Hermes-3 Documentation

Release 0.1.0

The BOUT++ team

Jun 12, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

Hermes-3 is a plasma simulation model built on BOUT++, developed mainly for simulating the edge of magnetically confined plasmas such as tokamaks. The source code is available on Github. The main aim of this model is multi-species simulation of fusion reactors, where the plasma will contain a mixture of deuterium, tritium, helium and other species.

An unusual feature of this model is that it is organised into reusable components, which can be tested individually and then configured at run-time. For example a transport simulation with deuterium and tritium ions and atoms has an input file specifying the components

```
[hermes]
components = d+, d, t+, t, e, collisions, sheath_boundary, recycling, reactions
```

The governing equations for each species are specified e.g.

```
[d+]
type = evolve_density, evolve_momentum, evolve_pressure, anomalous_diffusion
AA = 2   # Atomic mass
charge = 1
```

and other components have their configuration options e.g. for reactions

```
[reactions]
type = (
        d + e -> d+ + 2e,   # Deuterium ionisation
        t + e -> t+ + 2e,   # Tritium ionisation
       )
```

CHAPTER TWO

GETTING STARTED

2.1 Installing

Only CMake is supported for building Hermes-3 and running the tests. During configuration BOUT++ will be automatically downloaded as a submodule, together with some dependencies. NetCDF and FFTW are assumed to be installed already. The SUNDIALS library is strongly recommended for time-dependent simulations, and PETSc is needed to run some of the steady-state transport solver examples.

If you only want to run time-dependent simulations, then the recommended way to build Hermes-3 links to the SUNDIALS library:

1. Configure with cmake, downloading and linking to SUNDIALS:

   ```
   cmake . -B build -DBOUT_DOWNLOAD_SUNDIALS=ON
   ```

2. Build, compiling Hermes-3 and all dependencies:

   ```
   cmake --build build
   ```

3. Run the unit and integrated tests to check that everything is working:

   ```
   cd build
   ctest
   ```

Note that the integrated tests require MPI, and so may not run on the head nodes of many computing clusters.

The CMake configuration can be customised: See the [BOUT++ documentation](https://bout-dev.readthedocs.io/en/latest/user_docs/installing.html#cmake) for examples of using `cmake` arguments, or edit the compile options interactively before building:

```
ccmake . -B build
```

If you have already installed BOUT++ and want to use that rather than configure and build BOUT++ again, set `HERMES_BUILD_BOUT` to `OFF` and pass CMake the path to the BOUT++ `build` directory e.g.

```
cmake . -B build -DHERMES_BUILD_BOUT=OFF -DCMAKE_PREFIX_PATH=$HOME/BOUT-dev/build
```

Note that Hermes-3 currently requires BOUT++ version 5.

2.2 Building with PETSC

When building PETSc it is recommended to include `hypre`. The following PETSc configure line is a good starting point:

```
./configure --with-mpi=yes --download-hypre --download-make --with-fortran-bindings=0 --
↪with-debugging=0
```

To configure Hermes-3 with PETSc, use the `-DBOUT_USE_PETSC=ON` flag:

```
cmake . -B build -DBOUT_DOWNLOAD_SUNDIALS=ON -DBOUT_USE_PETSC=ON
```

If the `PETSC_DIR` and `PETSC_ARCH` environment variables have been set, then CMake should pick them up.

2.3 Numerical methods

Advection operators in Hermes-3 use slope limiters, also called `flux limiters <https://en.wikipedia.org/wiki/Flux_limiter` to suppress spurious numerical oscillations near sharp features, while converging at 2nd-order in smooth regions. In general there is a trade-off between suppression of numerical oscillations and dissipation: Too little dissipation results in oscillations that can cause problems (e.g. negative densities), while too much dissipation smooths out real features and requires higher resolution to converge to the same accuracy. The optimal choice of method is problem-dependent.

The CMake option `HERMES_SLOPE_LIMITER` sets the choice of slope limiter. The default method is `MinMod`, which has been found to provide a good balance for problems of interest. If less dissipation is required then this can be changed to `MC` (for Monotonized Central); For more dissipation (but 1st-order convergence) change it to `Upwind`.

CHAPTER

THREE

EXAMPLES

3.1 1D flux-tube

These simulations follow the dynamics of one or more species along the magnetic field. By putting a source at one end of the domain, and a sheath at the other, this can be a useful model of plasma dynamics in the Scrape-Off Layer (SOL) of a tokamak or other magnetised plasma.

3.1.1 1D periodic domain, Te and Ti

A fluid is evolved in 1D, imposing quasineutrality and zero net current. Both electron and ion pressures are evolved, but there is no exchange of energy between them, or heat conduction.

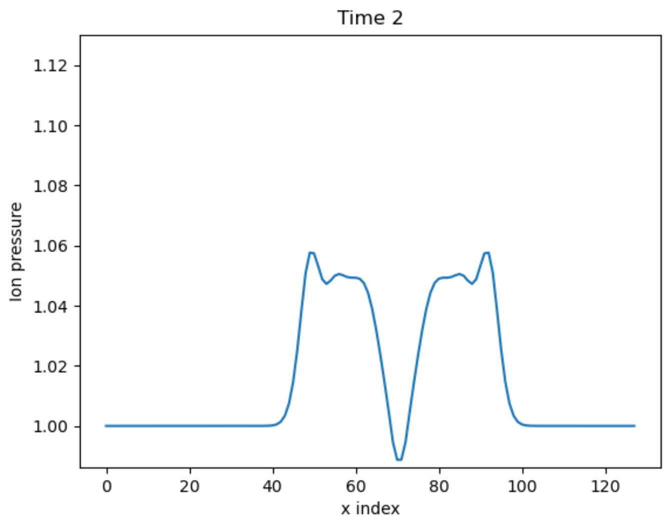


Fig. 3.1: Evolution of pressure, starting from a top hat

The model components are ions (i) and electrons (e), and a component which uses the force on the electrons to calculate the parallel electric field, which transfers the force to the ions.

```
[hermes]
components = i, e, electron_force_balance
```

The ion density, pressure and momentum equations are evolved:

```
[i]  # Ions
type = evolve_density, evolve_pressure, evolve_momentum
```

which solves the equations

$$\begin{aligned}\frac{\partial n_i}{\partial t} &= -\nabla\cdot\left(n_i\mathbf{b}v_{||i}\right)\\ \frac{\partial p_i}{\partial t} &= -\nabla\cdot\left(p_i\mathbf{b}v_{||i}\right) - \frac{2}{3}p_i\nabla\cdot\left(\mathbf{b}v_{||i}\right)\\ \frac{\partial}{\partial t}\left(n_iv_{||i}\right) &= -\nabla\cdot\left(n_iv_{||i}\mathbf{b}v_{||i}\right) - \partial_{||}p_i + E\end{aligned}$$

The electron density is set to the ion density by quasineutrality, the parallel velocity is set by a zero current condition, and only the electron pressure is evolved.

```
[e] # Electrons
type = quasineutral, zero_current, evolve_pressure
```

which adds the equations:

$$\begin{aligned}n_e &= n_i\\ \frac{\partial p_e}{\partial t} &= -\nabla\cdot\left(p_e\mathbf{b}v_{||e}\right) - \frac{2}{3}p_e\nabla\cdot\left(\mathbf{b}v_{||e}\right)\end{aligned}$$

The *zero_current* component sets:

$$\begin{aligned}E &= -\partial_{||}p_e\\ v_{||e} &= v_{||i}\end{aligned}$$

3.1.2 1D Scrape-off Layer (SOL)

This simulates a similar setup to the SD1D code: A 1D domain, with a source of heat and particles on one side, and a sheath boundary on the other. Ions recycle into neutrals, which charge exchange and are ionised. A difference is that separate ion and electron temperatures are evolved here.

Due to the short length-scales near the sheath, the grid is packed close to the target, by setting the grid spacing to be a linear function of index:

```
[mesh]
dy = (length / ny) * (1 + (1-dymin)*(1-y/pi))
```

where `dymin` is 0.1 here, and sets the smallest grid spacing (at the target) as a fraction of the average grid spacing.

The components are ion species `d+`, atoms `d`, electrons `e`:

```
[hermes]
components = (d+, d, e,
             zero_current, sheath_boundary, collisions, recycling, reactions,
             neutral_parallel_diffusion)
```

The electron velocity is set to the ion by specifying *zero_current*; A sheath boundary is included; Collisions are needed to be able to calculate heat conduction, as well as neutral diffusion rates; Recycling at the targets provides a source of atoms; *neutral_parallel_diffusion* simulates cross-field diffusion in a 1D system.

The sheath boundary is only imposed on the upper Y boundary:

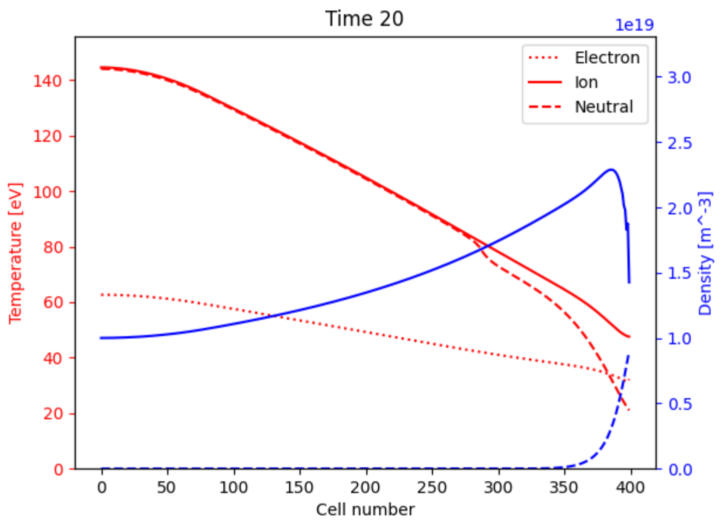


Fig. 3.2: Evolution of ion and neutral density (blue); ion, electron and neutral temperature (red), starting from flat profiles.

```
[sheath_boundary]

lower_y = false
upper_y = true
```

The reactions component is a group, which lists the reactions included:

```
[reactions]
type = (
        d + e -> d+ + 2e,   # Deuterium ionisation
        d + d+ -> d+ + d,   # Charge exchange
       )
```

To run this example:

```
nice -n 10 ./hermes-3 -d examples/1D-recycling
```

This should take 5-10 minutes to run. There is a `makeplots.py` script in the `examples/1D-recycling` directory which will generate plots and a gif animation (if ImageMagick is installed).

3.2 2D drift-plane

Simulations where the dynamics along the magnetic field is not included, or only included in a parameterised way as sources or sinks. These are useful for the study of the basic physics of plasma "blobs" / filaments, and tokamak edge turbulence.

3.2.1 Blob2d

A seeded plasma filament in 2D. This version is isothermal and cold ion, so only the electron density and vorticity are evolved. A sheath-connected closure is used for the parallel current.

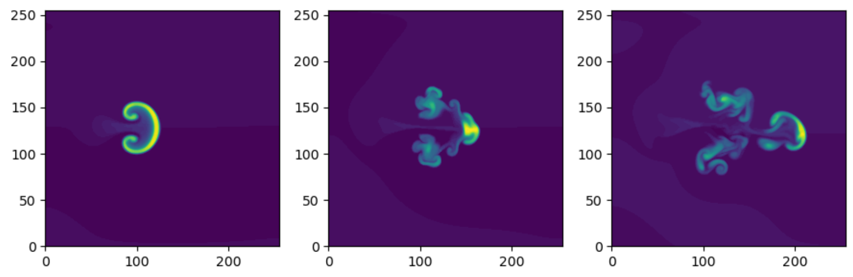


Fig. 3.3: Electron density Ne at three times, showing propagation to the right

The model components are

```
[hermes]
components = e, vorticity, sheath_closure
```

The electron component consists of two types:

```
[e]  # Electrons
type = evolve_density, isothermal
```

The *evolve_density* component type evolves the electron density `Ne`. This component has several options, which are set in the same section e.g.

```
poloidal_flows = false  # Y flows due to ExB
```

and so solves the equation:

$$\frac{\partial n_e}{\partial t} = -\nabla\cdot\left(n_e\mathbf{v}_{E\times B}\right) + \nabla\cdot\frac{1}{e}\mathbf{j}_{sh}$$

The *isothermal* component type sets the temperature to be a constant, and using the density then sets the pressure. The constant temperature is also set in this `[e]` section:

```
temperature = 5  # Temperature in eV
```

so that the equation solved is

$$p_e = en_eT_e$$

where T_e is the fixed electron temperature (5eV).

The *vorticity* component uses the pressure to calculate the diamagnetic current, so must come after the `e` component. This component then calculates the potential. Options to control the vorticity component are set in the `[vorticity]` section.

$$\frac{\partial \omega}{\partial t} = -\nabla\cdot\left(\omega\mathbf{v}_{E\times B}\right) + \nabla\left(p_e\nabla\times\frac{\mathbf{b}}{B}\right) + \nabla\cdot\mathbf{j}_{sh}$$

$$\nabla\cdot\left(\frac{1}{B^2}\nabla_\perp\phi\right) = \omega$$

The `sheath_closure` component uses the potential, so must come after *vorticity*. Options are also set as

```
[sheath_closure]
connection_length = 10 # meters
```

This adds the equation

$$\nabla \cdot \mathbf{j}_{sh} = \frac{n_e \phi}{L_{||}}$$

where $L_{||}$ is the connection length.

3.2.2 Blob2D-Te-Ti

A seeded plasma filament in 2D. This version evolves both electron and ion temperatures. A sheath-connected closure is used for the parallel current.

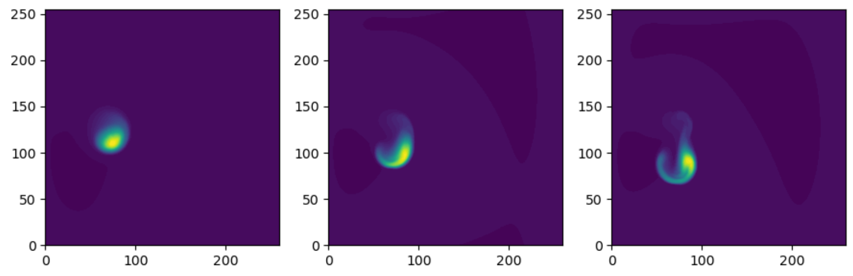


Fig. 3.4: Electron density Ne at three times, showing propagation to the right and downwards

The model components are

```
[hermes]
components = e, h+, vorticity, sheath_closure
```

The electron component evolves density (saved as `Ne`) and pressure (`Pe`), and from these the temperature is calculated.

```
[e]
type = evolve_density, evolve_pressure
```

The ion component sets the ion density from the electron density, by using the quasineutrality of the plasma; the ion pressure (`Ph+`) is evolved.

```
[h+]
type = quasineutral, evolve_pressure
```

The equations this solves are similar to the previous *Blob2d* case, except now there are pressure equations for both ions and electrons:

$$
\begin{aligned}
\frac{\partial n_e}{\partial t} =& - \nabla\cdot\left(n_e \mathbf{v}_{E\times B}\right) + \nabla\cdot\frac{1}{e}\mathbf{j}_{sh} \\
\frac{\partial p_e}{\partial t} =& - \nabla\cdot\left(p_e \mathbf{v}_{E\times B}\right) - \gamma_e p_e c_s \\
n_{h+} =& n_e \\
\frac{\partial p_{h+}}{\partial t} =& - \nabla\cdot\left(p_{h+} \mathbf{v}_{E\times B}\right) \\
\frac{\partial \omega}{\partial t} =& - \nabla\cdot\left(\omega \mathbf{v}_{E\times B}\right) + \nabla\left[\left(p_e + p_{h+}\right)\nabla\times\frac{\mathbf{b}}{B}\right] + \nabla\cdot\mathbf{j}_{sh} \\
\nabla\cdot\left[\frac{1}{B^2}\nabla_\perp\left(\phi + p_{h+}\right)\right] =& \omega \\
\nabla\cdot\mathbf{j}_{sh} =& \frac{n_e\phi}{L_{||}}
\end{aligned}
$$

3.2.3 2D-drift-plane-turbulence-te-ti

A 2D turbulence simulation, similar to the *Blob2D-Te-Ti* case, but with extra source and sink terms, so that a statistical steady state of source-driven turbulence can be reached.

