Mirheo documentation

Release v1.6.0

Dmitry Alexeev **Lucas Amoudruz** **Ivica Kicic**

Jan 13, 2022

Mesoscale flow solver for biological and medical applications.

Mirheo [alexeev2020] is designed as a classical molecular dynamics code adapted for inclusion of large (consisting of thousands of particles) rigid bodies and cells. The main features of the code include:

- fluids represented as free particles interacting with different pairwise potentials (i.e. DPD or Lennard-Jones),
- static walls of arbitrary complexity and size,
- rigid bodies with arbitrary shapes and sizes [amoudruz2021],
- viscoelastic cell membranes [economides2021], that can separate inner from outer fluids

The multi-process GPU implementation enables very fast time-to-solution without compromising physical complexity and Python front-end ensures fast and easy simulation setup. Some benchmarks are listed in *Benchmarks*.

The following documentation is aimed at providing users a comprehensive simulation guide as well as exposing code internals for the developers wishing to contribute to the project.

1 Overview

This section describes the Mirheo interface and introduces the reader to installing and running the code.

The Mirheo code is designed as a classical molecular dynamics code adapted for inclusion of rigid bodies and cells. The simulation consists of multiple time-steps during which the particles and bodies will be displaces following laws of mechanics and hydrodynamics. One time-step roughly consists of the following steps:

- compute all the forces in the system, which are mostly pairwise forces between different particles,
- move the particles by integrating the equations of motions,
- bounce particles off the wall surfaces so that they cannot penetrate the wall even in case of soft-core interactions,
- bounce particles off the bodies (i.e. rigid bodies and elastic membranes),
- perform additional operations dictated by plug-ins (modifications, statistics, data dumps, etc.).

1.1 Python interface

The code uses Python scripting language for the simulation setup. The script defines simulation domain, number of MPI ranks to run; data containers, namely *Particle Vectors* and data handlers: *Initial conditions*, *Integrators*, *Interactions*, *Walls*, *Object bouncers*, *Object belonging checkers* and *Plugins*.

The setup script usually starts with importing the module, e.g.:

```
import mirheo as mir
```

Warning: Loading the module will set the `sys.exepthook` to invoke `MPI_Abort`. Otherwise single failed MPI process will not trigger shutdown, and a deadlock will happen.

The coordinator class, *Mirheo*, and several submodules will be available after that:

- ***ParticleVectors*.** Consists of classes that store the collections of particles or objects like rigid bodies or cell membranes. The handlers from the other submodules usually work with one or several *ParticleVector*. Typically classes of this submodule define liquid, cell membranes, rigid objects in the flow, etc.
- ***InitialConditions*.** Provides various ways of creating initial distributions of particles or objects of a *ParticleVector*.
- ***BelongingCheckers*.** Provides a way to create a new *ParticleVector* by splitting an existing one. The split is based on a given *ObjectVector*: all the particles that were inside the objects will form one *ParticleVector*, all the outer particles – the other *ParticleVector*. Removing inner or outer particles is also possible. Typically, that checker will be used to remove particles of fluid from within the suspended bodies, or to create a *ParticleVector* describing cytoplasm of cells. See also *applyObjectBelongingChecker*.
- ***Interactions*.** Various interactions that govern forces between particles. Pairwise force-fields (DPD, Lennard-Jones) and membrane forces are available.
- ***Integrators*.** Various integrators used to advance particles' coordinates and velocities.
- ***Walls*.** Provides ways to create various static obstacles in the flow, like a sphere, pipe, cylinder, etc. See also *makeFrozenWallParticles*
- ***Bouncers*.** Provides ways to ensure that fluid particles don't penetrate inside of objects (or the particles from inside of membranes don't leak out of them).
- ***Plugins*.** Some classes from this submodule may influence simulation in one way or another, e.g. adding extra forces, adding or removing particles, and so on. Other classes are used to write simulation data, like particle trajectories, averaged flow-fields, object coordinates, etc.

A simple script may look this way:

```
#!/usr/bin/env python
import mirheo as mir

# Simulation time-step
dt = 0.001

# 1 simulation task
ranks  = (1, 1, 1)

# Domain setup
domain = (8, 16, 30)

# Applied extra force for periodic poiseuille flow
f = 1

# Create the coordinator, this should precede any other setup from mirheo package
u = mir.Mirheo(ranks, domain, debug_level=2, log_filename='log')

pv = mir.ParticleVectors.ParticleVector('pv', mass = 1)    # Create a simple particle␣
↪vector
```

(continues on next page)

(continued from previous page)

```
ic = mir.InitialConditions.Uniform(number_density=8)      # Specify uniform random␣
↪initial conditions
u.registerParticleVector(pv=pv, ic=ic)                    # Register the PV and␣
↪initialize its particles

# Create and register DPD interaction with specific parameters
dpd = mir.Interactions.Pairwise('dpd', rc=1.0, kind="DPD", a=10.0, gamma=10.0, kBT=1.
↪0, power=0.5)
u.registerInteraction(dpd)

# Tell the simulation that the particles of pv interact with dpd interaction
u.setInteraction(dpd, pv, pv)

# Create and register Velocity-Verlet integrator with extra force
vv = mir.Integrators.VelocityVerlet_withPeriodicForce('vv', force=f, direction='x')
u.registerIntegrator(vv)

# This integrator will be used to advance pv particles
u.setIntegrator(vv, pv)

# Set the dumping parameters
sample_every = 2
dump_every   = 1000
bin_size     = (1., 1., 1.)

# Write some simulation statistics on the screen
u.registerPlugins(mir.Plugins.createStats('stats', every=500))

# Create and register XDMF plugin
u.registerPlugins(mir.Plugins.createDumpAverage('field', [pv], sample_every, dump_
↪every, bin_size, ["velocities"], 'h5/solvent-'))

# Run 5002 time-steps
u.run(5002, dt=dt)
```

1.2 Running the simulation

Mirheo is intended to be executed within MPI environments, e.g.:

```
mpirun -np 2 python3 script.py
```

The code employs simple domain decomposition strategy (see *Mirheo*) with the work mapping fixed in the beginning of the simulation.

Warning: When the simulation is started, every subdomain will have 2 MPI tasks working on it. One of the tasks, referred to as *compute task* does the simulation itself, another one (*postprocessing task*) is used for asynchronous data-dumps and postprocessing.

Note: Recommended strategy is to place two tasks per single compute node with one GPU or 2 tasks per one GPU in multi-GPU configuration. The postprocessing tasks will not use any GPU calls, so you may not need multiprocess GPU mode or MPS.

Note: If the code is started with number of tasks exactly equal to the number specified in the script, the postprocessing will be disabled. All the plugins that use the postprocessing will not work (all the plugins that write anything, for example). This execution mode is mainly aimed at debugging.

The running code will produce several log files (one per MPI task): see *Mirheo*. Most errors in the simulation setup (like setting a negative particle mass) will be reported to the log. In case the code finishes unexpectedly, the user is advised to take a look at the log.

2 Installation

2.1 Mirheo

Mirheo requires at least Kepler-generation NVIDIA GPU and depends on a few external tools and libraries:

- Unix-based OS
- NVIDIA CUDA toolkit version >= 9.2
- gcc compiler with c++14 support compatible with CUDA installation
- CMake version >= 3.8
- Python interpreter version >= 3.4
- MPI library
- HDF5 parallel library
- libbfd for pretty debug information in case of an error

Note: The code has been tested with `mpich-3.2.1`, `mpich-3.3.1` and `openmpi-3.1.3`. A known bug in `mpich-3.3` causes Mirheo to deadlock, use another version instead.

With all the prerequisites installed, you can take the following steps to run Mirheo:

1. Get the up-to-date version of the code:

```
$ git clone --recursive https://github.com/cselab/Mirheo.git mirheo
```

2. In most cases automatic installation will work correctly, you should try it in the first place. Navigate to the folder with the code and run the installation command:

```
$ cd mirheo
$ make install
```

   In case of any issues, check the prerequisites or try a more "manual" way:

   1. From the mirheo folder, create a build folder and run CMake:

```
$ mkdir -p build/
$ cd build
$ cmake ../
```

      If CMake reports some packages are not found, make sure you have all the prerequisites installed and corresponding modules loaded. If that doesn't help, or you have some packages installed in non-default locations, you will need to manually point CMake to the correct locations.

See CMake documentation for more details on how to provide package installation files.

Note: On CRAY systems you may need to tell CMake to dynamically link the libraries by the following flag:

```
$ cmake -DCMAKE_EXE_LINKER_FLAGS="-dynamic" ../
```

Note: Usually CMake will correctly determine compute capability of your GPU. However, if compiling on a machine without a GPU (for example on a login node of a cluster), you may manually specify the compute capability (use your version instead of 6.0):

```
$ cmake -DMIR_CUDA_ARCH_NAME=6.0 ../
```

Note that in case you don't specify any capability, Mirheo will be compiled for all supported architectures, which increases compilation time and slightly increases application startup. Performance, however, should not be affected.

---

2. Now you can compile the code:

   ```
   $ make -j <number_of_jobs>
   ```

   The library will be generated in the current build folder.

3. A simple way to use Mirheo after compilation is to install it with pip. Navigate to the root folder of Mirheo and run the following command:

   ```
   $ pip install --user --upgrade .
   ```

3. Now you should be able to use the Mirheo in your Python scripts:

   ```
   import mirheo
   ```

2.2 Compile Options

Additional compile options are provided through `cmake`:

- `MIR_MEMBRANE_DOUBLE:BOOL=OFF`: Computes membrane forces (see *MembraneForces*) in double precision if set to `ON`; default: single precision
- `MIR_ROD_DOUBLE:BOOL=OFF`: Computes rod forces (see *RodForces*) in double precision if set to `ON`; default: single precision
- `MIR_DOUBLE_PRECISION:BOOL=OFF`: Use double precision everywhere if set to `ON` (including membrane forces and rod forces); default: single precision
- `MIR_USE_NVTX:BOOL=OFF`: Add NVIDIA Tools Extension (NVTX) trace support for more profiling informations if set to `ON`; default: no NVTX

Note: Compile options can be passed by using the `-D` prefix:

```
cmake -DMIR_DOUBLE_PRECISION=ON
```

When using the *Tools*, the compile options can be passed using the `CMAKE_FLAGS` variable:

```
CMAKE_FLAGS="-DMIR_DOUBLE_PRECISION=ON" mir.make
```

Note: The compile options of the current installation can be viewd by typing in a terminal:

```
python -m mirheo compile_opt all
```

2.3 Tools

Additional helper tools can be installed for convenience and are required for testing the code.

Configuration

The tools will automatically load modules for installing and running the code. Furthermore, CMake options can be saved in those wrapper tools for convenience. The list of modules and cmake flags can be customised by adding corresponding files in `tools/config` (see available examples). The `__default` files can be modified accordingly to your system.

Installation

The tools can be installed by typing:

```
$ cd tools/
$ ./configure
$ make install
```

Note: By default, the tools are installed in your `$HOME/bin` directory. It is possible to choose another location by setting the `--bin-prefix` option:

```
$ ./configure --bin-prefix <my-custom-tools-location>
```

Note: In order to run on a cluster with a job scheduler (e.g. slurm), the `--exec-cmd` option should be set to the right command (e.g. `srun`):

```
$ ./configure --exec-cmd <my-custom-command>
```

The default value is `mpiexec`

After installation, it is advised to test the tools by invoking

```
$ make test
```

The above command requires the atest framework (see *Testing*).

Tools description

mir.load

This tool is not executable but need to be sourced instead. This simply contains the list of of possible modules required by Mirheo. `mir.load.post` is similar and contains modules required only for postprocessing as it migh conflict with `mir.load`.

mir.make

Wrapper used to compile Mirheo. It calls the `make` command and additionally loads the correct modules and pass optional CMake flags. The arguments are the same as the `make` command. The compilation options shown previously, or any cmake flag, can be passed through the `CMAKE_FLAGS` variable, e.g.:

```
$ CMAKE_FLAGS="-DUSE_NVTX=ON"  mir.make
```

mir.run

Wrapper used to run Mirheo. It runs a given command after loading the correct modules. Internally calls the `--exec-cmd` passed during the configuation. Additionally, the user can execute profiling or debugging tools (see `mir.run --help` for more information). The parameters for the exec-cmd can be passed through the `--runargs` option, e.g.

```
$ mir.run --runargs "-n 2" echo "Hello!"
Hello!
Hello!
```

Alternatively, these arguments can be passed through the environment variable `MIR_RUNARGS`:

```
$ MIR_RUNARGS="-n 2" mir.run echo "Hello!"
Hello!
Hello!
```

The latter use is very useful when passing a common run option to all tests for example.

mir.post

Wrapper used to run postprocess tools. This is different from `mir.run` as it does not execute in parallel and can load a different set of modules (see `mir.load.post`)

mir.avgh5

a simple postprocessing tool used in many tests. It allows to average a grid field contained in one or multiple h5 files along given directions. See more detailed documentation in

```
$ mir.avgh5 --help
```

mir.restart.id

Convenience tool to manipulate the restart ID from multiple restart files. See more detailed documentation in

```
$ mir.restart.id --help
```

3 Testing

Mirheo can be tested with a set of regression tests (located in `tests`) and unit tests (located in `units`).

3.1 Regression tests

Regression testing makes use of the atest framework. This can be installed as follows:

```
$ git clone https://gitlab.ethz.ch/mavt-cse/atest.git
$ cd atest
$ make bin
```

Note: By default, this will install the atest executables in `$HOME/bin` folder. This location should be in your `PATH` variable

---

The regression tests are a set of python scripts. They make use of additional dependencies:

- numpy
- trimesh
- mpi4py

Which can all be installed via `pip`. All tests can be run by typing:

```
$ cd tests
$ make test
```

Note: You need to install the tools before running the tests

3.2 Units tests

Unit tests are compiled together with the google-test framework. The unit tests are compiled by adding the option `-DBUILD_TESTS=ON` to cmake (see *Installation*). The binaries are placed in the `build` folder.

```
$ mir.make units
$ cd build
$ mir.make test
```

Note: You need to install the tools before running the unit tests

---

3.3 Double precision

If compiled with `DOUBLE_PRECISION=ON` (see *Installation*), the reference files for the regression tests are inside the `tests/test_data_double` folder. The tests can be run by typing:

```
$ cd tests
$ make test_double
```

4 Tutorials

This section will guide you in the **Mirheo** interface step by step with examples.

4.1 Hello World: run Mirheo

We start with a very minimal script running **Mirheo**.

Listing 1: *hello.py*

```
#!/usr/bin/env python

# first we need to import the module
import mirheo as mir

dt = 0.001                  # simulation time step
ranks  = (1, 1, 1)          # number of ranks in x, y, z directions
domain = (32.0, 16.0, 16.0) # domain size in x, y, z directions

# create the coordinator
u = mir.Mirheo(ranks, domain, debug_level=3, log_filename='log')

u.run(100, dt=dt) # Run 100 time-steps
```

The time step of the simulation and the domain size are common to all objects in the simulation, hence it has to be passed to the coordinator (see its *constructor*). We do not add anything more before running the simulation (last line).

Note: We also specified the number of ranks in **each** direction. Together with the domain size, this tells **Mirheo** how the simulation domain will be split across MPI ranks. The number of simulation tasks must correspond to this variable.

The above script can be run as:

```
mpirun -np 1 python3 hello.py
```

Running *hello.py* will only print the "hello world" message of **Mirheo**, which consists of the version and git SHA1 of the code. Furthermore, **Mirheo** will dump log files (one per MPI rank) which name is specified when creating the coordinator. Depending on the `debug_level` variable, the log files will provide information on the simulation progress.

4.2 DPD solvent at rest

We will now run a simulation of particles in a periodic box interacting with *Pairwise* forces of type "DPD". We use a *VelocityVerlet* integrator to advance particles in time. The initial conditions are *Uniform* randomly placed particles in the domain with a given density.

Listing 2: *rest.py*

```
#!/usr/bin/env python

import mirheo as mir

dt = 0.001
rc = 1.0      # cutoff radius
number_density = 8.0

ranks  = (1, 1, 1)
domain = (16.0, 16.0, 16.0)

u = mir.Mirheo(ranks, domain, debug_level=3, log_filename='log')

pv = mir.ParticleVectors.ParticleVector('pv', mass = 1.0) # Create a simple Particle␣
↪Vector (PV) named 'pv'
ic = mir.InitialConditions.Uniform(number_density)        # Specify uniform random␣
↪initial conditions
u.registerParticleVector(pv, ic)                          # Register the PV and␣
↪initialize its particles

# Create and register DPD interaction with specific parameters and cutoff radius
dpd = mir.Interactions.Pairwise('dpd', rc, kind="DPD", a=10.0, gamma=10.0, kBT=1.0,␣
↪power=0.5)
u.registerInteraction(dpd)

# Tell the simulation that the particles of pv interact with dpd interaction
u.setInteraction(dpd, pv, pv)

# Create and register Velocity-Verlet integrator
vv = mir.Integrators.VelocityVerlet('vv')
u.registerIntegrator(vv)

# This integrator will be used to advance pv particles
u.setIntegrator(vv, pv)

# Write some simulation statistics on the screen every 500 time steps
u.registerPlugins(mir.Plugins.createStats('stats', every=500))

# Dump particle data
dump_every = 500
u.registerPlugins(mir.Plugins.createDumpParticles('part_dump', pv, dump_every, [],
↪'h5/solvent_particles-'))

u.run(5002, dt=dt)
```

This example demonstrates how to build a simulation:

1. **Create** the *coordinator*
2. **Create** the simulation objects (particle vectors, initial conditions...)

3. **Register** the above objects into the *coordinator* (see `register*` functions)
4. **link** the registered objects together in the *coordinator* (see `set*` functions)

The above script can be run as:

```
mpirun -np 2 python3 rest.py
```

Note: The *rest.py* script contains plugins of type `Stats` and `ParticleDumper`, which needs a **postprocess** rank additionally to the **simulation** rank in order to be active. The simulation is then launched with 2 ranks.

The execution should output the *stats.txt* file as well as information output in the console. Additionally, the particle positions and velocities are dumped in the `h5` folder.

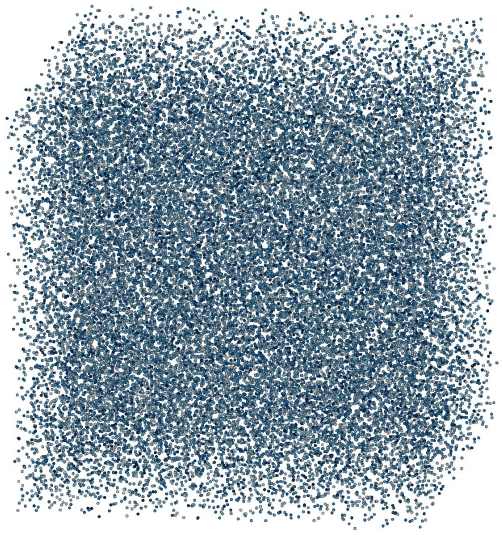


Fig. 1: Snapshot of the particles dumped by executing the *rest.py* script. Visualization made in visit.

4.3 Adding Walls

We extend the previous example by introducing *Walls* in the simulation. Two components are required to form walls:

- a geometry representation of the wall surface. In **Mirheo**, wall surfaces are represented as zero level set of a Signed Distance Function (SDF). This is used to decide which particles are kept at the beginning of the simulation, but also to prevent penetrability of the walls by solvent particles.
- frozen particles, a layer of particles outside of the wall geometry which interact with the inside particles to prevent density fluctuations in the vicinity of the walls.

Note: The user has to set the interactions with the frozen particles explicitly

---

Listing 3: *walls.py*

```
#!/usr/bin/env python

import mirheo as mir

rc = 1.0      # cutoff radius
number_density = 8.0
dt = 0.001
ranks  = (1, 1, 1)
domain = (16.0, 16.0, 16.0)

u = mir.Mirheo(ranks, domain, debug_level=3, log_filename='log')

pv  = mir.ParticleVectors.ParticleVector('pv', mass = 1.0)
ic  = mir.InitialConditions.Uniform(number_density)
dpd = mir.Interactions.Pairwise('dpd', rc, kind="DPD", a=10.0, gamma=10.0, kBT=1.0,␣
↪power=0.5)
vv  = mir.Integrators.VelocityVerlet('vv')

u.registerInteraction(dpd)
u.registerParticleVector(pv, ic)
u.registerIntegrator(vv)

# creation of the walls
# we create a cylindrical pipe passing through the center of the domain along
center = (domain[1]*0.5, domain[2]*0.5) # center in the (yz) plane
radius = 0.5 * domain[1] - rc           # radius needs to be smaller than half of the␣
↪domain
                                        # because of the frozen particles

wall = mir.Walls.Cylinder("cylinder", center=center, radius=radius, axis="x",␣
↪inside=True)

u.registerWall(wall) # register the wall in the coordinator

# we now create the frozen particles of the walls
# the following command is running a simulation of a solvent with given density␣
↪equilibrating with dpd interactions and vv integrator
# for 1000 steps.
# It then selects the frozen particles according to the walls geometry, register and␣
↪returns the newly created particle vector.
pv_frozen = u.makeFrozenWallParticles(pvName="wall", walls=[wall], interactions=[dpd],
↪ integrator=vv, number_density=number_density, dt=dt)

# set the wall for pv
# this is required for non-penetrability of the solvent thanks to bounce-back
# this will also remove the initial particles which are not inside the wall geometry
u.setWall(wall, pv)

# now the pv also interacts with the frozen particles!
u.setInteraction(dpd, pv, pv)
u.setInteraction(dpd, pv, pv_frozen)

# pv_frozen do not move, only pv needs an integrator in this case
u.setIntegrator(vv, pv)

u.registerPlugins(mir.Plugins.createStats('stats', every=500))
```

(continues on next page)

(continued from previous page)

```
dump_every = 500
u.registerPlugins(mir.Plugins.createDumpParticles('part_dump', pv, dump_every, [],
↪'h5/solvent_particles-'))

# we can also dump the frozen particles for visualization purpose
u.registerPlugins(mir.Plugins.createDumpParticles('part_dump_wall', pv_frozen, dump_
↪every, [], 'h5/wall_particles-'))

# we can dump the wall sdf in xdmf + h5 format for visualization purpose
# the dumped file is a grid with spacings h containing the SDF values
u.dumpWalls2XDMF([wall], h = (0.5, 0.5, 0.5), filename = 'h5/wall')

u.run(5002, dt=dt)
```

Note: A *ParticleVector* returned by *makeFrozenWallParticles* is automatically registered in the coordinator. There is therefore no need to provide any `InitialConditions` object.

This example demonstrates how to construct walls:

1. **Create** *Walls* representation
2. **Create** *Interactions* and an *Integrator* to equilibrate frozen particles
3. **Create** the frozen particles with *mmirheo.Mirheo.makeFrozenWallParticles()*
4. **Set** walls to given PVs with *mmirheo.Mirheo.setWall()*
5. **Set** interactions with the frozen particles as normal PVs

The execution of *walls.py* should output the *stats.txt* file as well as information output in the console. Additionally, frozen and solvent particles, as well as the walls SDF are dumped in the `h5` folder.

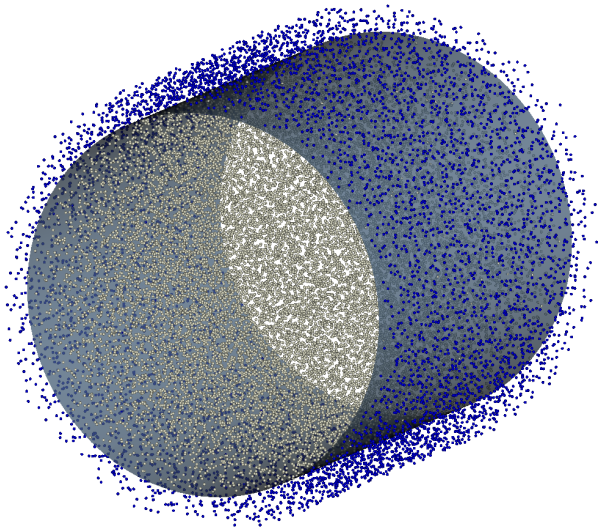


Fig. 2: Snapshot of the data dumped by executing the *walls.py* script. The white particles represent the solvent, the blue particles are the frozen wall particles and the surface is the 0 level set of the SDF file.

4.4 Membranes

Membranes are a set of particles connected into a triangle mesh. They can interact as normal *PVs* but have additional *internal* interactions, which we will use in this example. Here we simulate one membrane with a given initial mesh "rbc_mesh.py" which can be taken from the `data/` folder of the repository. The membrane is subjected to shear, bending, viscous and constraint forces and evolves over time thanks to a *VelocityVerlet* integrator.

Listing 4: *membrane.py*

```
#!/usr/bin/env python

import mirheo as mir

dt = 0.001

ranks  = (1, 1, 1)
domain = (12, 12, 12)

u = mir.Mirheo(ranks, domain, debug_level=3, log_filename='log')

# we need to first create a mesh before initializing the membrane vector
mesh_rbc = mir.ParticleVectors.MembraneMesh("rbc_mesh.off")

# create a MembraneVector with the given mesh
pv_rbc   = mir.ParticleVectors.MembraneVector("rbc", mass=1.0, mesh=mesh_rbc)
# place initial membrane
# we need a position pos and an orientation described by a quaternion q
# here we create only one membrane at the center of the domain
pos_q = [0.5*domain[0], 0.5*domain[1], 0.5*domain[2], # position
         1.0, 0.0, 0.0, 0.0]                          # quaternion
ic_rbc   = mir.InitialConditions.Membrane([pos_q])
u.registerParticleVector(pv_rbc, ic_rbc)

# next we store the parameters in a dictionary
prms_rbc = {
    "x0"     : 0.457,
    "ka_tot" : 4900.0,
    "kv_tot" : 7500.0,
    "ka"     : 5000,
    "ks"     : 0.0444 / 0.000906667,
    "mpow"   : 2.0,
    "gammaC" : 52.0,
    "kBT"    : 0.0,
    "tot_area"   : 62.2242,
    "tot_volume" : 26.6649,
    "kb"     : 44.4444,
    "theta"  : 6.97
}

# now we create the internal interaction
# here we take the WLC model for shear forces and Kantor model for bending forces.
# the parameters are passed in a kwargs style
int_rbc = mir.Interactions.MembraneForces("int_rbc", "wlc", "Kantor", **prms_rbc)

# then we proceed as usual to make th membrane particles evolve in time
vv = mir.Integrators.VelocityVerlet('vv')
u.registerIntegrator(vv)
```

(continues on next page)

(continued from previous page)

```
u.setIntegrator(vv, pv_rbc)
u.registerInteraction(int_rbc)
u.setInteraction(int_rbc, pv_rbc, pv_rbc)

# dump the mesh every 50 steps in ply format to the folder 'ply/'
u.registerPlugins(mir.Plugins.createDumpMesh("mesh_dump", pv_rbc, 50, "ply/"))

u.run(5002, dt=dt)
```

Note: The interactions handle different combinations of shear and bending models. Each model may require different parameters. Refer to *mmirheo.Interactions.MembraneForces()* for more information on the models and their corresponding parameters.

