are required to build this class:

`kappa0` : Reference diffusion coefficient.
Possible values: Any positive double.

`T0` : Reference kinetic energy.
Possible values: Any positive double.

`r0` : Reference radial distance.
Possible values: Any positive double.

`pow_law_T` : Index of kinetic energy power law.
Possible values: Any double.

`pow_law_r` : Index of radial distance power law.
Possible values: Any double.

`kap_rat` : Ratio of perpendicular to parallel diffusion coefficients.
Possible values: Any positive double.

8.3.10 Strauss et al. 2013 Diffusion

The `DiffusionKineticEnergyRadialDistancePowerLaw` class calculates anisotropic spatial diffusion coefficients that are power laws of the particle's rigidity and magnetic field magnitude but also take into account indicator variables stored in the `_spdata.region`,

→ Subsection 9.8

$$\kappa_{\parallel} = \frac{\lambda_0(i_1)v}{3}\left(\frac{R}{R_0}\right)^{\alpha_1(R)}\left(\frac{B}{B_0}\right)^{\alpha_2(i_1)}, \qquad \kappa_{\perp} = \eta_r(i_2)\eta(i_1)\kappa_{\parallel}, \tag{151}$$

where $\lambda_0(i_1)$ is a reference parallel mean free path which depends on the i_1 component of `_spdata.region`, R is the particle's rigidity, R_0 is a reference rigidity, $\alpha_1(R)$ is the index of the rigidity power law which also changes with rigidity, B_0 is a reference mangetic field magnitude, α_2 is the index of the magnetic field power law which changes with the i_1 component of `_spdata.region`, $\eta(i_1)$ is the ratio of perpendicular to parallel diffusion which depends on the i_1 component of `_spdata.region`, and $\eta_r(i_2)$ is a reduction factor for η which depends on the i_2 component of `_spdata.region`. In this model, the rigidity index has the form

$$\alpha_1 = \begin{cases} \frac{1}{3}, & R < R_0, \\ 1, & R \geq R_0, \end{cases} \tag{152}$$

and $\alpha_1 = 1$ where the i_1 component of `_spdata.region` is positive while $\alpha_1 = 0$ where the i_1 component of `_spdata.region` is negative. Similarly, $\eta_r = 1$ where the i_2 component of `_spdata.region` is positive while $\eta_r = \eta_{r,0}$ where the i_2 component of `_spdata.region` is negative.

This highly contrived diffusion model was designed for solar wind applications in which the i_1 indicator variable separates the heliosphere from the local insterstellar medium (LISM) and the i_2 indicator variable distinguishes between sectored and unipolar magnetic field regions. For example, the `BackgroundSolarWind` class and its derived classes do this for $i_1 = 0$ and $i_2 = 1$.

→ Subsection 4.5

The following inputs are required to build this class:

`LISM_idx` : Index i_1.
Possible values: Any non-negative integer (between 0 and 2).

`lambda_in` : Reference parallel mean free path where the i_1 component of `_spdata.region` is positive.
Possible values: Any positive double.

`lambda_out` : Reference parallel mean free path where the i_1 component of `_spdata.region` is negative.
Possible values: Any positive double.

`R0` : Reference rigidity.
Possible values: Any positive double.

`B0` : Reference magnetic field magnitude.
Possible values: Any positive double.

`kap_rat_in` : Ratio of perpendicular to parallel diffusion coefficients where the i_1 component of `_spdata.region` is positive.
Possible values: Any positive double.

`kap_rat_out` : Ratio of perpendicular to parallel diffusion coefficients where the i_1 component of `_spdata.region` is negative.
Possible values: Any positive double.

`Bmix_idx` : Index i_2.
Possible values: Any non-negative integer (between 0 and 2).

`kap_rat_red` : Reduction factor for the ratio of perpendicular to parallel diffusion coefficients where the i_2 component of `_spdata.region` is negative.
Possible values: Any positive double.

9 Common Modules

In this section, we describe the modules within the `$SPECTRUM/common` folder currently supported within the existing SPECTRUM architecture.