The model components are

```
[hermes]
components = e, h+, vorticity, sheath_closure
```

The electron component evolves density (saved as `Ne`) and pressure (`Pe`), and from these the temperature is calculated.

```
[e]
type = evolve_density, evolve_pressure
```

The ion component sets the ion density from the electron density, by using the quasineutrality of the plasma; the ion pressure (Ph+) is evolved.

```
[h+]
type = quasineutral, evolve_pressure
```

The sheath closure now specifies that additional sink terms should be added

```
[sheath_closure]
connection_length = 50 # meters
potential_offset = 0.0  # Potential at which sheath current is zero
sinks = true
```

and radially localised sources are added in the `[Ne]`, `[Pe]`, and `[Ph+]` sections.

The equations this solves are the same as the previous *Blob2D-Te-Ti* case, except wih extra source and sink terms. In

SI units (except temperatures in eV) the equations are:

$$
\begin{aligned}
p_{\text{total}} =& \sum_a e n_a T_a \\
\rho_{\text{total}} =& \sum_a A_a m_p n_a \\
c_s =& \sqrt{\frac{p_{\text{total}}}{\rho_{\text{total}}}} \\
\frac{\partial n_e}{\partial t} =& -\nabla\cdot\left(n_e\mathbf{v}_{E\times B}\right) + \nabla\cdot\frac{1}{e}\mathbf{j}_{sh} - \frac{n_e c_s}{L_{||}} + S_n \\
\frac{\partial p_e}{\partial t} =& -\nabla\cdot\left(p_e\mathbf{v}_{E\times B}\right) - \frac{\gamma_e p_e c_s}{L_{||}} + S_{p_e} \\
n_{h+} =& n_e \\
\frac{\partial p_{h+}}{\partial t} =& -\nabla\cdot\left(p_{h+}\mathbf{v}_{E\times B}\right) - \frac{\gamma_i p_{h+} c_s}{L_{||}} + S_{p_{h+}} \\
\frac{\partial \omega}{\partial t} =& -\nabla\cdot\left(\omega\mathbf{v}_{E\times B}\right) + \nabla\cdot\left[\left(p_e + p_{h+}\right)\nabla\times\frac{\mathbf{b}}{B}\right] + \nabla\cdot\mathbf{j}_{sh} \\
\nabla\cdot\left[\frac{\overline{A}m_p}{B^2}\left(\overline{n}\nabla_\perp\phi + \nabla_\perp p_{h+}\right)\right] =& \omega \\
\nabla\cdot\mathbf{j}_{sh} =& \frac{e n_e \overline{c_s}\phi}{\overline{T}L_{||}} \\
\mathbf{v}_{E\times B} =& \frac{\mathbf{B}\times\nabla\phi}{B^2}
\end{aligned}
$$

Where $\overline{T}$ and $\overline{n}$ are the reference temperature (units of eV) and density (in units of m^{-3}) used for normalisation. $\overline{c_s} = \sqrt{e\overline{T}/m_p}$ is the reference sound speed, where m_p is the proton mass. The mean ion atomic mass $\overline{A}$ is set to 1 here.

These reference values enter into the sheath current $\mathbf{j}_{sh}$ because that is a simplified, linearised form of the full expression. Likewise the vorticity (ω) equation used the Boussinesq approximation to simplify the polarisation current term, leading to constant reference values being used.

The sheath heat transmission coefficients default to $\gamma_e = 6.5$ and $\gamma_i = 2.0$ (γ_i as suggested in Stangeby's textbook between equations (2.92) and (2.93)). Note the sinks in may not be correct or the best choices, especially for cases with multiple ion species; they were chosen as being simple to implement by John Omotani in May 2022.

3.3 2D axisymmetric tokamak

These are transport simulations, where the cross-field transport is given by diffusion, and fluid-like equations are used for the parallel dynamics (as in the 1D flux tube cases).

The input settings (in BOUT.inp) are set to read the grid from a file `tokamak.nc`. This is linked to a default file `compass-36x48.grd.nc`, a COMPASS-like lower single null tokamak equilibrium. Due to the way that BOUT++ uses communications between processors to implement branch cuts, these simulations require a multiple of 6 processors. You don't usually need 6 physical cores to run these cases, if MPI over-subscription is enabled.

3.3.1 heat-transport

In `examples/tokamak/heat-transport`, this evolves only electron pressure with a fixed density. It combines cross-field diffusion with parallel heat conduction and a sheath boundary condition.

To run this simulation with the default inputs requires (at least) 6 processors because it is a single-null tokamak grid. From the build directory:

```
cd examples/tokamak
mpirun -np 6 ../../hermes-3 -d heat-transport
```

That will read the grid from `tokamak.nc`, which by default links to the `compass-36x48.grd.nc` file.

The components of the model are given in `heat-transport/BOUT.inp`:

```
[hermes]
components = e, h+, collisions, sheath_boundary_simple
```

We have two species, electrons and hydrogen ions, and add collisions between them and a simple sheath boundary condition.

The electrons have the following components to fix the density, evolve the pressure, and include anomalous cross-field diffusion:

```
[e]
type = fixed_density, evolve_pressure, anomalous_diffusion
```

The `fixed_density` takes these options:

```
AA = 1/1836
charge = -1
density = 1e18 # Fixed density [m^-3]
```

so in this simulation the electron density is a uniform and constant value. If desired, that density can be made a function of space (`x` and `y` coordinates).

The `evolve_pressure` component has thermal conduction enabled, and outputs extra diagnostics i.e. the temperature `Te`:

```
thermal_conduction = true   # Spitzer parallel heat conduction
diagnose = true   # Output additional diagnostics
```

There are other options that can be set to modify the behavior, such as setting `kappa_limit_alpha` to a value between 0 and 1 to impose a free-streaming heat flux limit.

Since we're evolving the electron pressure we should set initial and boundary conditions on `Pe`:

```
[Pe]
function = 1
bndry_core = dirichlet(1.0)  # Core boundary high pressure
bndry_all = neumann
```

That sets the pressure initially uniform, to a normalised value of 1, and fixes the pressure at the core boundary. Other boundaries are set to zero-gradient (neumann) so there is no cross-field diffusion of heat out of the outer (SOL or PF) boundaries. Flow of heat through the sheath is governed by the `sheath_boundary_simple` top-level component.

The hydrogen ions need a density and temperature in order to calculate the collision frequencies. If the ion temperature is fixed to be the same as the electron temperature then there is no transfer of energy between ions and electrons:

```
[h+]
type = quasineutral, set_temperature
```

The `quasineutral` component sets the ion density so that there is no net charge in each cell. In this case that means the hydrogen ion density is set equal to the electron density. To perform this calculation the component requires that the ion atomic mass and charge are specified:

```
AA = 1
charge = 1
```

The `set_temperature` component sets the ion temperature to the temperature of another species. The name of that species is given by the `temperature_from` option:

```
temperature_from = e  # Set Th+ = Te
```

The `collisions` component is described in the manual, and calculates both electron-electron and electron-ion collisions. These can be disabled if desired, using individual options. There are also ion-ion, electron-neutral, ion-neutral and neutral-neutral collisions that are not used here.

The `sheath_boundary_simple` component is a simplified Bohm-Chodura sheath boundary condition, that allows the sheath heat transmission coefficient to be specified for electrons and (where relevant) for ions.

The equations solved by this example are:

$$
\begin{aligned}
\frac{3}{2}\frac{\partial P_e}{\partial t} =& \nabla\cdot\left(\kappa_{e||}\mathbf{b}\mathbf{b}\cdot\nabla T_e\right) + \nabla\cdot\left(n_e\chi\nabla_\perp T_e\right) \\
\kappa_{e||} =& 3.16 P_e\tau_e/m_e \\
\tau_e =& 1/\left(\nu_{ee}+\nu_{ei}\right) \\
\nu_{ee} =& \frac{2e^4 n_e\ln\Lambda_{ee}}{3\epsilon_0^2 m_e^2\left(4\pi e T_e/m_e\right)^{3/2}} \\
\ln\Lambda_{ee} =& 30.4 - \frac{1}{2}\ln n_e + \frac{5}{4}\ln T_e - \sqrt{10^{-5} + \left(\ln T_e - 2\right)^2/16} \\
\nu_{ei} =& \frac{e^4 n_e\ln\Lambda_{ei}\left(1 + m_e/m_i\right)}{3\epsilon_0^2 m_e^2\left(2\pi T_e(1/m_e + 1/m_i)\right)^{3/2}} \\
\ln\Lambda_{ei} =& 31 - \frac{1}{2}\ln n_e + \ln T_e
\end{aligned}
$$

The calculation of the Coulomb logarithms follows the NRL formulary, and the above expression is used for temperatures above 10eV. See the `collisions` manual section for the expressions used in other regimes.

3.3.2 recycling-dthene

The `recycling-dthene` example includes cross-field diffusion, parallel flow and heat conduction, collisions between species, sheath boundary conditions and recycling. It simulates the density, parallel flow and pressure of the electrons; ion species D+, T+, He+, Ne+; and neutral species D, T, He, Ne.

The model components are a list of species, and then collective components which couple multiple species.

```
[hermes]
components = (d+, d, t+, t, he+, he, ne+, ne, e,
              collisions, sheath_boundary, recycling, reactions)
```

Note that long lists like this can be split across multiple lines by using parentheses.

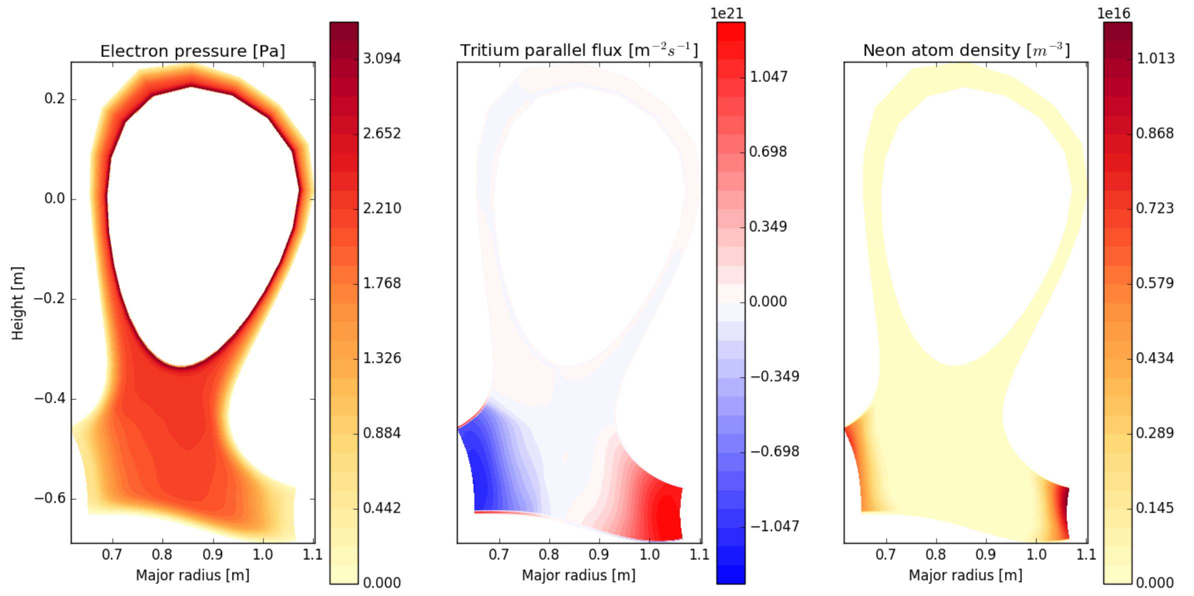


Fig. 3.5: Electron pressure, parallel tritium flux, and neon atom density. Simulation evolves D, T, He, Ne and electron species, including ions and neutral atoms.

Each ion species has a set of components, to evolve the density, momentum and pressure. Anomalous diffusion adds diffusion of particles, momentum and energy. For example deuterium ions contain:

```
[d+]
type = evolve_density, evolve_momentum, evolve_pressure, anomalous_diffusion
AA = 2
charge = 1
```

Atomic reactions are specified as a list:

```
[reactions]
type = (
     d + e -> d+ + 2e,   # Deuterium ionisation
     t + e -> t+ + 2e,   # Tritium ionisation
     he + e -> he+ + 2e, # Helium ionisation
     he+ + e -> he,      # Helium+ recombination
     ne + e -> ne+ + 2e, # Neon ionisation
     ne+ + e -> ne,      # Neon+ recombination
    )
```

CHAPTER

FOUR

TESTS

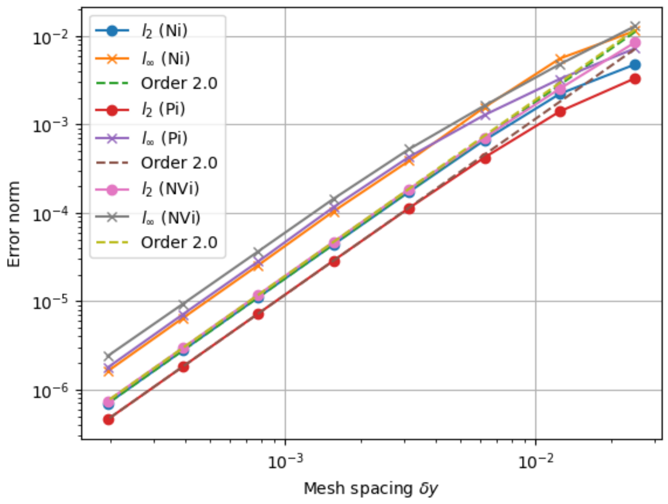
The specification of the Toro tests used here is taken from Walker (2012), originally from Toro's book Riemann Solvers and Numerical Methods for Fluid Dynamics.

