The *muphy*II Multiphysics Simulation Framework

User Manual

Insitute for Theoretical Physics I
Ruhr University Bochum
2019 – 2023

Last updated: May 3, 2023

Contents

1 Important information for external users

This document covers both aspects that are relevant to interested external people (i.e. not from Theoretical Physics I Bochum and research partners) and aspects that are only relevant internally.

Due to limited personnel,

- we unfortunately cannot provide support
- the language quality and the extent of this manual are not as high as we would want them to be
- there is no consistent error handling in the code.

There is absolutely no warranty of any kind!

2 License

3 About *muphy*II

The *muphy*II multi-physics code provides a framework for coupling physical models and is applied in the context of plasma simulations. Kinetic, hybrid and multi-fluid continuum models can be run either individually or spatially coupled to each other. The name muphy is an acronym for multi physics and *muphy*II is the new generation of the code with origins in the earlier code *muphy*.

The framework was developed with high performance, high maintainability and high flexibility in mind, three essential features of a scientific code. Its logic is written in C++ making use of the language's flexibility, whereas for the numeric calculations Fortran is employed which offers best performance in multi-dimensional array operations and good maintainability at the same time.

Parallelization is done via MPI and OpenACC so that the code can utilize GPUs of various types as well as CPUs with the same code base. This gives maximum flexibility in the choice of target architecture.

4 Compile

The Makefile is located in `bin/Makefile`. Thus, open a command line, change to the `muphy2src/bin/` directory and type

```
make -j 8
```

to compile with 8 processes (replace 8 with whatever you like). By typing

```
make clean
```

all build files can be removed.

4.1 Compilation for GPU and other compilation settings

To adjust the compilation process, use an editor of your choice and open the Makefile.

*muphy*II can be compiled for CPU or for GPU. This can be adjusted manually in the Makefile by commenting in/out the respective commands for CPU and GPU compilation.

Using the SETTINGS variable in the Makefile, you can turn on single precision (default double precision) and/or 2D GPU optimizations (default optimized for 3D). Example:

```
SETTINGS = -DSINGLE_PRECISION -DACC_2D_OPT
```

leads to compilation with single precision and OpenAcc optimizations for 2D.

By default the GCC compiler is used for CPU compilation and the PGI/Nvidia compiler for GPU compilation. To use other compilers, you have to manually comment in/out the respective commands in the Makefile:

Compilation for CPU:

```
# GCC with OpenMPI or ParaStationMPI
# tested on davinci and JUWELS-booster
# optionally before compiling on davinci use
# module purge; module load gcc mpi/openmpi3-x86_64
# on JUWELS-booster use
# module load GCC ParaStationMPI
    FLAGS = -Ofast -Wall $(SETTINGS) -I$(SRCDIRECTORY)
    CXXFLAGS = $(FLAGS) -std=c++11 -lgfortran -MMD -Wno-unknown-pragmas
    FORTRANFLAGS = $(FLAGS) -ffree-line-length-none
    MPICXX = mpicxx
    MPIF90 = mpifort

# Nvidia/PGI compiler with OpenMPI or ParaStationMPI
# tested on davinci and JUWELS-booster
# before compiling on davinci use
# module purge; module load nvhpc
# on JUWELS-booster use
# module load NVHPC ParaStationMPI
    #FLAGS = -fast $(SETTINGS) -I$(SRCDIRECTORY)
    #ACCFLAGS = -acc -ta=tesla,fastmath,keepgpu -Minfo=acc,ccff -Mcuda
    #CXXFLAGS = $(FLAGS) -std=c++11 -pgf90libs -MMD $(ACCFLAGS)
    #FORTRANFLAGS = $(FLAGS) $(ACCFLAGS)
    #MPICXX = mpicxx
    #MPIF90 = mpifort
```

Compilation for GPU (same as above, but other part commented in):

```
# GCC with OpenMPI or ParaStationMPI
# tested on davinci and JUWELS-booster
# optionally before compiling on davinci use
# module purge; module load gcc mpi/openmpi3-x86_64
# on JUWELS-booster use
# module load GCC ParaStationMPI
    #FLAGS = -Ofast -Wall $(SETTINGS) -I$(SRCDIRECTORY)
    #CXXFLAGS = $(FLAGS) -std=c++11 -lgfortran -MMD -Wno-unknown-pragmas
    #FORTRANFLAGS = $(FLAGS) -ffree-line-length-none
    #MPICXX = mpicxx
    #MPIF90 = mpifort

# Nvidia/PGI compiler with OpenMPI or ParaStationMPI
# tested on davinci and JUWELS-booster
# before compiling on davinci use
# module purge; module load nvhpc
# on JUWELS-booster use
# module load NVHPC ParaStationMPI
    FLAGS = -fast $(SETTINGS) -I$(SRCDIRECTORY)
    ACCFLAGS = -acc -ta=tesla,fastmath,keepgpu -Minfo=acc,ccff -Mcuda
    CXXFLAGS = $(FLAGS) -std=c++11 -pgf90libs -MMD $(ACCFLAGS)
    FORTRANFLAGS = $(FLAGS) $(ACCFLAGS)
    MPICXX = mpicxx
    MPIF90 = mpifort
```

4.2 Prerequisites

- Linux operating system. Running on Windows using the Windows Subsystem for Linux (WSL2) or on MacOS is also possible, but not covered here.
- A recent C++ and Fortran compiler. Tested are GCC, PGI/Nvidia compiler, Intel compiler.
- A recent MPI version. Tested are OpenMPI, ParaStationMPI, Intel MPI.

5 Run

In the `bin/` directory after compilation type

```
mpirun -np 16 ./muphy2
```

to run with 16 processes.

5.1 Adjust MPI processes, output directory and initial conditions

Simulation settings and initial conditions are set in `framework/initial_condtions.cc` and defined with default values in `framework/parameter.h`. The physical setups that can be chosen with the `parameter::setup` variable are defined in `physics/plasma/setup.F90`. In this document, we refer to the variables that are set in `framework/initial_conditions.cc` by the `parameter::` prefix. **Recompilation is necessary after changes to the setup files!**

5.1.1 MPI processes

The number of MPI processes is defined in `parameter::nproc` which determines the number of processes in each physical direction. Therefore, the number of processes given to mpirun must be identical to `nproc[0]*nproc[1]*nproc[2]`. Example: In a 2D domain with a resolution of 128x64x1 cells one could use `nproc[0]=16`, `nproc[1]=8`, `nproc[2]=1` so that each process deals with a block of 8x8x1 cells and mpirun must be started with `-np 128` as 16*8*1=128.