9.1 MPI_Config

The `MPI_Config` structure is designed to help coordinate the MPI communications between the SimW, SimS, and SimM processes. The structure holds four MPI Communicator objects, a global communicator (`glob_comm`), a node communicator (`node_comm`), and server communicator (`boss_comm`)[14], and a worker communicator (`work_comm`), along with the size and rank information for each one. Auxiliary variables are also defined to speed up the communicatior process, such as booleans to help determine the role of a CPU within the MPI scheme (`is_worker`, `is_boss`, `is_master`), counts for number of nodes (`n_nodes`), number of workers (`n_workers`), number of workers per node (`workers_in_node`), number of physical sockets per node (`n_sockets_per_node`), and number of servers per physical socket (`n_bosses_per_socket`).

The communicators are automatically built at the beginning of the code execution according to the following procedure:

1. A node communicator is split for the global communicator. This split is based on the `MPI_COMM_TYPE_SHARED` split type.
2. Each node communicator is subdivided into more node communicators if multiple servers per node where requested. The first split in this process is done using the `OMPI_COMM_TYPE_SOCKET` split type, since CPUs in the same socket will share memory and should communicate faster, and further splits will take place if necessary.
3. All the SimS processes, which have rank 0 within their respectively node communicator, will coalesce to form the server communicator. The SimM process, which has rank 0 in the global communicator, is always present in the server communicator.
4. Finally, all SimW processes will coalesce to form the worker communicator. The SimM process is also present in the worker communicator.

The file `$SPECTRUM/common/mpi_config.hh` defines the following pre-processor macros:

`ALLOW_MASTER_BOSS` : This macro controls whether the master process is also allowed to be a server. It is automatically set during the configuration stage.

`ALLOW_BOSS_WORKER` : This macro controls whether server processes are also allowed to be workers. It is automatically set during the configuration stage.

`N_BOSSES_PER_NODE` : This macro controls the number of servers per hardware node. Possible values: any positive integer.

`NEED_SERVER` : This macro controls whether server processes are needed. It is automatically set during the configuration stage.

9.2 DataContainer

The `DataContainer` class is designed to help pass arbitrary sets of parameters within functions, mainly for the purposes of construction and initialization of derived classes. The data is contiguously stored in memory with a binary format

[14] The "boss" terminology is an old nomenclature that refers to servers. The use of this term persists in the code but should be thought of as a synonym for "server".

(_storage) and a tally for the number of parameters (n_records) and the total size of the data array (total_size) are kept separately.

The templated functions

```
template <typename T> void Insert(T data);
template <typename T> void Insert(std::vector<T> data);
```

add parameters to the DataContainer object as binary chunks, while the templated functions

```
template <typename T> void Read(T* data_ptr);
template <typename T> void Read(std::vector<T>* data_ptr);
```

read those parameters sequentially. The function

```
void Clear(void);
```

will delete all the stored parameters, updating the parameter count and total size to zero. A private pointer member (_params) helps keep track of which parameters have been read already. The function

```
void Reset(void);
```

will reset the private pointer to the start of the binary chunk array.

9.3 SimpleArray

The SimpleArray templated structure is designed to facilitate operations and manipulations of other derived structures holding ordered collections of numbers, like MultiIndices or GeoVectors. Simple arrays are templated by the type of variables (data_type) and how many variables (n_vars) they hold. The first three variables they hold can be accessed directly with [0], [1], and [2], but also with .x, .y, and .z or .i, .j, and .k for convenience.

The basic arithmetic operators (=, +=, -=, *=, /=, %=) have been overloaded to work component-wise and with a single variable of the same type stored in the SimpleArray. Note that part of this functionality is implemented in the $SPECTRUM/common/arithmetic.hh file. The templated functions

```
data_type Norm2(void) const;
data_type Sum(void) const;
data_type Prod(void) const;
data_type Smallest(void) const;
data_type Largest(void) const;
void Negate(void);
data_type ScalarProd(const SimpleArray& other) const;
```

implement other useful operations typically of ordered collections of numbers. The function

```
int size(void) const;
```

returns the number of components in the array, while the templated functions

```
const data_type* Data(void) const;
data_type* Data(void);
```

returns a pointer to the data for reading and writing, respectively.