4.1 1D fluid (MMS)

`tests/integrated/1D-fluid`

This convergence test using the Method of Manufactured Solutions (MMS) solves fluid equations in the pressure form:

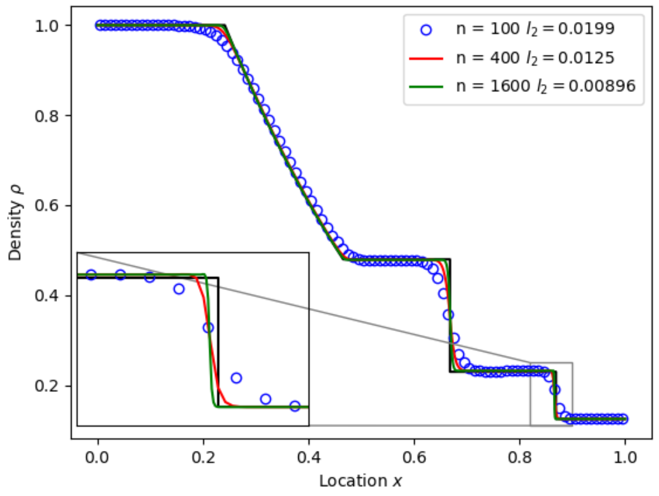
$$\begin{aligned}\frac{\partial n}{\partial t} &= -\nabla\cdot\left(n\mathbf{b}v_{||}\right)\\ \frac{\partial p}{\partial t} &= -\nabla\cdot\left(p\mathbf{b}v_{||}\right) - \frac{2}{3}p\nabla\cdot\left(\mathbf{b}v_{||}\right)\\ \frac{\partial}{\partial t}\left(mnv_{||}\right) &= -\nabla\cdot\left(nv_{||}\mathbf{b}v_{||}\right) - \partial_{||}p\end{aligned}$$



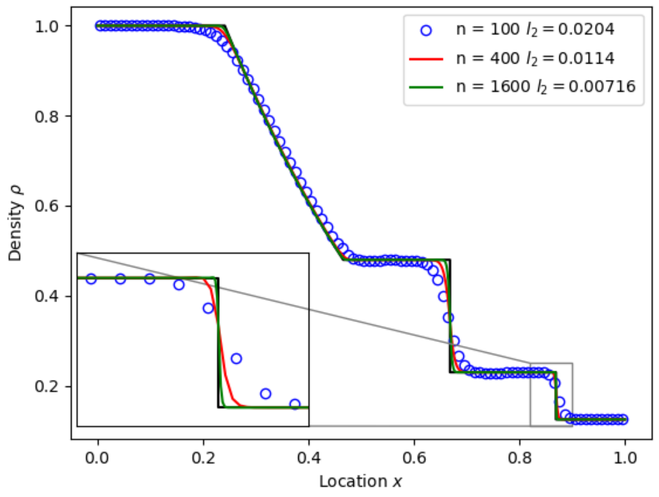
4.2 Sod shock

`tests/integrated/sod-shock` and `tests/integrated/sod-shock-energy`

Euler equations in 1D. Starting from a state with a jump at the middle of the domain. Left state density, velocity and pressure are $(\rho_L, u_L, p_L) = (1.0, 0, 1.0)$ Right state $(\rho_R, u_R, p_R) = (0.125, 0, 0.1)$. The result is shown in figure below at time $t = 0.2$ for different resolutions in a domain of length 1. The solid black line is the analytic solution.



When evolving pressure the position of the shock front lags the analytic solution, with the pressure behind the front slightly too high. This is a known consequence of solving the Euler equations in non-conservative form. If instead we evolve energy (internal + kinetic) then the result is much closer to the analytic solution.



4.3 Toro test 1

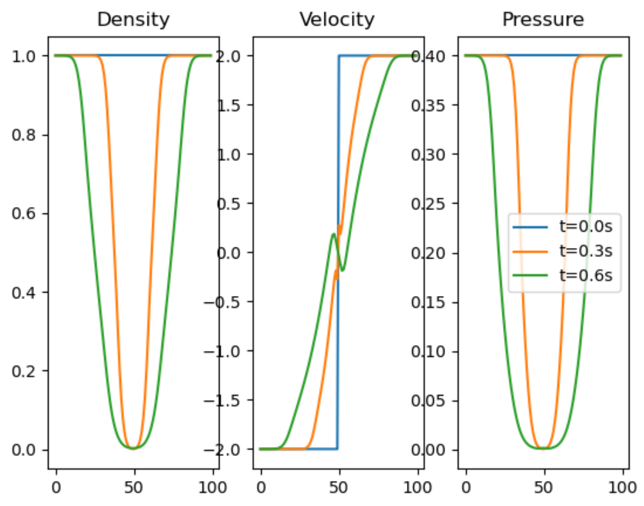
`tests/integrated/toro-1`

Toro's test problem #1, from Riemann Solvers and Numerical Methods for Fluid Dynamics is a variation of Sod's shock tube problem. The left state is moving into the right, increasing the speed of the resulting shock. Left state $(\rho_L, u_L, p_L) = (1.0, 0.75, 1.0)$ Right state $(\rho_R, u_R, p_R) = (0.125, 0, 0.1)$. The size of the domain is 5, and the reference result is given at time $t = 0.8$.

4.4 Toro test 2

`tests/integrated/toro-2` and `tests/integrated/toro-2-energy`

Toro's test problem #2 tests robustness to diverging flows and near-zero densities. The initial state has constant density and temperature, but a jump in velocity. Left state $(\rho_L, u_L, p_L) = (1.0, -2.0, 0.4)$ Right state $(\rho_R, u_R, p_R) = (1.0, 2.0, 0.4)$. The result in a domain of length 5 at time $t = 0.6$ is shown below.



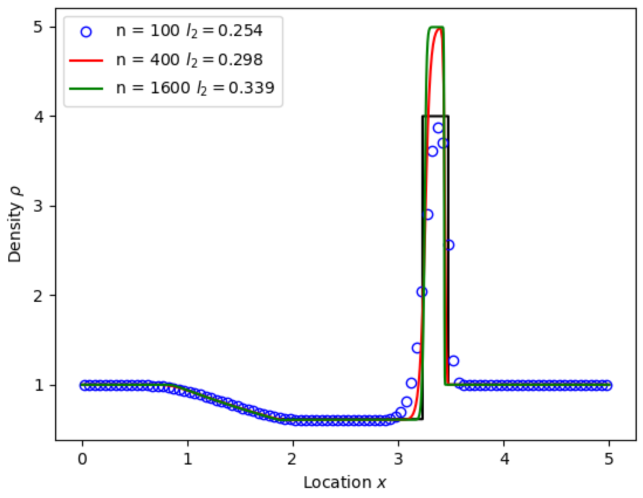
4.5 Toro test 3

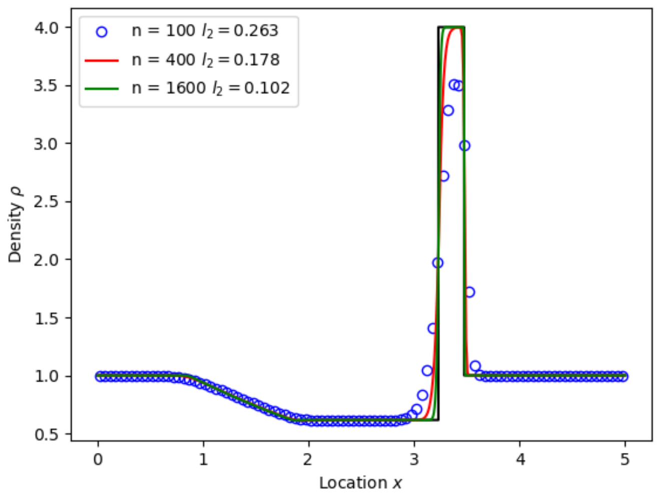
`tests/integrated/toro-3` and `tests/integrated/toro-3-energy`

Toro's test problem #3 contains a strong shock close to a contact discontinuity. Left initial state $(\rho_L, u_L, p_L) = (1.0, 0, 1000.0)$ Right state $(\rho_R, u_R, p_R) = (1.0, 0, 0.01)$. Time $t = 0.04$.

When evolving pressure, the simulation is robust but the density peak does not converge to the analytic solution (solid black line):

However by evolving energy the result converges towards the analytic solution:


n = 100 $l_2 = 0.254$
n = 400 $l_2 = 0.298$
n = 1600 $l_2 = 0.339$
Density ρ
Location x


n = 100 $l_2 = 0.263$
n = 400 $l_2 = 0.178$
n = 1600 $l_2 = 0.102$
Density ρ
Location x

4.6 Toro test 4

`tests/integrated/toro-4` and `tests/integrated/toro-4-energy`

Toro's test problem #4 produces two right-going shocks with a contact between them. Left state $(\rho_L, u_L, p_L) = (5.99924, 19.5975, 460.894)$ Right state $(\rho_R, u_R, p_R) = (5.99242, -6.19633, 46.0950)$. Result at time $t = 0.15$.

4.7 Toro test 5

`tests/integrated/toro-5` and `tests/integrated/toro-5-energy`

The initial conditions for Toro's test problem #5 are the same as test #3, but the whole system is moving to the left at a uniform speed. The velocity is chosen so that the contact discontinuity remains almost stationary at the initial jump location. Left state $(\rho_L, u_L, p_L) = (1, -19.59745, 1000.0)$ Right state $(\rho_R, u_R, p_R) = (1, -19.59745, 0.01)$. Result at time $t = 0.03$.

CHAPTER

FIVE

TOKAMAK AXISYMMETRIC TRANSPORT

Simulations of transport in axisymmetric tokamak geometries, with cross-field diffusion and interaction of plasma with neutral gas.

5.1 Finding steady state solutions

These models can be run as a time-dependent problem, for example to study power transients, but the primary application is to finding steady-state solutions.

5.1.1 Backward Euler solver

This solver uses PETSc to solve the nonlinear system of equations, with a Backward Euler timestep to improve the condition number. There are many choices of algorithm and settings, so the following are guidelines and may not be optimal for all cases.

```
[solver]
type = beuler            # Backward Euler steady-state solver
snes_type = newtonls     # Nonlinear solver
ksp_type = gmres         # Linear solver
max_nonlinear_iterations = 10
pc_type = hypre          # Preconditioner type
pc_hypre_type = euclid   # Hypre preconditioner type
lag_jacobian = 500       # Iterations between jacobian recalculations
atol = 1e-7              # Absolute tolerance
rtol = 1e-5              # Relative tolerance
```

PETSc can print quite extensive performance diagnostics. These can be enabled by putting in the BOUT.inp options file:

```
[petsc]
log_view = true
```

This section can also be used to set other PETSc flags, just omitting the leading - from the PETSc option.

5.1.2 cvode solver

CVODE is primarily intended for high-accuracy time integration, rather than finding steady-state solutions, but can be effective and quite robust. It tends to struggle at high order, so here we limit it to a maximum of 3rd order:

```
[solver]
type = cvode
use_precon = true   # Use the user-provided preconditioner
mxstep = 1e5
mxorder = 3         # Limit to 3rd order
atol = 1e-12
rtol = 1e-5
```

Here `use_precon = true` tells the solver to use the Hermes-3 preconditioners, which are implemented in some components. This includes preconditioning of parallel heat conduction, and of cross-field diffusion of neutrals.

5.1.3 Mesh interpolation

A useful strategy is to start with a low resolution grid, run until close to steady-state, then interpolate the solution onto a finer mesh and restart. This process can be repeated as a kind of simplified multigrid method.

5.2 Post-processing

CHAPTER

SIX

CODE STRUCTURE

A hermes-3 model, like all BOUT++ models, is an implementation of a set of Ordinary Differential Equations (ODEs). The time integration solver drives the simulation, calling the `Hermes::rhs` function to calculate the time-derivatives of all the evolving variables.

The calculation of the time derivatives is coordinated by passing a state object between components. The state is a nested tree, and can have values inserted and retrieved by the components. The components are created and then run by a scheduler, based on settings in the input (BOUT.inp) file.

In terms of design patterns, the method used here is essentially a combination of the Encapsulate Context and Command patterns.

6.1 Simulation state

The simulation state is passed between components, and is a tree of objects (Options objects). At the start of each iteration (rhs call) a new state is created and contains:

- `time` BoutReal, the current simulation time
- `units`
  - `seconds` Multiply by this to get units of seconds
  - `eV` Temperature normalisation
  - `Tesla` Magnetic field normalisation
  - `meters` Length normalisation
  - `inv_meters_cubed` Density normalisation

so the temperature normalisation can be extracted using:

```
BoutReal Tnorm = state["units"]["eV"];
```

As the components of a model are run, they set, modify and use values stored in this state. To ensure that components use consistent names for their input and output variables, a set of conventions are used for new variables which are added to the state:

- `species` Plasma species
  - `e` Electron species
  - `species1` Example "h", "he+2"
    - `AA` Atomic mass, proton = 1
    - `charge` Charge, in units of proton charge (i.e. electron=-1)

        * `density`
        * `momentum`
        * `pressure`
        * `velocity` Parallel velocity
        * `temperature`
        * `collision_frequency` Normalised collision frequency
        * `density_source` Normalised particle source
        * `momentum_source` Normalised momentum source
        * `energy_source` Normalised energy source

- `fields`
    - `vorticity`
    - `phi` Electrostatic potential
    - `DivJdia` Divergence of diamagnetic current
    - `DivJcol` Divergence of collisional current
    - `DivJextra` Divergence of current, including 2D parallel current closures. Not including diamagnetic, parallel current due to flows, or polarisation currents

For example to get the electron density:

```
Field3D ne = state["species"]["e"]["density"];
```

This way of extracting values from the state will print the value to the log file, and is intended mainly for initialisation. In `Component::transform()` and `Component::finally()` functions which run frequently, faster access methods are used which don't print to the log. To get a value:

```
Field3D ne = get<Field3D>(state["species"]["e"]["density"]);
```

If the value isn't set, or can't be converted to the given type, then a `BoutException` will be thrown.

To set a value in the state, there is the `set()` function:

```
set(state["species"]["h"]["density"], ne);
```

A common need is to add or subtract values from fields, such as density sources:

```
add(state["species"]["h"]["density_source"], recombination_rate);
subtract(state["species"]["h+"]["density_source"], recombination_rate);
```

Notes:

- When checking if a subsection exists, use `option.isSection`, since `option.isSet` is false if it is a section and not a value.
- The species name convention is that the charge state is last, after the + or - sign: `n2+` is a singly charged nitrogen molecule, while `n+2` is a +2 charged nitrogen atom.

6.2 Components

The basic building block of all Hermes-3 models is the `Component`. This defines an interface to a class which takes a state (a tree of dictionaries/maps), and transforms (modifies) it. After all components have modified the state in turn, all components may then implement a `finally` method to take the final state but not modify it. This allows two components to depend on each other, but makes debugging and testing easier by limiting the places where the state can be modified.

struct **`Component`**

Interface for a component of a simulation model

The constructor of derived types should have signature (std::string name, Options &options, Solver *solver)

Subclassed by AmjuelReaction, AnomalousDiffusion, *Collisions*, *DiamagneticDrift*, *Electromagnetic*, *ElectronForceBalance*, *ElectronViscosity*, *EvolveDensity*, *EvolveEnergy*, *EvolveMomentum*, *EvolvePressure*, *FixedDensity*, FixedFractionIons, FixedFractionRadiation< CoolingCurve >, *FixedTemperature*, *FixedVelocity*, *HydrogenChargeExchange*, Ionisation, *IonViscosity*, *Isothermal*, *NeutralBoundary*, NeutralFullVelocity, NeutralMixed, *NeutralParallelDiffusion*, *NoFlowBoundary*, OpenADAS, OpenADASChargeExchange, *PolarisationDrift*, Quasineutral, *Recycling*, *RelaxPotential*, ScaleTimeDerivs, SetTemperature, SheathBoundary, SheathBoundaryInsulating, SheathBoundarySimple, SheathClosure, *SimpleConduction*, *SNBConduction*, SOLKITHydrogenChargeExchange, SOLKITNeutralParallelDiffusion, SoundSpeed, *ThermalForce*, Transform, *UpstreamDensityFeedback*, *Vorticity*, ZeroCurrent

Public Functions

virtual void **`transform`**(Options &state) = 0

Modify the given simulation state All components must implement this function

inline virtual void **`finally`**(const Options &state)

Use the final simulation state to update internal state (e.g. time derivatives)

inline virtual void **`outputVars`**(Options &state)

Add extra fields for output, or set attributes e.g docstrings.

inline virtual void **`restartVars`**(Options &state)

Add extra fields to restart files.

inline virtual void **`precon`**(const Options &state, BoutReal gamma)

Preconditioning.