5.1.2 Output directory

The output directory is by default `bin/output/` and can be changed in the `parameter::output_directory` variable.

5.1.3 Setup files

Alternative setup files files are located in `physics/plasma/output/initial_conditions/` and can by used by simply copying them to `framework/initial_conditions.cc`, e.g. change to the muphy2src directory and type:

```
cp physics/plasma/setup/initial_conditions/
initial_conditions_landau_damping.cc framework/initial_conditions.cc; make
```

5.2 Restart simulations

By default, restart information is written at each VTK output which can be used to restart a cancelled simulation at this point. This can be done by setting `restart = true;` in `framework/initial_conditions.cc`. Apart from the current state of the simulation, no information is written in the restart output. The rest of the parameters is read from the `initial_conditions.cc` file so that parameters like resolution, number of processes etc. given in this file must be the same as in the original simulation. The cfl number may be adjusted.

The common procedure is to copy the initial conditions file from
`output_directory/log/initial_conditions_cc.txt`
to

`muphy2src/framework/initial_conditions.cc`
and add `restart = true;` (if not present already).

Known issues

- To prevent data loss in case of simulation cancel during output, restart data from the output before is also kept. In the case of corrupted output, this older output must be renamed manually. Some time a more elegant solution should be implemented.

5.3 On the davinci cluster at TPI

In the .bashrc on davinci put for GPU calculations:

```
module purge; module load nvhpc
```

Or instead for CPU calculations:

```
module purge; module load gcc mpi/openmpi3-x86_64
```

Standard machinefiles for davinci exist in `bin/machinefiles/` which define the node names and maximum number of processes on each node; adjust as required. Run e.g. on eight davinci c2 gpus:

```
mpirun -np 8 --machinefiles/machines-c2-gpu ./muphy2
```

Run on davinci c1 and c2 cpus:

```
mpirun -np 128 --machinefiles/machines-all ./muphy2
```

5.4 On JUWELS booster at FZ Jülich

In your .bashrc you may want to put

```
alias sq="squeue -u $USER -o '%A %u %j %t %C %D %R %Q %S %V %M %l' |
column -t"

# GPU computations
module load NVHPC ParaStationMPI

# python
#module load GCC ParaStationMPI/5.5.0-1 SciPy-Stack VTK texlive gnuplot

# CPU computations
#module load GCC ParaStationMPI
# or
#module load Intel ParaStationMPI
```

and comment in/out the `module load` that you need. The `sq` alias lets you monitor your runs by simply typing `sq`. A typical job script using 4 nodes (i.e. 16 MPI processes as each node has 4 GPUs) would look like this:

```
#!/bin/bash -x
#SBATCH --account=multiphysics
#SBATCH --nodes=4
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=4
#SBATCH --output=out/muphy2-stdout.%j
#SBATCH --error=out/muphy2-stdout.%j
#SBATCH --time=24:00:00
#SBATCH --partition=booster
#SBATCH --gres=gpu:4

srun ./muphy2
```

Save this in the `bin/` directory as `job.sh`. Then you can start a job with `sbatch job.sh`. To cancel a job find the job id with `sq` and type `scancel JOBID` where you replace `JOBID` with the number shown by `sq`. The std::out and std::err output of *muphy*II is written to a file in `bin/out/muphy2-stdout.JOBID` where again `JOBID` is replaced by the respective job id number. The `bin/out/` directory needs to be created by the user.

For coupled simulations you may want to utilize the CUDA Multi-Process Service (MPS) and higher numbers of tasks per GPU. An example job script with MPS is:

```
#!/bin/bash -x
#SBATCH --account=multiphysics
#SBATCH --nodes=16
#SBATCH --ntasks=768
#SBATCH --ntasks-per-node=48
#SBATCH --output=out/muphy2-stdout.%j
#SBATCH --error=out/muphy2-stdout.%j
#SBATCH --time=24:00:00
#SBATCH --partition=booster
#SBATCH --gres=gpu:4
#SBATCH --cuda-mps

srun --cpu-bind=cores ./muphy2
```

6 Visualize the output

*muphy*II uses the parallel VTK format for output which can be viewed using e.g. the open-source software ParaView (https://www.paraview.org/). To visualize the output, start ParaView and in ParaView's File→Open dialog choose the output..pvti files in *muphy*II's output directory.

Depending on the settings, Vlasov simulations may also write phase space slices of the

distribution function which can be found in the phase_space subdirectory within the output directory.

Some quantities (total mass, energy, ...) are also written into csv files in the csv subdirectory within the output directory. Those can be visualized with the python script in `muphy2src/physics/plasma/scripts/postproc_csv.py`. Usage:

```
python postproc_csv.py path/to/output/directory/
```

The VTK data can also be loaded into Python numpy arrays which is illustrated in some of the scripts in `muphy2src/physics/plasma/scripts/`.

7 Running a first simulation – concise and complete guide

7.1 Ten-moment fluid simulation of magnetic reconnection

The default initial configuration is the GEM reconnection setup (Birn et al., 2001) modelled by the ten-moment fluid equations with a gradient closure, leading to the results in Allmann-Rahn, Trost and Grauer (2018). Open a command line, change to the `muphy2src/bin/` directory and type:

```
make -j 8
```

Then run with:

```
mpirun -np 16 --oversubscribe ./muphy2
```

Start ParaView and from ParaView open the output*.pvti files in `muphy2src/bin/output/` to visualize the output. Note that only the lower right quarter of the domain is simulated to save computational time. ParaView can mirror the data using the Filter→Reflect functionality, so you can reflect once at X Min and reflect that again at Y Max (however, this does not take antisymmetries into account so is only valid for n, jz, T, ...).