9.4 MultiIndex

The `MultiIndex` structure is derived from `SimpleArray` with integer variable types and 3 components. Aside from the arithmetic operations inherited from `SimpleArray`, other logical and arithmetic operators have been overloaded for convenience:

```
MultiIndex& operator ++(void);
MultiIndex& operator --(void);
friend bool operator >(const MultiIndex& left, const MultiIndex& right);
friend bool operator <(const MultiIndex& left, const MultiIndex& right);
friend bool operator >=(const MultiIndex& left, const MultiIndex& right);
friend bool operator <=(const MultiIndex& left, const MultiIndex& right);
friend bool operator ==(const MultiIndex& left, const MultiIndex& right);
friend bool operator !=(const MultiIndex& left, const MultiIndex& right);
```

In addition, the function

```
long LinIdx(const MultiIndex& other) const;
```

computes a linear index in a 3D array using a second MultiIndex as the dimension lengths, which is extremely useful in distribution binning, and the function

→ Subsection 5.1

```
void Flip(void);
```

flips the order of the components. Keep in mind that MultiIndex objects are typically used to hold non-negative integer values which represent the length along each dimension of a 3D array. In order to preserve the flexibility of this object for other purposes, this is not strictly required by the logic of the class itself and the user is responsible for checking that the components are set properly.

The file `$SPECTRUM/common/vector.hh` defines the following pre-processor macros and global constants:

`SZMI` : This macro is the size (in bytes) of a MultiIndex object.

`mi_zeros` : This is the zero multi-index.

`mi_ones` : This is a multi-index with 1 in all components.

9.5 GeoVector

The `GeoVector` structure is derived from `SimpleArray` with double variable types and 3 components. Aside from the arithmetic operations inherited from `SimpleArray`, a vector product (`^=`) is defined for convenience. The functions

```
double Norm(void) const;
GeoVector& Normalize(void);
GeoVector& Normalize(double& norm);
double Theta(void);
double Phi(void);
void RTP_XYZ(void);
void XYZ_RTP(void);
void ToSpherical(double sintheta, double costheta, double sinphi, double
    cosphi);
void ToCartesian(double sintheta, double costheta, double sinphi, double
    cosphi);
void Rotate(const GeoVector& n, double sina, double cosa);
void Rotate(const GeoVector& axis, double angle);
void SubtractParallel(const GeoVector& axis);
void ChangeToBasis(const GeoVector* basis);
void ChangeToBasis2(const GeoVector* basis);
```

```
void ChangeFromBasis(const GeoVector* basis);
void ProjectToPlane(int projection, const GeoVector& normal, const
    GeoVector& north, double& xi, double& eta) const;
bool InsideWedge(int n_verts, const GeoVector* verts, double tol) const;
```

implement a variety routines helpful in common vector manipulations.

The file `$SPECTRUM/common/vector.hh` defines the following pre-processor macros and global constants:

`SZGV` : This macro is the size (in bytes) of a GeoVector object.

`gv_nx` : This is the unit-vector in the **x** direction.

`gv_ny` : This is the unit-vector in the **y** direction.

`gv_nz` : This is the unit-vector in the **z** direction.

`gv_nr` : This is the unit-vector in the **r** direction.

`gv_nt` : This is the unit-vector in the θ direction.

`gv_np` : This is the unit-vector in the ϕ direction.

`gv_zeros` : This is the zero vector.

`gv_ones` : This is a vector with 1 in all components.

`cart_unit_vec` : This is an array of 3 GeoVectors representing the standard Cartesian basis.

9.6 GeoMatrix

The `GeoMatrix` structure holds 3 GeoVectors representing the 3 rows of a 3×3 matrix. For convenience, the 9 (double type) components are also stored contiguously in an array (`linear`) for fast access. Various useful matrix operations have been implemented in this structure, including but not limited to trace, transpose, determinant, and basic arithmetic operations with scalars (doubles), vectors (GeoVectors) and other matrices (GeoMatrices) in the following functions

```
double Trace(void) const;
GeoMatrix& Transpose(void);
void Transpose(const GeoMatrix& matr_in);
GeoMatrix& operator +=(const GeoMatrix& matr_r);
GeoMatrix& operator -=(const GeoMatrix& matr_r);
GeoMatrix& operator *=(double sclr_r);
GeoMatrix& operator /=(double sclr_r);
double Det(void) const;
void Dyadic(const GeoVector& vect);
void Dyadic(const GeoVector& vect_l, const GeoVector& vect_r);
void IncrementCov(const GeoVector& vect_l, const GeoVector& vect_r);
void AxisymmetricBasis(const GeoVector& ez);
void ChangeToBasis(const GeoMatrix& basis);
void ChangeFromBasis(const GeoMatrix& basis);
double Minor(int i, int j) const;
GeoMatrix Inverse(void) const;
GeoVector Eigenvalues(void) const;
GeoVector Eigensystem(GeoMatrix& evec) const;
```

The file `$SPECTRUM/common/matrix.hh` defines the following pre-processor macros and global constants:

`SZGM` : This macro is the size (in bytes) of a GeoMatrix object.