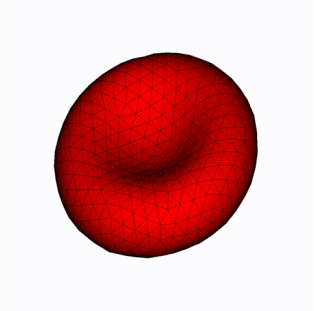


Fig. 3: Sequence data dumped by executing the *membrane.py* script.

4.5 Creating Cells with Different inner and outer liquids

It is easy to extend the above simple examples into quite complicated simulation setups. In this example we simulate a suspension of a few membranes inside a solvent. We also show here how to split inside from outside solvents into 2 *ParticleVectors*. This is useful when the 2 solvents do not have the same properties (such as viscosity). The example also demonstrates how to avoid penetration of the solvents through the membranes thanks to *mmirheo.Bouncers*.

Note that in this example, we also show that it is easy to add many different interactions between given particle vectors.

Listing 5: *membranes_solvents.py*

```
#!/usr/bin/env python

import mirheo as mir

dt = 0.001
rc = 1.0
number_density = 8.0

ranks  = (1, 1, 1)
domain = (16.0, 16.0, 16.0)

u = mir.Mirheo(ranks, domain, debug_level=3, log_filename='log')

# create the particle vectors
```

(continues on next page)

(continued from previous page)

```
##############################

# create MembraneVector for membranes
mesh_rbc = mir.ParticleVectors.MembraneMesh("rbc_mesh.off")

pv_rbc = mir.ParticleVectors.MembraneVector("rbc", mass=1.0, mesh=mesh_rbc)
pos_qs = [[ 5.0,  5.0, 2.0,     1.0, 0.0, 0.0, 0.0], # we create 4 membranes
          [11.0,  5.0, 6.0,     1.0, 0.0, 0.0, 0.0],
          [5.0,  11.0, 10.0,    1.0, 0.0, 0.0, 0.0],
          [11.0, 11.0, 14.0,    1.0, 0.0, 0.0, 0.0]]
ic_rbc   = mir.InitialConditions.Membrane(pos_qs)
u.registerParticleVector(pv_rbc, ic_rbc)

# create particleVector for outer solvent
pv_outer = mir.ParticleVectors.ParticleVector('pv_outer', mass = 1.0)
ic_outer = mir.InitialConditions.Uniform(number_density)
u.registerParticleVector(pv_outer, ic_outer)

# To create the inner solvent, we split the outer solvent (which originally occupies
# the whole domain) into outer and inner solvent
# This is done thanks to the belonging checker:
inner_checker = mir.BelongingCheckers.Mesh("inner_solvent_checker")

# the checker needs to be registered, as any other object; it is associated to a␣
↪given object vector
u.registerObjectBelongingChecker(inner_checker, pv_rbc)

# we can now apply the checker to create the inner PV
pv_inner = u.applyObjectBelongingChecker(inner_checker, pv_outer, correct_every = 0,␣
↪inside = "pv_inner")


# interactions
###############

prms_rbc = {
    "x0"     : 0.457,
    "ka_tot" : 4900.0,
    "kv_tot" : 7500.0,
    "ka"     : 5000,
    "ks"     : 0.0444 / 0.000906667,
    "mpow"   : 2.0,
    "gammaC" : 52.0,
    "kBT"    : 0.0,
    "tot_area"   : 62.2242,
    "tot_volume" : 26.6649,
    "kb"     : 44.4444,
    "theta"  : 6.97
}

int_rbc = mir.Interactions.MembraneForces("int_rbc", "wlc", "Kantor", **prms_rbc)
int_dpd_oo = mir.Interactions.Pairwise('dpd_oo', rc, kind="DPD", a=10.0, gamma=10.0,␣
↪kBT=1.0, power=0.5)
int_dpd_ii = mir.Interactions.Pairwise('dpd_ii', rc, kind="DPD", a=10.0, gamma=20.0,␣
↪kBT=1.0, power=0.5)
int_dpd_io = mir.Interactions.Pairwise('dpd_io', rc, kind="DPD", a=10.0, gamma=15.0,␣
↪kBT=1.0, power=0.5)
```

(continues on next page)

(continued from previous page)

```
int_dpd_sr = mir.Interactions.Pairwise('dpd_sr', rc, kind="DPD", a=0.0, gamma=15.0,␣
↪kBT=1.0, power=0.5)

u.registerInteraction(int_rbc)
u.registerInteraction(int_dpd_oo)
u.registerInteraction(int_dpd_ii)
u.registerInteraction(int_dpd_io)
u.registerInteraction(int_dpd_sr)

u.setInteraction(int_dpd_oo, pv_outer, pv_outer)
u.setInteraction(int_dpd_ii, pv_inner, pv_inner)
u.setInteraction(int_dpd_io, pv_inner, pv_outer)

u.setInteraction(int_rbc, pv_rbc, pv_rbc)

u.setInteraction(int_dpd_sr, pv_outer, pv_rbc)
u.setInteraction(int_dpd_sr, pv_inner, pv_rbc)

# integrators
#############

vv = mir.Integrators.VelocityVerlet('vv')
u.registerIntegrator(vv)

u.setIntegrator(vv, pv_outer)
u.setIntegrator(vv, pv_inner)
u.setIntegrator(vv, pv_rbc)

# Bouncers
##########

# The solvent must not go through the membrane
# we can enforce this by setting a bouncer
bouncer = mir.Bouncers.Mesh("membrane_bounce", "bounce_maxwell", kBT=0.5)

# we register the bouncer object as any other object
u.registerBouncer(bouncer)

# now we can set what PVs bounce on what OV:
u.setBouncer(bouncer, pv_rbc, pv_outer)
u.setBouncer(bouncer, pv_rbc, pv_inner)

# plugins
#########

u.registerPlugins(mir.Plugins.createStats('stats', every=500))

dump_every = 500
u.registerPlugins(mir.Plugins.createDumpParticles('part_dump_inner', pv_inner, dump_
↪every, [], 'h5/inner-'))
u.registerPlugins(mir.Plugins.createDumpParticles('part_dump_outer', pv_outer, dump_
↪every, [], 'h5/outer-'))
u.registerPlugins(mir.Plugins.createDumpMesh("mesh_dump", pv_rbc, dump_every, "ply/"))

u.run(5002, dt=dt)
```

Note: A *ParticleVector* returned by *applyObjectBelongingChecker* is automatically registered in the coordinator. There is therefore no need to provide any `InitialConditions` object.

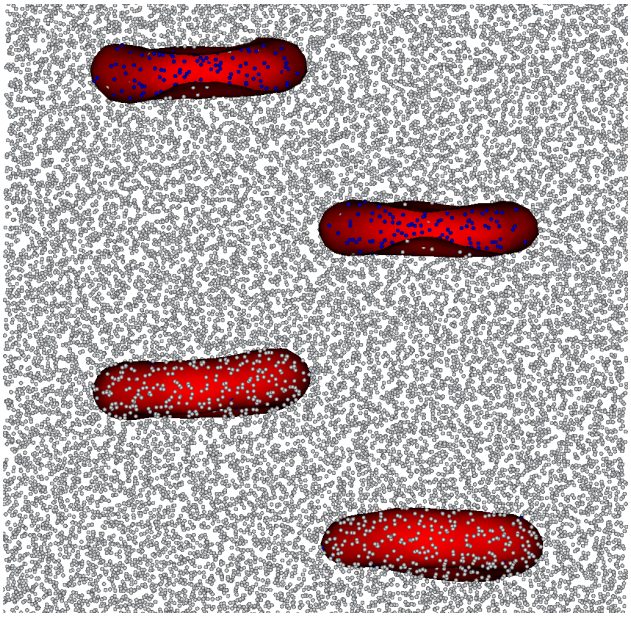


Fig. 4: Snapshots of the output files from *membranes_solvents.py*. White particles are the outer solvent, blue particles are inner. Two of the four membranes are cut for visualization purpose.

4.6 Creating Rigid Objects

Rigid objects are modeled as frozen particles moving together in a rigid motion, together with bounce back of particles, similarly to the walls. In **Mirheo**, we need to create a *RigidObjectVector*, in which each rigid object has the **same** frozen particles template. Generating these frozen particles can be done in a separate simulation using a *BelongingChecker*. This is shown in the following script for the simple mesh *sphere_mesh.off* which can be found in the *data/* folder:

Listing 6: *generate_frozen_rigid.py*

```
import mirheo as mir
import numpy as np
import trimesh

def recenter(coords, com):
    coords = [[r[0]-com[0], r[1]-com[1], r[2]-com[2]] for r in coords]
    return coords

dt = 0.001
rc = 1.0
mass = 1.0
number_density = 10.0
niter = 1000

# the triangle mesh used to create the object
# here we load the file using trimesh for convenience
```

(continues on next page)

(continued from previous page)

```
m = trimesh.load("sphere_mesh.off");

# trimesh is able to compute the inertia tensor
# we assume it is diagonal here
inertia = [row[i] for i, row in enumerate(m.moment_inertia)]

ranks  = (1, 1, 1)
domain = (16, 16, 16)

u = mir.Mirheo(ranks, domain, debug_level=3, log_filename='log')

dpd = mir.Interactions.Pairwise('dpd', rc, kind="DPD", a=10.0, gamma=10.0, kBT=0.5,␣
↪power=0.5)
vv  = mir.Integrators.VelocityVerlet('vv')

# we create here a fake rigid object in the center of the domain with only 2 particles
# those particles will be used to compute the extents in the object belonging, so they
# must be located in the to corners of the bounding box of the object
# this is only to be able to make use of the belonging checker

bb_hi = m.vertices.max(axis=0).tolist()
bb_lo = m.vertices.min(axis=0).tolist()

coords = [bb_lo, bb_hi]
com_q = [[0.5 * domain[0], 0.5 * domain[1], 0.5 * domain[2],   1, 0, 0, 0]]

mesh = mir.ParticleVectors.Mesh(m.vertices.tolist(), m.faces.tolist())
fake_ov = mir.ParticleVectors.RigidObjectVector('fake_ov', mass, inertia, len(coords),
↪ mesh)
fake_ic = mir.InitialConditions.Rigid(com_q, coords)

belonging_checker = mir.BelongingCheckers.Mesh("mesh_checker")

# similarly to wall creation, we freeze particles inside a rigid object
pv_rigid = u.makeFrozenRigidParticles(belonging_checker, fake_ov, fake_ic, [dpd], vv,␣
↪number_density, mass, dt=dt, nsteps=niter)

if u.isMasterTask():
    coords = pv_rigid.getCoordinates()
    coords = recenter(coords, com_q[0])
    np.savetxt("rigid_coords.txt", coords)
```

Note: here we make use of trimesh as we need some properties of the mesh. This would also allow to load many other formats not supported by **Mirheo**, such as ply.

Note: The saved coordinates must be in the frame of reference of the rigid object, hence the shift at the end of the script.

We can now run a simulation using our newly created rigid object. Let us build a suspension of spheres in a DPD solvent:

Listing 7: *rigid_suspension.py*

```
#!/usr/bin/env python

import mirheo as mir
import numpy as np
import trimesh

dt = 0.001
rc = 1.0
mass = 1.0
number_density = 10

m = trimesh.load("sphere_mesh.off");
inertia = [row[i] for i, row in enumerate(m.moment_inertia)]

ranks  = (1, 1, 1)
domain = (16, 8, 8)

u = mir.Mirheo(ranks, domain, debug_level=3, log_filename='log', no_splash=True)

pv_solvent = mir.ParticleVectors.ParticleVector('solvent', mass)
ic_solvent = mir.InitialConditions.Uniform(number_density)

dpd = mir.Interactions.Pairwise('dpd', rc, kind="DPD", a=10.0, gamma=10.0, kBT=0.01,␣
↪power=0.5)
# repulsive LJ to avoid overlap between spheres
cnt = mir.Interactions.Pairwise('cnt', rc, kind="RepulsiveLJ", epsilon=0.28, sigma=0.
↪8, max_force=400.0)

vv = mir.Integrators.VelocityVerlet_withPeriodicForce('vv', force=1.0, direction="x")

com_q = [[2.0, 6.0, 5.0,   1.0, 0.0, 0.0, 0.0],
         [6.0, 7.0, 5.0,   1.0, 0.0, 0.0, 0.0],
         [10., 6.0, 5.0,   1.0, 0.0, 0.0, 0.0],
         [4.0, 2.0, 4.0,   1.0, 0.0, 0.0, 0.0],
         [8.0, 3.0, 2.0,   1.0, 0.0, 0.0, 0.0]]

coords = np.loadtxt("rigid_coords.txt").tolist()
mesh = mir.ParticleVectors.Mesh(m.vertices.tolist(), m.faces.tolist())

pv_rigid = mir.ParticleVectors.RigidObjectVector('spheres', mass, inertia,␣
↪len(coords), mesh)
ic_rigid = mir.InitialConditions.Rigid(com_q, coords)
vv_rigid = mir.Integrators.RigidVelocityVerlet("vv_rigid")

u.registerParticleVector(pv_solvent, ic_solvent)
u.registerIntegrator(vv)
u.setIntegrator(vv, pv_solvent)

u.registerParticleVector(pv_rigid, ic_rigid)
u.registerIntegrator(vv_rigid)
u.setIntegrator(vv_rigid, pv_rigid)

u.registerInteraction(dpd)
u.registerInteraction(cnt)
u.setInteraction(dpd, pv_solvent, pv_solvent)
```

(continues on next page)

(continued from previous page)

```
u.setInteraction(dpd, pv_solvent, pv_rigid)
u.setInteraction(cnt, pv_rigid, pv_rigid)

# we need to remove the solvent particles inside the rigid objects
belonging_checker = mir.BelongingCheckers.Mesh("mesh checker")
u.registerObjectBelongingChecker(belonging_checker, pv_rigid)
u.applyObjectBelongingChecker(belonging_checker, pv_solvent, correct_every=0, inside=
↪"none", outside="")

# apply bounce
bb = mir.Bouncers.Mesh("bounce_rigid", "bounce_maxwell", kBT=0.01)
u.registerBouncer(bb)
u.setBouncer(bb, pv_rigid, pv_solvent)

# dump the mesh every 200 steps in ply format to the folder 'ply/'
u.registerPlugins(mir.Plugins.createDumpMesh("mesh_dump", pv_rigid, 200, "ply/"))

u.run(10000, dt=dt)
```

Note: We again used a *BelongingChecker* in order to remove the solvent inside the rigid objects.

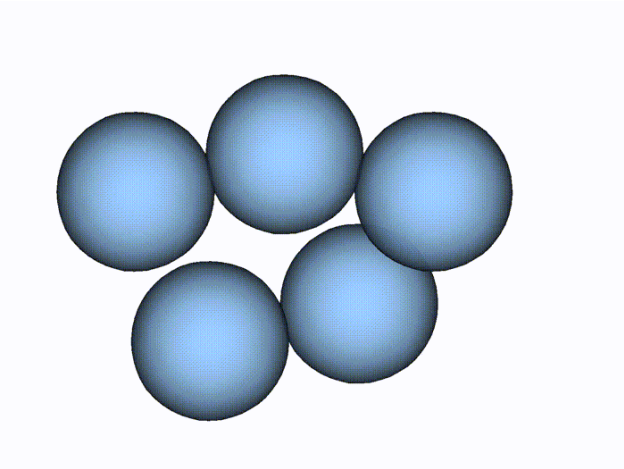


Fig. 5: Snapshots of the output files from *rigid_suspension.py*.

4.7 Going further

A set of maintained tests can be used as examples in the *tests/* folder. These tests use many features of **Mirheo** and can serve as a baseline for building more complex simulations. See also the *Testing* section of this documentation.

5 Benchmarks

The following benchmarks represent typical use cases of *Mirheo*. They were performed on the Piz-Daint supercomputer for both strong and weak scaling. See in *benchmarks/cases/* for more informations about the run scripts.

5.1 Bulk Solvent

Periodic Poiseuille flow in a periodic domain in every direction, with solvent only. Timings are based on the average time-step wall time, measured from the `createStats` plugin.

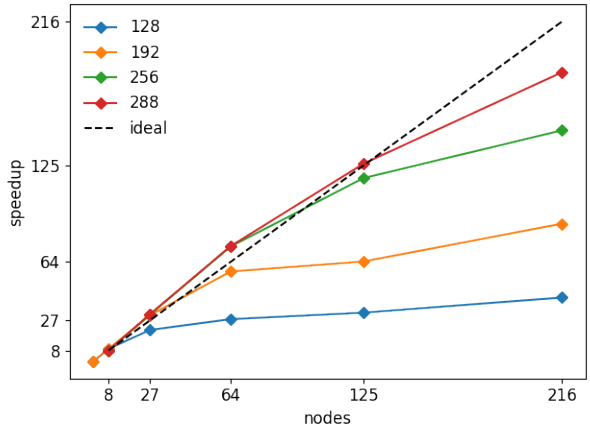


Fig. 6: strong scaling for multiple domain sizes

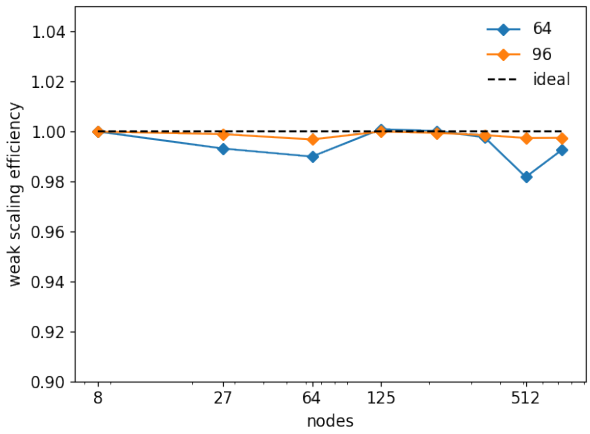


Fig. 7: weak scaling efficiency for multiple subdomain sizes

The weak scaling efficiency is very close to 1 thanks to the almost perfect overlapping of communication and computation.

5.2 Bulk Blood

Periodic Poiseuille flow for blood with 45% Hematocrite in a periodic domain in every direction. Timings are based on the average time-step wall time, measured from the `createStats` plugin.

The weak scaling efficiency is lower than in the solvent only case because of the complexity of the problem:

- Multiple solvents
- FSI interactions
- contact interactions

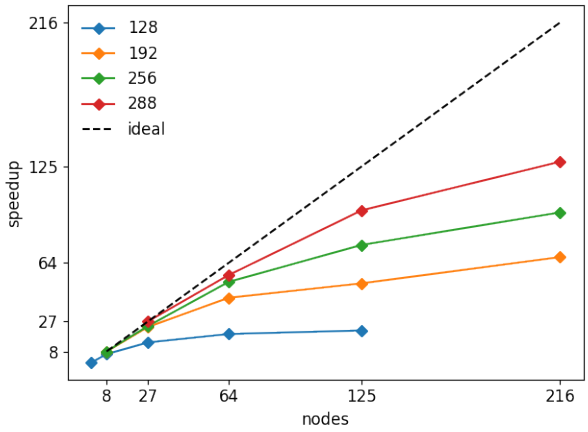


Fig. 8: strong scaling for multiple domain sizes

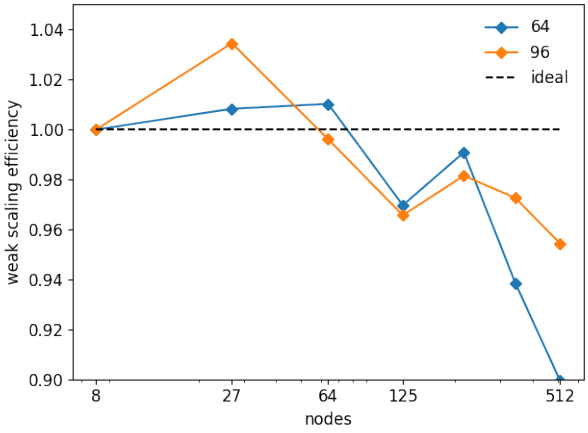


Fig. 9: weak scaling efficiency for multiple subdomain sizes

- Many objects
- Bounce back on membranes

The above induces a lot more communication than the simple solvent only case.

5.3 Poiseuille Flow

Poiseuille flow between two plates (walls), with solvent only. Timings are based on the average time-step wall time, measured from the `createStats` plugin.

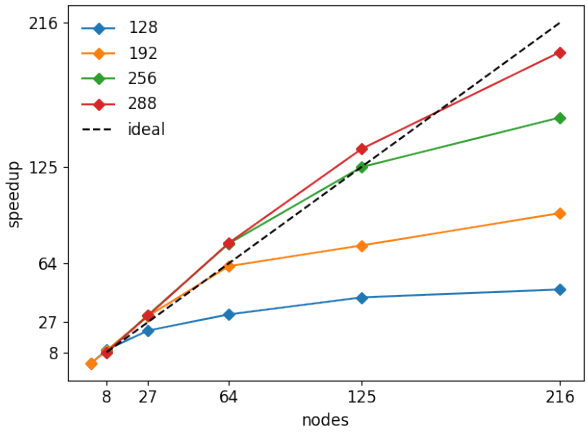


Fig. 10: strong scaling for multiple domain sizes

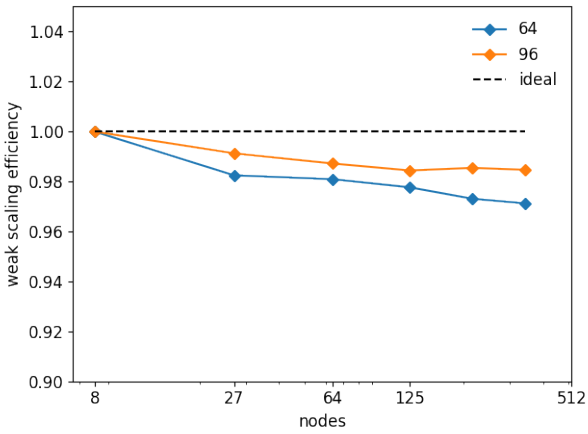


Fig. 11: weak scaling efficiency for multiple subdomain sizes

5.4 Rigid Objects suspension

Periodic Poiseuille flow for rigid suspensions in a periodic domain. Timings are based on the average time-step wall time, measured from the `createStats` plugin.

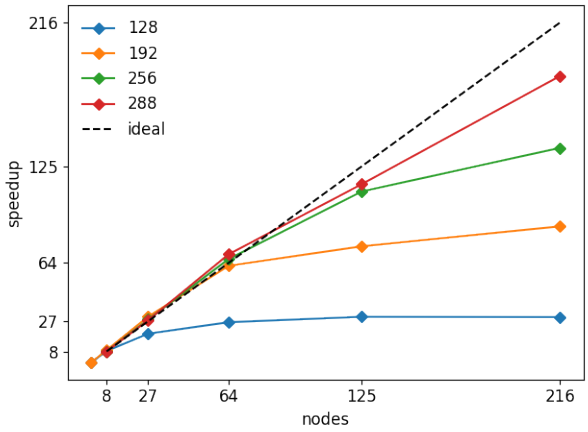


Fig. 12: strong scaling for multiple domain sizes

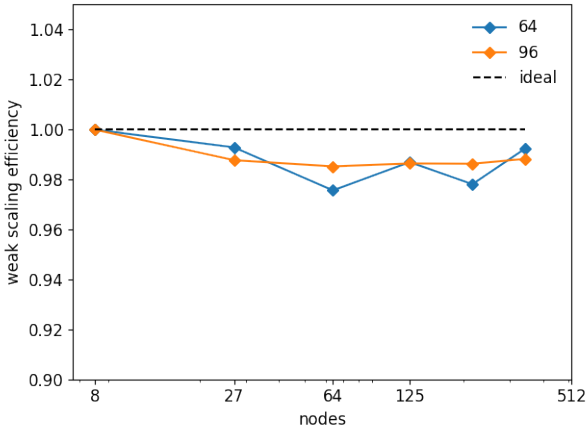


Fig. 13: weak scaling efficiency for multiple subdomain sizes

5.5 I/O overlap with computation

Data dump every 100 steps for the periodic Poiseuille flow benchmark. Computation timings are based on the average time-step wall time, measured from the `createStats` plugin when no I/O is performed. The I/O timings are extracted from the log files. The total timings are based on the average time-step wall time when I/O is active.

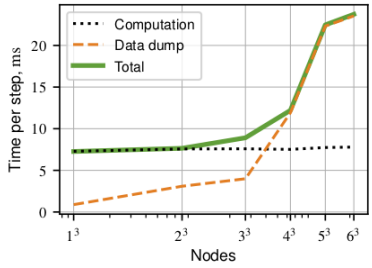


Fig. 14: Overlap of data dump and computation

6 Mirheo coordinator

The coordinator class stitches together data containers, *Particle Vectors*, and all the handlers, and provides functions to manipulate the system components.

A single instance of this class should be created in the beginning of any simulation setup.