7.2 Vlasov simulation of Landau damping

One-dimensional Landau damping setup, e.g. in Filbet, Sonnendrücker and Bertrand (2001). Open a command line, change to the `muphy2src/bin/` directory and type:

```
cp ../physics/plasma/setup/initial_conditions/
initial_conditions_landau_damping.cc ../framework/initial_conditions.cc
```

to use the Landau damping setup. Then compile

```
make -j 8
```

and run

```
mpirun -np 8 --oversubscribe ./muphy2
```

The phase-space plot of the plasma distribution function can be viewed with ParaView. From ParaView open `bin/output/phase_space/output*.pvti`. Of course you can also have a look at the 1D output of density, velocity etc. in the standard `bin/output/output*.pvti` files. The python script in `physics/plasma/scripts/postproc_csv.py` can visualize the damping of the electric energy (bottom right plot).

8 Coding notes

C++ is used for the program logic, Fortran for calculations. For C++, we loosely follow the Google C++ Style Guide (https://google.github.io/styleguide/cppguide.html). For Fortran, the Fortran90 Best Practices (https://www.fortran90.org/src/best-practices.html) is a helpful resource. Don't forget to put the `_PRC` appendix on Fortran floating point numbers to get the correct precision.

RULE 1: Keep it simple and stupid! The code needs to be understood by researchers and students ranging from coding experts to beginners. Prefer simple over elegant. Be as abstract as necessary, but as explicit as possible.

RULE 2: Never change a running system! Often it is necessary to refactor existing code when new features are implemented. However, improving the elegance or prettiness of existing code is in most cases not a good reason to change it.

Other thoughts

- Fix warnings
- Prefer understandable code and good variable/function names over extensive comments
- Don't keep unused code (not even commented out)
- Spaces, no tabs

9 Git branches

If unsure, devel is the branch to go for. There are two regular branches: devel for the most up-to-date production code and master for robust and well-tested but maybe out-dated code. Feature branches should be merged and/or deleted as soon as possible so that the number of branches is kept at a minimum.

10 General information on implementation and numerics

If you want to know which parameters/options/variables are available and what they mean, you can have a look into `framework/parameter.h`.

10.1 Files

- `bin/`: Contains the Makefile and after compilation the executable and build files.
- `framework/`: Files that are not limited to a special use case (such as plasma simulations), e.g.: main, parameter, block, MPI/boundary, file writing, interpolation, ...
- `physics/*/models/`: Physical models, i.e. their data, and functions implementing the respective numerical solvers.
- `physics/*/schemes/`: Schemes that determine the execution order of the models' solver functions.
- `physics/*/criteria/`: Criteria that determine which scheme is run in which block and functions for allocating schemes.
- `physics/*/converters/`: Functions that convert physical quantities or models into each other, or convert initial conditions to models and models to output.
- `physics/*/output/`: Preparation of output and call of file writing functions.
- `physics/*/setup/`: Initial configurations. In setup.F90, typical setups are defined that can be further specified in the `framework/initial_conditions.cc` file.

10.2 Program layout/structure

Each block corresponds to one MPI process and holds a pointer to the scheme that is present in the block. Each scheme holds pointers to the models that are used in the scheme. The block also holds an MPI class providing boundary exchange functions and a parameter struct that groups physical and numerical parameters. Both the MPI class and the parameter struct are handed on to the schemes and models as references.

In cases of mixed-language programming with Fortran, there is usually a triplet of files with the same name but extensions .cc, .h and .F90, where the .F90 file implements the actual code, the .h file contains definitions of the Fortran functions and of C++-wrappers for these functions and the .c file implements the wrapper functions.

10.3 Grid

In Fortran, we chose the spatial grid (in this example 1D) to go from -BD to (resolution - 1)+BD, where BD is the number of boundary/ghost cells per side and resolution is the number of total cells. Thus, the cell with index 0 is the first actual cell and the one with index (resolution - 1) the last, whereas e.g. two boundary cells on the left have the indices

-2 and -1. The velocity grid does not have boundary cells and thus uses conventional Fortran indexing (the first cell has index 1 and the last cell has index resolution_v).

Therefore, the location x_i of the center of the spatial cell with index i is given by

$$x_i = x_{begin} + (i + 0.5)\Delta x$$

where x_{begin} is the beginning of the simulation domain and Δx is the cell width. On the other hand, for velocity space cell v_j at index j it is

$$v_j = v_{begin} + (j - 0.5)\Delta v.$$

In C++ indexing naturally starts at zero and all arrays are treated as one-dimensional, even if they are multidimensional in the Fortran code.

10.4 Normalization

The implementation is based on SI units and Boltzmann constant is set to one ($k_B = 1$). The actual normalization depends on the initialization. There are two types of normalization which are used most commonly:

10.4.1 Electromagnetic setups

Length in ion inertial length $d_{i,0}$ based on density n_0, velocity in ion Alfvén velocity $v_{A,0}$ based on the magnetic field B_0, time in inverse of the ion cyclotron frequency $\Omega_{i,0}^{-1}$, mass in ion mass m_i, vacuum permeability $\mu_0 = 1$.

10.4.2 Electrostatic setups

Length in electron Debye length $\lambda_{D,e,0}$ based on density n_0, velocity in electron thermal velocity $v_{th,e,0}$, time in inverse of the electron plasma frequency $\omega_{e,0}^{-1}$, mass in electron mass m_e, vacuum permittivity $\epsilon_0 = 1$.

11 Add new source code

Open the Makefile. To add a new C++ source file add

```
file_name.o: ../path/to/file/file_name.cc
  $(MPICXX) -o $@ -c $<
```

where other definitions of this kind are. Replace `file_name` with the file's name and `path/to/file` with the path to the file. For a new Fortran file instead use

```
file_name_f.o: ../path/to/file/file_name.F90 definitions.mod
  $(MPIF90) -o $@ -c $<
```

(note the `_f` suffix and the `definitions.mod`).

In the end add `file_name.o` or `file_name_f.o` to the `OBJS` variable.

12 Setup

Physical setups/initial configurations are implemented in the directory `physics/plasma/setup/` in `setup.F90` with function definitions in `setup.h` and an `init` function called by a scheme in `setup.cc`. These basic initial configurations usually contain parameters that can then be specified in `framework/initial_conditions.cc`.