Public Static Functions

static std::unique_ptr<Component> **`create`**(const std::string &type, const std::string &name, Options &options, Solver *solver)

Create a *Component*

Parameters

- **`type`** – The name of the component type to create (e.g. "evolve_density")
- **`name`** – The species/name for this instance.
- **`options`** – *Component* settings: options[name] are specific to this component
- **`solver`** – Time-integration solver

Components are usually defined in separate files; sometimes multiple components in one file if they are small and related to each other (e.g. atomic rates for the same species). To be able to create components, they need to be registered in the factory. This is done in the header file using a code like:

```
#include "component.hxx"

struct MyComponent : public Component {
  MyComponent(const std::string &name, Options &options, Solver *solver);
  ...
};

namespace {
RegisterComponent<MyComponent> registercomponentmine("mycomponent");
}
```

where `MyComponent` is the component class, and "mycomponent" is the name that can be used in the BOUT.inp settings file to create a component of this type. Note that the name can be any string except it can't contain commas or brackets (), and shouldn't start or end with whitespace.

Inputs to the component constructors are:

- `name`
- `alloptions`
- `solver`

The `name` is a string labelling the instance. The `alloptions` tree contains at least:

- `alloptions[name]` options for this instance
- `alloptions['units']`

6.3 Component scheduler

The simulation model is created in `Hermes::init` by a call to the `ComponentScheduler`:

```
scheduler = ComponentScheduler::create(options, Options::root(), solver);
```

and then in `Hermes::rhs` the components are run by a call:

```
scheduler->transform(state);
```

The call to `ComponentScheduler::create()` treats the "components" option as a comma-separated list of names. The order of the components is the order that they are run in. For each name in the list, the scheduler looks up the options under the section of that name.

```
[hermes]
components = component1, component2

[component1]

# options to control component1

[component2]
```

(continues on next page)

(continued from previous page)

```

# options to control component2
```

This would create two `Component` objects, of type `component1` and `component2`. Each time `Hermes::rhs` is run, the `transform` functions of `component1` amd then `component2` will be called, followed by their `finally` functions.

It is often useful to group components together, for example to define the governing equations for different species. A `type` setting in the option section overrides the name of the section, and can be another list of components

```
[hermes]
components = group1, component3

[group1]
type = component1, component2

# options to control component1 and component2

[component3]

# options to control component3
```

This will create three components, which will be run in the order `component1`, `component2`, `component3`: First all the components in `group1`, and then `component3`.

class **`ComponentScheduler`**

Creates and schedules model components

Currently only one implementation, but in future alternative scheduler types could be created. There is therefore a static create function which in future could switch between types.

Public Functions

void **`transform`**(Options &state)

Run the scheduler, modifying the state. This calls all components' *transform()* methods, then all component's finally() methods.

void **`outputVars`**(Options &state)

Add metadata, extra outputs. This would typically be called only for writing to disk, rather than every internal timestep.

void **`restartVars`**(Options &state)

Add variables to restart files.

void **`precon`**(const Options &state, BoutReal gamma)

Preconditioning.

Public Static Functions

static std::unique_ptr<ComponentScheduler> `create`(Options &scheduler_options, Options &component_options, Solver *solver)

Inputs

Parameters

- **`scheduler_options`** – Configuration of the scheduler Should contain "components", a comma-separated list of component names
- **`component_options`** – Configuration of the components.
  - <name>
    - type = *Component* classes, ... If not provided, the type is the same as the name Multiple classes can be given, separated by commas. All classes will use the same Options section.
    - ... Options to control the component(s)
- **`solver`** – Used for time-dependent components to evolve quantities

CHAPTER

SEVEN

COMPONENTS

This section describes the model components currently available.

7.1 Species density

The density of a species can be calculated in several different ways, and are usually needed by other components.

7.1.1 fixed_density

Set the density to a value which does not change in time. For example:

```
[d]
type = fixed_density, ...

AA = 2 # Atomic mass
charge = 0
density = 1e17 # In m^-3
```

Note that the density can be a function of `x`, `y` and `z` coordinates for spatial variation.

The implementation is in the `FixedDensity` class:

struct **`FixedDensity`** : public Component

Set ion density to a fixed value

Public Functions

inline **`FixedDensity`**(std::string name, Options &alloptions, Solver *solver)

Inputs

- <name>
  - AA
  - charge
  - density value (expression) in units of m^-3

inline virtual void **transform**(Options &state) override

Sets in the state the density, mass and charge of the species

- species
  - <name>
    - AA
    - charge
    - density

inline virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.1.2 evolve_density

This component evolves the species density in time, using the BOUT++ time integration solver. The species charge and atomic mass must be set, and the initial density should be specified in its own section:

```
[d]
type = evolve_density, ...

AA = 2 # Atomic mass
charge = 0

[Nd]
function = 1 - 0.5x # Initial condition, normalised to Nnorm
```

The implementation is in the `EvolveDensity` class:

struct **EvolveDensity** : public Component

Evolve species density in time

Mesh inputs

N<name>_src A source of particles, per cubic meter per second. This can be over-ridden by the `source` option setting.

Public Functions

EvolveDensity(std::string name, Options &options, Solver *solver)

Inputs

- <name>
  - charge Particle charge e.g. hydrogen = 1
  - AA Atomic mass number e.g. hydrogen = 1
  - bndry_flux Allow flow through radial boundaries? Default is true.
  - poloidal_flows Include poloidal ExB flows? Default is true.

    - density_floor Minimum density floor. Default is 1e-5 normalised units
    - low_n_diffuse Enhance parallel diffusion at low density? Default false
    - hyper_z Hyper-diffusion in Z. Default off.
    - evolve_log Evolve logarithm of density? Default false.
    - diagnose Output additional diagnostics?

- N<name> e.g. “Ne”, “Nd+”
    - source Source of particles [/m^3/s] NOTE: This overrides mesh input N<name>_src
    - source_only_in_core Zero the source outside the closed field-line region?
    - neumann_boundary_average_z Apply Neumann boundaries with Z average?

virtual void **`transform`**(Options &state) override

This sets in the state

- species
    - <name>
        - AA
        - charge
        - density

virtual void **`finally`**(const Options &state) override

Calculate ddt(N).

Requires state components

- species
    - <name>
        - density

Optional components

- species
    - <name>
        - velocity If included, requires sound_speed or temperature
        - density_source
- fields
    - phi If included, ExB drift is calculated

virtual void **`outputVars`**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.1.3 upstream_density_feedback

This is intended for 1D simulations, where the density at $y = 0$ is set by adjusting an input source. This component uses a PI controller method to scale the density source up and down, to maintain the specified upstream density. The source, e.g. `Sd+_feedback`, is calculated as a product of the control signal `density_source_multiplier`, and the array `density_source_shape` which defines the source region. The signal is non-dimensional and the controller depends on the value of `density_source_shape` to have a good initial guess of the source. It should be set to a reasonable value in the units of `[m-3s-1]`. A good reasonable value is the expected steady state domain particle loss (for example due to unrecycled ions at the target).

For example:

```
[d+]
type = ..., upstream_density_feedback

density_upstream = 1e19      # Density in m^-3
density_controller_p = 1e-2  # Feedback controller proportional (p) parameter
density_controller_i = 1e-3  # Feedback controller integral (i) parameter

[Nd+]
source_shape = h(pi - y) * 1e20  # Source shape
```

There are two additional settings which can make the controller more robust without excessive tuning:

`density_source_positive` ensures the controller never takes particles away, which can prevent oscillatory behaviour. Note that this requires some other domain particle sink to ensure control, or else the particle count can never reduce.

`density_integral_positive` This makes sure the integral component only adds particles. The integral component takes a long time to change value, which can result in large overshoots if the initial guess was too small. This setting mitigates this by disabling the integral term if the density is above the desired value.

Notes:

- The example cases have their PI parameters tuned properly without the need of the above two settings.
- Under certain conditions, the use of the PI controller can make the upstream density enter a very small oscillation (~0.05% of upstream value).
- There is a separate `source` setting that includes a fixed (non varying) density source.

The implementation is in the `UpstreamDensityFeedback` class:

struct **`UpstreamDensityFeedback`** : public Component

Adds a time-varying density source, depending on the difference between the upstream density at y=0 and the specified value

Public Functions

inline **`UpstreamDensityFeedback`**(std::string name, Options &alloptions, Solver*)

Inputs

- <name> (e.g. "d+")
  - density_upstream Upstream density (y=0) in m^-3
  - density_controller_p Feedback proportional to error
  - density_controller_i Feedback proportional to error integral

- density_integral_positive Force integral term to be positive? (default: false)
- density_source_positive Force density source to be positive? (default: true)
- diagnose Output diagnostic information?

- N<name> (e.g. "Nd+")
  - source_shape The initial source that is scaled by a time-varying factor

virtual void **transform**(Options &state) override

Inputs

- <name>
  - density

Outputs

- <name>
  - density_source

inline virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

inline virtual void **restartVars**(Options &state) override

Add extra fields to restart files.

7.1.4 fixed_fraction_ions

This sets the density of a species to a fraction of the electron density.

7.1.5 quasineutral

This component sets the density of one species, so that the overall charge density is zero everywhere. This must therefore be done after all other charged species densities have been calculated. It only makes sense to use this component for species with a non-zero charge.

7.2 Species pressure and temperature

7.2.1 isothermal

Sets the temperature of a species to a fixed value which is constant in space and time. If the species density is set then this component also calculates the pressure.

By default only saves the temperature once as a non-evolving variable. If `diagnose` is set then pressure is also saved as a time-evolving variable.

```
[e]
type = ..., isothermal

temperature = 10   # Constant temperature [eV]
```

struct **`Isothermal`** : public Component

Set temperature to a fixed value

Public Functions

virtual void **`transform`**(Options &state) override

Inputs

- species
  - <name>
    - density (optional)

Sets in the state

- species
  - <name>
    - temperature
    - pressure (if density is set)

virtual void **`outputVars`**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.2.2 fixed_temperature

Sets the temperature of a species to a fixed value which is constant in time but can vary in space. If the species density is set then this component also calculates the pressure.

By default only saves the temperature once as a non-evolving variable. If `diagnose` is set then pressure is also saved as a time-evolving variable.

```
[e]
type = ..., fixed_temperature

temperature = 10 - x   # Spatially dependent temperature [eV]
```

struct **`FixedTemperature`** : public Component

Set species temperature to a fixed value

Public Functions

inline **`FixedTemperature`**(std::string name, Options &alloptions, Solver *solver)

Inputs

- <name>
  - temperature value (expression) in units of eV

inline virtual void **transform**(Options &state) override

Sets in the state the temperature and pressure of the species

Inputs

- species
  - <name>
    - density (optional)

Sets in the state

- species
  - <name>
    - temperature
    - pressure (if density is set)

inline virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.2.3 evolve_pressure

Evolves the pressure in time. This pressure is named `P` where `<species>` is the short name of the evolving species e.g. `Pe`.

By default parallel thermal conduction is included, which requires a collision time. If collisions are not calculated, then thermal conduction should be turned off by setting `thermal_conduction = false` in the input options.

If the component option `diagnose = true` then additional fields will be saved to the dump files: The species temperature `T + name` (e.g. `Td+` or `Te`), the time derivative `ddt(P + name)` (e.g. `ddt(Pd+)` or `ddt(Pe)`), and the source of pressure from other components is saved as `SP + name` (e.g. `SPd+` or `SPe`). The pressure source is the energy density source multiplied by `2/3` (i.e. assumes a monatomic species).

$$\frac{\partial P}{\partial t} = -\nabla \cdot (P\mathbf{v}) - \frac{2}{3}P\nabla \cdot \mathbf{b}v_{||} + \frac{2}{3}\nabla \cdot \left(\kappa_{||}\mathbf{b}\mathbf{b} \cdot \nabla T\right) + \frac{2}{3}S_E + S_N\frac{1}{2}mNV^2$$

where S_E is the `energy_source` (thermal energy source), and S_N is the density source.

Notes:

- Heat conduction through the boundary is turned off currently. This is because heat losses are usually calculated at the sheath, so any additional heat conduction would be in addition to the sheath heat transmission already included.

The implementation is in `EvolvePressure`:

struct **EvolvePressure** : public Component

Evolves species pressure in time

Mesh inputs

P<name>_src A source of pressure, in Pascals per second This can be over-ridden by the `source` option setting.

Public Functions

EvolvePressure(std::string name, Options &options, Solver *solver)

Inputs

- <name>
  - bndry_flux Allow flows through radial boundaries? Default is true
  - density_floor Minimum density floor. Default 1e-5 normalised units.
  - diagnose Output additional diagnostic fields?
  - evolve_log Evolve logarithm of pressure? Default is false
  - hyper_z Hyper-diffusion in Z
  - kappa_coefficient Heat conduction constant. Default is 3.16 for electrons, 3.9 otherwise
  - kappa_limit_alpha Flux limiter, off by default.
  - poloidal_flows Include poloidal ExB flows? Default is true
  - precon Enable preconditioner? Note: solver may not use it even if enabled.
  - p_div_v Use p * Div(v) form? Default is v * Grad(p) form
  - thermal_conduction Include parallel heat conduction? Default is true
- P<name> e.g. "Pe", "Pd+"
  - source Source of pressure [Pa / s]. NOTE: This overrides mesh input P<name>_src
  - source_only_in_core Zero the source outside the closed field-line region?
  - neumann_boundary_average_z Apply Neumann boundaries with Z average?

virtual void **transform**(Options &state) override

Inputs

- species
  - <name>
    - density

Sets

- species
  - <name>
    - pressure
    - temperature Requires density

virtual void **finally**(const Options &state) override

Optional inputs

- species
  - <name>
    - velocity. Must have sound_speed or temperature

      * energy_source
      * collision_rate (needed if thermal_conduction on)

  - fields
    - phi Electrostatic potential -> ExB drift

virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

virtual void **precon**(const Options &state, BoutReal gamma) override

Preconditioner

7.2.4 evolve_energy

Note This is currently under development and has some unresolved issues with boundary conditions. Only for testing purposes.