Note: Creating the coordinator will internally call the MPI_Init() function, and its destruction will call MPI_Finalize(). Therefore if using a mpi4py Python module, it should be imported in the following way:

```
import mpi4py
mpi4py.rc(initialize=False, finalize=False)
from mpi4py import MPI
```

---

class Mirheo

Main coordination class, should only be one instance at a time

Methods

__init__(*nranks: int3, domain: real3, log_filename: str='log', debug_level: int=-1, checkpoint_every: int=0, checkpoint_folder: str='restart/', checkpoint_mode: str='PingPong', max_obj_half_length: float=0.0, cuda_aware_mpi: bool=False, no_splash: bool=False, comm_ptr: int=0*) → None

Create the Mirheo coordinator.

Warning: Debug level determines the amount of output produced by each of the simulation processes:

0. silent: no log output
1. only report fatal errors

2. report serious errors
3. report important information steps of simulation and warnings (this is the default level)
4. report not critical information
5. report some debug information
6. report more debug
7. report all the debug
8. force flushing to the file after each message

Debug levels above 4 or 5 may significanlty increase the runtime, they are only recommended to debug errors. Flushing increases the runtime yet more, but it is required in order not to lose any messages in case of abnormal program abort.

The default debug level may be modified by setting the `MIRHEO_DEBUG_LEVEL` environment variable to the desired value. This variable may be useful when Mirheo is linked as part of other codes, in which case the `debug_level` variable affects only parts of the execution.

Parameters

- **`nranks`** – number of MPI simulation tasks per axis: x,y,z. If postprocess is enabled, the same number of the postprocess tasks will be running
- **`domain`** – size of the simulation domain in x,y,z. Periodic boundary conditions are applied at the domain boundaries. The domain will be split in equal chunks between the MPI ranks. The largest chunk size that a single MPI rank can have depends on the total number of particles, handlers and hardware, and is typically about $120^3 - 200^3$.
- **`log_filename`** – prefix of the log files that will be created. Logging is implemented in the form of one file per MPI rank, so in the simulation folder NP files with names log_00000.log, log_00001.log, ... will be created, where NP is the total number of MPI ranks. Each MPI task (including postprocess) writes messages about itself into his own log file, and the combined log may be created by merging all the individual ones and sorting with respect to time. If this parameter is set to 'stdout' or 'stderr' standard output or standard error streams will be used instead of the file, however, there is no guarantee that messages from different ranks are synchronized.
- **`debug_level`** – Debug level from 0 to 8, see above.
- **`checkpoint_every`** – save state of the simulation components (particle vectors and handlers like integrators, plugins, etc.)
- **`checkpoint_folder`** – folder where the checkpoint files will reside (for Checkpoint mechanism), or folder prefix (for Snapshot mechanism)
- **`checkpoint_mode`** – set to "PingPong" to keep only the last 2 checkpoint states; set to "Incremental" to keep all checkpoint states.
- **`max_obj_half_length`** – Half of the maximum size of all objects. Needs to be set when objects are self interacting with pairwise interactions.
- **`cuda_aware_mpi`** – enable CUDA Aware MPI. The MPI library must support that feature, otherwise it may fail.
- **`no_splash`** – don't display the splash screen when at the start-up.
- **`comm_ptr`** – pointer to communicator. By default MPI_COMM_WORLD will be used

applyObjectBelongingChecker (*checker: mirheo::ObjectBelongingChecker*, *pv: mirheo::ParticleVector*, *correct_every: int=0*, *inside: str=''*, *outside: str=''*) → mirheo::ParticleVector

Apply the **checker** to the given particle vector. One and only one of the options **inside** or **outside** has to be specified.

Parameters

- **checker** – instance of *BelongingChecker*
- **pv** – *ParticleVector* that will be split (source PV)
- **inside** – if specified and not "none", a new *ParticleVector* with name **inside** will be returned, that will keep the inner particles of the source PV. If set to "none", None object will be returned. In any case, the source PV will only contain the outer particles
- **outside** – if specified and not "none", a new *ParticleVector* with name **outside** will be returned, that will keep the outer particles of the source PV. If set to "none", None object will be returned. In any case, the source PV will only contain the inner particles
- **correct_every** – If greater than zero, perform correction every this many time-steps. Correction will move e.g. *inner* particles of outer PV to the :inner PV and viceversa. If one of the PVs was defined as "none", the 'wrong' particles will be just removed.

Returns New *ParticleVector* or None depending on **inside** and **outside** options

computeVolumeInsideWalls (*walls: List[mirheo::Wall], nSamplesPerRank: int=100000*) → float

Compute the volume inside the given walls in the whole domain (negative values are the 'inside' of the simulation). The computation is made via simple Monte-Carlo.

Parameters

- **walls** – sdf based walls
- **nSamplesPerRank** – number of Monte-Carlo samples used per rank

deregisterIntegrator (*integrator: mirheo::Integrator*) → None

Deregister a integrator.

deregisterPlugins (*arg0: mirheo::SimulationPlugin*, *arg1: mirheo::PostprocessPlugin*) → None

Deregister a plugin.

dumpWalls2XDMF (*walls: List[mirheo::Wall], h: real3, filename: str='xdmf/wall'*) → None

Write Signed Distance Function for the intersection of the provided walls (negative values are the 'inside' of the simulation)

Parameters

- **walls** – list of walls to dump; the output sdf will be the union of all walls inside
- **h** – cell-size of the resulting grid
- **filename** – base filename output, will create to files filename.xmf and filename.h5

getState (*self: Mirheo*) → MirState

Return mirheo state

isComputeTask (*self: Mirheo*) → bool

Returns `True` if the current rank is a simulation task and `False` if it is a postrprocess task

isMasterTask (*self: Mirheo*) → bool

Returns `True` if the current rank is the root

log_compile_options (*self: Mirheo*) → None

output compile times options in the log

makeFrozenRigidParticles(*checker: mirheo::ObjectBelongingChecker, shape: mirheo::ObjectVector, icShape: mirheo::InitialConditions, interactions: List[mirheo::Interaction], integrator: mirheo::Integrator, number_density: float, mass: float=1.0, dt: float, nsteps: int=1000*) → mirheo::ParticleVector

Create particles frozen inside object.

Note: A separate simulation will be run for every call to this function, which may take certain amount of time. If you want to save time, consider using restarting mechanism instead

---

Parameters

- **checker** – object belonging checker
- **shape** – object vector describing the shape of the rigid object
- **icShape** – initial conditions for shape
- **interactions** – list of *Interaction* that will be used to construct the equilibrium particles distribution
- **integrator** – this *Integrator* will be used to construct the equilibrium particles distribution
- **number_density** – target particle number density
- **mass** – the mass of a single frozen particle
- **dt** – time step
- **nsteps** – run this many steps to achieve equilibrium

Returns New *ParticleVector* that will contain particles that are close to the wall boundary, but still inside the wall.

makeFrozenWallParticles(*pvName: str, walls: List[mirheo::Wall], interactions: List[mirheo::Interaction], integrator: mirheo::Integrator, number_density: float, mass: float=1.0, dt: float, nsteps: int=1000*) → mirheo::ParticleVector

Create particles frozen inside the walls.

Note: A separate simulation will be run for every call to this function, which may take certain amount of time. If you want to save time, consider using restarting mechanism instead

---

Parameters

- **pvName** – name of the created particle vector
- **walls** – array of instances of *Wall* for which the frozen particles will be generated
- **interactions** – list of *Interaction* that will be used to construct the equilibrium particles distribution
- **integrator** – this *Integrator* will be used to construct the equilibrium particles distribution
- **number_density** – target particle number density
- **mass** – the mass of a single frozen particle

- **dt** – time step
- **nsteps** – run this many steps to achieve equilibrium

Returns New *ParticleVector* that will contain particles that are close to the wall boundary, but still inside the wall.

registerBouncer (*bouncer: mirheo::Bouncer*) → None
Register Object Bouncer

Parameters **bouncer** – the *Bouncer* to register

registerIntegrator (*integrator: mirheo::Integrator*) → None
Register an *Integrator* to the coordinator

Parameters **integrator** – the *Integrator* to register

registerInteraction (*interaction: mirheo::Interaction*) → None
Register an *Interaction* to the coordinator

Parameters **interaction** – the *Interaction* to register

registerObjectBelongingChecker (*checker: mirheo::ObjectBelongingChecker*, *ov: mirheo::ObjectVector*) → None
Register Object Belonging Checker

Parameters

- **checker** – instance of *BelongingChecker*
- **ov** – *ObjectVector* belonging to which the **checker** will check

registerParticleVector (*pv: mirheo::ParticleVector*, *ic: mirheo::InitialConditions=None*) → None
Register particle vector

Parameters

- **pv** – *ParticleVector*
- **ic** – *InitialConditions* that will generate the initial distibution of the particles

registerPlugins (*arg0: mirheo::SimulationPlugin*, *arg1: mirheo::PostprocessPlugin*) → None
Register Plugins

registerWall (*wall: mirheo::Wall*, *check_every: int=0*) → None
Register a *Wall*.

Parameters

- **wall** – the *Wall* to register
- **check_every** – if positive, check every this many time steps if particles penetrate the walls

restart (*folder: str='restart/'*) → None
Restart the simulation. This function should typically be called just before running the simulation. It will read the state of all previously registered instances of *ParticleVector*, *Interaction*, etc. If the folder contains no checkpoint file required for one of those, an error occur.

> **Warning:** Certain *Plugins* may not implement restarting mechanism and will not restart correctly. Please check the documentation for the plugins.

Parameters **folder** – folder with the checkpoint files

run (*niters: int*, *dt: float*) → None
Advance the system for a given amount of time steps.

Parameters

- **niters** – number of time steps to advance
- **dt** – time step duration

save_dependency_graph_graphml (*fname: str*, *current: bool=True*) → None
Exports GraphML file with task graph for the current simulation time-step

Parameters

- **fname** – the output filename (without extension)
- **current** – if True, save the current non empty tasks; else, save all tasks that can exist in a simulation

Warning: if current is set to True, this must be called **after** *mmirheo.Mirheo.run()*.

setBouncer (*bouncer: mirheo::Bouncer*, *ov: mirheo::ObjectVector*, *pv: mirheo::ParticleVector*) → None
Assign a *Bouncer* between an *ObjectVector* and a *ParticleVector*.

Parameters

- **bouncer** – *Bouncer* compatible with the object vector
- **ov** – the *ObjectVector* to be bounced on
- **pv** – the *ParticleVector* to be bounced

setIntegrator (*integrator: mirheo::Integrator*, *pv: mirheo::ParticleVector*) → None
Set a specific *Integrator* to a given *ParticleVector*

Parameters

- **integrator** – the *Integrator* to assign
- **pv** – the concerned *ParticleVector*

setInteraction (*interaction: mirheo::Interaction*, *pv1: mirheo::ParticleVector*, *pv2: mirheo::ParticleVector*) → None
Forces between two instances of *ParticleVector* (they can be the same) will be computed according to the defined interaction.

Parameters

- **interaction** – *Interaction* to apply
- **pv1** – first *ParticleVector*
- **pv2** – second *ParticleVector*

setWall (*wall: mirheo::Wall*, *pv: mirheo::ParticleVector*, *maximum_part_travel: float=0.25*) → None
Assign a *Wall* bouncer to a given *ParticleVector*. The current implementation does not support *ObjectVector*.

Parameters

- **wall** – the *Wall* surface which will bounce the particles

- **pv** – the *ParticleVector* to be bounced
- **maximum_part_travel** – maximum distance that one particle travels in one time step. this should be as small as possible for performance reasons but large enough for correctness

start_profiler (*self: Mirheo*) → None
Tells nvprof to start recording timeline

stop_profiler (*self: Mirheo*) → None
Tells nvprof to stop recording timeline

6.1 Unit system

Mirheo assumes all values are dimensionless. However, users may use Mirheo in combination with the pint Python package, by defining Mirheo's unit system using `set_unit_registry`:

```
import mirheo as mir
import pint
ureg = pint.UnitRegistry()

# Define Mirheo's unit system.
ureg.define('mirL = 1 um')
ureg.define('mirT = 1 us')
ureg.define('mirM = 1e-20 kg')
mir.set_unit_registry(ureg)

# dt automatically converted to 0.01, matching the value of 1e-8 s in the Mirheo unit␣
↪system.
u = mir.Mirheo(..., dt=ureg('1e-8 s'), ...)
```

6.2 Global Simulation State

Some information about the simulation is global to all Mirheo components. They are stored in the following binded objects:

class DomainInfo
Convert between local domain coordinates (specific to each rank) and global domain coordinates.

Methods

—

global_size
Size of the whole simulation domain.

global_start
Subdomain lower corner position of the current rank, in global coordinates.

global_to_local (*x: real3*) → real3
Convert from global coordinates to local coordinates.

Parameters **x** – Position in global coordinates.

global_to_local_shift
shift to transform global coordinates to local coordinates.

is_in_subdomain (*x: real3*) → bool
Returns True if the given position (in global coordinates) is inside the subdomain of the current rank, False otherwise.

Parameters **x** – Position in global coordinates.

local_size
Subdomain extents of the current rank.

local_to_global (*x: real3*) → real3
Convert local coordinates to global coordinates.

Parameters **x** – Position in local coordinates.

local_to_global_shift
shift to transform local coordinates to global coordinates.

class MirState
state of the simulation shared by all simulation objects.

Methods

—

current_dt
Current simulation step size dt. Note: this property is accessible only while Mirheo::run() is running.

current_step
Current simulation step.

current_time
Current simulation time.

domain_info
The *DomainInfo* of the current rank.

7 Particle Vectors

A *ParticleVector* (or PV) is a collection of particles in the simulation with identical properties. PV is the minimal unit of particles that can be addressed by most of the processing utilities, i.e. it is possible to specify interactions between different (or same) PVs, apply integrators, plugins, etc. to the PVs.

Each particle in the PV keeps its coordinate, velocity and force. Additional quantities may also be stored in a form of extra channels. These quantities are usually added and used by specific handlers, and can in principle be written in XDMF format (*createDumpParticles*), see more details in the Developer documentation.

A common special case of a *ParticleVector* is an *ObjectVector* (or OV). The OV **is** a Particle Vector with the particles separated into groups (objects) of the same size. For example, if a single cell membrane is represented by say 500 particles, an object vector consisting of the membranes will contain all the particles of all the membranes, grouped by membrane. Objects are assumed to be spatially localized, so they always fully reside within a single MPI process. OV can be used in most of the places where a regular PV can be used, and more

7.1 Reserved names

A list of name are reserved by Mirheo. When a user provides a custom channel name, it needs to be different than these reserved fields:

- **Reserved particle channel fields:**
  - "ids"
  - "positions"
  - "velocities"
  - "__forces"
  - "stresses"
  - "densities"
  - "old_positions"
- **Reserved object channel fields:**
  - "ids"
  - "motions"
  - "old_motions"
  - "com_extents"
  - "area_volumes"
  - "membrane_type_id"
  - "areas"
  - "mean_curvatures"
  - "len_theta_tot"
- **Reserved bisegment channel fields:**
  - "states"
  - "energies"
  - "biseg_kappa"
  - "biseg_tau_l"

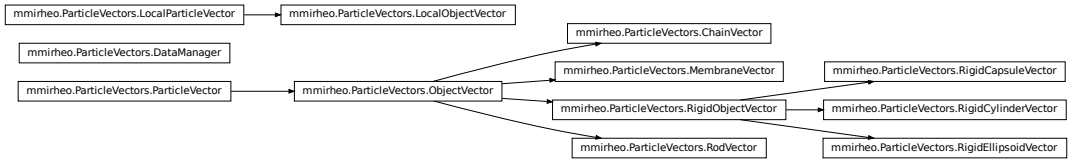
7.2 Summary

ChainVector()	Object Vector representing chain of particles.
DataManager()	A collection of channels in pinned memory.
LocalObjectVector()	Object vector local data storage, additionally contains object channels.
LocalParticleVector()	Particle local data storage, composed of particle channels.
MembraneMesh()	Internally used class for desctibing a triangular mesh that can be used with the Membrane Interactions.
MembraneVector()	Membrane is an Object Vector representing cell membranes.
Mesh()	Internally used class for describing a simple triangular mesh
ObjectVector()	Basic Object Vector.

Continued on next page

Table 4 – continued from previous page

ParticleVector()	Basic particle vector, consists of identical disconnected particles.
RigidCapsuleVector()	*RigidObjectVector* specialized for capsule shapes.
RigidCylinderVector()	*RigidObjectVector* specialized for cylindrical shapes.
RigidEllipsoidVector()	*RigidObjectVector* specialized for ellipsoidal shapes.
RigidObjectVector()	Rigid Object is an Object Vector representing objects that move as rigid bodies, with no relative displacement against each other in an object.
RodVector()	Rod Vector is an *ObjectVector* which reprents rod geometries.
getReservedBisegmentChannels()	Return the list of reserved channel names per bisegment fields
getReservedObjectChannels()	Return the list of reserved channel names for object fields
getReservedParticleChannels()	Return the list of reserved channel names for particle fields



7.3 Details

class ChainVector

Bases: *mmirheo.ParticleVectors.ObjectVector*

Object Vector representing chain of particles.

__init__ (*name: str*, *mass: float*, *chain_length: int*) → None

Parameters

- **name** – name of the created PV
- **mass** – mass of a single particle
- **chain_length** – number of particles per chain

getCoordinates (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of coordinate for every of the N particles

Return type A list of $N \times 3$ reals

getForces (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of force for every of the N particles

Return type A list of $N \times 3$ reals

getVelocities (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of velocity for every of the N particles

Return type A list of $N \times 3$ reals

get_indices (*self: ParticleVectors.ParticleVector*) → List[int]

Returns A list of unique integer particle identifiers

halo
The halo LocalObjectVector instance, the storage of halo objects.

local
The local LocalObjectVector instance, the storage of local objects.

setCoordinates (*coordinates: List[real3]*) → None

Parameters **coordinates** – A list of $N \times 3$ reals: 3 components of coordinate for every of the N particles

setForces (*forces: List[real3]*) → None

Parameters **forces** – A list of $N \times 3$ reals: 3 components of force for every of the N particles

setVelocities (*velocities: List[real3]*) → None

Parameters **velocities** – A list of $N \times 3$ reals: 3 components of velocity for every of the N particles

class DataManager
Bases: object

A collection of channels in pinned memory.

__init__ ()
Initialize self. See help(type(self)) for accurate signature.

class LocalObjectVector
Bases: *mmirheo.ParticleVectors.LocalParticleVector*

Object vector local data storage, additionally contains object channels.

__init__ ()
Initialize self. See help(type(self)) for accurate signature.

per_object
The *DataManager* that contains the object channels.

per_particle
The *DataManager* that contains the particle channels.

class LocalParticleVector
Bases: object

Particle local data storage, composed of particle channels.

__init__ ()
Initialize self. See help(type(self)) for accurate signature.

per_particle
The *DataManager* that contains the particle channels.

class MembraneMesh

Bases: *mmirheo.ParticleVectors.Mesh*

Internally used class for desctibing a triangular mesh that can be used with the Membrane Interactions. In contrast with the simple *Mesh*, this class precomputes some required quantities on the mesh, including connectivity structures and stress-free quantities.

__init__ (**args*, ***kwargs*)

Overloaded function.

1. __init__(off_filename: str) -> None

   Create a mesh by reading the OFF file. The stress free shape is the input initial mesh

   Args: off_filename: path of the OFF file

2. __init__(off_initial_mesh: str, off_stress_free_mesh: str) -> None

   Create a mesh by reading the OFF file, with a different stress free shape.

   Args: off_initial_mesh: path of the OFF file : initial mesh off_stress_free_mesh: path of the OFF file : stress-free mesh)

3. __init__(vertices: List[real3], faces: List[int3]) -> None

   Create a mesh by giving coordinates and connectivity

   Args: vertices: vertex coordinates faces: connectivity: one triangle per entry, each integer corresponding to the vertex indices

4. __init__(vertices: List[real3], stress_free_vertices: List[real3], faces: List[int3]) -> None

   Create a mesh by giving coordinates and connectivity, with a different stress-free shape.

   Args: vertices: vertex coordinates stress_free_vertices: vertex coordinates of the stress-free shape faces: connectivity: one triangle per entry, each integer corresponding to the vertex indices

getFaces (*self: ParticleVectors.Mesh*) → List[List[int[3]]]

returns the vertex indices for each triangle of the mesh.

getVertices (*self: ParticleVectors.Mesh*) → List[List[float[3]]]

returns the vertex coordinates of the mesh.

class MembraneVector

Bases: *mmirheo.ParticleVectors.ObjectVector*

Membrane is an Object Vector representing cell membranes. It must have a triangular mesh associated with it such that each particle is mapped directly onto single mesh vertex.

__init__ (*name: str*, *mass: float*, *mesh: ParticleVectors.MembraneMesh*) → None

Parameters

- **name** – name of the created PV
- **mass** – mass of a single particle
- **mesh** – *MembraneMesh* object

getCoordinates (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of coordinate for every of the N particles

Return type A list of $N \times 3$ reals

getForces (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of force for every of the N particles

Return type A list of $N \times 3$ reals

getVelocities (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of velocity for every of the N particles

Return type A list of $N \times 3$ reals

get_indices (*self: ParticleVectors.ParticleVector*) → List[int]

Returns A list of unique integer particle identifiers

halo
The halo LocalObjectVector instance, the storage of halo objects.

local
The local LocalObjectVector instance, the storage of local objects.

setCoordinates (*coordinates: List[real3]*) → None

Parameters **coordinates** – A list of $N \times 3$ reals: 3 components of coordinate for every of the N particles

setForces (*forces: List[real3]*) → None

Parameters **forces** – A list of $N \times 3$ reals: 3 components of force for every of the N particles

setVelocities (*velocities: List[real3]*) → None

Parameters **velocities** – A list of $N \times 3$ reals: 3 components of velocity for every of the N particles

class Mesh
Bases: `object`

Internally used class for describing a simple triangular mesh

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(off_filename: str) -> None

   Create a mesh by reading the OFF file

   Args: off_filename: path of the OFF file

2. __init__(vertices: List[real3], faces: List[int3]) -> None

   Create a mesh by giving coordinates and connectivity

   Args: vertices: vertex coordinates faces: connectivity: one triangle per entry, each integer corresponding to the vertex indices

getFaces (*self: ParticleVectors.Mesh*) → List[List[int[3]]]
returns the vertex indices for each triangle of the mesh.

getVertices (*self: ParticleVectors.Mesh*) → List[List[float[3]]]
returns the vertex coordinates of the mesh.

class ObjectVector
Bases: *mmirheo.ParticleVectors.ParticleVector*

Basic Object Vector. An Object Vector stores chunks of particles, each chunk belonging to the same object.

Warning: In case of interactions with other *ParticleVector*, the extents of the objects must be smaller than a subdomain size. The code only issues a run time warning but it is the responsibility of the user to ensure this condition for correctness.

__init__()
Initialize self. See help(type(self)) for accurate signature.

getCoordinates(*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of coordinate for every of the N particles

Return type A list of $N \times 3$ reals

getForces(*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of force for every of the N particles

Return type A list of $N \times 3$ reals

getVelocities(*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of velocity for every of the N particles

Return type A list of $N \times 3$ reals

get_indices(*self: ParticleVectors.ParticleVector*) → List[int]

Returns A list of unique integer particle identifiers

halo
The halo LocalObjectVector instance, the storage of halo objects.

local
The local LocalObjectVector instance, the storage of local objects.

setCoordinates(*coordinates: List[real3]*) → None

Parameters **coordinates** – A list of $N \times 3$ reals: 3 components of coordinate for every of the N particles

setForces(*forces: List[real3]*) → None

Parameters **forces** – A list of $N \times 3$ reals: 3 components of force for every of the N particles

setVelocities(*velocities: List[real3]*) → None

Parameters **velocities** – A list of $N \times 3$ reals: 3 components of velocity for every of the N particles

class ParticleVector
Bases: object

Basic particle vector, consists of identical disconnected particles.

__init__(*name: str*, *mass: float*) → None

Parameters

- **name** – name of the created PV
- **mass** – mass of a single particle

getCoordinates(*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of coordinate for every of the N particles

Return type A list of $N \times 3$ reals

getForces (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of force for every of the N particles

Return type A list of $N \times 3$ reals

getVelocities (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of velocity for every of the N particles

Return type A list of $N \times 3$ reals

get_indices (*self: ParticleVectors.ParticleVector*) → List[int]

Returns A list of unique integer particle identifiers

halo
The halo LocalParticleVector instance, the storage of halo particles.

local
The local LocalParticleVector instance, the storage of local particles.

setCoordinates (*coordinates: List[real3]*) → None

Parameters **coordinates** – A list of $N \times 3$ reals: 3 components of coordinate for every of the N particles

setForces (*forces: List[real3]*) → None

Parameters **forces** – A list of $N \times 3$ reals: 3 components of force for every of the N particles

setVelocities (*velocities: List[real3]*) → None

Parameters **velocities** – A list of $N \times 3$ reals: 3 components of velocity for every of the N particles

class RigidCapsuleVector
Bases: *mmirheo.ParticleVectors.RigidObjectVector*

RigidObjectVector specialized for capsule shapes. The advantage is that it doesn't need mesh and moment of inertia define, as those can be computed analytically.

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(name: str, mass: float, object_size: int, radius: float, length: float) -> None

   Args: name: name of the created PV mass: mass of a single particle object_size: number of frozen particles per object radius: radius of the capsule length: length of the capsule between the half balls. The total height is then "length + 2 * radius"

2. __init__(name: str, mass: float, object_size: int, radius: float, length: float, mesh: ParticleVectors.Mesh) -> None

   Args: name: name of the created PV mass: mass of a single particle object_size: number of frozen particles per object radius: radius of the capsule length: length of the capsule between the half balls. The total height is then "length + 2 * radius" mesh: *Mesh* object representing the shape of the object. This is used for dump only.

getCoordinates (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of coordinate for every of the N particles

Return type A list of $N \times 3$ reals

getForces (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of force for every of the N particles

Return type A list of $N \times 3$ reals

getVelocities (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of velocity for every of the N particles

Return type A list of $N \times 3$ reals

get_indices (*self: ParticleVectors.ParticleVector*) → List[int]

Returns A list of unique integer particle identifiers

halo
The halo LocalObjectVector instance, the storage of halo objects.

local
The local LocalObjectVector instance, the storage of local objects.

setCoordinates (*coordinates: List[real3]*) → None

Parameters **coordinates** – A list of $N \times 3$ reals: 3 components of coordinate for every of the N particles

setForces (*forces: List[real3]*) → None

Parameters **forces** – A list of $N \times 3$ reals: 3 components of force for every of the N particles

setVelocities (*velocities: List[real3]*) → None

Parameters **velocities** – A list of $N \times 3$ reals: 3 components of velocity for every of the N particles

class RigidCylinderVector
Bases: *mmirheo.ParticleVectors.RigidObjectVector*

RigidObjectVector specialized for cylindrical shapes. The advantage is that it doesn't need mesh and moment of inertia define, as those can be computed analytically.