12.1 Create a new setup

Always use existing code as an orientation. Create a new function describing the setup in `setup.F90` and define it in `setup.h`. In `setup.cc` move to the suitable position, e.g. "five moment setups two species" (five moment meaning that the setup is based on density, velocity and a scalar temperature) and insert the function call in the `if (p.setup == "setup_name")` block. Choose the `"setup_name"` string identical to function name in the Fortran code minus the `setup_` prefix and the underscore suffix. Now the setup will be used when the `parameter::setup` variable is set to this string in `framework/initial_conditions.cc`. Parameters that are specific to the respective setup should be implemented using the `parameter::setup_var` variable which is of type `std::unordered_map`.

13 Models

Files associated with physical models are located in the `physics/plasma/models/` directory. A model consists of a C++ source file, a C++ header file and one or more Fortran source files implementing the solver. The model classes inherit from the base class in `framework/model.h`. Each model has a unique identifier (model ID).

13.1 Create a new model

Create the triplet of .cc, .h and .F90 files, using e.g. the five-moment model files as an orientation. Add the new model ID (must be unique!) to the `model_names` enum in `framework/parameter.h`. Expand the files `physics/plasma/setup/setup.cc` and `physics/plasma/output/output.cc` so that the framework knows how to load initial conditions into the model and how to do output for the model. If necessary add new converters for the model (cf. Sec. 16).

13.1.1 Create a new solver for an existing model

This is straightforward: Create a new .F90 file for the model that contains the new solver functions. Add the function definitions to the model's .h file and adjust the .cc file to use the new solver. As required add a variable to `framework/parameter.h` to switch between the solvers.

13.2 Vlasov

Files: `vlasov.cc`, `vlasov.h`, `vlasov_pfc.F90`, `vlasov_lagrange5.F90`. Identifier: `kVlasov`.

Solves the Vlasov equation:

$$\frac{\partial f_s}{\partial t} + \mathbf{v} \cdot \nabla f_s + \frac{q_s}{m_s}(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \nabla_v f_s = 0. \tag{1}$$

Different solvers can be used by adjusting `parameter::vlasov_solver`. Currently the variable may be set to `"pfc"` or `"lagrange5"`. Positive and flux conservative (PFC) solver (Filbet, Sonnendrücker and Bertrand, 2001) or simple five point Lagrange interpolation (cf. e.g. Kormann, Reuter and Rampp, 2019). The PFC solver uses the backsubstitution method for the velocity updates (Schmitz and Grauer, 2006).

13.3 10 moments

Files: `fluid10.cc`, `fluid10.h`, `fluid_10mom_cweno.F90`. Identifier: `kFluid10`.

Solves the 10-moment fluid equations:

$$\frac{\partial n_s}{\partial t} + \nabla \cdot (n_s \mathbf{u}_s) = 0\,, \tag{2}$$

$$m_s \frac{\partial (n_s \mathbf{u}_s)}{\partial t} = n_s q_s (\mathbf{E} + \mathbf{u}_s \times \mathbf{B}) - \nabla \cdot \mathcal{P}_s\,, \tag{3}$$

$$\frac{\partial \mathcal{P}^s_{ij}}{\partial t} - q_s (n_s u^s_{[i} E_{j]} + \frac{1}{m_s} \epsilon_{[ikl} \mathcal{P}^s_{kj]} B_l) = -(\nabla \cdot \mathcal{Q}_s)_{ij}\,. \tag{4}$$

$\mathcal{P}_s = m_s \int \mathbf{v} \otimes \mathbf{v} f_s \mathrm{d}\mathbf{v}$ and $\mathcal{Q}_s = m_s \int \mathbf{v} \otimes \mathbf{v} \otimes \mathbf{v} f_s \mathrm{d}\mathbf{v}$ are the second and third moment of the distribution function (multiplied by mass), ϵ_{ikl} is the Levi-Civita symbol and the square brackets denote the sum over as many permutations of indices as needed to make the tensors symmetric, for example

$$u_{[i} E_{j]} = u_i E_j + u_j E_i.$$

This set of equations needs a closure approximation for the divergence of the heat flux which is discussed hereafter. Heat flux Q is related to the distribution function's moments according to $\mathrm{Q}_{ijk} = \mathcal{Q}_{ijk} - u_{[i} \mathcal{P}_{jk]} + 2mnu_i u_j u_k$.

The relevant quantities are grouped into one `ten_moments` variable. It is `ten_moments(:,:,:,1)` $= n = \int f_s \mathrm{d}\mathbf{v}$, `ten_moments(:,:,:,2:4)` $= n\,\mathbf{u} = \int \mathbf{v} f_s \mathrm{d}\mathbf{v}$ and `ten_moments(:,:,:,5:10)` $= \mathcal{P}/m = \int \mathbf{v} \otimes \mathbf{v} f_s \mathrm{d}\mathbf{v}$. $\mathcal{P}$ is symmetric and therefore has 6 independent components.

The fluid solver employs the centrally-weighted essentially non-oscillating (CWENO) method (Kurganov and Levy, 2000). The CWENO method features a free parameter `parameter::cweno_epsilon` which controls the artificial dissipation. Low value means less numeric oscillations but more dissipation. Typical choice is `parameter::cweno_epsilon` $= 10^{-5}$. In order to avoid negative temperatures/pressures, an explicit floor is for the diagonal elements of the pressure tensor given by `parameter::cweno_limiter`. This is also used when e.g. dividing by very small n.

13.3.1 10 moment heat flux closures

The heat flux closure can be chosen with the `parameter::ten_moment_closure` variable. Possible closures are `"none"`, `"gradient_P"`, `"gradient_T"`, `"gradient_sym"`, `"isotropization"`. The closures have a free parameter `parameter::ten_moment_closure` that can be adjusted. For details on the closures and the choice of the free parameter see the comments in the respective functions in the `fluid_10mom_cweno.F90` file. Also see Allmann-Rahn, Lautenbach, Grauer and Sydora (2021).

13.4 5 moments

Files: `fluid5.cc`, `fluid5.h`, `fluid_5mom_cweno.F90`. Identifier: `kFluid5`.