`gm_unit` : This is the identity matrix.

`gm_zeros` : This is the zero matrix.

`gm_ones` : This is a matrix with 1 in all components.

9.7 Params

The `Params` class is a stub (common ancestor) for nearly every module in the `$SPECTRUM/src` folder. It holds basic state information like a double for the current time, t (`_t`), and GeoVectors for position, $\mathbf{r}$ (`_pos`), velocity, $\mathbf{v}$ (`_vel`), and momentum, $\mathbf{p}$ (`_mom`). Additionally, it possesses other members that are useful for applications of derived classes: a string to hold the class name (`class_name`), an unsigned integer to identify the specie associated with that class (`specie`), a random number generator if necessary (`rng`), a DataContainer object for construction and setup, and a status flag used at various points in the code (`_status`).

9.8 SpatialData

The `SpatialData` structure is designed to encode the background information (scalars, vector fields, and derivatives) at a particular point in time and space. Its data members include GeoVectors for the plasma velocity, $\mathbf{u}$ (`Uvec`), magnetic field, $\mathbf{B}$ (`Bvec`), and electric field, $\mathbf{E}$ (`Evec`), along with GeoMatrices for their spatial gradients, $\nabla\mathbf{X}$ (`gradXvec`), and GeoVectors for their time derivatives, $d\mathbf{X}/dt$ (`dXdt`). The derivatives are stored using the standard gradient notation,

$$\nabla\mathbf{X}_{ij} = \frac{\partial X_j}{\partial x^i}. \tag{153}$$

Note that there are dedicated variables for the magnetic field magnitude, B (`Bmag`), and direction, $\mathbf{b}$ (`bhat`), as well as the gradient/time derivative of B.

This structure also stores some scalar background quantities like number density, n (`n_dens`), thermal pressure, P (`p_ther`) and minimum and maximum magnetic field magnitudes encountered during a pseudo-trajectory integration, $B_{\min}$ (`Bmag_min`) and $B_{\max}$ (`Bmag_max`). Additionally, the maximum spatial displacement, $\Delta r_{\max}$ (`dmax`) is kept for the timestep CFL conditions. The GeoVector `region` holds indicator variables for the background.

The file `$SPECTRUM/common/spatial_data.hh` defines series of global constants beginning with `BACKGROUND_` that help signal the background modules which fields should be computed. These constants act like bit flags raised or lowered in the `_mask` member of this structure. The constants containing `U`, `B`, and `E` refer to the bulk velocity, magnetic field, and electric field, respetively, while those with `ALL` compute all three fields. The constants featuring `grad` are associated with spatial derivatives, while those with `d*dt` are for time derivatives. The variables with `_dr` and `_dt` in their names are used for numerical derivative calculations.

The spatial derivative information is fully contained in the GeoMatrix objects previously mentioned, and specific differential operators, like divergence or curl, are computed from these using the following functions

```
double divU(void);
double divB(void);
double divE(void);
GeoVector curlU(void);
GeoVector curlB(void);
GeoVector curlE(void);
```

The derivatives of $\mathbf{b}$ can also be derived from $\nabla\mathbf{B}$ and ∇B, and these relations are exploited in the following functions

```
double divbhat(void);
GeoVector curlbhat(void);
GeoMatrix gradbhat(void);
GeoVector dbhatdt(void);
```

9.9 TurbProp

→ Subsection 4.10

The `TurbProp` structure is designed to encode the turbulent properties of a specific turbulent geometry for use in the `BackgroundWaves` structure. Four turbulent geometries (modes) are offered: Alfvén or slab (A), transverse 2D (T), longitudinal 2D (L), and isotropic (I). The enumeration type `turb_type` refers to these geometries. Its data members include doubles for the smallest and the largest wavenumbers, $k_{\min}$ (`kmin`) and $k_{\max}$ (`kmax`), the bendover length, l_0 (`l0`), the number of wave modes, N (`n_waves`), the magnetic variance, $\langle \delta B^2 \rangle$ (`variance`) and the power law slope, γ (`slope`).

Index