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(name: str, mass: float, object_size: int, radius: float, length: float) -> None

   Args: name: name of the created PV mass: mass of a single particle object_size: number of frozen particles per object radius: radius of the cylinder length: length of the cylinder

2. __init__(name: str, mass: float, object_size: int, radius: float, length: float, mesh: ParticleVectors.Mesh) -> None

   Args: name: name of the created PV mass: mass of a single particle object_size: number of frozen particles per object radius: radius of the cylinder length: length of the cylinder mesh: *Mesh* object representing the shape of the object. This is used for dump only.

getCoordinates (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of coordinate for every of the N particles

Return type A list of $N \times 3$ reals

getForces (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of force for every of the N particles

Return type A list of $N \times 3$ reals

getVelocities (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of velocity for every of the N particles

Return type A list of $N \times 3$ reals

get_indices (*self: ParticleVectors.ParticleVector*) → List[int]

Returns A list of unique integer particle identifiers

halo
The halo LocalObjectVector instance, the storage of halo objects.

local
The local LocalObjectVector instance, the storage of local objects.

setCoordinates (*coordinates: List[real3]*) → None

Parameters **coordinates** – A list of $N \times 3$ reals: 3 components of coordinate for every of the N particles

setForces (*forces: List[real3]*) → None

Parameters **forces** – A list of $N \times 3$ reals: 3 components of force for every of the N particles

setVelocities (*velocities: List[real3]*) → None

Parameters **velocities** – A list of $N \times 3$ reals: 3 components of velocity for every of the N particles

class RigidEllipsoidVector
Bases: *mmirheo.ParticleVectors.RigidObjectVector*

RigidObjectVector specialized for ellipsoidal shapes. The advantage is that it doesn't need mesh and moment of inertia define, as those can be computed analytically.

__init__ (**args*, ***kwargs*)
Overloaded function.

1. __init__(name: str, mass: float, object_size: int, semi_axes: real3) -> None

   Args: name: name of the created PV mass: mass of a single particle object_size: number of frozen particles per object semi_axes: ellipsoid principal semi-axes

2. __init__(name: str, mass: float, object_size: int, semi_axes: real3, mesh: ParticleVectors.Mesh) -> None

   Args: name: name of the created PV mass: mass of a single particle object_size: number of frozen particles per object radius: radius of the cylinder semi_axes: ellipsoid principal semi-axes mesh: *Mesh* object representing the shape of the object. This is used for dump only.

getCoordinates (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of coordinate for every of the N particles

Return type A list of $N \times 3$ reals

getForces (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of force for every of the N particles

Return type A list of $N \times 3$ reals

getVelocities (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of velocity for every of the N particles

Return type A list of $N \times 3$ reals

get_indices (*self: ParticleVectors.ParticleVector*) → List[int]

Returns A list of unique integer particle identifiers

halo
The halo LocalObjectVector instance, the storage of halo objects.

local
The local LocalObjectVector instance, the storage of local objects.

setCoordinates (*coordinates: List[real3]*) → None

Parameters **coordinates** – A list of $N \times 3$ reals: 3 components of coordinate for every of the N particles

setForces (*forces: List[real3]*) → None

Parameters **forces** – A list of $N \times 3$ reals: 3 components of force for every of the N particles

setVelocities (*velocities: List[real3]*) → None

Parameters **velocities** – A list of $N \times 3$ reals: 3 components of velocity for every of the N particles

class RigidObjectVector
Bases: *mmirheo.ParticleVectors.ObjectVector*

Rigid Object is an Object Vector representing objects that move as rigid bodies, with no relative displacement against each other in an object. It must have a triangular mesh associated with it that defines the shape of the object.

__init__ (*name: str*, *mass: float*, *inertia: real3*, *object_size: int*, *mesh: ParticleVectors.Mesh*) → None

Parameters

- **name** – name of the created PV
- **mass** – mass of a single particle
- **inertia** – moment of inertia of the body in its principal axes. The principal axes of the mesh are assumed to be aligned with the default global *OXYZ* axes
- **object_size** – number of frozen particles per object
- **mesh** – *Mesh* object used for bounce back and dump

getCoordinates (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of coordinate for every of the N particles

Return type A list of $N \times 3$ reals

getForces (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of force for every of the N particles

Return type A list of $N \times 3$ reals

getVelocities (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of velocity for every of the N particles

Return type A list of $N \times 3$ reals

get_indices (*self: ParticleVectors.ParticleVector*) → List[int]

Returns A list of unique integer particle identifiers

halo
The halo LocalObjectVector instance, the storage of halo objects.

local
The local LocalObjectVector instance, the storage of local objects.

setCoordinates (*coordinates: List[real3]*) → None

Parameters **coordinates** – A list of $N \times 3$ reals: 3 components of coordinate for every of the N particles

setForces (*forces: List[real3]*) → None

Parameters **forces** – A list of $N \times 3$ reals: 3 components of force for every of the N particles

setVelocities (*velocities: List[real3]*) → None

Parameters **velocities** – A list of $N \times 3$ reals: 3 components of velocity for every of the N particles

class RodVector
Bases: *mmirheo.ParticleVectors.ObjectVector*

Rod Vector is an *ObjectVector* which reprents rod geometries.

__init__ (*name: str*, *mass: float*, *num_segments: int*) → None

Parameters

- **name** – name of the created Rod Vector
- **mass** – mass of a single particle
- **num_segments** – number of elements to discretize the rod

getCoordinates (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of coordinate for every of the N particles

Return type A list of $N \times 3$ reals

getForces (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of force for every of the N particles

Return type A list of $N \times 3$ reals

getVelocities (*self: ParticleVectors.ParticleVector*) → List[List[float[3]]]

Returns 3 components of velocity for every of the N particles

Return type A list of $N \times 3$ reals

get_indices (*self: ParticleVectors.ParticleVector*) → List[int]

Returns A list of unique integer particle identifiers

halo
The halo LocalObjectVector instance, the storage of halo objects.

local
The local LocalObjectVector instance, the storage of local objects.

setCoordinates (*coordinates: List[real3]*) → None

Parameters **coordinates** – A list of $N \times 3$ reals: 3 components of coordinate for every of the N particles

setForces (*forces: List[real3]*) → None

Parameters **forces** – A list of $N \times 3$ reals: 3 components of force for every of the N particles

setVelocities(*velocities: List[real3]*) → None

Parameters **velocities** – A list of $N \times 3$ reals: 3 components of velocity for every of the N particles

getReservedBisegmentChannels() → List[str]
Return the list of reserved channel names per bisegment fields

getReservedObjectChannels() → List[str]
Return the list of reserved channel names for object fields

getReservedParticleChannels() → List[str]
Return the list of reserved channel names for particle fields

8 Initial conditions

Initial conditions create the distribution of the particles or objects at the beginning of any simulation. Several variants include random placement, reading ICs from Python or restarting from the previous state

8.1 Summary

FromArray()	Set particles according to given position and velocity arrays.
InitialConditions()	Base class for initial conditions
Membrane()	Can only be used with Membrane Object Vector, see *Initial conditions*.
MembraneWithTypeId()	Same as `Membrane` with an additional *type id* field which distinguish membranes with different properties.
RandomChains()	Creates chains of particles with random shapes at prescribed positions.
Restart()	Read the state of the particle vector from restart files.
Rigid()	Can only be used with Rigid Object Vector or Rigid Ellipsoid, see *Initial conditions*.
Rod()	Can only be used with Rod Vector.
StraightChains()	Creates chains of particles of the same orientations and lengths at prescribed positions.
Uniform()	The particles will be generated with the desired number density uniformly at random in all the domain.
UniformFiltered()	The particles will be generated with the desired number density uniformly at random in all the domain and then filtered out by the given filter.
UniformSphere()	The particles will be generated with the desired number density uniformly at random inside or outside a given sphere.

8.2 Details

class FromArray
Bases: *mmirheo.InitialConditions.InitialConditions*

Set particles according to given position and velocity arrays.

__init__ (*pos: List[real3], vel: List[real3]*) → None

Parameters

- **pos** – array of positions
- **vel** – array of velocities

class InitialConditions

Bases: `object`

Base class for initial conditions

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

class Membrane

Bases: *mmirheo.InitialConditions.InitialConditions*

Can only be used with Membrane Object Vector, see *Initial conditions*. These IC will initialize the particles of each object according to the mesh associated with Membrane, and then the objects will be translated/rotated according to the provided initial conditions.

__init__ (*com_q: List[ComQ], global_scale: float=1.0*) → None

Parameters

- **com_q** – List describing location and rotation of the created objects. One entry in the list corresponds to one object created. Each entry consist of 7 reals: <*com_x*> <*com_y*> <*com_z*> <*q_x*> <*q_y*> <*q_z*> <*q_w*>, where *com* is the center of mass of the object, *q* is the quaternion of its rotation, not necessarily normalized
- **global_scale** – All the membranes will be scaled by that value. Useful to implement membranes growth so that they can fill the space with high volume fraction

class MembraneWithTypeId

Bases: *mmirheo.InitialConditions.Membrane*

Same as `Membrane` with an additional *type id* field which distinguish membranes with different properties. This is may be used with `MembraneForces` with the corresponding filter.

__init__ (*com_q: List[ComQ], type_ids: List[int], global_scale: float=1.0*) → None

Parameters

- **com_q** – List describing location and rotation of the created objects. One entry in the list corresponds to one object created. Each entry consist of 7 reals: <*com_x*> <*com_y*> <*com_z*> <*q_x*> <*q_y*> <*q_z*> <*q_w*>, where *com* is the center of mass of the object, *q* is the quaternion of its rotation, not necessarily normalized
- **type_ids** – list of type ids. Each entry corresponds to the id of the group to which the corresponding membrane belongs.
- **global_scale** – All the membranes will be scaled by that value. Useful to implement membranes growth so that they can fill the space with high volume fraction

class RandomChains

Bases: *mmirheo.InitialConditions.InitialConditions*

Creates chains of particles with random shapes at prescribed positions. Each chain is generated by a random walk with a constant step size.

__init__ (*positions: List[real3], length: float*) → None

Parameters

- **positions** – center of mass of each chain
- **length** – length of the chains.

class Restart

Bases: *mmirheo.InitialConditions.InitialConditions*

Read the state of the particle vector from restart files.

__init__ (*path: str='restart/'*) → None

Parameters **path** – folder where the restart files reside.

class Rigid

Bases: *mmirheo.InitialConditions.InitialConditions*

Can only be used with Rigid Object Vector or Rigid Ellipsoid, see *Initial conditions*. These IC will initialize the particles of each object according to the template .xyz file and then the objects will be translated/rotated according to the provided initial conditions.

__init__ (**args*, ***kwargs*)

Overloaded function.

1. __init__(com_q: List[ComQ], xyz_filename: str) -> None

   Args:

   com_q: List describing location and rotation of the created objects. One entry in the list corresponds to one object created. Each entry consist of 7 reals: <*com_x*> <*com_y*> <*com_z*> <*q_x*> <*q_y*> <*q_z*> <*q_w*>, where *com* is the center of mass of the object, *q* is the quaternion of its rotation, not necessarily normalized

   xyz_filename: Template that describes the positions of the body particles before translation or rotation is applied. Standard .xyz file format is used with first line being the number of particles, second comment, third and onwards - particle coordinates. The number of particles in the file must be the same as in number of particles per object in the corresponding PV

2. __init__(com_q: List[ComQ], coords: List[real3]) -> None

   Args:

   com_q: List describing location and rotation of the created objects. One entry in the list corresponds to one object created. Each entry consist of 7 reals: <*com_x*> <*com_y*> <*com_z*> <*q_x*> <*q_y*> <*q_z*> <*q_w*>, where *com* is the center of mass of the object, *q* is the quaternion of its rotation, not necessarily normalized

   coords: Template that describes the positions of the body particles before translation or rotation is applied. The number of coordinates must be the same as in number of particles per object in the corresponding PV

3. __init__(com_q: List[ComQ], coords: List[real3], init_vels: List[real3]) -> None

   Args:

   com_q: List describing location and rotation of the created objects. One entry in the list corresponds to one object created. Each entry consist of 7 reals: <*com_x*> <*com_y*> <*com_z*> <*q_x*> <*q_y*> <*q_z*> <*q_w*>, where *com* is the center of mass of the object, *q* is the quaternion of its rotation, not necessarily normalized

   coords: Template that describes the positions of the body particles before translation or rotation is applied. The number of coordinates must be the same as in number of particles per object in the corresponding PV

com_q: List specifying initial Center-Of-Mass velocities of the bodies. One entry (list of 3 reals) in the list corresponds to one object

class Rod

Bases: *mmirheo.InitialConditions.InitialConditions*

Can only be used with Rod Vector. These IC will initialize the particles of each rod according to the the given explicit center-line position aand torsion mapping and then the objects will be translated/rotated according to the provided initial conditions.

__init__(*com_q: List[ComQ], center_line: Callable[[float], real3], torsion: Callable[[float], float], a: float, initial_frame: real3=real3(inf, inf, inf)*) → None

Parameters

- **com_q** – List describing location and rotation of the created objects. One entry in the list corresponds to one object created. Each entry consist of 7 reals: <*com_x*> <*com_y*> <*com_z*> <*q_x*> <*q_y*> <*q_z*> <*q_w*>, where *com* is the center of mass of the object, *q* is the quaternion of its rotation, not necessarily normalized
- **center_line** – explicit mapping $\mathbf{r} : [0, 1] \to R^3$. Assume $|r'(s)|$ is constant for all $s \in [0, 1]$.
- **torsion** – explicit mapping $\tau : [0, 1] \to R$.
- **a** – width of the rod
- **initial_frame** – Orientation of the initial frame (optional) By default, will come up with any orthogonal frame to the rod at origin

class StraightChains

Bases: *mmirheo.InitialConditions.InitialConditions*

Creates chains of particles of the same orientations and lengths at prescribed positions.

__init__(*positions: List[real3], orientations: List[real3], length: float*) → None

Parameters

- **positions** – center of mass of each chain
- **orientations** – array of unit vectors indicating the orientation of the chains
- **length** – length of the chains.

class Uniform

Bases: *mmirheo.InitialConditions.InitialConditions*

The particles will be generated with the desired number density uniformly at random in all the domain. These IC may be used with any Particle Vector, but only make sense for regular PV.

__init__(*number_density: float*) → None

Parameters **number_density** – target number density

class UniformFiltered

Bases: *mmirheo.InitialConditions.InitialConditions*

The particles will be generated with the desired number density uniformly at random in all the domain and then filtered out by the given filter. These IC may be used with any Particle Vector, but only make sense for regular PV.

__init__(*number_density: float, filter: Callable[[real3], bool]*) → None

Parameters

- **number_density** – target number density

- **filter** – given position, returns True if the particle should be kept

class UniformSphere

Bases: *mmirheo.InitialConditions.InitialConditions*

The particles will be generated with the desired number density uniformly at random inside or outside a given sphere. These IC may be used with any Particle Vector, but only make sense for regular PV.

__init__(*number_density: float*, *center: real3*, *radius: float*, *inside: bool*) → None

Parameters

- **number_density** – target number density
- **center** – center of the sphere
- **radius** – radius of the sphere
- **inside** – whether the particles should be inside or outside the sphere

9 Object belonging checkers

Object belonging checkers serve two purpooses:

1. Split a *ParticleVector* into two disjointed parts (possibly forming a new Particle Vector): the particles that are *inside* any object of the given *ObjectVector* and the particles that are *outside*.
2. Maintain the mentioned *inside-outside* property of the particles in the resulting *ParticleVectors*. Such maintenance is performed periodically, and the particles of, e.g. inner PV that appear to mistakingly be outside of the reference *ObjectVector* will be moved to the outer PV (and viceversa). If one of the PVs was specified as "none", the erroneous particles will be deleted from the simulation.

See also *Mirheo.registerObjectBelongingChecker* and *Mirheo.applyObjectBelongingChecker*.

9.1 Summary

BelongingChecker()	Base class for checking if particles belong to objects
Capsule()	This checker will use the analytical representation of the capsule to detect *inside-outside* status.
Cylinder()	This checker will use the analytical representation of the cylinder to detect *inside-outside* status.
Ellipsoid()	This checker will use the analytical representation of the ellipsoid to detect *inside-outside* status.
Mesh()	This checker will use the triangular mesh associated with objects to detect *inside-outside* status.
Rod()	This checker will detect *inside-outside* status with respect to every segment of the rod, enlarged by a given radius.

9.2 Details

class BelongingChecker

Bases: `object`

Base class for checking if particles belong to objects

__init__()
Initialize self. See help(type(self)) for accurate signature.

class Capsule
Bases: *mmirheo.BelongingCheckers.BelongingChecker*

This checker will use the analytical representation of the capsule to detect *inside-outside* status.

__init__(*name: str*) → None

Parameters **name** – name of the checker

class Cylinder
Bases: *mmirheo.BelongingCheckers.BelongingChecker*

This checker will use the analytical representation of the cylinder to detect *inside-outside* status.

__init__(*name: str*) → None

Parameters **name** – name of the checker

class Ellipsoid
Bases: *mmirheo.BelongingCheckers.BelongingChecker*

This checker will use the analytical representation of the ellipsoid to detect *inside-outside* status.

__init__(*name: str*) → None

Parameters **name** – name of the checker

class Mesh
Bases: *mmirheo.BelongingCheckers.BelongingChecker*

This checker will use the triangular mesh associated with objects to detect *inside-outside* status.

__init__(*name: str*) → None

Parameters **name** – name of the checker

class Rod
Bases: *mmirheo.BelongingCheckers.BelongingChecker*

This checker will detect *inside-outside* status with respect to every segment of the rod, enlarged by a given radius.

__init__(*name: str*, *radius: float*) → None

Parameters

- **name** – name of the checker
- **radius** – radius of the rod

10 Integrators

Integrators are used to advance particle coordinates and velocities in time according to forces acting on them.

10.1 Summary

Integrator()	Base integration class
Minimize()	Energy minimization integrator.

Continued on next page

Table 7 – continued from previous page

Oscillate()	Move particles with the periodically changing velocity $\mathbf{u}(t) = \cos(2\pi\, t/T)\mathbf{u}_0$
RigidVelocityVerlet()	Integrate the position and rotation (in terms of quaternions) of the rigid bodies as per Velocity-Verlet scheme.
Rotate()	Rotate particles around the specified point in space with a constant angular velocity $\mathbf{\Omega}$
SubStep()	Takes advantage of separation of time scales between "fast" internal forces and other "slow" forces on an object vector.
SubStepShardlowSweep()	Takes advantage of separation of time scales between "fast" internal forces and other "slow" forces on a membrane vector.
Translate()	Translate particles with a constant velocity $\mathbf{u}$ regardless forces acting on them.
VelocityVerlet()	Classical Velocity-Verlet integrator with fused steps for coordinates and velocities.
VelocityVerlet_withConstForce()	Same as regular *VelocityVerlet*, but the forces on all the particles are modified with the constant pressure term:
VelocityVerlet_withPeriodicForce()	Same as regular Velocity-Verlet, but the forces on all the particles are modified with periodic Poiseuille term.

10.2 Details

class Integrator

Bases: `object`

Base integration class

__init__()

Initialize self. See help(type(self)) for accurate signature.

class Minimize

Bases: *mmirheo.Integrators.Integrator*

Energy minimization integrator. Updates particle positions according to a gradient-descent policy with respect to the energy potential (force). Does not read or modify particle velocities.

$$\mathbf{a}^n = \frac{1}{m}\mathbf{F}(\mathbf{x}^n, \mathbf{v}^{n-1/2})$$
$$\mathbf{x}^{n+1} = \mathbf{x}^n + \frac{\Delta t^2}{m}\mathbf{a}^n$$

__init__(*name: str*, *max_displacement: float*) → None

Parameters

- **name** – name of the integrator
- **max_displacement** – maximum displacement per time step

class Oscillate

Bases: *mmirheo.Integrators.Integrator*

Move particles with the periodically changing velocity $\mathbf{u}(t) = \cos(2\pi\, t/T)\mathbf{u}_0$

__init__(*name: str*, *velocity: real3*, *period: float*) → None

Parameters

- **name** – name of the integrator
- **velocity** – $\mathbf{u}_0$
- **period** – oscillation period T

class RigidVelocityVerlet

Bases: *mmirheo.Integrators.Integrator*

Integrate the position and rotation (in terms of quaternions) of the rigid bodies as per Velocity-Verlet scheme. Can only applied to *RigidObjectVector* or *RigidEllipsoidVector*.

__init__ (*name: str*) → None

Parameters **name** – name of the integrator

class Rotate

Bases: *mmirheo.Integrators.Integrator*

Rotate particles around the specified point in space with a constant angular velocity $\mathbf{\Omega}$

__init__ (*name: str*, *center: real3*, *omega: real3*) → None

Parameters

- **name** – name of the integrator
- **center** – point around which to rotate
- **omega** – angular velocity $\mathbf{\Omega}$

class SubStep

Bases: *mmirheo.Integrators.Integrator*

Takes advantage of separation of time scales between "fast" internal forces and other "slow" forces on an object vector. This integrator advances the object vector with constant slow forces for 'substeps' sub time steps. The fast forces are updated after each sub step. Positions and velocity are updated using an internal velocity verlet integrator.

__init__ (*name: str, substeps: int, fastForces: List[Interactions.Interaction]*) → None

Parameters

- **name** – name of the integrator
- **substeps** – number of sub steps
- **fastForces** – a list of fast interactions. Only accepts *MembraneForces* or *RodForces*

> **Warning:** The interaction will be set to the required object vector when setting this integrator to the object vector. Hence the interaction needs not to be set explicitely to the OV.

class SubStepShardlowSweep

Bases: *mmirheo.Integrators.Integrator*

Takes advantage of separation of time scales between "fast" internal forces and other "slow" forces on a membrane vector. This integrator advances the object vector with constant slow forces for 'substeps' sub time steps. The fast forces are updated after each sub step using the Shardlow method for viscous forces with multiple seeps.

__init__ (*name: str*, *substeps: int*, *fastForces: Interactions.MembraneForces*, *gammaC: float*, *kBT: float*, *nsweeps: int*) → None

Parameters

- **name** – Name of the integrator.
- **substeps** – Number of sub steps.
- **fastForces** – Membrane interactions. Only accepts *MembraneForces*. Must have zero gammaC and zero kBT.
- **gammaC** – Membrane viscous coefficient.
- **kBT** – temperature, in energy units. Set to zero to disable membrane fluctuations.
- **nsweeps** – Number of sweeps for the semi implicit step. Must be strictly more than 0.

> **Warning:** The interaction will be set to the required object vector when setting this integrator to the object vector. Hence the interaction needs not to be set explicitely to the OV.

class Translate

Bases: *mmirheo.Integrators.Integrator*

Translate particles with a constant velocity $\mathbf{u}$ regardless forces acting on them.

__init__ (*name: str*, *velocity: real3*) → None

Parameters

- **name** – name of the integrator
- **velocity** – translational velocity $\mathbf{\Omega}$

class VelocityVerlet

Bases: *mmirheo.Integrators.Integrator*

Classical Velocity-Verlet integrator with fused steps for coordinates and velocities. The velocities are shifted with respect to the coordinates by one half of the time-step

$$\begin{aligned} \mathbf{a}^n &= \frac{1}{m}\mathbf{F}(\mathbf{x}^n, \mathbf{v}^{n-1/2}) \\ \mathbf{v}^{n+1/2} &= \mathbf{v}^{n-1/2} + \mathbf{a}^n \Delta t \\ \mathbf{x}^{n+1} &= \mathbf{x}^n + \mathbf{v}^{n+1/2} \Delta t \end{aligned}$$

where bold symbol means a vector, m is a particle mass, and superscripts denote the time: $\mathbf{x}^k = \mathbf{x}(k\,\Delta t)$

__init__ (*name: str*) → None

Parameters **name** – name of the integrator

class VelocityVerlet_withConstForce

Bases: *mmirheo.Integrators.Integrator*

Same as regular *VelocityVerlet*, but the forces on all the particles are modified with the constant pressure term:

$$\mathbf{a}^n = \frac{1}{m}\left(\mathbf{F}(\mathbf{x}^n, \mathbf{v}^{n-1/2}) + \mathbf{F}_{extra}\right)$$

__init__ (*name: str*, *force: real3*) → None

Parameters

- **name** – name of the integrator
- **force** – $\mathbf{F}_{extra}$

class VelocityVerlet_withPeriodicForce

Bases: *mmirheo.Integrators.Integrator*

Same as regular Velocity-Verlet, but the forces on all the particles are modified with periodic Poiseuille term. This means that all the particles in half domain along certain axis (Ox, Oy or Oz) are pushed with force $F_{Poiseuille}$ parallel to Oy, Oz or Ox correspondingly, and the particles in another half of the domain are pushed in the same direction with force $-F_{Poiseuille}$

__init__ (*name: str*, *force: float*, *direction: str*) → None

Parameters

- **name** – name of the integrator
- **force** – force magnitude, $F_{Poiseuille}$
- **direction** – Valid values: "x", "y", "z". Defines the direction of the pushing force if direction is "x", the sign changes along "y". if direction is "y", the sign changes along "z". if direction is "z", the sign changes along "x".

11 Interactions

Interactions are used to calculate forces on individual particles due to their neighbours. Pairwise short-range interactions are currently supported, and membrane forces.