Solves the 5-moment fluid equations:

$$\frac{\partial n_s}{\partial t} + \nabla \cdot (n_s \mathbf{u}_s) = 0\,, \tag{5}$$

$$m_s \frac{\partial (n_s \mathbf{u}_s)}{\partial t} = n_s q_s (\mathbf{E} + \mathbf{u}_s \times \mathbf{B}) - \frac{1}{3}\nabla (2\mathcal{E}_s - m_s n_s u_s^2) - \nabla \cdot (m_s n_s \mathbf{u}_s \otimes \mathbf{u}_s)\,, \tag{6}$$

$$\frac{\partial \mathcal{E}_s}{\partial t} + \frac{1}{3}\nabla \cdot (\mathbf{u}_s (5\mathcal{E}_s - m_s n_s u_s^2)) - q_s n_s \mathbf{u}_s \cdot \mathbf{E} = 0. \tag{7}$$

with scalar energy density $\mathcal{E}_s = \frac{m_s}{2}\int v^2 f_s \mathrm{d}\mathbf{v}$.

The relevant quantities are grouped into one `five_moments` variable. It is `five_moments(:,:,:,1)` $= n = \int f_s \mathrm{d}\mathbf{v}$, `five_moments(:,:,:,2:4)` $= n\,\mathbf{u} = \int \mathbf{v} f_s \mathrm{d}\mathbf{v}$ and `five_moments(:,:,:,5)` $= 2\mathcal{E}/m = \int v^2 f_s \mathrm{d}\mathbf{v}$.

Information on the solver is given in the 10 moments section.

13.5 MHD

Files: `mhd.cc`, `mhd.h`, `fluid_mhd_cweno.F90`. Identifier: `kMHD1Temperature`.

Solves the magnetohydrodynamic (MHD) equations.

Known issues

- This model is experimental and not tested.
- No GPU support.

13.6 Maxwell

Files: `maxwell.cc`, `maxwell.h`, `maxwell_fdtd.F90`. Identifier: `kMaxwell`.

Solves Maxwell's equations

$$\nabla \cdot \mathbf{E} = \rho/\epsilon_0, \quad \nabla \cdot \mathbf{B} = 0, \quad \nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}, \quad \nabla \times \mathbf{B} = \mu_0 \mathbf{j} + \mu_0 \epsilon_0 \frac{\partial \mathbf{E}}{\partial t}. \tag{8}$$

The equations are solved by means of the Finite Difference Time Domain (FDTD) method. $\mathbf{E}, \mathbf{B}, \mathbf{j}$ are on a Yee cube, i.e. they are located on the faces or edges of a cell according to:
E_x: y, z center; x face
E_y: x, z center; y face
E_z: x, y center; z face
B_x: y, z face; x center
B_y: x, z face; y center
B_z: x, y face; z center.

A quantity which lives on the face of the cell has a location that is half a cell width smaller than that of a quantity which lives in the center of the cell.

13.7 Poisson

Files: `poisson.cc`, `poisson.h`, `poisson.F90`. Identifier: `kPoisson`.

Solves Poisson's equation

$$\nabla^2 \Phi = -\rho/\epsilon_0, \quad \mathbf{E} = -\nabla\Phi \tag{9}$$

with charge density ρ and electric potential Φ. Poisson solvers can be chosen with the `parameter::poisson_solver` variable. Allowed values are currently `"sor"` for using the successive over-relaxation (SOR) solver or `"gauss_seidel"` for using the Gauss-Seidel solver.

Parallelization is done via additive Schwarz iterations. The number of Schwarz iterations is controlled with `parameter::poisson_schwarz_iterations` and each iteration repeats the solver cycle until the result changes by less than the value in `parameter::poisson_convergence_threshold`.

Known issues

- Conjugate gradients (CG) or multigrid solvers would be desirable.
- Gauss-Seidel is not available on GPU (does not parallelize).

13.8 Ohm's law

Files: `ohm.cc`, `ohm.h`, `ohm.F90`. Identifier: `kOhm`.

Solves the generalized Ohm's law. To be combined with the MHD model.

Known issues

- This model is experimental and not tested.
- No GPU support.

14 Schemes

Files associated with numerical schemes are located in the `physics/plasma/schemes/` directory. A model consists of a C++ source file and a C++ header file. The scheme classes inherit from the base class in `framework/scheme.h`. Each scheme has a unique identifier (scheme ID). The following schemes are available:

- `kVeViM`: Vlasov electrons, Vlasov ions, Maxwell
- `kF10eViM`: 10 moment fluid electrons, Vlasov ions, Maxwell
- `kF10eF10iM`: 10 moment fluid electrons, 10 moment fluid ions, Maxwell
- `kF5eF10iM`: 5 moment fluid electrons, 10 moment fluid ions, Maxwell
- `kF5eF5iM`: 5 moment fluid electrons, 5 moment fluid ions, Maxwell
- `kMHD1TemperatureOhm`: one temperature MHD, Ohm's law
- `kVeViP`: Vlasov electrons, Vlasov ions, Poisson
- `kVeViMdeltaf`: Vlasov electrons, Vlasov ions, Maxwell. With delta f correction of the Vlasov solver.
- `kF10eViMdeltaf`: 10 moment fluid electrons, Vlasov ions, Maxwell. With delta f correction of the Vlasov solver.
- `kVeViPdeltaf`: Vlasov electrons, Vlasov ions, Poisson. With delta f correction of the Vlasov solver.

The electromagnetic schemes are described in Rieke, Trost and Grauer (2015), the Vlasov-Poisson scheme is described in Cheng and Knorr (1976), the dual Vlasov ("deltaf") scheme is described in Allmann-Rahn, Lautenbach and Grauer (2022).

14.1 Create a new scheme

Create the .cc and .h files, using e.g. the VeViP scheme files as an orientation. If the scheme is supposed to be coupled to other schemes, it may be sensible to use the FeFiM scheme files as an orientation. Add the new scheme ID (must be unique!) to the `scheme_names` enum in `framework/parameter.h`. Add the call to the scheme's constructor to the `allocate_scheme` function in the `physics/plasma/criteria/select_scheme.cc` file. If coupling to other schemes is desired, you may want to add it to the `parameter::scheme_hierarchy` variable and adjust other schemes' boundary exchange functions.

15 Criteria

Files in `physics/plasma/criteria/`.

Criteria determine which physical scheme is used in a block. By default, there is no criterion (i.e. `Parameter::criterion == nullptr`) and the scheme set in `Parameter::default_scheme` is used.