This evolves the sum of species internal energy and parallel kinetic energy, $\mathcal{E}$:

$$\mathcal{E} = \frac{1}{\gamma - 1}P + \frac{1}{2}mnv_{||}^2$$

Note that this component requires the parallel velocity $v_{||}$ to calculate the pressure. It must therefore be listed after a component that sets the velocity, such as `evolve_momentum`:

```
[d]
type = ..., evolve_momentum, evolve_energy
```

The energy density will be saved as `E` (e.g `Ed`) and the pressure as `P` (e.g. `Pd`). Additional diagnostics, such as the temperature, can be saved by setting the option `diagnose = true`.

struct **EvolveEnergy** : public Component

Evolves species internal energy in time

Mesh inputs

P<name>_src A source of pressure, in Pascals per second This can be over-ridden by the `source` option setting.

Public Functions

EvolveEnergy(std::string name, Options &options, Solver *solver)

Inputs

  - <name>
    - bndry_flux Allow flows through radial boundaries? Default is true
    - density_floor Minimum density floor. Default 1e-5 normalised units.
    - diagnose Output additional diagnostic fields?
    - evolve_log Evolve logarithm of pressure? Default is false
    - hyper_z Hyper-diffusion in Z
    - kappa_coefficient Heat conduction constant. Default is 3.16 for electrons, 3.9 otherwise

  - kappa_limit_alpha Flux limiter, off by default.
  - poloidal_flows Include poloidal ExB flows? Default is true
  - precon Enable preconditioner? Note: solver may not use it even if enabled.
  - thermal_conduction Include parallel heat conduction? Default is true
- E<name> e.g. “Ee”, “Ed+”
  - source Source of energy [W / s]. NOTE: This overrides mesh input P<name>_src
  - source_only_in_core Zero the source outside the closed field-line region?
  - neumann_boundary_average_z Apply Neumann boundaries with Z average?

virtual void **transform**(Options &state) override

Inputs

- species
  - <name>
    - density
    - velocity

Sets

- species
  - <name>
    - pressure
    - temperature

virtual void **finally**(const Options &state) override

Optional inputs

- species
  - <name>
    - velocity. Must have sound_speed or temperature
    - energy_source
    - collision_rate (needed if thermal_conduction on)
- fields
  - phi Electrostatic potential -> ExB drift

virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

virtual void **precon**(const Options &state, BoutReal gamma) override

Preconditioner

7.2.5 SNB nonlocal heat flux

Calculates the divergence of the electron heat flux using the Shurtz-Nicolai-Busquet (SNB) model. Uses the BOUT++ implementation which is documented here.

struct **SNBConduction** : public Component

Calculate electron heat flux using the Shurtz-Nicolai-Busquet (SNB) model

This component will only calculate divergence of heat flux for the electron (`e`) species.

Usage

Add as a top-level component after both electron temperature and collision times have been calculated.

Important: If evolving electron pressure, disable thermal conduction or that will continue to add Spitzer heat conduction.

```
[hermes]
components = e, ..., collisions, snb_conduction

[e]
type = evolve_pressure, ...
thermal_conduction = false # For evolve_pressure

[snb_conduction]
diagnose = true # Saves heat flux diagnostics
```

Useful references:

- Braginskii equations by R.Fitzpatrick: http://farside.ph.utexas.edu/teaching/plasma/Plasmahtml/node35.html
- J.P.Brodrick et al 2017: https://doi.org/10.1063/1.5001079 and https://arxiv.org/abs/1704.08963
- Shurtz, Nicolai and Busquet 2000: https://doi.org/10.1063/1.1289512

Public Functions

inline **SNBConduction**(std::string name, Options &alloptions, Solver*)

Inputs

- <name>
  - diagnose Saves Div_Q_SH and Div_Q_SNB

virtual void **transform**(Options &state) override

Inputs

- species
  - e
    - density
    - collision_frequency

Sets

- species

- e
  - energy_source

virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.3 Species parallel dynamics

7.3.1 fixed_velocity

Sets the velocity of a species to a fixed value which is constant in time but can vary in space. If the species density is set then this component also calculates the momentum.

Saves the temperature once as a non-evolving variable.

```
[e]
type = ..., fixed_velocity

velocity = 10 + sin(z)    # Spatially dependent velocity [m/s]
```

struct **FixedVelocity** : public Component

Set parallel velocity to a fixed value

Public Functions

inline virtual void **transform**(Options &state) override

This sets in the state

- species
  - <name>
    - velocity
    - momentum

inline virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.3.2 evolve_momentum

Evolves the momentum `NV` in time. The evolving quantity includes the atomic mass number, so should be divided by `AA` to obtain the particle flux.

If the component option `diagnose = true` then additional fields will be saved to the dump files: The velocity `V + name` (e.g. `Vd+` or `Ve`), the time derivative `ddt(NV + name)` (e.g. `ddt(NVd+)` or `ddt(NVe)`), and the source of momentum density (i.e force density) from other components is saved as `SNV + name` (e.g. `SNVd+` or `SNVe`).

The implementation is in `EvolveMomentum`:

struct **EvolveMomentum** : public Component

Evolve parallel momentum.

Public Functions

virtual void **transform**(Options &state) override

This sets in the state

- species
  - <name>
    - momentum
    - velocity if density is defined

virtual void **finally**(const Options &state) override

Calculate ddt(NV).

Inputs

- species
  - <name>
    - density
    - velocity
    - pressure (optional)
    - momentum_source (optional)
    - sound_speed (optional, used for numerical dissipation)
    - temperature (only needed if sound_speed not provided)
- fields
  - phi (optional)

virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.3.3 zero_current

This calculates the parallel flow of one charged species so that there is no net current, using flows already calculated for other species. It is used like `quasineutral`:

```
[hermes]
components = h+, ..., e, ...   # Note: e after all other species

[e]
type = ..., zero_current,... # Set e:velocity

charge = -1 # Species must have a charge
```

7.3.4 electron_force_balance

This calculates a parallel electric field which balances the electron pressure gradient and other forces on the electrons (including collisional friction, thermal forces):

$$E_{||} = \left(-\nabla p_e + F\right)/n_e$$

where F is the `momentum_source` for the electrons. This electric field is then used to calculate a force on the other species:

$$F_z = Zn_zE_{||}$$

which is added to the ion's `momentum_source`.

The implementation is in `ElectronForceBalance`:

struct **`ElectronForceBalance`** : public Component

Balance the parallel electron pressure gradient against the electric field. Use this electric field to calculate a force on the other species

E = (-?p_e + F) / n_e

where F is the momentum source for the electrons.

Then uses this electric field to calculate a force on all ion species.

Note: This needs to be put after collisions and other components which impose forces on electrons

Public Functions

virtual void **`transform`**(Options &state) override

Required inputs

- species
  - e
    - pressure
    - density
    - momentum_source [optional] Asserts that charge = -1

Sets in the input

- species
  - <all except=”” e>=””> if both density and charge are set
    - momentum_source

7.4 electron_viscosity

Calculates the Braginskii electron parallel viscosity, adding a force (momentum source) to the electron momentum equation:

$$F = \sqrt{B}\nabla\cdot\left[\frac{\eta_e}{B}\mathbf{b}\mathbf{b}\cdot\nabla\left(\sqrt{B}V_{||e}\right)\right]$$

The electron parallel viscosity is

$$\eta_e = \frac{4}{3}0.73p_e\tau_e$$

where τ_e is the electron collision time. The collisions between electrons and all other species therefore need to be calculated before this component is run:

```
[hermes]
components = ..., e, ..., collisions, electron_viscosity
```

struct **ElectronViscosity** : public Component

Electron viscosity

Adds Braginskii parallel electron viscosity, with SOLPS-style viscosity flux limiter

Needs to be calculated after collisions, because collision frequency is used to calculate parallel viscosity

References

- https://farside.ph.utexas.edu/teaching/plasma/lectures1/node35.html

Public Functions

ElectronViscosity(std::string name, Options &alloptions, Solver*)

Braginskii electron viscosity.

Inputs

- <name>
  - diagnose: bool, default false Output diagnostic SNVe_viscosity?
  - eta_limit_alpha: float, default -1.0 Flux limiter coefficient. < 0 means no limiter

virtual void **transform**(Options &state) override

Inputs

- species
  - e
    - pressure (skips if not present)
    - velocity (skips if not present)
    - collision_frequency

Sets in the state

- species
  - e
    - momentum_source

virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.5 ion_viscosity

Adds ion viscosity terms to all charged species that are not electrons. The collision frequency is required so this is a top-level component that must be calculated after collisions:

```
[hermes]
components =  ..., collisions, ion_viscosity
```

By default only the parallel diffusion of momentum is included, adding a force to each ion's momentum equation:

$$F = \sqrt{B}\nabla \cdot \left[\frac{\eta_i}{B}\mathbf{b}\mathbf{b}\cdot\nabla\left(\sqrt{B}V_{||i}\right)\right]$$

The ion parallel viscosity is

$$\eta_i = \frac{4}{3}0.96 p_i\tau_i$$

If the `perpendicular` option is set:

```
[ion_viscosity]
perpendicular = true # Include perpendicular flows
```

Then the ion scalar viscous pressure is calculated as:

$$\Pi_{ci} = \Pi_{ci||} + \Pi_{ci\perp}$$

where $\Pi_{ci||}$ corresponds to the parallel diffusion of momentum above.

$$\Pi_{ci||} = -0.96\frac{2p_i\tau_i}{\sqrt{B}}\partial_{||}\left(\sqrt{B}V_{||i}\right)$$

The perpendicular part is calculated from:

$$\begin{aligned}\Pi_{ci\perp} =& 0.96 p_i\tau_i\kappa\cdot\left[\mathbf{V}_E + \mathbf{V}_{di} + 1.16\frac{\mathbf{b}\times\nabla T_i}{B}\right] \\ =& -0.96 p_i\tau_i\frac{1}{B}\left(\mathbf{b}\times\kappa\right)\cdot\left[\nabla\phi + \frac{\nabla p_i}{en_i} + 1.61\nabla T_i\right]\end{aligned}$$

A parallel force term is added, in addition to the parallel viscosity above:

$$F = -\frac{2}{3}B^{3/2}\partial_{||}\left(\frac{\Pi_{ci\perp}}{B^{3/2}}\right)$$

In the vorticity equation the viscosity appears as a divergence of a current:

$$\mathbf{J}_{ci} = \frac{\Pi_{ci}}{2}\nabla\times\frac{\mathbf{b}}{B} - \frac{1}{3}\frac{\mathbf{b}\times\nabla\Pi_{ci}}{B}$$

that transfers energy between ion internal energy and $E\times B$ energy:

$$\begin{aligned}\frac{\partial\omega}{\partial t} =& \ldots + \nabla\cdot\mathbf{J}_{ci} \\ \frac{\partial p_i}{\partial t} =& \ldots - \mathbf{J}_{ci}\cdot\nabla\left(\phi + \frac{p_i}{n_0}\right)\end{aligned}$$

Note that the sum of the perpendicular and parallel contributions to the ion viscosity act to damp the net poloidal flow. This can be seen by assuming that ϕ, p_i and T_i are flux functions. We can then write:

$$\Pi_{ci\perp} = -0.96 p_i \tau_i \frac{1}{B} \left(\mathbf{b}\times\kappa\right)\cdot\nabla\psi F\left(\psi\right)$$

where

$$F\left(\psi\right) = \frac{\partial\phi}{\partial\psi} + \frac{1}{en}\frac{\partial p_i}{\partial\psi} + 1.61\frac{\partial T_i}{\partial\psi}$$

Using the approximation

$$\left(\mathbf{b}\times\kappa\right)\cdot\nabla\psi \simeq -RB_\zeta\partial_{||}\ln B$$

expanding:

$$\frac{2}{\sqrt{B}}\partial_{||}\left(\sqrt{B}V_{||i}\right) = 2\partial_{||}V_{||i} + V_{||i}\partial_{||}\ln B$$

and neglecting parallel gradients of velocity gives:

$$\Pi_{ci} \simeq 0.96 p_i\tau_i \left[\frac{RB_\zeta}{B}F\left(\psi\right) - V_{||i}\right]\partial_{||}\ln B$$

Notes and implementation details: - The magnitude of $\Pi_{ci\perp}$ and $\Pi_{ci||}$ are individually

limited to be less than or equal to the scalar pressure Pi (though can have opposite sign). The reasoning is that if these off-diagonal terms become large then the model is likely breaking down. Occasionally happens in low-density regions.

struct **IonViscosity** : public Component

Ion viscosity terms

Adds a viscosity to all species which are not electrons

Uses Braginskii collisional form, combined with a SOLPS-like flux limiter.

Needs to be calculated after collisions, because collision frequency is used to calculate parallel viscosity

The ion stress tensor Pi_ci is split into perpendicular and parallel pieces:

Pi_ci = Pi_ciperp + Pi_cipar

In the parallel ion momentum equation the Pi_cipar term is solved as a parallel diffusion, so is treated separately All other terms are added to Pi_ciperp, even if they are not really parallel parts

Public Functions

IonViscosity(std::string name, Options &alloptions, Solver*)

Inputs

- <name>
  - eta_limit_alpha: float, default -1 Flux limiter coefficient. < 0 means off.
  - perpendicular: bool, default false Include perpendicular flows? Requires curvature vector and phi potential

virtual void **transform**(Options &state) override

Inputs

- species
  - <name> (skips "e")
    - pressure (skips if not present)
    - velocity (skips if not present)
    - collision_frequency

Sets in the state

- species
  - <name>
    - momentum_source

virtual void **outputVars**(Options &state) override

Save variables to the output.

7.6 simple_conduction

This is a simplified parallel heat conduction model that can be used when a linearised model is needed. If used, the thermal conduction term in `evolve_pressure` component should be disabled.

```
[hermes]
components = e, ...

[e]
type = evolve_pressure, simple_conduction

thermal_conduction = false  # Disable term in evolve_pressure
```

To linearise the heat conduction the temperature and density used in calculating the Coulomb logarithm and heat conduction coefficient can be fixed by specifying `conduction_temperature` and `conduction_density`.

Note: For hydrogenic plasmas this produces very similar parallel electron heat conduction as the `evolve_pressure` term with electron-electron collisions disabled.

struct **SimpleConduction** : public Component

Simplified models of parallel heat conduction

Intended mainly for testing.

Expressions taken from: https://farside.ph.utexas.edu/teaching/plasma/lectures1/node35.html

Public Functions

inline virtual void **transform**(Options &state) override

Modify the given simulation state All components must implement this function

7.7 Drifts

The ExB drift is included in the density, momentum and pressure evolution equations if potential is calculated. Other drifts can be added with the following components.

7.7.1 diamagnetic_drift

Adds diamagnetic drift terms to all species' density, pressure and parallel momentum equations. Calculates the diamagnetic drift velocity as

$$\mathbf{v}_{dia} = \frac{T}{q}\nabla \times \left(\frac{\mathbf{b}}{B}\right)$$

where the curvature vector $\nabla \times \left(\frac{\mathbf{b}}{B}\right)$ is read from the `bxcv` mesh input variable.

struct **DiamagneticDrift** : public Component

Calculate diamagnetic flows.