11.1 Summary

ChainFENE()	FENE forces between beads of a *ChainVector*.
Interaction()	Base interaction class
MembraneForces()	Abstract class for membrane interactions.
ObjBinding()	Forces attaching a *ParticleVector* to another via harmonic potentials between the particles of specific pairs.
ObjRodBinding()	Forces attaching a *RodVector* to a *RigidObjectVector*.
Pairwise()	Generic pairwise interaction class.
RodForces()	Forces acting on an elastic rod.

11.2 Details

class ChainFENE

Bases: *mmirheo.Interactions.Interaction*

FENE forces between beads of a *ChainVector*.

__init__ (*name: str*, *ks: float*, *rmax: float*, *stress_period: Optional[float]=None*) → None

Parameters

- **name** – name of the interaction

- **ks** – the spring constant
- **rmax** – maximal extension of the springs
- **stress_period** – if set, compute the stresses on particles at this given period, in simulation time.

class Interaction

Bases: object

Base interaction class

__init__()

Initialize self. See help(type(self)) for accurate signature.

class MembraneForces

Bases: *mmirheo.Interactions.Interaction*

Abstract class for membrane interactions. Mesh-based forces acting on a membrane according to the model in [Fedosov2010]

The membrane interactions are composed of forces comming from:

- bending of the membrane, potential U_b
- shear elasticity of the membrane, potential U_s
- constraint: area conservation of the membrane (local and global), potential U_A
- constraint: volume of the cell (assuming incompressible fluid), potential U_V
- membrane viscosity, pairwise force $\mathbf{F}^v$
- membrane fluctuations, pairwise force $\mathbf{F}^R$

The form of the constraint potentials are given by (see [Fedosov2010] for more explanations):

$$U_A = \frac{k_a (A_{tot} - A^0_{tot})^2}{2A^0_{tot}} + \sum_{j \in 1 \dots N_t} \frac{k_d (A_j - A_0)^2}{2A_0},$$

$$U_V = \frac{k_v (V - V^0_{tot})^2}{2V^0_{tot}}.$$

The viscous and dissipation forces are central forces and are the same as DPD interactions with $w(r) = 1$ (no cutoff radius, applied to each bond).

Several bending models are implemented. First, the Kantor enrgy reads (see [kantor1987]):

$$U_b = \sum_{j \in 1 \dots N_s} k_b \left[1 - \cos(\theta_j - \theta_0)\right].$$

The Juelicher energy is (see [Juelicher1996]):

$$U_b = 2k_b \sum_{\alpha=1}^{N_v} \frac{(M_\alpha - C_0)^2}{A_\alpha},$$

$$M_\alpha = \frac{1}{4} \sum_{<i,j>}^{(\alpha)} l_{ij} \theta_{ij}.$$

It is improved with the area-difference model (see [Bian2020]), which is a discretized version of:

$$U_{AD} = \frac{k_{AD} \pi}{2 D_0^2 A_0} \left(\Delta A - \Delta A_0\right)^2.$$

Currently, the stretching and shear energy models are:

WLC model:

$$U_s = \sum_{j \in 1 \dots N_s} \left[\frac{k_s l_m \left(3x_j^2 - 2x_j^3\right)}{4(1 - x_j)} + \frac{k_p}{l_0} \right].$$

Lim model: an extension of the Skalak shear energy (see [Lim2008]).

$$U_{Lim} = \sum_{i=1}^{N_t} (A_0)_i \left(\frac{k_a}{2} \left(\alpha_i^2 + a_3 \alpha_i^3 + a_4 \alpha_i^4\right) + \mu \left(\beta_i + b_1 \alpha_i \beta_i + b_2 \beta_i^2\right)\right),$$

where α and β are the invariants of the strains.

__init__(*name: str*, *shear_desc: str*, *bending_desc: str*, *filter_desc: str='keep_all'*, *stress_free: bool=False*, ***kwargs*) → None

Parameters

- **name** – name of the interaction
- **shear_desc** – a string describing what shear force is used
- **bending_desc** – a string describing what bending force is used
- **filter_desc** – a string describing which membranes are concerned
- **stress_free** – if True, stress Free shape is used for the shear parameters

kwargs:

- **tot_area**: total area of the membrane at equilibrium
- **tot_volume**: total volume of the membrane at equilibrium
- **ka_tot**: constraint energy for total area
- **kv_tot**: constraint energy for total volume
- **kBT**: fluctuation temperature (set to zero will switch off fluctuation forces)
- **gammaC**: dissipative forces coefficient
- **initial_length_fraction**: the size of the membrane increases linearly in time from this fraction of the provided mesh to its full size after grow_until time; the parameters are scaled accordingly with time. If this is set, **grow_until** must also be provided. Default value: 1.
- **grow_until**: the size increases linearly in time from a fraction of the provided mesh to its full size after that time; the parameters are scaled accordingly with time. If this is set, **initial_length_fraction** must also be provided. Default value: 0

Shear Parameters, warm like chain model (set **shear_desc** = 'wlc'):

- **x0**: x_0
- **ks**: energy magnitude for bonds
- **mpow**: m
- **ka**: energy magnitude for local area

Shear Parameters, Lim model (set **shear_desc** = 'Lim'):

- **ka**: k_a, magnitude of stretching force

- **mu**: μ, magnitude of shear force
- **a3**: a_3, non linear part for stretching
- **a4**: a_4, non linear part for stretching
- **b1**: b_1, non linear part for shear
- **b2**: b_2, non linear part for shear

Bending Parameters, Kantor model (set **bending_desc** = 'Kantor'):

- **kb**: local bending energy magnitude
- **theta**: spontaneous angle

Bending Parameters, Juelicher model (set **bending_desc** = 'Juelicher'):

- **kb**: local bending energy magnitude
- **C0**: spontaneous curvature
- **kad**: area difference energy magnitude
- **DA0**: area difference at relaxed state divided by the offset of the leaflet midplanes

filter_desc = "keep_all":

The interaction will be applied to all membranes

filter_desc = "by_type_id":

The interaction will be applied membranes with a given **type_id** (see `MembraneWithTypeId`)

- **type_id**: the type id that the interaction applies to

class ObjBinding

Bases: *mmirheo.Interactions.Interaction*

Forces attaching a *ParticleVector* to another via harmonic potentials between the particles of specific pairs.

> **Warning:** To deal with MPI, the force is zero if two particles of a pair are apart from more than half the subdomain size. Since this interaction is designed to bind objects to each other, this should not happen under normal conditions.

__init__ (*name: str, k_bound: float, pairs: List[int2]*) → None

Parameters

- **name** – Name of the interaction.
- **k_bound** – Spring force coefficient.
- **pairs** – The global Ids of the particles that will interact through the harmonic potential. For each pair, the first entry is the id of pv1 while the second is that of pv2 (see *setInteraction*).

class ObjRodBinding

Bases: *mmirheo.Interactions.Interaction*

Forces attaching a *RodVector* to a *RigidObjectVector*.

__init__ (*name: str*, *torque: float*, *rel_anchor: real3*, *k_bound: float*) → None

Parameters

- **name** – name of the interaction
- **torque** – torque magnitude to apply to the rod
- **rel_anchor** – position of the anchor relative to the rigid object
- **k_bound** – anchor harmonic potential magnitude

class Pairwise

Bases: *mmirheo.Interactions.Interaction*

Generic pairwise interaction class. Can be applied between any kind of *ParticleVector* classes. The following interactions are currently implemented:

- **DPD:** Pairwise interaction with conservative part and dissipative + random part acting as a thermostat, see [Groot1997]

$$\mathbf{F}_{ij} = \left(\mathbf{F}^C_{ij} + \mathbf{F}^D_{ij} + \mathbf{F}^R_{ij}\right)\hat{\mathbf{r}}$$
$$F^C_{ij} = \begin{cases} a(1-\frac{r}{r_c}), & r < r_c \\ 0, & r \geqslant r_c \end{cases}$$
$$F^D_{ij} = -\gamma w^2(\tfrac{r}{r_c})(\hat{\mathbf{r}}\cdot\mathbf{u})$$
$$F^R_{ij} = \sigma w(\tfrac{r}{r_c})\,\frac{\theta}{\sqrt{\Delta t}}$$

  where bold symbol means a vector, its regular counterpart means vector length: $x = \|\mathbf{x}\|$, hat-ed symbol is the normalized vector: $\hat{\mathbf{x}} = \mathbf{x}/\|\mathbf{x}\|$. Moreover, θ is the random variable with zero mean and unit variance, that is distributed independently of the interacting pair *i-j*, dissipation and random forces are related by the fluctuation-dissipation theorem: $\sigma^2 = 2\gamma\, k_B T$; and $w(r)$ is the weight function that we define as follows:

$$w(r) = \begin{cases} (1-r)^p, & r < 1 \\ 0, & r \geqslant 1 \end{cases}$$

- **MDPD:** Compute MDPD interaction as described in [Warren2003]. Must be used together with "Density" interaction with kernel "MDPD".

  The interaction forces are the same as described in "DPD" with the modified conservative term

$$F^C_{ij} = a w_c(r_{ij}) + b(\rho_i + \rho_j) w_d(r_{ij}),$$

  where ρ_i is computed from "Density" and

$$w_c(r) = \begin{cases} (1-\frac{r}{r_c}), & r < r_c \\ 0, & r \geqslant r_c \end{cases}$$
$$w_d(r) = \begin{cases} (1-\frac{r}{r_d}), & r < r_d \\ 0, & r \geqslant r_d \end{cases}$$

- **SDPD:** Compute SDPD interaction with angular momentum conservation, following [Hu2006] and

[Bian2012]. Must be used together with "Density" interaction with the same density kernel.

$$\begin{aligned}
\mathbf{F}_{ij} &= \left(F^C_{ij} + F^D_{ij} + F^R_{ij}\right) \\
F^C_{ij} &= -\left(\frac{p_i}{d_i^2} + \frac{p_j}{d_j^2}\right)\frac{\partial w_\rho}{\partial r_{ij}}, \\
F^D_{ij} &= -\eta\left[\left(\frac{1}{d_i^2} + \frac{1}{d_j^2}\right)\frac{-\zeta}{r_{ij}}\frac{\partial w_\rho}{\partial r_{ij}}\right](\mathbf{v}_{ij}\cdot\mathbf{e}_{ij}), \\
F^R_{ij} &= \sqrt{2k_BT\eta}\left[\left(\frac{1}{d_i^2} + \frac{1}{d_j^2}\right)\frac{-\zeta}{r_{ij}}\frac{\partial w_\rho}{\partial r_{ij}}\right]^{\frac{1}{2}}\xi_{ij},
\end{aligned}$$

where η is the viscosity, w_ρ is the density kernel, $\zeta = 2 + d = 5$, d_i is the density of particle i and $p_i = p(d_i)$ is the pressure of particle i.. The available density kernels are listed in "Density". The available equations of state (EOS) are:

Linear equation of state:

$$p(\rho) = c_S^2\left(\rho - \rho_0\right)$$

where c_S is the speed of sound and ρ_0 is a parameter.

Quasi incompressible EOS:

$$p(\rho) = p_0\left[\left(\frac{\rho}{\rho_r}\right)^\gamma - 1\right],$$

where p_0, ρ_r and $\gamma = 7$ are parameters to be fitted to the desired fluid.

- **LJ:** Pairwise interaction according to the classical Lennard-Jones potential

$$\mathbf{F}_{ij} = 24\epsilon\left(2\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^6\right)\frac{\mathbf{r}}{r^2}$$

  As opposed to `RepulsiveLJ`, the force is not bounded from either sides.

- **RepulsiveLJ:** Pairwise interaction according to the classical Lennard-Jones potential, truncated such that it is *always repulsive*.

$$\mathbf{F}_{ij} = \max\left[0.0, 24\epsilon\left(2\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^6\right)\frac{\mathbf{r}}{r^2}\right]$$

  Note that in the implementation, the force is bounded for stability at larger time steps.

- **GrowingRepulsiveLJ:** Same as **RepulsiveLJ**, but the length scale is growing linearly in time until a prespecified time, from a specified fraction to 1. This is useful when growing membranes while avoiding overlaps.

- **Morse:** Pairwise interaction according to the classical Morse potential

$$\mathbf{F}_{ij} = 2D_e\beta\left(e^{2\beta(r_0-r)} - e^{\beta(r_0-r)}\right)\frac{\mathbf{r}}{r},$$

  where r is the distance between the particles.

- **Density:** Compute density of particles with a given kernel.

$$\rho_i = \sum_{j \neq i} w_\rho(r_{ij})$$

where the summation goes over the neighbours of particle i within a cutoff range of r_c. The implemented densities are listed below:

  - kernel "MDPD":

    see [Warren2003]

$$w_\rho(r) = \begin{cases} \frac{15}{2\pi r_d^3}\left(1 - \frac{r}{r_d}\right)^2, & r < r_d \\ 0, & r \geqslant r_d \end{cases}$$

  - kernel "WendlandC2":

$$w_\rho(r) = \frac{21}{2\pi r_c^3}\left(1 - \frac{r}{r_c}\right)^4 \left(1 + 4\frac{r}{r_c}\right)$$

__init__ (*name: str*, *rc: float*, *kind: str*, ***kwargs*) → None

Parameters

- **name** – name of the interaction
- **rc** – interaction cut-off (no forces between particles further than **rc** apart)
- **kind** – interaction kind (e.g. DPD). See below for all possibilities.

Create one pairwise interaction handler of kind **kind**. When applicable, stress computation is activated by passing **stress = True**. This activates virial stress computation every **stress_period** time units (also passed in **kwars**)

- **kind** = "DPD"
  - **a**: a
  - **gamma**: γ
  - **kBT**: $k_B T$
  - **power**: p in the weight function
- **kind** = "MDPD"
  - **rd**: r_d
  - **a**: a
  - **b**: b
  - **gamma**: γ
  - **kBT**: temperature $k_B T$
  - **power**: p in the weight function
- **kind** = "SDPD"
  - **viscosity**: fluid viscosity
  - **kBT**: temperature $k_B T$

  - **EOS**: the desired equation of state (see below)
  - **density_kernel**: the desired density kernel (see below)
- **kind** = "LJ"
  - **epsilon**: ε
  - **sigma**: σ
- **kind** = "RepulsiveLJ"
  - **epsilon**: ε
  - **sigma**: σ
  - **max_force**: force magnitude will be capped to not exceed **max_force**
  - **aware_mode:**
    * if "None", all particles interact with each other.
    * if "Object", the particles belonging to the same object in an object vector do not interact with each other. That restriction only applies if both Particle Vectors in the interactions are the same and is actually an Object Vector.
    * if "Rod", the particles interact with all other particles except with the ones which are below a given a distance (in number of segment) of the same rod vector. The distance is specified by the kwargs parameter **min_segments_distance**.
- **kind** = "GrowingRepulsiveLJ"
  - **epsilon**: ε
  - **sigma**: σ
  - **max_force**: force magnitude will be capped to not exceed **max_force**
  - **aware_mode:**
    * if "None", all particles interact with each other.
    * if "Object", the particles belonging to the same object in an object vector do not interact with each other. That restriction only applies if both Particle Vectors in the interactions are the same and is actually an Object Vector.
    * if "Rod", the particles interact with all other particles except with the ones which are below a given a distance (in number of segment) of the same rod vector. The distance is specified by the kwargs parameter **min_segments_distance**.
  - **init_length_fraction**: tnitial length factor. Must be in [0, 1].
  - **grow_until**: time after which the length quantities are scaled by one.
- **kind** = "Morse"
  - **De**: D_e
  - **r0**: r_0
  - **beta**: β
  - **aware_mode**: See "RepulsiveLJ" kernel description.
- **kind** = "Density"
  - **density_kernel**: the desired density kernel (see below)

The available density kernels are “MDPD” and “WendlandC2”. Note that “MDPD” can not be used with SDPD interactions. MDPD interactions can use only “MDPD” density kernel.

For SDPD, the available equation of states are given below:

- **EOS** = “Linear” parameters:
  - **sound_speed**: the speed of sound
  - **rho_0**: background pressure in c_S units
- **EOS** = “QuasiIncompressible” parameters:
  - **p0**: p_0
  - **rho_r**: ρ_r

class RodForces

Bases: *mmirheo.Interactions.Interaction*

Forces acting on an elastic rod.

The rod interactions are composed of forces comming from:

- bending energy, E_{bend}
- twist energy, E_{twist}
- bounds energy, E_{bound}

The form of the bending energy is given by (for a bi-segment):

$$E_{\text{bend}} = \frac{l}{4} \sum_{j=0}^{1} \left(\kappa^j - \overline{\kappa}\right)^T B \left(\kappa^j - \overline{\kappa}\right),$$

where

$$\kappa^j = \frac{1}{l} \left((\kappa \mathbf{b}) \cdot \mathbf{m}_2^j, -(\kappa \mathbf{b}) \cdot \mathbf{m}_1^j \right).$$

See, e.g. [bergou2008] for more details. The form of the twist energy is given by (for a bi-segment):

$$E_{\text{twist}} = \frac{k_t l}{2} \left(\frac{\theta^1 - \theta^0}{l} - \overline{\tau} \right)^2.$$

The additional bound energy is a simple harmonic potential with a given equilibrium length.

__init__(*name: str*, *state_update: str='none'*, *save_energies: bool=False*, ***kwargs*) → None

Parameters

- **name** – name of the interaction
- **state_update** – description of the state update method; only makes sense for multiple states. See below for possible choices.
- **save_energies** – if *True*, save the energies of each bisegment

kwargs:

- **a0** (real): equilibrium length between 2 opposite cross vertices
- **l0** (real): equilibrium length between 2 consecutive vertices on the centerline
- **k_s_center** (real): elastic force magnitude for centerline
- **k_s_frame** (real): elastic force magnitude for material frame particles

- **k_bending** (real3): Bending symmetric tensor B in the order (B_{xx}, B_{xy}, B_{zz})
- **kappa0** (real2): Spontaneous curvatures along the two material frames $\overline{\kappa}$
- **k_twist** (real): Twist energy magnitude k_{twist}
- **tau0** (real): Spontaneous twist $\overline{\tau}$
- **E0** (real): (optional) energy ground state

state update parameters, for **state_update** = 'smoothing':

(not fully implemented yet; for now just takes minimum state but no smoothing term)

state update parameters, for **state_update** = 'spin':

- **nsteps** number of MC step per iteration
- **kBT** temperature used in the acceptance-rejection algorithm
- **J** neighbouring spin 'dislike' energy

The interaction can support multiple polymorphic states if **kappa0**, **tau0** and **E0** are lists of equal size. In this case, the **E0** parameter is required. Only lists of 1, 2 and 11 states are supported.

12 Object bouncers

Bouncers prevent particles from crossing boundaries of objects (maintaining no-through boundary conditions). The idea of the bouncers is to move the particles that crossed the object boundary after integration step back to the correct side. Particles are moved such that they appear very close (about 10^{-5} units away from the boundary). Assuming that the objects never come too close to each other or the walls, this approach ensures that recovered particles will not penetrate into a different object or wall. In practice maintaining separation of at least 10^{-3} units between walls and objects is sufficient. Note that particle velocities are also altered, which means that objects experience extra force from the collisions.

See also *Mirheo.registerBouncer* and *Mirheo.setBouncer*.

12.1 Summary

Bouncer()	Base class for bouncing particles off the objects.
Capsule()	This bouncer will use the analytical capsule representation of the rigid objects to perform the bounce.
Cylinder()	This bouncer will use the analytical cylinder representation of the rigid objects to perform the bounce.
Ellipsoid()	This bouncer will use the analytical ellipsoid representation of the rigid objects to perform the bounce.
Mesh()	This bouncer will use the triangular mesh associated with objects to detect boundary crossings.
Rod()	This bouncer will use the analytical representation of enlarged segments by a given radius.

12.2 Details

class Bouncer

Bases: `object`

Base class for bouncing particles off the objects. Take bounce kernel as argument:

- **kernel = "bounce_back":** Bounces back the particle. The new velocity of the particle is given by:

$$\mathbf{u}_{\text{new}} = \mathbf{u}_{\text{wall}} - \left(\mathbf{u}_{\text{old}} - \mathbf{u}_{\text{wall}}\right).$$

- **kernel = "bounce_maxwell":** Reinsert particle at the collision point with a velocity drawn from a maxwellian distribution. Need the additional parameter **kBT (real)**. The new velocity of the particle is given by:

$$\mathbf{u}_{\text{new}} = \mathbf{u}_{\text{wall}} + \sqrt{\frac{k_B T}{m}}\xi,$$

  where $\xi \sim \mathcal{N}(0, 1)$.

__init__ ()
Initialize self. See help(type(self)) for accurate signature.

class Capsule
Bases: *mmirheo.Bouncers.Bouncer*

This bouncer will use the analytical capsule representation of the rigid objects to perform the bounce. No additional correction from the Object Belonging Checker is usually required. The velocity of the particles bounced from the cylinder is reversed with respect to the boundary velocity at the contact point.

__init__ (*name: str*, *kernel: str*, *verbosity: int=0*, ***kwargs*) → None

Parameters

- **name** – name of the checker
- **kernel** – the kernel used to bounce the particles (see *Bouncer*)
- **verbosity** – 0 for no warning, 1 to display rescue failures, 2 to display all warnings

class Cylinder
Bases: *mmirheo.Bouncers.Bouncer*

This bouncer will use the analytical cylinder representation of the rigid objects to perform the bounce. No additional correction from the Object Belonging Checker is usually required. The velocity of the particles bounced from the cylinder is reversed with respect to the boundary velocity at the contact point.

__init__ (*name: str*, *kernel: str*, *verbosity: int=0*, ***kwargs*) → None

Parameters

- **name** – name of the checker
- **kernel** – the kernel used to bounce the particles (see *Bouncer*)
- **verbosity** – 0 for no warning, 1 to display rescue failures, 2 to display all warnings

class Ellipsoid
Bases: *mmirheo.Bouncers.Bouncer*

This bouncer will use the analytical ellipsoid representation of the rigid objects to perform the bounce. No additional correction from the Object Belonging Checker is usually required. The velocity of the particles bounced from the ellipsoid is reversed with respect to the boundary velocity at the contact point.

__init__ (*name: str*, *kernel: str*, *verbosity: int=0*, ***kwargs*) → None

Parameters

- **name** – name of the checker

- **kernel** – the kernel used to bounce the particles (see *Bouncer*)
- **verbosity** – 0 for no warning, 1 to display rescue failures, 2 to display all warnings

class Mesh

Bases: *mmirheo.Bouncers.Bouncer*

This bouncer will use the triangular mesh associated with objects to detect boundary crossings. Therefore it can only be created for Membrane and Rigid Object types of object vectors. Due to numerical precision, about 1 of $10^5 - 10^6$ mesh crossings will not be detected, therefore it is advised to use that bouncer in conjunction with correction option provided by the Object Belonging Checker, see *Object belonging checkers*.

Note: In order to prevent numerical instabilities in case of light membrane particles, the new velocity of the bounced particles will be a random vector drawn from the Maxwell distibution of given temperature and added to the velocity of the mesh triangle at the collision point.

__init__ (*name: str*, *kernel: str*, ***kwargs*) → None

Parameters

- **name** – name of the bouncer
- **kernel** – the kernel used to bounce the particles (see *Bouncer*)

class Rod

Bases: *mmirheo.Bouncers.Bouncer*

This bouncer will use the analytical representation of enlarged segments by a given radius. The velocity of the particles bounced from the segments is reversed with respect to the boundary velocity at the contact point.

__init__ (*name: str*, *radius: float*, *kernel: str*, ***kwargs*) → None

Parameters

- **name** – name of the checker
- **radius** – radius of the segments
- **kernel** – the kernel used to bounce the particles (see *Bouncer*)

13 Walls

Walls are used to represent time-independent stationary boundary conditions for the flows. They are described in the form of a signed distance function, such that a zero-level isosurface defines the wall surface. No slip and no through boundary conditions are enforced on that surface by bouncing the particles off the wall surface.

In order to prevent undesired density oscillations near the walls, so called frozen particles are used. These non-moving particles reside inside the walls and interact with the regular liquid particles. If the density and distribution of the frozen particles is the same as of the corresponding liquid particles, the density oscillations in the liquid in proximity of the wall is minimal. The frozen particles have to be created based on the wall in the beginning of the simulation, see *Mirheo.makeFrozenWallParticles*.

In the beginning of the simulation all the particles defined in the simulation (even not attached to the wall by *Mirheo*) will be checked against all of the walls. Those inside the wall as well as objects partly inside the wall will be deleted. The only exception are the frozen PVs, created by the *Mirheo.makeFrozenWallParticles* or the PVs manually set to be treated as frozen by *Wall.attachFrozenParticles*

13.1 Summary

Box()	Rectangular cuboid wall with edges aligned with the coordinate axes.
Cylinder()	Cylindrical infinitely stretching wall, the main axis is aligned along OX or OY or OZ
MovingPlane()	Planar wall that is moving along itself with constant velocity.
OscillatingPlane()	Planar wall that is moving along itself with periodically changing velocity:
Plane()	Planar infinitely stretching wall.
RotatingCylinder()	Cylindrical wall rotating with constant angular velocity along its axis.
SDF()	This wall is based on an arbitrary Signed Distance Function (SDF) defined in the simulation domain on a regular Cartesian grid.
Sphere()	Spherical wall.
Wall()	Base wall class.

13.2 Details

class Box

Bases: *mmirheo.Walls.Wall*

Rectangular cuboid wall with edges aligned with the coordinate axes.