A criterion is a class that is initialized in the `framework/initial_conditions.cc` file, see for example in the `physics/plasma/setup/initial_conditions/` directory the initial condition files `initial_conditions_gem.cc` and `initial_conditions_whistler.cc`. The respective header file (e.g. `physics/plasma/criteria/criterion_j.h`) needs to be included in `framework/initial_conditions.cc` in order to use the criterion.

The criterion classes are derived from the `framework/criterion.h` base class.

15.1 Create a new criterion

Always use existing code as an orientation. Copy for example the `criterion_j.{cc,h}` or the `criterion_position.{cc,h}` files into new files that are named like the supposed to be contained class and adjust as desired. Then add the criterion to the if-else branch in the `evaluate_criterion(...)` function in `physics/plasma/criteria/select_scheme.cc`. To use it, the criterion needs to be inialized in the `framework/initial_conditions.cc` as mentioned above.

16 Converters

Files in `physics/plasma/converters/`. Converters are simple Fortran functions with C++-wrappers that serve one of the following purposes:

- Convert between physical quantities, e.g. calculate density from the distribution function.
- Convert data of different models into each other, e.g. ten moment fluid data to five moment fluid data.
- Convert initial conditions to model data.
- Convert model data to output.

Creating a new converter function is straightforward: Simply add a new Fortran function together with the corresponding wrapper functions in the C++-files.

17 Output

File writing is done in the `framework/output_write.cc` and `.h` files and preparation of data for output is done in the files in `physics/plasma/output/`. There is VTK-output at regular times, csv-output for time series of e.g. energy, logging-output where parameters of the run are written, and restart-output. The time of a VTK-output can be read from a comment in the respective .pvtk file. The csv-output can be visualized with the `postproc_csv.py` script.

17.1 Options

In `framework/initial_conditions.cc`, several options for output can be set (default values to be found in `framework/parameter.h`).

- `output_directory`: Path to the directory where the output is written.
- `noutputs_vtk, noutputs_csv`: Total number of vtk/csv outputs throughout the simulation.
- `nconsecutive_outputs`: Number of consecutive outputs at each output time, default is 1. Needed for calculating time derivatives in postprocessing. If set to e.g. 3, then one additional output before and one additional output after the regular output is done. The additional outputs are not considered in `noutputs_vtk`, i.e. `noutputs_vtk=100` and `nconsecutive_outputs=3` results in 300 outputs.
- `heatflux_output, restart_output`: Write heatflux/restart output or not.
- `phase_space_output`: May be `"none", "slice", "full", "quarter"`. Slice means that e.g. for given y, z, v_y, v_z a phase space slice in x, v_x is output. Full means that the complete distribution function is output, which is currently implemented by simply keeping all restart output in a separate directory. Quarter is the same but only for a quarter of the full domain.
- `phase_space_output_direction`: For phase space slice output, 0 for x, v_x slice, 1 for y, v_y slice, 2 for z, v_z slice.
- `phase_space_output_slice_index`: For phase space slice output, set the coordinates of y, z, v_y, v_z if x, v_x plane is output and likewise x, z, v_x, v_z if y, v_y plane is output etc.

17.2 Add new output fields

Adding a new output field is done by extending the file `physics/plasma/output/output.cc`. This description is at the example of a new VTK output field; modifying other output works similarly and should be possible after studying the code. In the `prepare_and_write` function add a new variable to hold your output data. Take the existing code as an orientation and adjust depending on whether your output field has a copy for each species or not. In the `// prepare` section fill your variable with data. Modify the `write_fluid_quantities` function to take your field as an argument and adjust the call to the function accordingly. In `write_fluid_quantities`, add a name for your field to `field_names_species_dependent` or `field_names_other`. Finally add your field to the `fields` array of pointers at the correct position (again use the existing fields as an orientation).

Known issues

- There is no error message, if the output directory cannot be created or connot be written to.

- There is no post-processing script yet to visualize/analyze full distribution function output.

- In consecutive outputs, the restart output is always written, not only at the regular output time. This behaviour should be changed.

18 Boundary exchange and boundary conditions

Files: `framework/mpi_boundary.cc`, `framework/mpi_boundary.h`, `framework/mpi_boundary.F90`.

Boundary exchange is done via the `exchange_field` function which takes as arguments the field, boundary conditions, position of the field within the grid, optionally instructions in which direction the boundary exchange is supposed to be (default all) and the number of components of the field.

In dimensions lower than three, the boundary cells of the unused dimensions are automatically filled with the value from the one actual cell in the direction so that 3D implementations do not necessarily need special treatment of lower dimensionality.

Boundary conditions can be set in the `initial_conditions` file using the `bd_cond_*` variable for velocity (v), electric field (E), magnetic field (B), electric potential (phi), density (n), energy density/temperature (nuu), pressure tensor (P) and heat flux tensor (Q_h). For the ten and five moment data, boundary conditions are automatically derived from the previous quantities. Boundary conditions can also be defined ad hoc before the respective boundary exchange if required. The available boundary conditions are "p" periodic, "q" antiperiodic, "m" mirrored, "s" symmetric, "a" antisymmetric, "z" zero. These identifiers are grouped together in a string to describe boundary conditions in the six directions, x low, x high, y low, y high, z low and z high. A boundary condition could e.g. look like this: "ppmmpa" which means that the boundary conditions are periodic in the low and high x boundaries; symmetric in the low and high y boundaries; periodic in the low z boundary and antisymmetric in the high z boundary. For vector and tonsor quantities they are grouped into one array, e.g. velocity boundary conditions in a fully periodic domain would be `bd_cond_v = {"pppppp","pppppp","pppppp"}`.

Orientation within the cell is defined ad hoc before the boundary exchange and takes a similar form. The electric field on the Yee grid, for example, has the orientation `orientation_E[3] = {"fcc", "cfc", "ccf"}` (compare to Maxwell model section).

Symmetric-like boundary conditions for values on cell faces: For values on the cell faces the value from cell [dimX+1] is excactly at the border so that it has no counterpart in case of the symmetric/mirrored and antisymmetric/antimirrored boundary conditions. Thus it is handled the following way: symmetric: constant extrapolation from the neighbour cell mirrored: linear extrapolation from the two cells to the left antisymmetric/antimirrored: 0.