Public Functions

virtual void **transform**(Options &state) override

For every species, if it has:

- temperature
- charge

Modifies:

- density_source
- energy_source
- momentum_source

7.7.2 polarisation_drift

This calculates the polarisation drift of all charged species, including ions and electrons. It works by approximating the drift as a potential flow:

$$\mathbf{v}_{pol} = -\frac{m}{qB^2}\nabla_\perp\phi_{pol}$$

where ϕ_{pol} is approximately the time derivative of the electrostatic potential ϕ in the frame of the fluid, with an ion diamagnetic contribution. This is calculated by inverting a Laplacian equation similar to that solved in the vorticity equation.

This component needs to be run after all other currents have been calculated. It marks currents as used, so out-of-order modifications should raise errors.

See the `examples/blob2d-vpol` example, which contains:

```
[hermes]
components = e, vorticity, sheath_closure, polarisation_drift

[polarisation_drift]
diagnose = true
```

Setting `diagnose = true` saves `DivJ` to the dump files with the divergence of all currents except polarisation, and `phi_pol` which is the polarisation flow potential.

struct **`PolarisationDrift`** : public Component

Calculates polarisation drift terms for all charged species, both ions and electrons.

Approximates the polarisation drift by a generalised flow potential `phi_pol`

v_pol = - (A / (Z * B^2)) * Grad_perp(phi_pol)

phi_pol is approximately the time derivative of the electric potential in the frame of the flow, plus an ion diamagnetic contribution

phi_pol is calculated using:

Div(mass_density / B^2 * Grad_perp(phi_pol)) = Div(Jpar) + Div(Jdia) + …

Where the divergence of currents on the right is calculated from:

- species[…][“momentum”] The parallel momentum of charged species
- DivJdia, diamagnetic current, calculated in vorticity component
- DivJcol collisional current, calculated in vorticity component
- DivJextra Other currents, eg. 2D parallel closures

The mass_density quantity is the sum of density * atomic mass for all charged species (ions and electrons)

Public Functions

virtual void **`transform`**(Options &state) override

Inputs

- species
  - … All species with both charge and mass
    - AA
    - charge
    - density
    - momentum (optional)
- fields
  - DivJextra (optional)
  - DivJdia (optional)
  - DivJcol (optional)

Sets

- species
  - ... All species with both charge and mass
    - density_source
    - energy_source (if pressure set)
    - momentum_source (if momentum set)

virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.8 Neutral gas models

The `neutral_mixed` component solves fluid equations along y (parallel to the magnetic field), and uses diffusive transport in x and z. It was adopted from the approach used in UEDGE and this paper [Journal of Nuclear Materials, vol. 313-316, pp. 559-563 (2003)].

$$\begin{aligned}\frac{\partial n_n}{\partial t} =& -\nabla\cdot\left(n_n\mathbf{b}v_{||n} + n_n\mathbf{v}_{\perp n}\right) + S\\ \frac{\partial}{\partial t}\left(n_nv_{||n}\right) =& -\nabla\cdot\left(n_nv_{||n}\mathbf{b}v_{||n} + n_nv_{||n}\mathbf{v}_{\perp n}\right) - \partial_{||}p_n + \nabla_{||}\left(D_{nn}n_n\partial_{||}v_{||n}\right) + F\\ \frac{\partial p_n}{\partial t} =& -\nabla\cdot\left(p_n\mathbf{b}v_{||n} + p_n\mathbf{v}_{\perp n}\right) - \frac{2}{3}p_n\nabla\cdot\left(\mathbf{b}v_{||n}\right) + \nabla\cdot\left(D_{nn}n_n\nabla_\perp T_n\right) + \frac{2}{3}Q\end{aligned}$$

The parallel momentum is evolved, so that it can be exchanged with the plasma parallel momentum, but the mass is neglected for perpendicular motion. In the perpendicular direction, therefore, the motion is a balance between the friction (primarily with the plasma through charge exchange) and the pressure gradient:

$$\mathbf{v}_{\perp n} = -D_{nn}\frac{1}{p_n}\nabla_\perp p_n$$

At the moment there is no attempt to limit these velocities, which has been found necessary in UEDGE to get physical results in better agreement with kinetic neutral models [Discussion, T.Rognlien].

7.9 Boundary conditions

7.9.1 noflow_boundary

This is a species component which imposes a no-flow boundary condition on y (parallel) boundaries.

- Zero-gradient boundary conditions are applied to `density`, `temperature` and `pressure` fields, if they are set.
- Zero-value boundary conditions are applied to `velocity` and `momentum` if they are set.

By default both yup and ydown boundaries are set, but can be turned off by setting `noflow_lower_y` or `noflow_upper_y` to `false`.

Example: To set no-flow boundary condition on an ion `d+` at the lower y boundary, with a sheath boundary at the upper y boundary:

```
[hermes]
components = d+, sheath_boundary

[d+]
type = noflow_boundary

noflow_lower_y = true   # This is the default
noflow_upper_y = false  # Turn off no-flow at upper y for d+ species

[sheath_boundary]
lower_y = false         # Turn off sheath lower boundary for all species
upper_y = true
```

Note that currently `noflow_boundary` is set per-species, whereas `sheath_boundary` is applied to all species. This is because sheath boundary conditions couple all charged species together, and doesn't affect neutral species.

The implementation is in `NoFlowBoundary`:

struct **`NoFlowBoundary`** : public Component

Public Functions

virtual void **`transform`**(Options &state) override

Inputs

- species
  - <name>
    * density [Optional]
    * temperature [Optional]
    * pressure [Optional]
    * velocity [Optional]
    * momentum [Optional]

7.9.2 neutral_boundary

Sets Y (sheath/target) boundary conditions on neutral particle density, temperature and pressure. A no-flow boundary condition is set on parallel velocity and momentum. It is a species-specific component and so goes in the list of components for the species that the boundary condition should be applied to.

An energy sink is added to the flux of heat to the wall, with heat flux `q`:

$$q = \gamma_{heat} n T v_{th}$$
$$v_{th} = \sqrt{eT/m}$$

The factor `gamma_heat`

```
[hermes]
components = d

```

(continues on next page)

(continued from previous page)

```
[d]
type = ... , neutral_boundary

gamma_heat = 3  # Neutral boundary heat transmission coefficient
neutral_lower_y = true  # Boundary on lower y?
neutral_upper_y = true  # Boundary on upper y?
```

struct **NeutralBoundary** : public Component

Per-species boundary condition for neutral particles at sheath (Y) boundaries.

Sets boundary conditions:

- Free boundary conditions on logarithm of density, temperature and pressure
- No-flow boundary conditions on velocity and momentum.

Adds an energy sink corresponding to a flux of heat to the walls.

Heat flux into the wall is q = gamma_heat * n * T * v_th

where v_th = sqrt(eT/m) is the thermal speed

Public Functions

virtual void **transform**(Options &state) override

state

- species
  - <name>
    - density Free boundary
    - temperature Free boundary
    - pressure Free boundary
    - velocity [if set] Zero boundary
    - momentum [if set] Zero boundary
    - energy_source Adds wall losses

7.10 Collective quantities

These components combine multiple species together. They are typically listed after all the species groups in the component list, so that all the species are present in the state.

One of the most important is the *collisions* component. This sets collision times for all species, which are then used

7.10.1 sound_speed

Calculates the collective sound speed, by summing the pressure of all species, and dividing by the sum of the mass density of all species:

$$c_s = \sqrt{\sum_i P_i / \sum_i m_i n_i}$$

This is set in the state as `sound_speed`, and is used for the numerical diffusion terms in the parallel advection.

7.10.2 neutral_parallel_diffusion

This adds diffusion to **all** neutral species (those with no or zero charge), because it needs to be calculated after the collision frequencies are known.

```
[hermes]
components = ... , collisions, neutral_parallel_diffusion

[neutral_parallel_diffusion]
dneut = 1          # Diffusion multiplication factor
diagnose = true    # This enables diagnostic output for each species
```

It is intended mainly for 1D simulations, to provide effective parallel diffusion of particles, momentum and energy due to the projection of cross-field diffusion:

$$\begin{aligned}\frac{\partial n_n}{\partial t} &= \ldots + \nabla\cdot\left(\mathbf{b}D_n n_n \partial_{||}p_n\right)\\ \frac{\partial p_n}{\partial t} &= \ldots + \nabla\cdot\left(\mathbf{b}D_n p_n \partial_{||}p_n\right) + \frac{2}{3}\nabla\cdot\left(\mathbf{b}\kappa_n \partial_{||}T_n\right)\\ \frac{\partial}{\partial t}\left(n_n v_{||n}\right) &= \ldots + \nabla\cdot\left(\mathbf{b}D_n n_n v_{||n}\partial_{||}p_n\right) + \nabla\cdot\left(\mathbf{b}\eta_n \partial_{||}T_n\right)\end{aligned}$$

The diffusion coefficient is calculated as

$$D_n = \left(\frac{B}{B_{pol}}\right)^2 \frac{T_n}{A\nu}$$

where `A` is the atomic mass number; ν is the collision frequency. The factor B/B_{pol} is the projection of the cross-field direction on the parallel transport, and is the `dneut` input setting.

struct **`NeutralParallelDiffusion`** : public Component

Add effective diffusion of neutrals in a 1D system, by projecting cross-field diffusion onto parallel distance.

Note: This needs to be calculated after the collision frequency, so is a collective component. This therefore applies diffusion to all neutral species i.e. those with no (or zero) charge

If diagnose = true then the following outputs are saved for each neutral species

- D<name>_Dpar Parallel diffusion coefficient e.g. Dhe_Dpar
- S<name>_Dpar Density source due to diffusion
- E<name>_Dpar Energy source due to diffusion
- F<name>_Dpar Momentum source due to diffusion

Public Functions

virtual void **transform**(Options &state) override

Inputs

- species
  - <all neutrals>="”> # Applies to all neutral species
    - AA
    - collision_frequency
    - density
    - temperature
    - pressure [optional, or density * temperature]
    - velocity [optional]
    - momentum [if velocity set]

Sets

- species
  - <name>
    - density_source
    - energy_source
    - momentum_source [if velocity set]

virtual void **outputVars**(Options &state) override

Save variables to the output.

7.10.3 collisions

For collisions between charged particles. In the following all quantities are in SI units except the temperatures: T is in eV, so eT has units of Joules.

Debye length λ_D

$$\lambda_D = \sqrt{\frac{\epsilon_0 T_e}{n_e e}}$$

Coulomb logarithm, from [NRL formulary 2019], adapted to SI units

- For thermal electron-electron collisions

$$\ln \lambda_{ee} = 30.4 - \frac{1}{2}\ln(n_e) + \frac{5}{4}\ln(T_e) - \sqrt{10^{-5} + (\ln T_e - 2)^2/16}$$

  where the coefficient (30.4) differs from the NRL value due to converting density from cgs to SI units ($30.4 = 23.5 - 0.5\ln\left(10^{-6}\right)$).

- Electron-ion collisions

$$\ln \lambda_{ei} = \begin{cases} 10 & \text{if } T_e < 0.1\text{eV or } n_e < 10^{10}m^{-3} \\ 30 - \frac{1}{2}\ln(n_e) - \ln(Z) + \frac{3}{2}\ln(T_e) & \text{if } T_i m_e/m_i < T_e < 10Z^2 \\ 31 - \frac{1}{2}\ln(n_e) + \ln(T_e) & \text{if } T_i m_e/m_i < 10Z^2 < T_e \\ 23 - \frac{1}{2}\ln(n_i) + \frac{3}{2}\ln(T_i) - \ln\left(Z^2\mu\right) & \text{if } T_e < T_i m_e/m_i \end{cases}$$

- Mixed ion-ion collisions

  $$\ln\lambda_{ii'} = 29.91 - ln\left[\frac{ZZ'\left(\mu+\mu'\right)}{\mu T_{i'} + \mu' T_i}\left(\frac{n_i Z^2}{T_i} + \frac{n_{i'}Z'^2}{T_{i'}}\right)^{1/2}\right]$$

  where like the other expressions the different constant is due to converting from cgs to SI units: $29.91 = 23 - 0.5\ln\left(10^{-6}\right)$.

The frequency of charged species `a` colliding with charged species `b` is

$$\nu_{ab} = \frac{1}{3\pi^{3/2}\epsilon_0^2}\frac{Z_a^2 Z_b^2 n_b \ln\Lambda}{\left(v_a^2 + v_b^2\right)^{3/2}}\frac{\left(1 + m_a/m_b\right)}{m_a^2}$$

Note that the cgs expression in Hinton is divided by $\left(4\pi\epsilon_0\right)^2$ to get the expression in SI units. The thermal speeds in this expression are defined as:

$$v_a^2 = 2eT_a/m_a$$

Note that with this definition we recover the Braginskii expressions for e-i and i-i collision times.

For conservation of momentum, the collision frequencies ν_{ab} and ν_{ba} are related by:

$$m_a n_a \nu_{ab} = m_b n_b \nu_{ba}$$

Momentum exchange, force on species `a` due to collisions with species `b`:

$$F_{ab} = C_m \nu_{ab} m_a n_a \left(u_b - u_a\right)$$

Where the coefficient C_m for parallel flows depends on the species: For most combinations of species this is set to 1, but for electron-ion collisions the Braginskii coefficients are used: $C_m = 0.51$ if ion charge $Z_i = 1$; 0.44 for $Z_i = 2$; 0.40 for $Z_i = 3$; and 0.38 is used for $Z_i \geq 4$. Note that this coefficient should decline further with increasing ion charge, tending to 0.29 as $Z_i \rightarrow \infty$.

Frictional heating is included by default, but can be disabled by setting the `frictional_heating` option to `false`. When enabled it adds a source of thermal energy corresponding to the resistive heating term:

$$Q_{ab,F} = \frac{m_b}{m_a + m_b}\left(u_b - u_a\right)F_{ab}$$

This term has some important properties:

1. It is always positive: Collisions of two species with the same temperature never leads to cooling.
2. It is Galilean invariant: Shifting both species' velocity by the same amount leaves $Q_{ab,F}$ unchanged.
3. If both species have the same mass, the thermal energy change due to slowing down is shared equally between them.
4. If one species is much heavier than the other, for example electron-ion collisions, the lighter species is preferentially heated. This recovers e.g. Braginskii expressions for Q_{ei} and Q_{ie}.

This can be derived by considering the exchange of energy $W_{ab,F}$ between two species at the same temperature but different velocities. If the pressure is evolved then it contains a term that balances the change in kinetic energy due to changes in velocity:

$$\begin{aligned}\frac{\partial}{\partial t}\left(m_a n_a u_a\right) &= \ldots + F_{ab}\\ \frac{\partial}{\partial t}\left(\frac{3}{2}p_a\right) &= \ldots - F_{ab}u_a + W_{ab,F}\end{aligned}$$

For momentum and energy conservation we must have $F_{ab} = -F_{ba}$ and $W_{ab,F} = -W_{ba,F}$. Comparing the above to the Braginskii expression we see that for ion-electron collisions the term $-F_{ab}u_a + W_{ab,F}$ goes to zero, so $W_{ab,F} \sim u_a F_{ab}$ for $m_a \gg m_b$. An expression that has all these desired properties is

$$W_{ab,F} = \left(\frac{m_a u_a + m_b u_a}{m_a + m_b}\right) F_{ab}$$

which is not Galilean invariant but when combined with the $-F_{ab}u_a$ term gives a change in pressure that is invariant, as required.