__init__(*name: str*, *low: real3*, *high: real3*, *inside: bool=False*) → None

Parameters

- **name** – name of the wall
- **low** – lower corner of the box
- **high** – higher corner of the box
- **inside** – whether the domain is inside the box or outside of it

attachFrozenParticles(*arg0: ParticleVectors.ParticleVector*) → None

Let the wall know that the following *ParticleVector* should be treated as frozen. As a result, its particles will not be removed from the inside of the wall.

class Cylinder

Bases: *mmirheo.Walls.Wall*

Cylindrical infinitely stretching wall, the main axis is aligned along OX or OY or OZ

__init__(*name: str*, *center: real2*, *radius: float*, *axis: str*, *inside: bool=False*) → None

Parameters

- **name** – name of the wall
- **center** – point that belongs to the cylinder axis projected along that axis
- **radius** – cylinder radius
- **axis** – direction of cylinder axis, valid values are “x”, “y” or “z”
- **inside** – whether the domain is inside the cylinder or outside of it

attachFrozenParticles (*arg0: ParticleVectors.ParticleVector*) → None
Let the wall know that the following `ParticleVector` should be treated as frozen. As a result, its particles will not be removed from the inside of the wall.

class MovingPlane
Bases: `mmirheo.Walls.Wall`

Planar wall that is moving along itself with constant velocity. Can be used to produce Couette velocity profile in combination with The boundary conditions on such wall are no-through and constant velocity (specified).

__init__ (*name: str*, *normal: real3*, *pointThrough: real3*, *velocity: real3*) → None

Parameters

- **name** – name of the wall
- **normal** – wall normal, pointing *inside* the wall
- **pointThrough** – point that belongs to the plane
- **velocity** – wall velocity, should be orthogonal to the normal

attachFrozenParticles (*arg0: ParticleVectors.ParticleVector*) → None
Let the wall know that the following `ParticleVector` should be treated as frozen. As a result, its particles will not be removed from the inside of the wall.

class OscillatingPlane
Bases: `mmirheo.Walls.Wall`

Planar wall that is moving along itself with periodically changing velocity:

$$\mathbf{u}(t) = cos(2 * \pi * t/T);$$

__init__ (*name: str*, *normal: real3*, *pointThrough: real3*, *velocity: real3*, *period: float*) → None

Parameters

- **name** – name of the wall
- **normal** – wall normal, pointing *inside* the wall
- **pointThrough** – point that belongs to the plane
- **velocity** – velocity amplitude, should be orthogonal to the normal
- **period** – oscillation period dpd time units

attachFrozenParticles (*arg0: ParticleVectors.ParticleVector*) → None
Let the wall know that the following `ParticleVector` should be treated as frozen. As a result, its particles will not be removed from the inside of the wall.

class Plane
Bases: `mmirheo.Walls.Wall`

Planar infinitely stretching wall. Inside is determined by the normal direction .

__init__ (*name: str*, *normal: real3*, *pointThrough: real3*) → None

Parameters

- **name** – name of the wall
- **normal** – wall normal, pointing *inside* the wall
- **pointThrough** – point that belongs to the plane

attachFrozenParticles(*arg0: ParticleVectors.ParticleVector*) → None
Let the wall know that the following *ParticleVector* should be treated as frozen. As a result, its particles will not be removed from the inside of the wall.

class RotatingCylinder
Bases: *mmirheo.Walls.Wall*

Cylindrical wall rotating with constant angular velocity along its axis.

__init__(*name: str*, *center: real2*, *radius: float*, *axis: str*, *omega: float*, *inside: bool=False*) → None

Parameters

- **name** – name of the wall
- **center** – point that belongs to the cylinder axis projected along that axis
- **radius** – cylinder radius
- **axis** – direction of cylinder axis, valid values are "x", "y" or "z"
- **omega** – angular velocity of rotation along the cylinder axis
- **inside** – whether the domain is inside the cylinder or outside of it

attachFrozenParticles(*arg0: ParticleVectors.ParticleVector*) → None
Let the wall know that the following *ParticleVector* should be treated as frozen. As a result, its particles will not be removed from the inside of the wall.

class SDF
Bases: *mmirheo.Walls.Wall*

This wall is based on an arbitrary Signed Distance Function (SDF) defined in the simulation domain on a regular Cartesian grid. The wall reads the SDF data from a custom format `.sdf` file, that has a special structure.

First two lines define the header: three real number separated by spaces govern the size of the domain where the SDF is defined, and next three integer numbers ($Nx\ Ny\ Nz$) define the resolution. Next the $Nx \times Ny \times Nz$ single precision realing point values are written (in binary representation).

Negative SDF values correspond to the domain, and positive – to the inside of the wall. The boundary is defined by the zero-level isosurface.

__init__(*name: str*, *sdfFilename: str*, *h: real3=real3(0.25, 0.25, 0.25)*, *margin: real3=real3(5.0, 5.0, 5.0)*) → None

Parameters

- **name** – name of the wall
- **sdfFilename** – name of the `.sdf` file
- **h** – resolution of the resampled SDF. In order to have a more accurate SDF representation, the initial function is resampled on a finer grid. The lower this value is, the more accurate the wall will be represented, however, the more memory it will consume and the slower the execution will be.
- **margin** – Additional margin to store on each rank. This is used to e.g. bounce-back particles that are on the local rank but outside the local domain.

attachFrozenParticles(*arg0: ParticleVectors.ParticleVector*) → None
Let the wall know that the following *ParticleVector* should be treated as frozen. As a result, its particles will not be removed from the inside of the wall.

class Sphere
Bases: *mmirheo.Walls.Wall*

Spherical wall.

__init__(*name: str*, *center: real3*, *radius: float*, *inside: bool=False*) → None

Parameters

- **name** – name of the wall
- **center** – sphere center
- **radius** – sphere radius
- **inside** – whether the domain is inside the sphere or outside of it

attachFrozenParticles(*arg0: ParticleVectors.ParticleVector*) → None

Let the wall know that the following `ParticleVector` should be treated as frozen. As a result, its particles will not be removed from the inside of the wall.

class Wall

Bases: `object`

Base wall class.

__init__()

Initialize self. See help(type(self)) for accurate signature.

attachFrozenParticles(*arg0: ParticleVectors.ParticleVector*) → None

Let the wall know that the following `ParticleVector` should be treated as frozen. As a result, its particles will not be removed from the inside of the wall.

14 Plugins

Plugins are used to add specific data processing or to modify the regular pipeline in certain way. However, the functionality they provide is not considered essential.

If the simulation is started without postprocess part (see *Overview*), most of the plugins are disabled.

14.1 Summary

Classes

`PinObject`	Contains the special value *Unrestricted* for unrestricted axes in `createPinObject`.
`PostprocessPlugin`	Base postprocess plugin class
`SimulationPlugin`	Base simulation plugin class

Creation functions

`createAddForce`(state, name, pv, force)	This plugin will add constant force $\mathbf{F}_{extra}$ to each particle of a specific PV every time-step.
`createAddFourRollMillForce`(state, name, pv, ...)	This plugin will add a force $\mathbf{f} = (A \sin x \cos y, A \cos x \sin y, 0)$ to each particle of a specific PV every time-step.

Continued on next page

Table 12 – continued from previous page

createAddTorque(state, name, ov, torque)	This plugin will add constant torque $\mathbf{T}_{extra}$ to each *object* of a specific OV every time-step.
createAnchorParticles(state, name, pv, ...)	This plugin will set a given particle at a given position and velocity.
createBerendsenThermostat(state, name, pvs, ...)	Berendsen thermostat.
createDensityControl(state, name, file_name, ...)	This plugin applies forces to a set of particle vectors in order to get a constant density.
createDensityOutlet(state, name, pvs, ...)	This plugin removes particles from a set of ParticleVector in a given region if the number density is larger than a given target.
createDumpAverage(state, name, pvs, ...)	This plugin will project certain quantities of the particle vectors on the grid (by simple binning), perform time-averaging of the grid and dump it in XDMF format with HDF5 backend.The quantities of interest are represented as *channels* associated with particles vectors.Some interactions, integrators, etc.and more notable plug-ins can add to the Particle Vectors per-particles arrays to hold different values.These arrays are called *channels*.Any such channel may be used in this plug-in, however, user must explicitely specify the type of values that the channel holds.Particle number density is used to correctly average the values, so it will be sampled and written in any case into the field "number_densities"..
createDumpAverageRelative(state, name, pvs, ...)	This plugin acts just like the regular flow dumper, with one difference.
createDumpMesh(state, name, ov, dump_every, path)	This plugin will write the meshes of all the object of the specified Object Vector in a PLY format.
createDumpObjectStats(state, name, ov, ...)	This plugin will write the coordinates of the centers of mass of the objects of the specified Object Vector.
createDumpParticles(state, name, pv, ...)	This plugin will dump positions, velocities and optional attached data of all the particles of the specified Particle Vector.
createDumpParticlesWithMesh(state, name, ov, ...)	This plugin will dump positions, velocities and optional attached data of all the particles of the specified Object Vector, as well as connectivity information.
createDumpParticlesWithPolylines(state, ...)	This plugin will dump positions, velocities and optional attached data of all the particles of the specified ChainVector, as well as connectivity information representing polylines.
createDumpXYZ(state, name, pv, dump_every, path)	This plugin will dump positions of all the particles of the specified Particle Vector in the XYZ format.
createExchangePVSFluxPlane(state, name, pv1, ...)	This plugin exchanges particles from a particle vector crossing a given plane to another particle vector.
createExternalMagneticTorque(state, name, ...)	This plugin gives a magnetic moment $\mathbf{M}$ to every rigid objects in a given RigidObjectVector.
createForceSaver(state, name, pv)	This plugin creates an extra channel per particle inside the given particle vector named 'forces'.

Continued on next page

Table 12 – continued from previous page

createImposeProfile(state, name, pv, low, ...)	This plugin will set the velocity of each particle inside a given domain to a target velocity with an additive term drawn from Maxwell distribution of the given temperature.
createImposeVelocity(state, name, pvs, ...)	This plugin will add velocity to all the particles of the target PV in the specified area (rectangle) such that the average velocity equals to desired.
createMagneticDipoleInteractions(state, ...)	This plugin computes the forces and torques resulting from pairwise dipole-dipole interactions between rigid objects.
createMembraneExtraForce(state, name, pv, forces)	This plugin adds a given external force to a given membrane.
createMsd(state, name, pv, start_time, ...)	This plugin computes the mean square displacement of th particles of a given ParticleVector.
createParticleChannelAverager(state, name, ...)	This plugin averages a channel (per particle data) inside the given particle vector and saves it to a new channel.
createParticleChannelSaver(state, name, pv, ...)	This plugin creates an extra channel per particle inside the given particle vector with a given name.
createParticleChecker(state, name, check_every)	This plugin will check the positions and velocities of all particles in the simulation every given time steps.
createParticleDisplacement(state, name, pv, ...)	This plugin computes and save the displacement of the particles within a given particle vector.
createParticleDrag(state, name, pv, drag)	This plugin will add drag force $\mathbf{f} = -C_d\mathbf{u}$ to each particle of a specific PV every time-step.
createPinObject(state, name, ov, dump_every, ...)	This plugin will impose given velocity as the center of mass velocity (by axis) of all the objects of the specified Object Vector.
createPinRodExtremity(state, name, rv, ...)	This plugin adds a force on a given segment of all the rods in a RodVector.
createPlaneOutlet(state, name, pvs, plane)	This plugin removes all particles from a set of ParticleVector that are on the non-negative side of a given plane.
createRateOutlet(state, name, pvs, ...)	This plugin removes particles from a set of ParticleVector in a given region at a given mass rate.
createRdf(state, name, pv, max_dist, nbins, ...)	Compute the radial distribution function (RDF) of a given ParticleVector.
createStats(state, name, every, pvs, filename)	This plugin will report aggregate quantities of all the particles in the simulation: total number of particles in the simulation, average temperature and momentum, maximum velocity magnutide of a particle and also the mean real time per step in milliseconds.
createTemperaturize(state, name, pv, kBT, ...)	This plugin changes the velocity of each particles from a given ParticleVector.
createVacf(state, name, pv, start_time, ...)	This plugin computes the mean velocity autocorrelation over time from a given ParticleVector.
createVelocityControl(state, name, filename, ...)	This plugin applies a uniform force to all the particles of the target PVS in the specified area (rectangle).
createVelocityInlet(state, name, pv, ...)	This plugin inserts particles in a given ParticleVector.

Continued on next page

Table 12 – continued from previous page

createVirialPressurePlugin(state, name, pv, ...)	This plugin computes the virial pressure from a given *ParticleVector*.
createWallForceCollector(state, name, wall, ...)	This plugin collects and averages the total force exerted on a given wall.
createWallRepulsion(state, name, pv, wall, ...)	This plugin will add force on all the particles that are nearby a specified wall.

14.2 Details

class PinObject

Bases: *mmirheo.Plugins.SimulationPlugin*

Contains the special value *Unrestricted* for unrestricted axes in *createPinObject*.

class PostprocessPlugin

Bases: object

Base postprocess plugin class

class SimulationPlugin

Bases: object

Base simulation plugin class

createAddForce(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *force: real3*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will add constant force $\mathbf{F}_{extra}$ to each particle of a specific PV every time-step. Is is advised to only use it with rigid objects, since Velocity-Verlet integrator with constant pressure can do the same without any performance penalty.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with
- **force** – extra force

createAddFourRollMillForce(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *intensity: float*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will add a force $\mathbf{f} = (A \sin x \cos y, A \cos x \sin y, 0)$ to each particle of a specific PV every time-step.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with
- **intensity** – The intensity of the force

createAddTorque(*state: MirState*, *name: str*, *ov: ParticleVectors.ParticleVector*, *torque: real3*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will add constant torque $\mathbf{T}_{extra}$ to each *object* of a specific OV every time-step.

Parameters

- **name** – name of the plugin
- **ov** – *ObjectVector* that we'll work with
- **torque** – extra torque (per object)

createAnchorParticles(*state: MirState, name: str, pv: ParticleVectors.ParticleVector, positions: Callable[[float], List[real3]], velocities: Callable[[float], List[real3]], pids: List[int], report_every: int, path: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will set a given particle at a given position and velocity.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with
- **positions** – positions (at given time) of the particles
- **velocities** – velocities (at given time) of the particles
- **pids** – global ids of the particles in the given particle vector
- **report_every** – report the time averaged force acting on the particles every this amount of timesteps
- **path** – folder where to dump the stats

createBerendsenThermostat(*state: MirState, name: str, pvs: List[ParticleVectors.ParticleVector], tau: float, kBT: float, increaseIfLower: bool=True*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

Berendsen thermostat.

On each time step the velocities of all particles in given particle vectors are multiplied by the following factor:

$$\lambda = \sqrt{1 + \frac{\Delta t}{\tau}\left(\frac{T_0}{T} - 1\right)}$$

where Δt is a time step, τ relaxation time, T current temperature, T_0 target temperature.

Reference: Berendsen et al. (1984)

Parameters

- **name** – name of the plugin
- **pvs** – list of *ParticleVector* objects to apply the thermostat to
- **tau** – relaxation time τ
- **kBT** – target thermal energy $k_B T_0$
- **increaseIfLower** – whether to increase the temperature if it's lower than the target temperature

createDensityControl(*state: MirState, name: str, file_name: str, pvs: List[ParticleVectors.ParticleVector], target_density: float, region: Callable[[real3], float], resolution: real3, level_lo: float, level_hi: float, level_space: float, Kp: float, Ki: float, Kd: float, tune_every: int, dump_every: int, sample_every: int*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin applies forces to a set of particle vectors in order to get a constant density.

Parameters

- **name** – name of the plugin
- **file_name** – output filename
- **pvs** – list of *ParticleVector* that we'll work with

- **target_density** – target number density (used only at boundaries of level sets)
- **region** – a scalar field which describes how to subdivide the domain. It must be continuous and differentiable, as the forces are in the gradient direction of this field
- **resolution** – grid resolution to represent the region field
- **level_lo** – lower level set to apply the controller on
- **level_hi** – highest level set to apply the controller on
- **level_space** – the size of one subregion in terms of level sets
- **Ki, Kd** (*Kp,*) – pid control parameters
- **tune_every** – update the forces every this amount of time steps
- **dump_every** – dump densities and forces in file `filename`
- **sample_every** – sample to average densities every this amount of time steps

createDensityOutlet (*state: MirState, name: str, pvs: List[ParticleVectors.ParticleVector], number_density: float, region: Callable[[real3], float], resolution: real3*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin removes particles from a set of *ParticleVector* in a given region if the number density is larger than a given target.

Parameters

- **name** – name of the plugin
- **pvs** – list of *ParticleVector* that we'll work with
- **number_density** – maximum number_density in the region
- **region** – a function that is negative in the concerned region and positive outside
- **resolution** – grid resolution to represent the region field

createDumpAverage (*state: MirState, name: str, pvs: List[ParticleVectors.ParticleVector], sample_every: int, dump_every: int, bin_size: real3=real3(1.0, 1.0, 1.0), channels: List[str], path: str='xdmf/'*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will project certain quantities of the particle vectors on the grid (by simple binning), perform time-averaging of the grid and dump it in XDMF format with HDF5 backend. The quantities of interest are represented as *channels* associated with particles vectors. Some interactions, integrators, etc. and more notable plug-ins can add to the Particle Vectors per-particles arrays to hold different values. These arrays are called *channels*. Any such channel may be used in this plug-in, however, user must explicitely specify the type of values that the channel holds. Particle number density is used to correctly average the values, so it will be sampled and written in any case into the field "number_densities".

Note: This plugin is inactive if postprocess is disabled

Parameters

- **name** – name of the plugin
- **pvs** – list of *ParticleVector* that we'll work with
- **sample_every** – sample quantities every this many time-steps
- **dump_every** – write files every this many time-steps
- **bin_size** – bin size for sampling. The resulting quantities will be *cell-centered*

- **path** – Path and filename prefix for the dumps. For every dump two files will be created: <path>_NNNNN.xmf and <path>_NNNNN.h5
- **channels** – list of channel names. See *Reserved names*.

createDumpAverageRelative(*state: MirState, name: str, pvs: List[ParticleVectors.ParticleVector], relative_to_ov: ParticleVectors.ObjectVector, relative_to_id: int, sample_every: int, dump_every: int, bin_size: real3=real3(1.0, 1.0, 1.0), channels: List[str], path: str='xdmf/'*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin acts just like the regular flow dumper, with one difference. It will assume a coordinate system attached to the center of mass of a specific object. In other words, velocities and coordinates sampled correspond to the object reference frame.

Note: Note that this plugin needs to allocate memory for the grid in the full domain, not only in the corresponding MPI subdomain. Therefore large domains will lead to running out of memory

---

Note: This plugin is inactive if postprocess is disabled

The arguments are the same as for createDumpAverage() with a few additions:

Parameters

- **name** – name of the plugin
- **pvs** – list of *ParticleVector* that we'll work with
- **sample_every** – sample quantities every this many time-steps
- **dump_every** – write files every this many time-steps
- **bin_size** – bin size for sampling. The resulting quantities will be *cell-centered*
- **path** – Path and filename prefix for the dumps. For every dump two files will be created: <path>_NNNNN.xmf and <path>_NNNNN.h5
- **channels** – list of channel names. See *Reserved names*.
- **relative_to_ov** – take an object governing the frame of reference from this *ObjectVector*
- **relative_to_id** – take an object governing the frame of reference with the specific ID

createDumpMesh(*state: MirState*, *name: str*, *ov: ParticleVectors.ObjectVector*, *dump_every: int*, *path: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will write the meshes of all the object of the specified Object Vector in a PLY format.

Note: This plugin is inactive if postprocess is disabled

Parameters

- **name** – name of the plugin
- **ov** – *ObjectVector* that we'll work with
- **dump_every** – write files every this many time-steps
- **path** – the files will look like this: <path>/<ov_name>_NNNNN.ply

createDumpObjectStats(*state: MirState*, *name: str*, *ov: ParticleVectors.ObjectVector*, *dump_every: int*, *filename: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will write the coordinates of the centers of mass of the objects of the specified Object Vector. Instantaneous quantities (COM velocity, angular velocity, force, torque) are also written. If the objects are rigid bodies, also will be written the quaternion describing the rotation. The *type id* field is also dumped if the objects have this field activated (see `MembraneWithTypeId`).

The file format is the following:

<object id> <simulation time> <COM>x3 [<quaternion>x4] <velocity>x3 <angular velocity>x3 <force>x3 <torque>x3 [<type id>]

Note: Note that all the written values are *instantaneous*

Note: This plugin is inactive if postprocess is disabled

Parameters

- **name** – Name of the plugin.
- **ov** – *ObjectVector* that we'll work with.
- **dump_every** – Write files every this many time-steps.
- **filename** – The name of the resulting csv file.

createDumpParticles(*state: MirState, name: str, pv: ParticleVectors.ParticleVector, dump_every: int, channel_names: List[str], path: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will dump positions, velocities and optional attached data of all the particles of the specified Particle Vector. The data is dumped into hdf5 format. An additional xdfm file is dumped to describe the data and make it readable by visualization tools. If a channel from object data or bisegment data is provided, the data will be scattered to particles before being dumped as normal particle data.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with
- **dump_every** – write files every this many time-steps
- **channel_names** – list of channel names to be dumped.
- **path** – Path and filename prefix for the dumps. For every dump two files will be created: <path>_NNNNN.xmf and <path>_NNNNN.h5

createDumpParticlesWithMesh(*state: MirState, name: str, ov: ParticleVectors.ObjectVector, dump_every: int, channel_names: List[str], path: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will dump positions, velocities and optional attached data of all the particles of the specified Object Vector, as well as connectivity information. The data is dumped into hdf5 format. An additional xdfm file is dumped to describe the data and make it readable by visualization tools.

Parameters

- **name** – name of the plugin

- **ov** – *ObjectVector* that we'll work with
- **dump_every** – write files every this many time-steps
- **channel_names** – list of channel names to be dumped.
- **path** – Path and filename prefix for the dumps. For every dump two files will be created: <path>_NNNNN.xmf and <path>_NNNNN.h5

createDumpParticlesWithPolylines(*state: MirState, name: str, cv: ParticleVectors.ChainVector, dump_every: int, channel_names: List[str], path: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will dump positions, velocities and optional attached data of all the particles of the specified ChainVector, as well as connectivity information representing polylines. The data is dumped into hdf5 format. An additional xdfm file is dumped to describe the data and make it readable by visualization tools.

Parameters

- **name** – name of the plugin.
- **cv** – *ChainVector* to be dumped.
- **dump_every** – write files every this many time-steps.
- **channel_names** – list of channel names to be dumped.
- **path** – Path and filename prefix for the dumps. For every dump two files will be created: <path>_NNNNN.xmf and <path>_NNNNN.h5

createDumpXYZ(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *dump_every: int*, *path: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will dump positions of all the particles of the specified Particle Vector in the XYZ format.

Note: This plugin is inactive if postprocess is disabled

Parameters

- **name** – name of the plugin
- **pvs** – list of *ParticleVector* that we'll work with
- **dump_every** – write files every this many time-steps
- **path** – the files will look like this: <path>/<pv_name>_NNNNN.xyz

createExchangePVSFluxPlane(*state: MirState*, *name: str*, *pv1: ParticleVectors.ParticleVector*, *pv2: ParticleVectors.ParticleVector*, *plane: real4*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin exchanges particles from a particle vector crossing a given plane to another particle vector. A particle with position x, y, z has crossed the plane if ax + by + cz + d >= 0, where a, b, c and d are the coefficient stored in the 'plane' variable

Parameters

- **name** – name of the plugin
- **pv1** – `ParticleVector` source
- **pv2** – `ParticleVector` destination
- **plane** – 4 coefficients for the plane equation ax + by + cz + d >= 0

createExternalMagneticTorque(*state: MirState, name: str, rov: ParticleVectors.RigidObjectVector, moment: real3, magneticFunction: Callable[[float], real3]*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin gives a magnetic moment $\mathbf{M}$ to every rigid objects in a given *RigidObjectVector*. It also models a uniform magnetic field $\mathbf{B}$ (varying in time) and adds the induced torque to the objects according to:

$$\mathbf{T} = \mathbf{M} \times \mathbf{B}$$

The magnetic field is passed as a function from python. The function must take a real (time) as input and output a tuple of three reals (magnetic field).

Parameters

- **name** – name of the plugin
- **rov** – `RigidObjectVector` with which the magnetic field will interact
- **moment** – magnetic moment per object
- **magneticFunction** – a function that depends on time and returns a uniform (real3) magnetic field

createForceSaver(*state: MirState, name: str, pv: ParticleVectors.ParticleVector*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin creates an extra channel per particle inside the given particle vector named 'forces'. It copies the total forces at each time step and make it accessible by other plugins. The forces are stored in an array of real3.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with

createImposeProfile(*state: MirState, name: str, pv: ParticleVectors.ParticleVector, low: real3, high: real3, velocity: real3, kBT: float*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will set the velocity of each particle inside a given domain to a target velocity with an additive term drawn from Maxwell distribution of the given temperature.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with
- **low** – the lower corner of the domain
- **high** – the higher corner of the domain
- **velocity** – target velocity
- **kBT** – temperature in the domain (appropriate Maxwell distribution will be used)

createImposeVelocity(*state: MirState, name: str, pvs: List[ParticleVectors.ParticleVector], every: int, low: real3, high: real3, velocity: real3*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will add velocity to all the particles of the target PV in the specified area (rectangle) such that the average velocity equals to desired.

Parameters

- **name** – name of the plugin
- **pvs** – list of *ParticleVector* that we'll work with
- **every** – change the velocities once in **every** timestep

- **low** – the lower corner of the domain
- **high** – the higher corner of the domain
- **velocity** – target velocity

createMagneticDipoleInteractions(*state: MirState*, *name: str*, *rov: ParticleVectors.RigidObjectVector*, *moment: real3*, *mu0: float*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin computes the forces and torques resulting from pairwise dipole-dipole interactions between rigid objects. All rigid objects are assumed to be the same with a constant magnetic moment in their frame of reference.

Parameters

- **name** – name of the plugin
- **rov** – `RigidObjectVector` with which the magnetic field will interact
- **moment** – magnetic moment per object
- **mu0** – magnetic permeability of the medium

createMembraneExtraForce(*state: MirState, name: str, pv: ParticleVectors.ParticleVector, forces: List[real3]*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin adds a given external force to a given membrane. The force is defined vertex wise and does not depend on position. It is the same for all membranes belonging to the same particle vector.

Parameters

- **name** – name of the plugin
- **pv** – `ParticleVector` to which the force should be added
- **forces** – array of forces, one force (3 reals) per vertex in a single mesh

createMsd(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *start_time: float*, *end_time: float*, *dump_every: int*, *path: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin computes the mean square displacement of th particles of a given *ParticleVector*. The reference position‘ is that of the given *ParticleVector* at the given start time.