Zero boundary condition for reflecting/conducting walls: Fields that are zero at a conducting wall are on the cell face. Since the face of the first actual cell is excactly at the boundary, for the zero boundary condition not only the boundary cells are set to zero but also the first actual cell. Reflected particles lead to zero average velocity in the first

actual cell, which is therefore also set to zero when the zero boundary condition is used.

Reflecting walls and j from fluxes: The current density calculated from the CWENO cell fluxes does not get along well with the reflecting wall boundary condition in the current implementation. Therefore, in the cells directly at the reflecting wall the current density is calculated from interpolation of cell averages.

Known issues

- The electromagnetic Vlasov schemes use looping over boundary cells in the first half v-step to avoid one boundary exchange. Therefore the boundary exchange of the electric/magnetic field would need to be done with diagonal neighbours as well, which is not the case. Although in practice this is almost always negligible, it should be fixed.
- Also for the additive Schwarz Poisson solver a diagonal boundary exchange would be necessary for complete correctness.

19 Spatial coupling

Spatial coupling is possible between schemes that are next to each other in the `parameter::scheme_hierarchy` array. Each block is associated with one scheme. The assignment of schemes to blocks can be static or it can adaptively change during runtime, and in both cases is done according to a criterion function (cf. Sec. 15). The conversion between the models in the respective schemes is done using the converter functions in `physics/plasma/converters/converters.F90` (cf. Sec. 16).

19.1 Boundary exchange between different models

Schemes that are capable of being coupled implement the boundary exchange process for example in a `exchange_plasma_solvers(...)` function. There, it is decided (depending on the neighbour) whether boundary exchange is performed and if the boundary data needs to be converted to a different model. Boundary data is always sent in form of the physically more reduced model so that less data needs to be sent. Thus, it is the responsibility of the scheme that is higher in the hierarchy (i.e. the scheme with physically more complete models) to convert its boundary data before sending it to a neighbour that is lower in the hierarchy, and also to convert the data that it receives from the physically less complete models.

19.2 Adaptive/dynamic coupling

Schemes are assigned/updated in the `update_scheme()` function in the `framework/block.cc` file. How often schemes are updated during one simulation run is defined by the `parameter::nupdates_scheme` variable. Each scheme that is capable of being adaptively/dynamically coupled provides functions `calc_alloc_converted_model(...)` and `replace_dealloc_old_models(...)` which are called from the function

`convert_scheme` in the file `physics/plasma/converters/select_scheme.cc`. At each point where schemes are updated, it is first decided on the basis of the criterion which scheme is supposed to be used in each block. If the scheme given by the criterion differs from the currently used scheme in the block, it is updated in the following procedure: (1) The new scheme is created with dummy models. (2) The old scheme converts its models into the models that the new scheme needs and returns them using the `calc_alloc_converted_model(...)` function. (3) The dummy models of the new scheme are replaced by the returned converted models using the `replace_dealloc_old_models(...)` function. (4) The old scheme is deleted and the pointer to the old scheme is set to point to the new scheme.

19.3 Load Balancing

Load balancing is done when `Parameter::nslots_per_node` > 0 and is only useful in coupled simulations. There the idea is to use CUDA MPS in order to run multiple MPI processes on a GPU at the same time. Ideally, the GPU is then used to capacity by one or few Vlasov blocks (i.e. MPI processes) and in the second part of the scheme by many fluid blocks. The schemes are distributed among GPUs by the load balancing in a way that the approximately the same amount of expensive (e.g. Vlasov) schemes is on each GPU so that each GPU has approximately the same work load.

In order to use load balancing, the `Parameter::nslots_per_node` variable is set to the amount of GPUs that each node has (e.g. 4 GPUs per node in JUWELS-booster).

20 Simulation parameter tuning

If you want to know which parameters/options/variables are available and what they mean, look into `muphy2src/framework/parameter.h`.

For optimal performance and modelling at the limit of what is possible, it is often necessary to fine tune numerical settings. Numerical stability is in many cases a much more complex matter than with for example PIC simulations. This section is supposed to give an overview on how to properly choose simulation parameters.

20.1 Position space resolution

The minimum resolution in electromagnetic simulations is typically that the cell size is smaller than electron inertial length $d_e = \sqrt{\frac{m_s}{n_s q_s^2 \mu_0}}$ (when length is normalized over d_i, it is $d_e = \sqrt{m_e}$). Sometimes one needs higher resolutions in order to reduce numerical diffusion or capture certain physical phenomena or just to increase accuracy. Especially instabilities profit from a high resolution.

20.2 Velocity space resolution and extent

In order to fit the initial Maxwellian distribution into velocity space, it is sensible to choose a velocity space extent of $\pm 8 v_{th} = 8\sqrt{\frac{k_B T_s}{m_s}}$ or, if necessary to get a better resolution, $\pm 7 v_{th}$.

With the dual Vlasov solver ("deltaf" schemes) it is possible to use low velocity space resolutions without numerical heating. The necessary resolution depends on the velocity space extent. Experience has shown: Resolution $\geq 16^3$ for ions and $\geq 20^3$ for electrons at mass ratio $m_i/m_e = 25$. A typical choice is $\geq 32^3$. For higher mass ratios, higher resolutions should be used for the electrons.

In dual Vlasov simulations low velocity space resolutions can lead to a crash of the simulation in some cases when the diffusion in the Vlasov solver is too large and the heat flux too inaccurate. In coupled simulations a low velocity space resolution can lead to unsmooth borders and also to unstable simulations.

20.3 Time step

The time step is automatically derived from `Parameter::cfl`. In principle, the fluid Runge-Kutta solver needs CFL<0.3 and the Vlasov solver needs CFL<1 because of boundary exchange.

In practice a CFL between 0.05 and 0.25 is sensible. CFL=0.1 is often a safe choice: In the fluid solver one needs a safety factor and in the Vlasov solver one needs to account for the CFL in the velocity space advection which is not respected by the time step calculation.

When electron subcycling is used, one can choose `Parameter::cfl_fluid_ions` and `Parameter::cfl_vlasov_ions`. Since ions are accelerated less, the ion CFL can usually be higher. A sensible choice would be e.g. `Parameter::cfl_fluid_ions`=0.25 and `Parameter::cfl_vlasov_ions`=0.5.