Thermal energy exchange, heat transferred to species a from species b due to temperature differences, is given by:

$$Q_{ab,T} = \nu_{ab} \frac{3 n_a m_a \left(T_b - T_a\right)}{m_a + m_b}$$

- Ion-neutral and electron-neutral collisions

  The cross-section for elastic collisions between charged and neutral particles can vary significantly. Here for simplicity we just take a value of $5 \times 10^{-19} m^2$ from the NRL formulary.

- Neutral-neutral collisions

  The cross-section is given by

$$\sigma = \pi \left(\frac{d_1 + d_2}{2}\right)^2$$

where d_1 and d_2 are the kinetic diameters of the two species. Typical values are [Wikipedia] for H2 2.89e-10m, He 2.60e-10m, Ne 2.75e-10m.

The mean relative velocity of the two species is

$$v_{rel} = \sqrt{\frac{eT_1}{m_1} + \frac{eT_2}{m_2}}$$

and so the collision rate of species 1 on species 2 is:

$$\nu_{12} = v_{rel} n_2 \sigma$$

The implementation is in `Collisions`:

struct **`Collisions`** : public Component

Calculates the collision rate of each species with all other species

Important: Be careful when including both ion_neutral collisions and reactions such as charge exchange, since that may result in double counting. Similarly for electron_neutral collisions and ionization reactions.

Public Functions

`Collisions`(std::string name, Options &alloptions, Solver*)

The following boolean options under alloptions[name] control which collisions are calculated:

- electron_electron
- electron_ion
- electron_neutral

- ion_ion
- ion_neutral
- neutral_neutral

There are also switches for other terms:

- frictional_heating Include R dot v heating term as energy source? (includes Ohmic heating)

Parameters
`alloptions` – Settings, which should include:

- units
  - eV
  - inv_meters_cubed
  - meters
  - seconds

virtual void `transform(Options &state)` override

Modify the given simulation state All components must implement this function

7.10.4 thermal_force

This implements simple expressions for the thermal force. If the `electron_ion` option is true (which is the default), then a momentum source is added to all ions:

$$F_z = 0.71 n_z Z^2 \nabla_{||} T_e$$

where n_z is the density of the ions of charge Z. There is an equal and opposite force on the electrons.

If the `ion_ion` option is true (the default), then forces are calculated between light species (atomic mass < 4) and heavy species (atomic mass > 10). If any combinations of ions are omitted, then a warning will be printed once. The force on the heavy ion is:

$$\begin{aligned} F_z =& \beta \nabla_{||} T_i \\ \beta =& \frac{3\left(\mu + 5\sqrt{2}Z^2\left(1.1\mu^{5/2} - 0.35\mu^{3/2}\right) - 1\right)}{2.6 - 2\mu + 5.4\mu^2} \\ \mu =& m_z / \left(m_z + m_i\right) \end{aligned}$$

where subscripts z refer to the heavy ion, and i refers to the light ion. The force on the light ion fluid is equal and opposite: $F_i = -F_z$.

The implementation is in the `ThermalForce` class:

struct `ThermalForce` : public Component

Simple calculation of the thermal force

Important: This implements a quite crude approximation, which is intended for initial development and testing. The expressions used are only valid for trace heavy ions and light main ion species, and would not be valid for Helium impurities in a D-T plasma, for example. For this reason only collisions where one ion has an atomic mass < 4, and the other an atomic mass > 10 are considered. Warning messages will be logged for species combinations which are not calculated.

Options used:

- <name>
  - electron_ion : bool Include electron-ion collisions?
  - ion_ion : bool Include ion-ion elastic collisions?

Public Functions

virtual void **transform**(Options &state) override

Inputs

- species
  - e [if electron_ion true]
    - charge
    - density
    - temperature
  - <species>
    - charge [Checks, skips species if not set]
    - AA
    - temperature [If AA < 4 i.e. "light" species]

Outputs

- species
  - e
    - momentum_source [if electron_ion true]
  - <species> [if AA < 4 ("light") or AA > 10 ("heavy")]
    - momentum_source

7.10.5 recycling

This component calculates the flux of a species into a Y boundary, due to recycling of flow out of the boundary of another species.

The boundary fluxes might be set by sheath boundary conditions, which potentially depend on the density and temperature of all species. Recycling therefore can't be calculated until all species boundary conditions have been set. It is therefore expected that this component is a top-level component which comes after boundary conditions are set.

The recycling component has a `species` option, that is a list of species to recycle. For each of the species in that list, `recycling` will look in the corresponding section for the options `recycle_as`, `recycle_multiplier` and `recycle_energy`.

For example, recycling `d+` ions into `d` atoms with a recycling fraction of 1. Each returning atom has an energy of 3.5eV:

```
[hermes]
components = d+, d, sheath_boundary, recycling

[recycling]
species = d+   # Comma-separated list of species to recycle

[d+]
recycle_as = d         # Species to recycle as
recycle_multiplier = 1 # Recycling fraction
recycle_energy = 3.5   # Energy of recycled particles [eV]
```

struct **`Recycling`** : public Component

Convert fluxes of species at boundaries

Since this must be calculated after boundary fluxes (e.g. sheath), it is included as a top-level component

Public Functions

`Recycling`(std::string name, Options &alloptions, Solver*)

Inputs

- <name>
  - species A comma-separated list of species to recycle
- <species>
  - recycle_as The species to recycle into
  - recycle_multiplier The recycled flux multiplier, between 0 and 1
  - recycle_energy The energy of the recycled particles [eV]

virtual void **`transform`**(Options &state) override

Inputs

- species
  - <species>
    - density
    - velocity

Outputs

- species
  - <species>
    - density_source

7.11 Atomic and molecular reactions

The formula for the reaction is used as the name of the component. This makes writing the input file harder, since the formula must be in the exact same format (e.g. `h + e` and `e + h` won't be recognised as being the same thing), but makes reading and understanding the file easier.

To include a set of reactions, it is probably easiest to group them, and then include the group name in the components list

```
[hermes]
components = ..., reactions

[reactions]
type = (
        h + e -> h+ + 2e,  # ionisation
        h+ + e -> h,    # Radiative + 3-body recombination
       )
```

Note that brackets can be used to split the list of reactions over multiple lines, and trailing commas are ignored. Comments can be used if needed to add explanation. The name of the section does not need to be `reactions`, and multiple components could be created with different reaction sets. Be careful not to include the same reaction twice.

When reactions are added, all the species involved must be included, or an exception should be thrown.

Notes:

1. Charge exchange channel diagnostics: For two species `a` and `b`, the channel `Fab_cx` is a source of momentum for species `a` due to charge exchange with species `b`. There are corresponding sinks for the products of the charge exchange reaction which are not saved.

   For example,reaction `d + t+ -> d+ + t` will save the following forces (momentum sources): - `Fdt+_cx` is a source of momentum for deuterium atoms `d` and sink of momentum for deuterium ions `d+`. - `Ft+d_cx` is a source of momentum for tritium ions `t+` and sink of momentum for tritium atoms `t`

   The reason for this convention is the existence of the inverse reactions: `t + d+ -> t+ + d` outputs diagnostics `Ftd+_cx` and `Fd+t_cx`.

2. Reactions typically convert species from one to another, leading to a transfer of mass momentum and energy. For a reaction converting species a to species b at rate R (units of events per second per volume) we have transfers:

$$\begin{aligned}
\frac{\partial}{\partial t}n_a &= \ldots - R \\
\frac{\partial}{\partial t}n_b &= \ldots + R \\
\frac{\partial}{\partial t}\left(mn_a u_a\right) &= \ldots + F_{ab} \\
\frac{\partial}{\partial t}\left(mn_a u_a\right) &= \ldots + F_{ba} \\
\frac{\partial}{\partial t}\left(\frac{3}{2}p_a\right) &= \ldots - F_{ab}u_a + W_{ab} - \frac{1}{2}mRu_a^2 \\
\frac{\partial}{\partial t}\left(\frac{3}{2}p_b\right) &= \ldots - F_{ba}u_b + W_{ba} + \frac{1}{2}mRu_b^2
\end{aligned}$$

   where both species have the same mass: $m_a = m_b = m$. In the pressure equations the $-F_{ab}u_a$ comes from splitting the kinetic and thermal energies; $W_{ab} = -W_{ba}$ is the energy transfer term that we need to find; The final term balances the loss of kinetic energy at fixed momentum due to a particle source or sink.

The momentum transfer $F_{ab} = -Fba$ is the momentum carried by the converted ions: $F_{ab} = -mRu_a$. To find W_{ab} we note that for $p_a = 0$ the change in pressure must go to zero: $-F_{ab}u_a + W_{ab} - \frac{1}{2}mRu_a^2 = 0$.

$$\begin{aligned} W_{ab} =& F_{ab}u_a + \frac{1}{2}mRu_a^2 \\ =& -mRu_a^2 + \frac{1}{2}mRu_a^2 \\ =& -\frac{1}{2}mRu_a^2 \end{aligned}$$

Substituting into the above gives:

$$\begin{aligned} \frac{\partial}{\partial t}\left(\frac{3}{2}p_b\right) =& \ldots - F_{ba}u_b + W_{ba} + \frac{1}{2}mRu_b^2 \\ =& \ldots - mRu_au_b + \frac{1}{2}mRu_a^2 + \frac{1}{2}mRu_a^2 \\ =& \ldots + \frac{1}{2}mR\left(u_a - u_b\right)^2 \end{aligned}$$

This has the property that the change in pressure of both species is Galilean invariant. This transfer term is included in the Amjuel reactions and hydrogen charge exchange.

7.11.1 Hydrogen

Multiple isotopes of hydrogen can be evolved, so to keep track of this the species labels `h`, `d` and `t` are all handled by the same hydrogen atomic rates calculation. The following might therefore be used

```
[hermes]
components = d, t, reactions

[reactions]
type = (
        d + e -> d+ + 2e,   # Deuterium ionisation
        t + e -> t+ + 2e,   # Tritium ionisation
       )
```

Reaction	Description
h + e -> h+ + 2e	Hydrogen ionisation (Amjuel 2.1.5)
d + e -> d+ + 2e	Deuterium ionisation (Amjuel 2.1.5)
t + e -> t+ + 2e	Tritium ionisation (Amjuel 2.1.5)
h + h+ -> h+ + h	Hydrogen charge exchange
d + d+ -> d+ + d	Deuterium charge exchange
t + t+ -> t+ + t	Tritium charge exchange
h + d+ -> h+ + d	Mixed hydrogen isotope CX
d + h+ -> d+ + h	
h + t+ -> h+ + t	
t + h+ -> t+ + h	
d + t+ -> d+ + t	
t + d+ -> t+ + d	
h+ + e -> h	Hydrogen recombination (Amjuel 2.1.8)
d+ + e -> d	Deuterium recombination (Amjuel 2.1.8)
t+ + e -> t	Tritium recombination (Amjuel 2.1.8)

The code to calculate the charge exchange rates is in `hydrogen_charge_exchange.[ch]xx`. This implements reaction 3.1.8 from Amjuel (p43), scaled to different isotope masses and finite neutral particle temperatures by using the effective temperature (Amjuel p43):

$$T_{eff} = \frac{M}{M_1}T_1 + \frac{M}{M_2}T_2$$

The effective hydrogenic ionisation rates are calculated using Amjuel reaction 2.1.5, by D.Reiter, K.Sawada and T.Fujimoto (2016). Effective recombination rates, which combine radiative and 3-body contributions, are calculated using Amjuel reaction 2.1.8.

struct **`HydrogenChargeExchange`** : public Component

Hydrogen charge exchange total rate coefficient

p + H(1s) -> H(1s) + p

Reaction 3.1.8 from Amjuel (p43)

Scaled to different isotope masses and finite neutral particle temperatures by using the effective temperature (Amjuel p43)

T_eff = (M/M_1)T_1 + (M/M_2)T_2

Important: If this is included then ion_neutral collisions should probably be disabled in the `collisions` component, to avoid double-counting.

Subclassed by HydrogenChargeExchangeIsotope< Isotope1, Isotope2 >

Public Functions

inline **`HydrogenChargeExchange`**(std::string name, Options &alloptions, Solver*)

Parameters

`alloptions` – Settings, which should include:

- units
  - eV
  - inv_meters_cubed
  - seconds

7.11.2 Helium

Reaction	Description
he + e -> he+ + 2e	He ionisation, unresolved metastables (Amjuel 2.3.9a)
he+ + e -> he	He+ recombination, unresolved metastables (Amjuel 2.3.13a)

The implementation of these rates are in the `AmjuelHeIonisation01` and `AmjuelHeRecombination10` classes:

struct **`AmjuelHeIonisation01`** : public AmjuelReaction

e + he -> he+ + 2e Amjuel reaction 2.3.9a, page 161 Not resolving metastables, only transporting ground state

Public Functions

inline virtual void **transform**(Options &state) override

Modify the given simulation state All components must implement this function

struct **AmjuelHeRecombination10** : public AmjuelReaction

e + he+ -> he Amjuel reaction 2.3.13a Not resolving metastables. Includes radiative + threebody + dielectronic. Fujimoto Formulation II

Public Functions

inline virtual void **transform**(Options &state) override

Modify the given simulation state All components must implement this function

7.11.3 Neon

These rates are taken from ADAS (96): SCD and PLT are used for the ionisation rate and radiation energy loss; ACD and PRB for the recombination rate and radiation energy loss; and CCD (89) for the charge exchange coupling to hydrogen. The ionisation potential is also included as a source or sink of energy for the electrons.