Parameters

- **name** – Name of the plugin.
- **pv** – Concerned `ParticleVector`.
- **start_time** – Simulation time of the reference positions.
- **end_time** – End time until which to compute the MSD.
- **dump_every** – Report MSD every this many time-steps.
- **path** – The folder name in which the file will be dumped.

createParticleChannelAverager(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *channelName: str*, *averageName: str*, *updateEvery: float*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin averages a channel (per particle data) inside the given particle vector and saves it to a new channel. This new channel (containing the averaged data) is updated every fixed number of time steps.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with

- **channelName** – The name of the source channel.
- **averageName** – The name of the average channel.
- **updateEvery** – reinitialize the averages every this number of steps.

createParticleChannelSaver(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *channelName: str*, *savedName: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin creates an extra channel per particle inside the given particle vector with a given name. It copies the content of an extra channel of pv at each time step and make it accessible by other plugins.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with
- **channelName** – the name of the source channel
- **savedName** – name of the extra channel

createParticleChecker(*state: MirState*, *name: str*, *check_every: int*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will check the positions and velocities of all particles in the simulation every given time steps. To be used for debugging purpose.

Parameters

- **name** – name of the plugin
- **check_every** – check every this amount of time steps

createParticleDisplacement(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *update_every: int*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin computes and save the displacement of the particles within a given particle vector. The result is stored inside the extra channel "displacements" as an array of real3.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with
- **update_every** – displacements are computed between positions separated by this amount of timesteps

createParticleDrag(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *drag: float*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will add drag force $\mathbf{f} = -C_d\mathbf{u}$ to each particle of a specific PV every time-step.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with
- **drag** – drag coefficient

createPinObject(*state: MirState*, *name: str*, *ov: ParticleVectors.ObjectVector*, *dump_every: int*, *path: str*, *velocity: real3*, *angular_velocity: real3*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will impose given velocity as the center of mass velocity (by axis) of all the objects of the specified

Object Vector. If the objects are rigid bodies, rotation may be restricted with this plugin as well. The *time-averaged* force and/or torque required to impose the velocities and rotations are reported in the dumped file, with the following format:

<object id> <simulation time> <force>x3 [<torque>x3]

Note: This plugin is inactive if postprocess is disabled

Parameters

- **name** – name of the plugin
- **ov** – *ObjectVector* that we'll work with
- **dump_every** – write files every this many time-steps
- **path** – the files will look like this: <path>/<ov_name>.csv
- **velocity** – 3 reals, each component is the desired object velocity. If the corresponding component should not be restricted, set this value to `PinObject::Unrestricted`
- **angular_velocity** – 3 reals, each component is the desired object angular velocity. If the corresponding component should not be restricted, set this value to `PinObject::Unrestricted`

createPinRodExtremity (*state: MirState*, *name: str*, *rv: ParticleVectors.RodVector*, *segment_id: int*, *f_magn: float*, *target_direction: real3*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin adds a force on a given segment of all the rods in a *RodVector*. The force has the form deriving from the potential

$$E = k\left(1 - \cos\theta\right),$$

where θ is the angle between the material frame and a given direction (projected on the concerned segment). Note that the force is applied only on the material frame and not on the center line.

Parameters

- **name** – name of the plugin
- **rv** – *RodVector* that we'll work with
- **segment_id** – the segment to which the plugin is active
- **f_magn** – force magnitude
- **target_direction** – the direction in which the material frame tends to align

createPlaneOutlet (*state: MirState, name: str, pvs: List[ParticleVectors.ParticleVector], plane: real4*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin removes all particles from a set of *ParticleVector* that are on the non-negative side of a given plane.

Parameters

- **name** – name of the plugin
- **pvs** – list of *ParticleVector* that we'll work with
- **plane** – Tuple (a, b, c, d). Particles are removed if *ax + by + cz + d >= 0*.

createRateOutlet(*state: MirState, name: str, pvs: List[ParticleVectors.ParticleVector], mass_rate: float, region: Callable[[real3], float], resolution: real3*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin removes particles from a set of `ParticleVector` in a given region at a given mass rate.

Parameters

- **name** – name of the plugin
- **pvs** – list of `ParticleVector` that we'll work with
- **mass_rate** – total outlet mass rate in the region
- **region** – a function that is negative in the concerned region and positive outside
- **resolution** – grid resolution to represent the region field

createRdf(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *max_dist: float*, *nbins: int*, *basename: str*, *every: int*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

Compute the radial distribution function (RDF) of a given `ParticleVector`. For simplicity, particles that are less that *max_dist* from the subdomain border are not counted.

Parameters

- **name** – Name of the plugin.
- **pv** – The `ParticleVector` that we ant to compute the RDF from.
- **max_dist** – The RDF will be computed on the interval [0, max_dist]. Must be strictly less than half the minimum size of one subdomain.
- **nbins** – The RDF is computed on nbins bins.
- **basename** – Each RDF dump will be dumped in csv format to <basename>-XXXXX.csv.
- **every** – Computes and dump the RDF every this amount of timesteps.

createStats(*state: MirState*, *name: str*, *every: int*, *pvs: List[ParticleVectors.ParticleVector]=[]*, *filename: str=''*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will report aggregate quantities of all the particles in the simulation: total number of particles in the simulation, average temperature and momentum, maximum velocity magnutide of a particle and also the mean real time per step in milliseconds.

Note: This plugin is inactive if postprocess is disabled

Parameters

- **name** – Name of the plugin.
- **every** – Report to standard output every that many time-steps.
- **pvs** – List of pvs to compute statistics from. If empty, will use all the pvs registered in the simulation.
- **filename** – The statistics are saved in this csv file. The name should either end with *.csv* or have no extension, in which case *.csv* is added.

createTemperaturize(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *kBT: float*, *keepVelocity: bool*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin changes the velocity of each particles from a given `ParticleVector`. It can operate under two modes: *keepVelocity* = *True*, in which case it adds a term drawn from a Maxwell distribution to the current velocity; *keepVelocity* = *False*, in which case it sets the velocity to a term drawn from a Maxwell distribution.

Parameters

- **name** – name of the plugin
- **pv** – the concerned *ParticleVector*
- **kBT** – the target temperature
- **keepVelocity** – True for adding Maxwell distribution to the previous velocity; False to set the velocity to a Maxwell distribution.

createVacf(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *start_time: float*, *end_time: float*, *dump_every: int*, *path: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin computes the mean velocity autocorrelation over time from a given *ParticleVector*. The reference velocity *v0* is that of the given *ParticleVector* at the given start time.

Parameters

- **name** – Name of the plugin.
- **pv** – Concerned `ParticleVector`.
- **start_time** – Simulation time of the reference velocities.
- **end_time** – End time until which to compute the VACF.
- **dump_every** – Report the VACF every this many time-steps.
- **path** – The folder name in which the file will be dumped.

createVelocityControl(*state: MirState, name: str, filename: str, pvs: List[ParticleVectors.ParticleVector], low: real3, high: real3, sample_every: int, tune_every: int, dump_every: int, target_vel: real3, Kp: float, Ki: float, Kd: float*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin applies a uniform force to all the particles of the target PVS in the specified area (rectangle). The force is adapted bvia a PID controller such that the velocity average of the particles matches the target average velocity.

Parameters

- **name** – Name of the plugin.
- **filename** – Dump file name. Must have a csv extension or no extension at all.
- **pvs** – List of concerned `ParticleVector`.
- **high** (*low,*) – boundaries of the domain of interest
- **sample_every** – sample velocity every this many time-steps
- **tune_every** – adapt the force every this many time-steps
- **dump_every** – write files every this many time-steps
- **target_vel** – the target mean velocity of the particles in the domain of interest
- **Ki, Kd** (*Kp,*) – PID controller coefficients

createVelocityInlet(*state: MirState, name: str, pv: ParticleVectors.ParticleVector, implicit_surface_func: Callable[[real3], float], velocity_field: Callable[[real3], real3], resolution: real3, number_density: float, kBT: float*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin inserts particles in a given *ParticleVector*. The particles are inserted on a given surface with given velocity inlet. The rate of insertion is governed by the velocity and the given number density.

Parameters

- **name** – name of the plugin
- **pv** – the *ParticleVector* that we ll work with
- **implicit_surface_func** – a scalar field function that has the required surface as zero level set
- **velocity_field** – vector field that describes the velocity on the inlet (will be evaluated on the surface only)
- **resolution** – grid size used to discretize the surface
- **number_density** – number density of the inserted solvent
- **kBT** – temperature of the inserted solvent

createVirialPressurePlugin(*state: MirState, name: str, pv: ParticleVectors.ParticleVector, regionFunc: Callable[[real3], float], h: real3, dump_every: int, path: str*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin computes the virial pressure from a given *ParticleVector*. Note that the stress computation must be enabled with the corresponding stressName. This returns the total internal virial part only (no temperature term). Note that the volume is not devided in the result, the user is responsible to properly scale the output.

Parameters

- **name** – name of the plugin
- **pv** – concerned ParticleVector
- **regionFunc** – predicate for the concerned region; positive inside the region and negative outside
- **h** – grid size for representing the predicate onto a grid
- **dump_every** – report total pressure every this many time-steps
- **path** – the folder name in which the file will be dumped

createWallForceCollector(*state: MirState*, *name: str*, *wall: Walls.Wall*, *pvFrozen: ParticleVectors.ParticleVector*, *sample_every: int*, *dump_every: int*, *filename: str*, *detailed_dump: bool=False*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin collects and averages the total force exerted on a given wall. The result has 2 components:

- bounce back: force necessary to the momentum change
- frozen particles: total interaction force exerted on the frozen particles

Parameters

- **name** – name of the plugin
- **wall** – The *Wall* to collect forces from
- **pvFrozen** – corresponding frozen *ParticleVector*
- **sample_every** – sample every this number of time steps
- **dump_every** – dump every this number of time steps
- **filename** – output filename (csv format)
- **detailed_dump** – if True, will dump separately the bounce contribution and the rest. If False, only the sum is dumped.

createWallRepulsion(*state: MirState*, *name: str*, *pv: ParticleVectors.ParticleVector*, *wall: Walls.Wall*, *C: float*, *h: float*, *max_force: float*) → Tuple[Plugins.SimulationPlugin, Plugins.PostprocessPlugin]

This plugin will add force on all the particles that are nearby a specified wall. The motivation of this plugin is as follows. The particles of regular PVs are prevented from penetrating into the walls by Wall Bouncers. However, using Wall Bouncers with Object Vectors may be undesirable (e.g. in case of a very viscous membrane) or impossible (in case of rigid objects). In these cases one can use either strong repulsive potential between the object and the wall particle or alternatively this plugin. The advantage of the SDF-based repulsion is that small penetrations won't break the simulation.

The force expression looks as follows:

$$\mathbf{F}(\mathbf{r}) = \nabla S(\mathbf{r}) \cdot \begin{cases} 0, & S(\mathbf{r}) < -h, \\ \min(F_{\max}, C(S(\mathbf{r}) + h)), & S(\mathbf{r}) \geqslant -h, \end{cases}$$

where S is the SDF of the wall, C, $F_{\max}$ and h are parameters.

Parameters

- **name** – name of the plugin
- **pv** – *ParticleVector* that we'll work with
- **wall** – *Wall* that defines the repulsion
- **C** – C
- **h** – h
- **max_force** – F_{max}

15 Utils

Utility functions that do not need any mirheo coordinator. Most of these functions are wrapped by the `__main__.py` file and can be called directly from the command line:

```
python -m mirheo --help
```

15.1 Summary

get_all_compile_options()	Return all compile time options used in the current installation in the form of a dictionary.
get_compile_option(key)	Fetch a given compile time option currently in use.

15.2 Details

get_all_compile_options() → Dict[str, str]

Return all compile time options used in the current installation in the form of a dictionary.

get_compile_option(*key: str*) → str

Fetch a given compile time option currently in use. :param key: the option name.

Available names can be found from the *get_all_compile_options* command.

16 Overview

16.1 Task Dependency Graph

The simulation is composed of a set of tasks that have dependencies between them, e.g. the forces must be computed before integrating the particles velocities and positions. The following graph represents the tasks that are executed at every time step and there dependencies:

17 Coding Conventions

In this section we list a guidelines to edit/add code to Mirheo.

17.1 Naming

Variables

Local variable names and paramters follow camelCase format starting with a lower case:

```
int myInt;    // OK
int MyInt;    // not OK
int my_int;   // not OK
```

Member variable names inside a `class` (not for `struct`) have a trailing _:

```
class MyClass
{
private:
     int myInt_;    // OK
     int myInt;     // not OK
     int my_int_;   // not OK
};
```

Types, classes

Class names (and all types) have a camelCase format and start with an upper case letter:

```
class MyClass;          // OK
using MyIntType = int; // OK
class My_Class;         // Not OK
```

Functions

Functions and public member functions follow the same rules as local variables. They should state an action and must be meaningfull, especially when they are exposed to the rest of the library.

```
Mesh readOffFile(std::string fileName); // OK
Mesh ReadOffFile(std::string fileName); // not OK
Mesh read(std::string fileName);        // not precise enough naming out of context
```

private member functions have an additional _ in front:

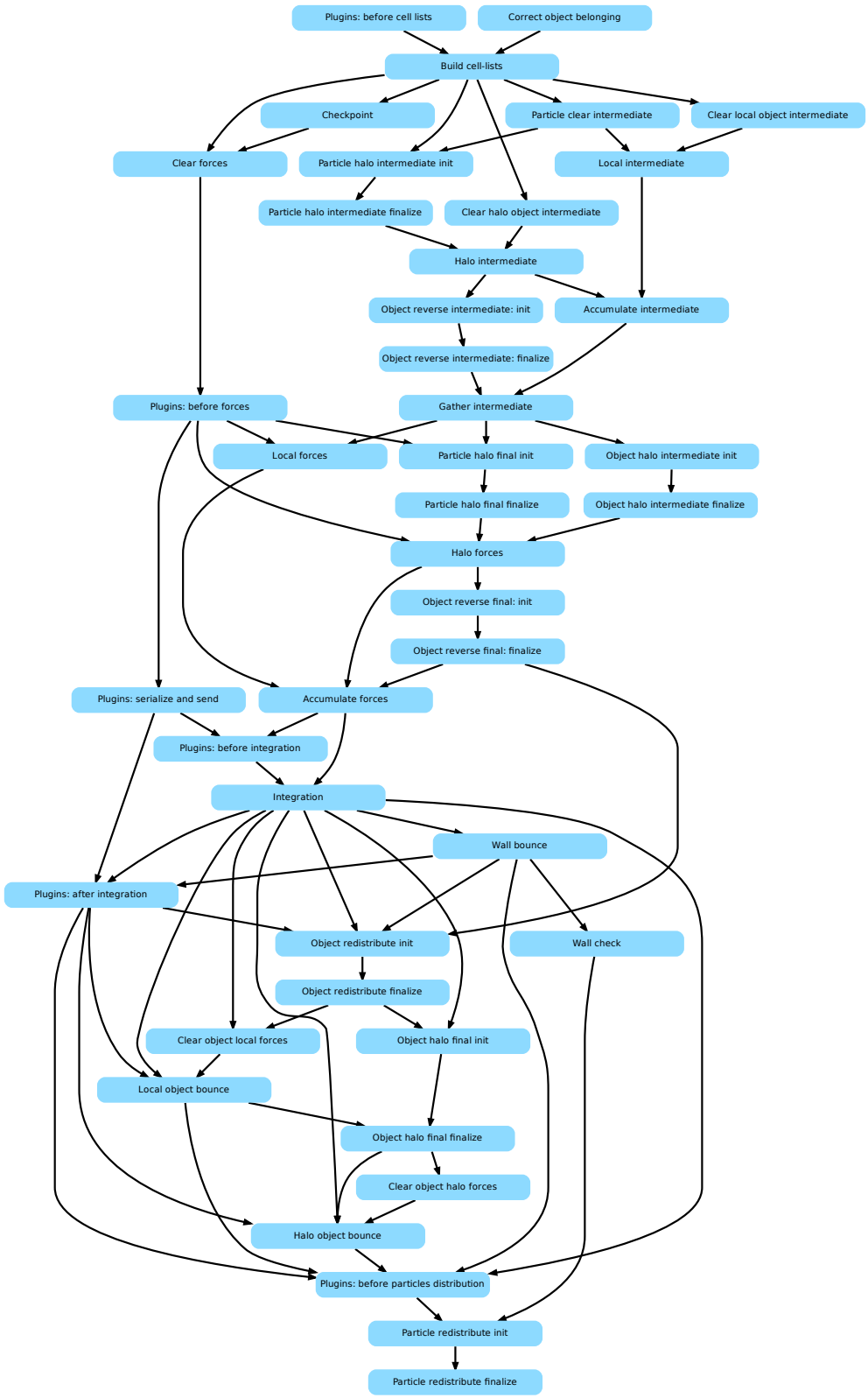


Fig. 15: The task dependency graph of a single time step in Mirheo.

```
class MyClass
{
public:
    void doSomething();
private:
    void _doSubTask();    // OK
    void doSubTask();     // Not OK
    void _do_sub_task(); // Not OK
};
```

namespaces

Namespaces are written in `snake_case`. This allows to distinguish them from class names easily.

17.2 Coding practices

Use modern C++ whenever possible. Refer to c++ core guidelines up to C++ 14. Some exceptions in Mirheo:

- Do not fail with exceptions. Mirheo crashes with the `die` method from the logger. This will print the full stacktrace.

17.3 Style

The indentation uses 4 spaces (no tabs). Here are a few coding examples of the style:

```
// loops
for (int i = 0; i < n; ++i)
{
    // multi line commands
    ...
}

for (int i = 0; i < n; ++i)
    // one line command

// if
if (condition)
    doThat();
else
    doThis();

// for multi line, all entries must have braces
if (condition)
{
    doThat();
    andThis();
}
else
{
    doThis();
}
```

More can be found directly in the code.

17.4 Includes

Every header file must include all files required such that it compiles on its own. The includes must be grouped into 3 groups with the following order:

1. local files (relative path)
2. mirheo files (path relative to mirheo src dir)
3. external libraries and std library

Each subgroup must be sorted alphabetically. The first group has the quotes style while the other groups must use bracket style.

Example:

```
#include "data_manager.h"

#include <mirheo/core/containers.h>
#include <mirheo/core/datatypes.h>
#include <mirheo/core/mirheo_object.h>
#include <mirheo/core/utils/pytypes.h>

#include <memory>
#include <string>
#include <vector>
```

18 Library API

18.1 Analytic Shapes

Utility classes used to represent closed shapes with an implicit function. They must all have the same interface.

class Capsule

Represents a capsule.

A capsule is represented by a segment and a radius. Its surfae is the set of points whose distance to the segment is equal to the radius.

In more visual terms, a capsule looks like a finite cylinder with two half spheres on its ends.

The capsule is centered at the origin and oriented along the z axis.

Public Functions

Capsule (real *R*, real *L*)

Construct a *Capsule*.

Parameters

- R: the radius of the capsule. Must be positive.
- L: the length of the segment used to represent the capsule. Must be positive.

real **inOutFunction** (real3 *r*) **const**
Implicit surface representation.

This scalar field is a smooth function of the position. It is negative inside the capsule and positive outside. The zero level set of that field is the surface of the capsule.

Return The value of the field at the given position.

Parameters

- `r`: The position at which to evaluate the in/out field.

real3 **normal** (real3 *r*) **const**
Get the normal pointing outside the capsule.

This vector field is defined everywhere in space. On the surface, it represents the normal vector of the surface.

Return The normal at r (length of this return must be 1).

Parameters

- `r`: The position at which to evaluate the normal.

real3 **inertiaTensor** (real *totalMass*) **const**
Get the inertia tensor of the capsule in its frame of reference.

Return The diagonal of the inertia tensor.

Parameters

- `totalMass`: The total mass of the capsule.

Public Static Attributes

const char ***desc**
the description of shape.

class Cylinder
Represents a finite cylinder.

The cylinder is centered at the origin and is oriented along the z axis. It is fully described by its length and its radius.

Public Functions

Cylinder (real *R*, real *L*)
Constructs a *Cylinder*.

Parameters

- `R`: the radius of the cylinder. Must be positive
- `L`: the length of the cylinder. Must be positive.

real **inOutFunction** (real3 *r*) **const**
Implicit surface representation.

This scalar field is a smooth function of the position. It is negative inside the cylinder and positive outside. The zero level set of that field is the surface of the cylinder.

Return The value of the field at the given position.

Parameters

- `r`: The position at which to evaluate the in/out field.

real3 **`normal`**(real3 *r*) **`const`**
Get the normal pointing outside the cylinder.

This vector field is defined everywhere in space. On the surface, it represents the normal vector of the surface.

Return The normal at r (length of this return must be 1).

Parameters

- `r`: The position at which to evaluate the normal.

real3 **`inertiaTensor`**(real *totalMass*) **`const`**
Get the inertia tensor of the cylinder in its frame of reference.

Return The diagonal of the inertia tensor.

Parameters

- `totalMass`: The total mass of the cylinder.

Public Static Attributes

`const` char ***`desc`**
the description of shape.

`class Ellipsoid`
Represents an ellipsoid.

The ellipsoid is centered at the origin and oriented along its principal axes. the three radii are passed through the *axes* variable. The surface is described implicitly by the zero level set of:

$$\left(\frac{x}{a_x}\right)^2 + \left(\frac{y}{a_y}\right)^2 + \left(\frac{z}{a_z}\right)^2 = 1$$

Public Functions

`Ellipsoid`(real3 *axes*)
Construct a *`Ellipsoid`* object.

Parameters

- `axes`: the "radius" along each principal direction.

real **`inOutFunction`**(real3 *r*) **`const`**
Implicit surface representation.

This scalar field is a smooth function of the position. It is negative inside the ellipsoid and positive outside. The zero level set of that field is the surface of the ellipsoid.

Return The value of the field at the given position.

Parameters

- `r`: The position at which to evaluate the in/out field.

real3 **`normal`**(real3 *r*) **`const`**
Get the normal pointing outside the ellipsoid.

This vector field is defined everywhere in space. On the surface, it represents the normal vector of the surface.

Return The normal at r (length of this return must be 1).

Parameters

- `r`: The position at which to evaluate the normal.

real3 **`inertiaTensor`**(real *totalMass*) **`const`**
Get the inertia tensor of the ellipsoid in its frame of reference.

Return The diagonal of the inertia tensor.

Parameters

- `totalMass`: The total mass of the ellipsoid.

Public Static Attributes

`const` char ***`desc`**
the description of shape.

18.2 Bouncers

See also *the user interface*.

Base class

`class Bouncer` : **`public`** *mirheo*::*MirSimulationObject*
Avoid penetration of particles inside onjects.

Interface class for Bouncers. Bouncers are responsible to reflect particles on the surface of the attached object. Each `Bouncer` class needs to attach exactly one *ObjectVector* before performing the bounce.

Subclassed by *mirheo::BounceFromMesh*, *mirheo::BounceFromRigidShape< Shape >*, *mirheo::BounceFromRod*

Public Functions

`Bouncer`(**`const`** *MirState* **state*, std::string *name*)
Base `Bouncer` constructor.

Parameters

- `state`: *Simulation* state.
- `name`: Name of the bouncer.

virtual void **setup**(*ObjectVector* *ov)
Second initialization stage.

This method must be called before calling any other method of this class.

Parameters

- ov: The *ObjectVector* to attach to that Bouncer.

ObjectVector ***getObjectVector**()

Return The attached *ObjectVector*

virtual void **setPrerequisites**(*ParticleVector* *pv)
Setup prerequisites to a given *ParticleVector*.

Add additional properties to a *ParticleVector* to make it compatible with the exec() method. The default implementation does not add any properties.

Parameters

- pv: The *ParticleVector* that will be bounced

void **bounceLocal**(*ParticleVector* *pv, *CellList* *cl, cudaStream_t *stream*)
Perform the reflection of local particles onto the **local** attached objects surface.

Parameters

- pv: The *ParticleVector* that will be bounced
- cl: The *CellList* attached to pv
- stream: The cuda stream used for execution

void **bounceHalo**(*ParticleVector* *pv, *CellList* *cl, cudaStream_t *stream*)
Perform the reflection of local particles onto the **halo** attached objects surface.

Parameters

- pv: The *ParticleVector* that will be bounced
- cl: The *CellList* attached to pv
- stream: The cuda stream used for execution

virtual std::vector<std::string> **getChannelsToBeExchanged**() **const** = 0

Return list of channel names of the attached object needed before bouncing

virtual std::vector<std::string> **getChannelsToBeSentBack**() **const**

Return list of channel names of the attached object that need to be exchanged after bouncing

Derived classes

class BounceFromMesh : **public** *mirheo*::*Bouncer*
Bounce particles against a triangle mesh.

- if the attached object is a *RigidObjectVector*, the bounced particles will transfer (atomically) their change of momentum into the force and torque of the rigid object.

- if the attached object is a not *RigidObjectVector*, the bounced particles will transfer (atomically) their change of momentum into the force of the three vertices which form the colliding triangle.

  This class will fail if the object does not have a mesh representing its surfece.

Public Functions

BounceFromMesh(**const** *MirState* *state, **const** std::string &name, VarBounceKernel *varBounceKernel*)
Construct a *BounceFromMesh* object.

Parameters

- `state`: *Simulation* state
- `name`: Name of the bouncer
- `varBounceKernel`: How are the particles bounced

void **setup**(*ObjectVector* *ov)
If `ov` is a rigid object, this will ask it to keep its old motions accross exchangers.

Otherwise, ask `ov` to keep its old positions accross exchangers.

void **setPrerequisites**(*ParticleVector* *pv)
Will ask `pv` to keep its old positions (not in persistent mode)

std::vector<std::string> **getChannelsToBeExchanged**() **const**

Return list of channel names of the attached object needed before bouncing

std::vector<std::string> **getChannelsToBeSentBack**() **const**

Return list of channel names of the attached object that need to be exchanged after bouncing

The following class employs *Analytic Shapes* implicit surface representation.

template <**class** Shape>
class BounceFromRigidShape : **public** *mirheo*::*Bouncer*
Bounce particles against an *RigidShapedObjectVector*.