In the time step calculation it is not respected whether typical time scales such as plasma oscillations and gyrations are resolved. Experience has shown that unless a very high speed of light is chosen, above mentioned CFL choices are usually also sufficient from a physical point of view.

20.4 Gradient closure subcycles

The gradient heat flux closures generally need a time step that is smaller than that of the fluid solver, therefore it is subcycled. A typcial choice is to use 4 subcycles at a resolution comparable to the electron inertial length and then double the subcycles when doubling the resolution. However, there is no clear rule and one has to try out. When the subcycles are not sufficient, the simulation will usually become unstable very fast. In rare cases also crashes later in the simulation can be attributed to too few closure subcycles.

20.5 Initial Maxwell steps per fluid step

Maxwell's equations are subcycled with respect to the plasma solver in order to be able to consistently adapt the time step size during the simulation. More initial Maxwell subcycles increase the computational cost but also allow for a finer adaption of dt. And of course sometimes it is desirable to use many Maxwell subcycles for better physical resolution. A typcial choice would be something between 7 and 21 initial subcycles.

20.6 CWENO epsilon

The `Parameter::cweno_epsilon` controls the influence of the limiter in the CWENO fluid solver. Small epsilon (e.g. 1e-5) leads to smooth results, but high diffusion. High epsilon (e.g. 1) leads to less diffusion and often more accurate results, but a less stable simulation.

When simulation instabilities and turbulence, often a high epsilon is advisable. In reconnection often a low epsilon is advisable. Generally, at high resolutions with respect to electron intertial length it is fine to use a small epsilon. At low resolutions one should consider using a higher epsilon.

Small epsilon at low resolution can sometimes lead to problems with the fluid solver at reflecting walls when `Parameter::cweno_calc_j_from_flux == true` (see also next section). In hybrid simulations of Harris sheet reconnection it is sometimes sensible to use a high epsilon for the fluid electrons for accuracy and a low epsilon for the dual Vlasov ion scheme for stability. This is possible with setting `Parameter::cweno_epsilon_deltaf` (do not use this with coupled simulations).

20.7 j from flux

With `Parameter::cweno_calc_j_from_flux == true` the current density is calculated from the CWENO cell interface fluxes which leads to a good representation of Gauss's law. If `false`, the current density is calculated from the cell averages which is the much more stable option, but can violate Gauss's law heavily. For the pure Vlasov schemes, the latter is the only option.

If possible, always use `Parameter::cweno_calc_j_from_flux = true`! There can be the following problems: Bad accuracy of the current density for low resolutions combined with low CWENO epsilon; more difficult reflecting walls; unstable with electron subcycling and low resolutions; unstable with `five_ten_moment_coupling_fit == true`.

20.8 Coupling fit

In coupled simulations there is the option to fit the models at the coupling borders with `Parameter::ten_moment_vlasov_coupling_fit = true` and `Parameter::five_ten_moment_coupling_fit = true`. Both lead to smoother coupling at the respective boundaries, but can also lead to unstable simulations in some cases. In particular the `five_ten_moment_coupling_fit = true` option seems to be problematic when `cweno_calc_j_from_flux == true`, (possibly because the Riemann fan computation is different for the five and ten moment cases?). Thus, `five_ten_moment_coupling_fit` is set to `false` by default.

20.9 Electron subcycling

The `Parameter::electron_subcycling = true` option is usually combined with the ion CFL options described in the *Time step* section above. So far it seems that electron subcycling does not lead to notably reduced physical accuracy. It

can lead to unstable simulations in particular when the resolution is low and `Parameter::cweno_calc_j_from_flux = true`.

References

Allmann-Rahn, F., S. Lautenbach and R. Grauer (2022). "An Energy Conserving Vlasov Solver That Tolerates Coarse Velocity Space Resolutions: Simulation of MMS Reconnection Events". In: *Journal of Geophysical Research: Space Physics* 127.2. e2021JA029976 2021JA029976, e2021JA029976.

Allmann-Rahn, F., S. Lautenbach, R. Grauer and R. D. Sydora (2021). "Fluid simulations of three-dimensional reconnection that capture the lower-hybrid drift instability". In: *Journal of Plasma Physics* 87.1, p. 905870115.

Allmann-Rahn, F., T. Trost and R. Grauer (2018). "Temperature gradient driven heat flux closure in fluid simulations of collisionless reconnection". In: *Journal of Plasma Physics* 84.3, p. 905840307.

Birn, J., J. F. Drake, M. A. Shay, B. N. Rogers, R. E. Denton, M. Hesse, M. Kuznetsova, Z. W. Ma, A. Bhattacharjee, A. Otto and P. L. Pritchett (2001). "Geospace Environmental Modeling (GEM) Magnetic Reconnection Challenge". In: *Journal of Geophysical Research: Space Physics* 106.A3, pp. 3715–3719.

Cheng, C. and G. Knorr (1976). "The integration of the vlasov equation in configuration space". In: *Journal of Computational Physics* 22.3, pp. 330–351.

Filbet, F., E. Sonnendrücker and P. Bertrand (2001). "Conservative Numerical Schemes for the Vlasov Equation". In: *Journal of Computational Physics* 172.1, pp. 166–187.

Kormann, K., K. Reuter and M. Rampp (2019). "A massively parallel semi-Lagrangian solver for the six-dimensional Vlasov–Poisson equation". In: *The International Journal of High Performance Computing Applications* 33.5, pp. 924–947. eprint: `https://doi.org/10.1177/1094342019834644`.

Kurganov, A. and D. Levy (2000). "A Third-Order Semidiscrete Central Scheme for Conservation Laws and Convection-Diffusion Equations". In: *SIAM Journal on Scientific Computing* 22.4, pp. 1461–1488.

Rieke, M., T. Trost and R. Grauer (2015). "Coupled Vlasov and two-fluid codes on GPUs". In: *Journal of Computational Physics* 283, pp. 436–452.

Schmitz, H. and R. Grauer (2006). "Kinetic Vlasov simulations of collisionless magnetic reconnection". In: *Physics of Plasmas* 13.9, 092309, p. 092309.