Reaction	Description
ne + e -> ne+ + 2e	Neon ionisation
ne+ + e -> ne+2 + 2e	
ne+2 + e -> ne+3 + 2e	
ne+3 + e -> ne+4 + 2e	
ne+4 + e -> ne+5 + 2e	
ne+5 + e -> ne+6 + 2e	
ne+6 + e -> ne+7 + 2e	
ne+7 + e -> ne+8 + 2e	
ne+8 + e -> ne+9 + 2e	
ne+9 + e -> ne+10 + 2e	
ne+ + e -> ne	Neon recombination
ne+2 + e -> ne+	
ne+3 + e -> ne+2	
ne+4 + e -> ne+3	
ne+5 + e -> ne+4	
ne+6 + e -> ne+5	
ne+7 + e -> ne+6	
ne+8 + e -> ne+7	
ne+9 + e -> ne+8	
ne+10 + e -> ne+9	
ne+ + h -> ne + h+	Charge exchange with hydrogen
ne+2 + h -> ne+ + h+	
ne+3 + h -> ne+2 + h+	
ne+4 + h -> ne+3 + h+	
ne+5 + h -> ne+4 + h+	
ne+6 + h -> ne+5 + h+	
ne+7 + h -> ne+6 + h+	
ne+8 + h -> ne+7 + h+	

continues on next page

Table 7.1 – continued from previous page

Reaction	Description
ne+9 + h -> ne+8 + h+	
ne+10 + h -> ne+9 + h+	
ne+ + d -> ne + d+	Charge exchange with deuterium
ne+2 + d -> ne+ + d+	
ne+3 + d -> ne+2 + d+	
ne+4 + d -> ne+3 + d+	
ne+5 + d -> ne+4 + d+	
ne+6 + d -> ne+5 + d+	
ne+7 + d -> ne+6 + d+	
ne+8 + d -> ne+7 + d+	
ne+9 + d -> ne+8 + d+	
ne+10 + d -> ne+9 + d+	
ne+ + t -> ne + t+	Charge exchange with tritium
ne+2 + t -> ne+ + t+	
ne+3 + t -> ne+2 + t+	
ne+4 + t -> ne+3 + t+	
ne+5 + t -> ne+4 + t+	
ne+6 + t -> ne+5 + t+	
ne+7 + t -> ne+6 + t+	
ne+8 + t -> ne+7 + t+	
ne+9 + t -> ne+8 + t+	
ne+10 + t -> ne+9 + t+	

The implementation of these rates is in `ADASNeonIonisation`, `ADASNeonRecombination` and `ADASNeonCX` template classes:

template<int **level**>

struct **ADASNeonIonisation** : public OpenADAS

ADAS effective ionisation (ADF11)

Template Parameters

level – The ionisation level of the ion on the left of the reaction

Public Functions

inline virtual void **transform**(Options &state) override

Modify the given simulation state All components must implement this function

template<int **level**>

struct **ADASNeonRecombination** : public OpenADAS

ADAS effective recombination coefficients (ADF11)

Template Parameters

level – The ionisation level of the ion on the right of the reaction

Public Functions

inline **ADASNeonRecombination**(std::string, Options &alloptions, Solver*)

Parameters
alloptions – The top-level options. Only uses the ["units"] subsection.

inline virtual void **transform**(Options &state) override
Modify the given simulation state All components must implement this function

template<int **level**, char **Hisotope**>

struct **ADASNeonCX** : public OpenADASChargeExchange

Template Parameters

- **level** – The ionisation level of the ion on the right of the reaction
- **Hisotope** – The hydrogen isotope ('h', 'd' or 't')

Public Functions

inline **ADASNeonCX**(std::string, Options &alloptions, Solver*)

Parameters
alloptions – The top-level options. Only uses the ["units"] subsection.

inline virtual void **transform**(Options &state) override
Modify the given simulation state All components must implement this function

7.11.4 Fixed fraction radiation

These components produce volumetric electron energy losses, but don't otherwise modify the plasma solution: Their charge and mass density are not calculated, and there are no interactions with other species or boundary conditions.

The fixed_fraction_carbon component calculates radiation due to carbon in coronal equilibrium, using a simple formula from I.H.Hutchinson Nucl. Fusion 34 (10) 1337 - 1348 (1994):

$$L\left(T_e\right) = 2\times 10^{-31}\frac{\left(T_e/10\right)^3}{1 + \left(T_e/10\right)^{4.5}}$$

which has units of Wm^3 with T_e in eV.

To use this component you can just add it to the list of components and then configure the impurity fraction:

```
[hermes]
components = ..., fixed_fraction_carbon, ...

[fixed_fraction_carbon]
fraction = 0.05    # 5% of electron density
diagnose = true    # Saves Rfixed_fraction_carbon to output
```

Or to customise the name of the radiation output diagnostic a section can be defined like this:

```
[hermes]
components = ..., c, ...

[c]
type = fixed_fraction_carbon
fraction = 0.05   # 5% of electron density
diagnose = true   # Saves Rc (R + section name)
```

The fixed_fraction_nitrogen component works in the same way, calculating nitrogen radiation using a formula from Bruce Lipschultz et al 2016 Nucl. Fusion 56 056007:

$$L\left(T_e\right) = \begin{cases} 5.9\times 10^{-34}\frac{\sqrt{T_e - 1}\left(80 - T_e\right)}{1 + 3.1\times 10^{-3}\left(T_e - 1\right)^2} & \text{If } 1 < T_e < 80\text{eV} \\ 0 & \text{Otherwise} \end{cases}$$

The fixed_fraction_neon component use a piecewise polynomial fit to the neon cooling curve (Ryoko 2020 Nov):

$$L\left(T\right) = \begin{cases} \sum_{i=0}^5 a_i T_e^i & \text{If } 3 \le T_e < 100\text{eV} \\ 7\times 10^{-35}\left(T_e - 2\right) + 10^{-35} & \text{If } 2 \le T_e < 3\text{eV} \\ 10^{-35}\left(T_e - 1\right) & \text{If } 1 < T_e < 2\text{eV} \\ 0 & \text{Otherwise} \end{cases}$$

where the coefficients of the polynomial fit are $a_0 = -3.2798\times 10^{-34}$, $a_1 = -3.4151\times 10^{-34}$, $a_2 = 1.7347\times 10^{-34}$, $a_3 = -5.119\times 10^{-36}$, $a_4 = 5.4824\times 10^{-38}$, $a_5 = -2.0385\times 10^{-40}$.

The fixed_fraction_argon components uses a piecewise polynomial fit to the argon cooling curve (Ryoko 2020 Nov):

$$L\left(T\right) = \begin{cases} \sum_{i=0}^9 b_i T_e^i & \text{If } 1.5 \le T_e < 100\text{eV} \\ 5\times 10^{-35}\left(T_e - 1\right) & \text{If } 1 \le T_e < 1.5\text{eV} \\ 0 & \text{Otherwise} \end{cases}$$

where polynomial coefficients $b_0 \dots b_9$ are $-9.9412e-34$, $4.9864e-34$, $1.9958e-34$, $8.6011e-35$, $-8.341e-36$, $3.2559e-37$, $-6.9642e-39$, $8.8636e-41$, $-6.7148e-43$, $2.8025e-45$, $-4.9692e-48$.

7.12 Electromagnetic fields

These are components which calculate the electric and/or magnetic fields.

7.12.1 vorticity

Evolves a vorticity equation, and at each call to transform() uses a matrix inversion to calculate potential from vorticity.

In this component the Boussinesq approximation is made, so the vorticity equation solved is

$$\nabla\cdot\left(\frac{\overline{A}\overline{n}}{B^2}\nabla_\perp\phi\right) \underbrace{+\nabla\cdot\left(\sum_i\frac{A_i}{Z_i B^2}\nabla_\perp p_i\right)}_{\text{ifdiamagnetic\_polarisation}} = \Omega$$

Where the sum is over species, $\overline{A}$ is the average ion atomic number, and $\overline{n}$ is the normalisation density (i.e. goes to 1 in the normalised equations). The ion diamagnetic flow terms in this Boussinesq approximation can be written in terms of an effective ion pressure $\hat{p}$:

$$\hat{p} \equiv \sum_i \frac{A_i}{\overline{A}Z_i}p_i$$

as

$$\nabla \cdot \left[\frac{\overline{A}\overline{n}}{B^2}\nabla_\perp\left(\phi + \frac{\hat{p}}{\overline{n}}\right)\right] = \Omega$$

Note that if `diamagnetic_polarisation = false` then the ion pressure terms are removed from the vorticity, and also from other ion pressure terms coming from the polarisation current (i.e. $\hat{p} \rightarrow 0$.

This is a simplified version of the full vorticity definition which is:

$$\nabla \cdot \left(\sum_i \frac{A_i n_i}{B^2}\nabla_\perp \phi + \sum_i \frac{A_i}{Z_i B^2}\nabla_\perp p_i\right) = \Omega$$

and is derived by replacing

$$\sum_i A_i n_i \rightarrow \overline{A}\overline{n}$$

In the case of multiple species, this Boussinesq approximation means that the ion diamagnetic flow terms

The vorticity equation that is integrated in time is

$$\begin{aligned}\frac{\partial \Omega}{\partial t} =& \nabla \cdot \left(\mathbf{b}\sum_s Z_s n_s V_{||s}\right) \\ &+ \underbrace{\nabla \cdot \left(\nabla \times \frac{\mathbf{b}}{B}\sum_s p_s\right)}_{\text{if diamagnetic}} + \underbrace{\nabla \cdot \mathbf{J_{exb}}}_{\text{ifexb\_advection}} \\ &+ \nabla \cdot \left(\mathbf{b}J_{extra}\right)\end{aligned}$$

The nonlinearity $\nabla \cdot \mathbf{J_{exb}}$ is part of the divergence of polarisation current. In its simplified form when `exb_advection_simplified = true`, this is the $E \times B$ advection of vorticity:

$$\nabla \cdot \mathbf{J_{exb}} = -\nabla \cdot \left(\Omega \mathbf{V}_{E\times B}\right)$$

When `exb_advection_simplified = false` then the more complete (Boussinesq approximation) form is used:

$$\nabla \cdot \mathbf{J_{exb}} = -\nabla \cdot \left[\frac{\overline{A}}{2B^2}\nabla_\perp\left(\mathbf{V}_{E\times B} \cdot \nabla \hat{p}\right) + \frac{\Omega}{2}\mathbf{V}_{E\times B} + \frac{\overline{A}\overline{n}}{2B^2}\nabla^2_\perp \phi \left(\mathbf{V}_{E\times B} + \frac{\mathbf{b}}{B} \times \nabla \hat{p}\right)\right]$$

The form of the vorticity equation is based on Simakov & Catto (corrected in erratum 2004), in the Boussinesq limit and with the first term modified to conserve energy. In the limit of zero ion pressure and constant B it reduces to the simplified form.

struct **`Vorticity`** : public Component

Evolve electron density in time

Public Functions

`Vorticity`(std::string name, Options &options, Solver *solver)

Options

- <name>

  - average_atomic_mass: float, default 2.0 Weighted average ion atomic mass for polarisation current
  - bndry_flux: bool, default true Allow flows through radial (X) boundaries?
  - collisional_friction: bool, default false Damp vorticity based on mass-weighted collision frequency?
  - diagnose: bool, false Output additional diagnostics?
  - diamagnetic: bool, default true Include diamagnetic current, using curvature vector?
  - diamagnetic_polarisation: bool, default true Include ion diamagnetic drift in polarisation current?
  - exb_advection: bool, default true Include ExB advection (nonlinear term)?
  - hyper_z: float, default -1.0 Hyper-viscosity in Z. < 0 means off
  - laplacian: subsection Options for the Laplacian phi solver
  - phi_boundary_relax: bool, default false Relax radial phi boundaries towards zero-gradient?
  - phi_boundary_timescale: float, 1e-4 Timescale for phi boundary relaxation [seconds]
  - phi_dissipation: bool, default true Parallel dissipation of potential (Recommended)
  - poloidal_flows: bool, default true Include poloidal ExB flow?
  - sheath_boundary: bool, default false If phi_boundary_relax is false, set the radial boundary to the sheath potential?
  - split_n0: bool, default false Split phi into n=0 and n!=0 components?
  - viscosity: Field2D, default 0.0 Kinematic viscosity [m^2/s]
  - vort_dissipation: bool, default false Parallel dissipation of vorticity?

virtual void **transform**(Options &state) override

Optional inputs

- species
  - pressure and charge => Calculates diamagnetic terms [if diamagnetic=true]
  - pressure, charge and mass => Calculates polarisation current terms [if diamagnetic_polarisation=true]

Sets in the state

- species
  - [if has pressure and charge]
    - energy_source
- fields
  - vorticity
  - phi Electrostatic potential
  - DivJdia Divergence of diamagnetic current [if diamagnetic=true]

Note: Diamagnetic current calculated here, but could be moved to a component with the diamagnetic drift advection terms virtual void **finally**(const Options &state) override

Optional inputs

- fields
  - DivJextra Divergence of current, including parallel current Not including diamagnetic or polarisation currents

virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

inline virtual void **restartVars**(Options &state) override

Add extra fields to restart files.

7.12.2 relax_potential

This component evolves a vorticity equation, similar to the `vorticity` component. Rather than inverting an elliptic equation at every timestep, this component evolves the potential in time as a diffusion equation.

struct **RelaxPotential** : public Component

Evolve vorticity and potential in time.

Uses a relaxation method for the potential, which is valid for steady state, but not for timescales shorter than the relaxation timescale.

Public Functions

RelaxPotential(std::string name, Options &options, Solver *solver)

Options

- <name>
  - diamagnetic
  - diamagnetic_polarisation
  - average_atomic_mass
  - bndry_flux
  - poloidal_flows
  - split_n0
  - laplacian Options for the Laplacian phi solver

virtual void **transform**(Options &state) override

Optional inputs

- species
  - pressure and charge => Calculates diamagnetic terms [if diamagnetic=true]
  - pressure, charge and mass => Calculates polarisation current terms [if diamagnetic_polarisation=true]

Sets in the state

- species
  - [if has pressure and charge]
    - energy_source
- fields
  - vorticity
  - phi Electrostatic potential
  - DivJdia Divergence of diamagnetic current [if diamagnetic=true]

Note: Diamagnetic current calculated here, but could be moved to a component with the diamagnetic drift advection terms

virtual void **finally**(const Options &state) override

Optional inputs

- fields
  - DivJextra Divergence of current, including parallel current Not including diamagnetic or polarisation currents

virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

7.12.3 electromagnetic

This component modifies the definition of momentum of all species, to include the contribution from the electromagnetic potential $A_{||}$.

Assumes that "momentum" p_s calculated for all species s is

$$p_s = m_s n_s v_{||s} + Z_s e n_s A_{||}$$

which arises once the electromagnetic contribution to the force on each species is included in the momentum equation. This is normalised so that in dimensionless quantities

$$p_s = A n v_{||} + Z n A_{||}$$

where A and Z are the atomic number and charge of the species.

The current density $j_{||}$ in SI units is

$$j_{||} = -\frac{1}{\mu_0}\nabla_\perp^2 A_{||}$$

which when normalised in Bohm units becomes

$$j_{||} = -\frac{1}{\beta_{em}}\nabla_\perp^2 A_{||}$$

where β_{em} is a normalisation parameter which is half the plasma electron beta as normally defined:

$$\beta_{em} = \frac{\mu_0 e \overline{n}\overline{T}}{\overline{B}^2}$$

To convert the species momenta into a current, we take the sum of $p_s Z_s e/m_s$. In terms of normalised quantities this gives:

$$-\frac{1}{\beta_{em}}\nabla_\perp^2 A_{||} + \sum_s \frac{Z^2 n_s}{A} A_{||} = \sum_s \frac{Z}{A} p_s$$

struct **Electromagnetic** : public Component

Electromagnetic potential A||

Reinterprets all species' parallel momentum as a combination of a parallel flow and a magnetic contribution, i.e. canonical momentum.

```
m n v_{||} + Z e n A_{||}
```

Changes the "momentum" of each species so that after this component the momentuum of each species is just

```
m n v_{||}
```

This component should be run after all species have set their momentum, but before the momentum is used e.g to set boundary conditions.

Calculates the electromagnetic potential A_{||} using

Laplace(Apar) - alpha_em * Apar = -Ajpar

By default outputs Apar every timestep. When `diagnose = true` in also saves alpha_em and Ajpar.

Public Functions

Electromagnetic(std::string name, Options &options, Solver *solver)

Options

- units
- <name>
  - diagnose Saves Ajpar and alpha_em time-dependent values

virtual void **transform**(Options &state) override

Inputs

- species
  - <..> All species with charge and parallel momentum
    - charge
    - momentum
    - density
    - AA

Sets

- species
  - <..> All species with charge and parallel momentum
    - momentum (modifies) to m n v||
    - velocity (modifies) to v||
- fields
  - Apar *Electromagnetic* potential

virtual void **outputVars**(Options &state) override

Add extra fields for output, or set attributes e.g docstrings.

INDEX