Particles are bounced against an analytical shape on each object of the attached *ObjectVector*. When bounced, the particles will transfer (atomically) their change of momentum into the force and torque of the rigid objects.

Template Parameters

- `Shape`: A class following the AnalyticShape interface

This class only works with RigidShapedObjectVector<Shape> objects. It will fail at setup time if the attached object is not rigid.

Public Functions

BounceFromRigidShape(**const** *MirState* *state, **const** std::string &name, VarBounceKernel *varBounceKernel*, int *verbosity*)
Construct a *BounceFromRigidShape* object.

Parameters

- `state`: *Simulation* state

- `name`: Name of the bouncer
- `varBounceKernel`: How are the particles bounced
- `verbosity`: 0: no print; 1 print to console the rescue failures; 2 print to console all failures.

void **setup**(*ObjectVector* **ov*)
Will ask `ov` to keep its old motions information persistently.

This method will die if `ov` is not of type *RigidObjectVector*.

void **setPrerequisites**(*ParticleVector* **pv*)
Will ask `pv` to keep its old positions (not in persistent mode)

std::vector<std::string> **getChannelsToBeExchanged**() **const**

Return list of channel names of the attached object needed before bouncing

std::vector<std::string> **getChannelsToBeSentBack**() **const**

Return list of channel names of the attached object that need to be exchanged after bouncing

class BounceFromRod : **public** *mirheo*::*Bouncer*
Bounce particles against rods.

The particles are reflacted against the set of capsules around each segment forming the rod. This class will fail if the attached object is not a RodObjectVector

Public Functions

BounceFromRod(**const** *MirState* **state*, **const** std::string &*name*, real *radius*, VarBounceKernel *varBounceKernel*)
Construct a *BounceFromRod* object.

Parameters

- `state`: *Simulation* state
- `name`: Name of the bouncer
- `radius`: The radius of the capsules attached to each segment
- `varBounceKernel`: How are the particles bounced

void **setup**(*ObjectVector* **ov*)
Ask `ov` to keep its old motions accross persistently.

This method will die if `ov` is not of type RodObjectVector.

void **setPrerequisites**(*ParticleVector* **pv*)
Will ask `pv` to keep its old positions (not in persistent mode)

std::vector<std::string> **getChannelsToBeExchanged**() **const**

Return list of channel names of the attached object needed before bouncing

std::vector<std::string> **getChannelsToBeSentBack**() **const**

Return list of channel names of the attached object that need to be exchanged after bouncing

Utilities

class BounceBack

Implements bounce-back reflection.

This bounce kernel reverses the velocity of the particle in the frame of reference of the surface.

Public Functions

void **update**(std::mt19937 &*rng*)

Does nothing, just to be consistent with the inteface.

real3 **newVelocity**(real3 *uOld*, real3 *uWall*, real3 *n*, real *mass*) **const**

Compute the velocity after bouncing the particle.

The velocity is chosen such that the average between the new and old velocities of the particle is that of the wall surface at the collision point.

Parameters

- `uOld`: The velocity of the particle at the previous time step.
- `uWall`: The velocity of the wall surface at the collision point.
- `n`: The wall surface normal at the collision point.
- `mass`: The particle mass.

class BounceMaxwell

Implements reflection with Maxwell scattering.

This bounce kernel sets the particle velocity to the surface one with an additional random term drawed from Maxwell distribution. The kernel tries to make the random term have a positive dot product with the surface normal.

Public Functions

BounceMaxwell(real *kBT*)

Construct a *BounceMaxwell* object.

Parameters

- `kBT`: The temperature used to sample the velocity

void **update**(std::mt19937 &*rng*)

Update internal state, must be called before use.

Parameters

- `rng`: A random number generator.

real3 **newVelocity**(real3 *uOld*, real3 *uWall*, real3 *n*, real *mass*) **const**

Compute the velocity after bouncing the particle.

The velocity is chosen such that it is sampled by a Maxwelian distribution and has a positive dot product with the wall surface normal.

Parameters

- `uOld`: The velocity of the particle at the previous time step.
- `uWall`: The velocity of the wall surface at the collision point.
- `n`: The wall surface normal at the collision point.
- `mass`: The particle mass.

18.3 Cell-Lists

Cell-lists are used to map from space to particles and vice-versa.

Internal structure

A cell list is composed of:

1. The representation of the cells geometry (here a uniform grid): see *`mirheo::CellListInfo`*
2. Number of particles per cell
3. Index of the first particle in each cell
4. The particle data, reordered to match the above structure.

API

`class CellListInfo`

A device-compatible structure that represents the cell-lists structure.

Contains geometric info (number of cells, cell sizes) and associated particles info (number of particles per cell and cell-starts).

Subclassed by *mirheo::CellList*

Public Functions

`CellListInfo` (real3 *h*, real3 *localDomainSize*)

Construct a *CellListInfo* object.

This will create a cell-lists structure with cell sizes which are larger or equal to `h`, such that the number of cells fit exactly inside the local domain size.

Parameters

- `h`: The size of a single cell along each dimension
- `localDomainSize`: Size of the local domain

`CellListInfo` (real *rc*, real3 *localDomainSize*)

Construct a *CellListInfo* object.

This will create a cell-lists structure with cell sizes which are larger or equal to `rc`, such that the number of cells fit exactly inside the local domain size.

Parameters

- `rc`: The minimum size of a single cell along every dimension

- `localDomainSize`: Size of the local domain

__device__ __host__ int **encode** (int *ix*, int *iy*, int *iz*) **const**
map 3D cell indices to linear cell index.

Return Linear cell index

Parameters

- `ix`: Cell index in the x direction
- `iy`: Cell index in the y direction
- `iz`: Cell index in the z direction

__device__ __host__ void **decode** (int *cid*, int &*ix*, int &*iy*, int &*iz*) **const**
map linear cell index to 3D cell indices.

Parameters

- `cid`: Linear cell index
- `ix`: Cell index in the x direction
- `iy`: Cell index in the y direction
- `iz`: Cell index in the z direction

__device__ __host__ int **encode** (int3 *cid3*) **const**
see *encode()*

__device__ __host__ int3 **decode** (int *cid*) **const**
see *decode()*

template <CellListsProjection *Projection* = CellListsProjection::Clamp>
__device__ __host__ int3 **getCellIdAlongAxes** (**const** real3 *x*) **const**
Map from position to cell indices.

Return cell indices

Template Parameters

- `Projection`: if the cell indices must be clamped or not

Parameters

- `x`: The position in **local coordinates**

template <CellListsProjection *Projection* = CellListsProjection::Clamp, **typename** T>
__device__ __host__ int **getCellId** (**const** T *x*) **const**
Map from position to linear indices.

> **Warning:** If The projection is set to CellListsProjection::NoClamp, this function will return -1 if the particle is outside the subdomain.

Return linear cell index

Template Parameters

- `Projection`: if the cell indices must be clamped or not

- `T`: The vector type that represents the position

Parameters

- `x`: The position in **local coordinates**

Public Members

int3 **`ncells`**
Number of cells along each direction in the local domain.

int **`totcells`**
total number of cells in the local domain

real3 **`localDomainSize`**
dimensions of the subdomain

real **`rc`**
cutoff radius

real3 **`h`**
dimensions of the cells along each direction

int ***`cellSizes`** = {nullptr}
number of particles contained in each cell

int ***`cellStarts`** = {nullptr}
exclusive prefix sum of cellSizes

int ***`order`** = {nullptr}
used to reorder particles when building the cell lists: `order`[pid] is the destination index of the particle with index `pid` before reordering

`class CellList` : **`public`** *mirheo*::*CellListInfo*
Contains the cell-list data for a given *ParticleVector*.

As opposed to the *PrimaryCellList* class, it contains a **copy** of the attached *ParticleVector*. This means that the original *ParticleVector* data will not be reorder but rather copied into this container. This is useful when several *CellList* object can be attached to the same *ParticleVector* or if the *ParticleVector* must not be reordered such as e.g. for *ObjectVector* objects.

Subclassed by *mirheo::PrimaryCellList*

Public Functions

`CellList`(*ParticleVector* **pv*, real *rc*, real3 *localDomainSize*)
Construct a *CellList* object.

Parameters

- `pv`: The *ParticleVector* to attach.
- `rc`: The maximum cut-off radius that can be used with that cell list.
- `localDomainSize`: The size of the local subdomain

`CellList`(*ParticleVector* **pv*, int3 *resolution*, real3 *localDomainSize*)
Construct a *CellList* object.

Parameters

- `pv`: The *ParticleVector* to attach.
- `resolution`: The number of cells along each dimension
- `localDomainSize`: The size of the local subdomain

CellListInfo **cellInfo**()

Return the device-compatible handler

virtual void **build**(cudaStream_t *stream*)
construct the cell-list associated with the attached *ParticleVector*

Parameters

- `stream`: The stream used to execute the process

virtual void **accumulateChannels**(**const** std::vector<std::string> &*channelNames*, cudaStream_t *stream*)
Accumulate the channels from the data contained inside the cell-list container to the attached *ParticleVector*.

Parameters

- `channelNames`: List that contains the names of all the channels to accumulate
- `stream`: Execution stream

virtual void **gatherChannels**(**const** std::vector<std::string> &*channelNames*, cudaStream_t *stream*)
Copy the channels from the attached *ParticleVector* to the cell-lists data.

Parameters

- `channelNames`: List that contains the names of all the channels to copy
- `stream`: Execution stream

void **clearChannels**(**const** std::vector<std::string> &*channelNames*, cudaStream_t *stream*)
Clear channels contained inside the cell-list.

Parameters

- `channelNames`: List that contains the names of all the channels to clear
- `stream`: Execution stream

template <**typename** ViewType>
ViewType **getView**() **const**
Create a view that points to the data contained in the cell-lists.

Return View that points to the cell-lists data

Template Parameters

- `ViewType`: The type of the view to create

template <**typename** T>
void **requireExtraDataPerParticle**(**const** std::string &*name*)
Add an extra channel to the cell-list.

Template Parameters

- `T`: The type of data to add

Parameters

- `name`: Name of the channel

LocalParticleVector ***getLocalParticleVector**()

Return The *LocalParticleVector* that contains the data in the cell-list

std::string **getName**() **const**

Return the name of the cell-list.

class PrimaryCellList : **public** *mirheo*::*CellList*

Contains the cell-list map for a given *ParticleVector*.

As opposed to the *CellList* class, the data is stored only in the *ParticleVector*. This means that the original *ParticleVector* data will be reorder according to this cell-list. This allows to save memory and avoid extra copies. On the other hand, this class must not be used with *ObjectVector* objects.

Public Functions

PrimaryCellList(*ParticleVector* **pv*, real *rc*, real3 *localDomainSize*)

Construct a *PrimaryCellList* object.

Parameters

- `pv`: The *ParticleVector* to attach.
- `rc`: The maximum cut-off radius that can be used with that cell list.
- `localDomainSize`: The size of the local subdomain

PrimaryCellList(*ParticleVector* **pv*, int3 *resolution*, real3 *localDomainSize*)

Construct a *PrimaryCellList* object.

Parameters

- `pv`: The *ParticleVector* to attach.
- `resolution`: The number of cells along each dimension
- `localDomainSize`: The size of the local subdomain

void **build**(cudaStream_t *stream*)

construct the cell-list associated with the attached *ParticleVector*

Parameters

- `stream`: The stream used to execute the process

void **accumulateChannels**(**const** std::vector<std::string> &*channelNames*, cudaStream_t *stream*)

Accumulate the channels from the data contained inside the cell-list container to the attached *ParticleVector*.

Parameters

- `channelNames`: List that contains the names of all the channels to accumulate
- `stream`: Execution stream

void **gatherChannels**(**const** std::vector<std::string> &*channelNames*, cudaStream_t *stream*)
Copy the channels from the attached *ParticleVector* to the cell-lists data.

Parameters

- `channelNames`: List that contains the names of all the channels to copy
- `stream`: Execution stream

18.4 Containers

A set of array containers to manage device, host and pinned memory.

class GPUcontainer
Interface of containers of device (GPU) data.

Subclassed by *mirheo::DeviceBuffer< BelongingTags >*, *mirheo::DeviceBuffer< char >*, *mirheo::DeviceBuffer< int >*, *mirheo::DeviceBuffer< int2 >*, *mirheo::DeviceBuffer< mirheo::Force >*, *mirheo::DeviceBuffer< mirheo::MapEntry >*, *mirheo::DeviceBuffer< mirheo::TemplRigidMotion >*, *mirheo::DeviceBuffer< real >*, *mirheo::DeviceBuffer< real4 >*, *mirheo::DeviceBuffer< T >*, *mirheo::PinnedBuffer< T >*, *mirheo::PinnedBuffer< bool >*, *mirheo::PinnedBuffer< ChannelType >*, *mirheo::PinnedBuffer< char >*, *mirheo::PinnedBuffer< CudaVarPtr >*, *mirheo::PinnedBuffer< double >*, *mirheo::PinnedBuffer< double3 >*, *mirheo::PinnedBuffer< int >*, *mirheo::PinnedBuffer< int2 >*, *mirheo::PinnedBuffer< int3 >*, *mirheo::PinnedBuffer< mirheo::Force >*, *mirheo::PinnedBuffer< mirheo::ParticleCheckerPlugin::Status >*, *mirheo::PinnedBuffer< msd_plugin::ReductionType >*, *mirheo::PinnedBuffer< rdf_plugin::CountType >*, *mirheo::PinnedBuffer< real *>*, *mirheo::PinnedBuffer< real >*, *mirheo::PinnedBuffer< real3 >*, *mirheo::PinnedBuffer< real4 >*, *mirheo::PinnedBuffer< size_t >*, *mirheo::PinnedBuffer< stats_plugin::ReductionType >*, *mirheo::PinnedBuffer< unsigned long long int >*, *mirheo::PinnedBuffer< vacf_plugin::ReductionType >*, *mirheo::PinnedBuffer< virial_pressure_plugin::ReductionType >*

Public Functions

virtual size_t **size**() **const** = 0

Return number of stored elements

virtual size_t **datatype_size**() **const** = 0

Return the size (in bytes) of a single element

virtual void ***genericDevPtr**() **const** = 0

Return pointer to device data

virtual void **resize_anew**(size_t *n*) = 0
resize the internal array.

No guarantee to keep the current data.

Parameters

- `n`: New size (in number of elements). Must be non negative.

virtual void **resize**(size_t *n*, cudaStream_t *stream*) = 0
resize the internal array.

Keeps the current data.

Parameters

- `n`: New size (in number of elements). Must be non negative.
- `stream`: Used to copy the data internally

virtual void **clearDevice**(cudaStream_t *stream*) = 0
Call cudaMemset on the array.

Parameters

- `stream`: Execution stream

virtual *GPUcontainer* ***produce**() **const** = 0
Create a new instance of the concrete container implementation.

template <typename *T*>
class DeviceBuffer : **public** *mirheo*::*GPUcontainer*
Data only on the device (GPU)

Never releases any memory, keeps a buffer big enough to store maximum number of elements it ever held (except in the destructor).

Template Parameters

- `T`: The type of a single element to store.

Public Functions

DeviceBuffer(size_t *n* = 0)
Construct a *DeviceBuffer* of given size.

Parameters

- `n`: The initial number of elements

DeviceBuffer(**const** *DeviceBuffer* &*b*)
Copy constructor.

DeviceBuffer &**operator=**(**const** *DeviceBuffer* &*b*)
Assignment operator.

DeviceBuffer(*DeviceBuffer* &&*b*)
Move constructor; To enable `std::swap()`

DeviceBuffer &**operator=**(*DeviceBuffer* &&*b*)
Move assignment; To enable `std::swap()`

size_t **datatype_size**() **const**

Return the size (in bytes) of a single element

size_t **size**() **const**

Return number of stored elements

void ***genericDevPtr**() **const**

Return pointer to device data

void **resize**(size_t *n*, cudaStream_t *stream*)
resize the internal array.

Keeps the current data.

Parameters

- `n`: New size (in number of elements). Must be non negative.
- `stream`: Used to copy the data internally

void **resize_anew**(size_t *n*)
resize the internal array.

No guarantee to keep the current data.

Parameters

- `n`: New size (in number of elements). Must be non negative.

GPUcontainer ***produce**() **const**
Create a new instance of the concrete container implementation.

T ***devPtr**() **const**

Return device pointer to data

void **clearDevice**(cudaStream_t *stream*)
Call cudaMemset on the array.

Parameters

- `stream`: Execution stream

void **clear**(cudaStream_t *stream*)
clear the device data

template <typename Cont**>**
auto **copy**(**const** Cont &*cont*, cudaStream_t *stream*)
Copy data from another container of the same template type.

Can only copy from another *DeviceBuffer* of *HostBuffer*, but not *PinnedBuffer*.

Template Parameters

- `Cont`: The source container type. Must have the same data type than the current instance.

Parameters

- `cont`: The source container
- `stream`: Execution stream

auto **copy**(**const** *DeviceBuffer*<T> &*cont*)
synchronous copy

void **copyFromDevice**(**const** *PinnedBuffer*<T> &*cont*, cudaStream_t *stream*)
Copy the device data of a *PinnedBuffer* to the internal buffer.

Note The copy is performed asynchronously. The user must manually synchronize with the stream if needed.

Parameters

- `cont`: the source container
- `stream`: The stream used to copy the data.

void **copyFromHost**(**const** *PinnedBuffer*<T> &*cont*, cudaStream_t *stream*)
Copy the host data of a *PinnedBuffer* to the internal buffer.

Note The copy is performed asynchronously. The user must manually synchronize with the stream if needed.

Parameters

- `cont`: the source container
- `stream`: The stream used to copy the data.

template <typename *T*>
class HostBuffer
Data only on the host.

The data is allocated as pinned memory using the CUDA utilities. This allows to transfer asynchronously data from the device (e.g. *DeviceBuffer*).

Never releases any memory, keeps a buffer big enough to store maximum number of elements it ever held (except in the destructor).

Template Parameters

- `T`: The type of a single element to store.

Public Functions

HostBuffer(size_t *n* = 0)
construct a *HostBuffer* with a given size

Parameters

- `n`: The initial number of elements

HostBuffer(**const** *HostBuffer* &*b*)
copy constructor.

HostBuffer &**operator=**(**const** *HostBuffer* &*b*)
Assignment operator.

HostBuffer(*HostBuffer* &&*b*)
Move constructor; To enable `std::swap()`

HostBuffer &**operator=**(*HostBuffer* &&*b*)
Move assignment; To enable `std::swap()`

size_t **datatype_size**() **const**

Return the size of a single element (in bytes)

size_t **size**() **const**

Return the number of elements

T ***hostPtr**() **const**

Return pointer to host memory

T ***data**() **const**
For uniformity with std::vector.

T &**operator[]**(size_t *i*)

Return element with given index

const T &**operator[]**(size_t *i*) **const**

Return element with given index

void **resize**(size_t *n*)
resize the internal array.

Keeps the current data.

Parameters

- n: New size (in number of elements). Must be non negative.

void **resize_anew**(size_t *n*)
resize the internal array.

No guarantee to keep the current data.

Parameters

- n: New size (in number of elements). Must be non negative.

T ***begin**()
To support range-based loops.

T ***end**()
To support range-based loops.

const T ***begin**() **const**
To support range-based loops.

const T ***end**() **const**
To support range-based loops.

void **clear**()
Set all the bytes to 0.

template <**typename** Cont>
auto **copy**(**const** Cont &*cont*)
Copy data from a *HostBuffer* of the same template type.

template <**typename** Cont>
auto **copy**(**const** Cont &*cont*, cudaStream_t *stream*)
Copy data from a *DeviceBuffer* of the same template type.

void **genericCopy**(**const** *GPUcontainer* ***cont**, cudaStream_t *stream*)
Copy data from an arbitrary *GPUcontainer*.

Note the type sizes must be compatible (equal or multiple of each other)

Parameters

- `cont`: a pointer to the source container.
- `stream`: Stream used to copy the data.

template <typename *T*>
class PinnedBuffer : **public** *mirheo*::*GPUcontainer*

Device data with mirror host data.

Useful to transfer arrays between host and device memory.

The host data is allocated as pinned memory using the CUDA utilities. This allows to transfer asynchronously data from the device.

Never releases any memory, keeps a buffer big enough to store maximum number of elements it ever held (except in the destructor).

Note: Host and device data are not automatically synchronized! Use downloadFromDevice() and uploadToDevice() MANUALLY to sync

---

Template Parameters

- `T`: The type of a single element to store.

Public Functions

PinnedBuffer(size_t *n* = 0)

Construct a *PinnedBuffer* with given number of elements.

Parameters

- `n`: initial number of elements. Must be non negative.

PinnedBuffer(**const** *PinnedBuffer* &*b*)

Copy constructor.

PinnedBuffer &**operator=**(**const** *PinnedBuffer* &*b*)

assignment operator

PinnedBuffer(*PinnedBuffer* &&*b*)

Move constructor; To enable `std::swap()`

PinnedBuffer &**operator=**(*PinnedBuffer* &&*b*)

Move assignment; To enable `std::swap()`

size_t **datatype_size**() **const**

Return the size (in bytes) of a single element

size_t **size**() **const**

Return number of stored elements

void ***genericDevPtr**() **const**

Return pointer to device data

void **resize**(size_t *n*, cudaStream_t *stream*)
resize the internal array.

Keeps the current data.

Parameters

- `n`: New size (in number of elements). Must be non negative.
- `stream`: Used to copy the data internally

void **resize_anew**(size_t *n*)
resize the internal array.

No guarantee to keep the current data.

Parameters

- `n`: New size (in number of elements). Must be non negative.

GPUcontainer ***produce**() **const**
Create a new instance of the concrete container implementation.

T ***hostPtr**() **const**

Return pointer to host data

T ***data**() **const**
For uniformity with std::vector.

T ***devPtr**() **const**

Return pointer to device data

T &**operator[]**(size_t *i*)
allow array-like bracketed access to HOST data

const T &**operator[]**(size_t *i*) **const**
allow array-like bracketed access to HOST data

T ***begin**()
To support range-based loops.

T ***end**()
To support range-based loops.

const T ***begin**() **const**
To support range-based loops.

const T ***end**() **const**
To support range-based loops.

void **downloadFromDevice**(cudaStream_t *stream*, ContainersSynch *synch* = ContainersSynch::Synch)
Copy internal data from device to host.

Parameters

- `stream`: The stream used to perform the copy

- `synch`: Synchronicity of the operation. If synchronous, the call will block until the operation is done.

void **uploadToDevice**(cudaStream_t *stream*)
Copy the internal data from host to device.

Parameters

- `stream`: The stream used to perform the copy

void **clear**(cudaStream_t *stream*)
Set all the bytes to 0 on both host and device.

void **clearDevice**(cudaStream_t *stream*)
Set all the bytes to 0 on device only.

void **clearHost**()
Set all the bytes to 0 on host only.

void **copy**(**const** *DeviceBuffer*<T> &*cont*, cudaStream_t *stream*)
Copy data from a *DeviceBuffer* of the same template type.

void **copy**(**const** *HostBuffer*<T> &*cont*)
Copy data from a *HostBuffer* of the same template type.

void **copy**(**const** *PinnedBuffer*<T> &*cont*, cudaStream_t *stream*)
Copy data from a *PinnedBuffer* of the same template type.

void **copyDeviceOnly**(**const** *PinnedBuffer*<T> &*cont*, cudaStream_t *stream*)
Copy data from device pointer of a *PinnedBuffer* of the same template type.

void **copy**(**const** *PinnedBuffer*<T> &*cont*)
synchronous copy

18.5 Datatypes

A set of simple POD structures.

struct Real3_int
Helper class for packing/unpacking *real3* + *integer* into *real4*.

Public Functions

Real3_int(**const** *Real3_int* &*x*)
copy constructor

Real3_int &**operator=**(**const** *Real3_int* &*x*)
assignment operator

Real3_int()
defult constructor; NO default values!

Real3_int(real3 *vecPart*, integer *intPart*)
Constructor from vector and integer.

Real3_int(**const** real4 *r4*)
Constructor from 4 components vector; the last one will be reinterpreted to integer (not converted)

real4 **toReal4**() **const**

Return reinterpreted values packed in a real4 (no conversion)

void **mark**()
Mark this object; see *isMarked()*.

Does not modify the integer part

bool **isMarked**() **const**

Return `true` if the object has been marked via *mark()*

Public Members

real3 **v**
vector part

integer **i**
integer part

Public Static Attributes

constexpr real **mark_val** = -8.0e10_r
A special value used to mark particles.

Marked particles will be deleted during cell list rebuild. For objects, objects with all particles marked will be removed during object redistribution.

struct Particle
Structure that holds position, velocity and global index of one particle.

Due to performance reasons it should be aligned to 16 bytes boundary, therefore 8 bytes = 2 integer numbers are extra. The integer fields are used to store the global index

Public Functions

Particle(**const** *Particle* &*x*)
Copy constructor uses efficient 16-bytes wide copies.

Particle &**operator=**(*Particle* *x*)
Assignment operator uses efficient 16-bytes wide copies.

Particle()
Default constructor.

> **Attention:** The default constructor DOES NOT initialize any members!

void **setId**(int64_t *id*)
Set the global index of the particle.

int64_t **getId**() **const**

Return the global index of the particle

Particle(**const** real4 *r4*, **const** real4 *u4*)
Construct a *Particle* from two real4 entries.

Parameters

- `r4`: first three reals will be position (*r*), last one .*w* - *i1* (reinterpreted, not converted)
- `u4`: first three reals will be velocity (*u*), last one .*w* - *i2* (reinterpreted, not converted)

void **readCoordinate**(**const** real4 **addr*, **const** int *pid*)
read position from array and stores it internally

Parameters

- `addr`: start of the array with size > `pid`
- `pid`: particle index

void **readVelocity**(**const** real4 **addr*, **const** int *pid*)
read velocity from array and stores it internally

Parameters

- `addr`: pointer to the start of the array. Must be larger than `pid`
- `pid`: particle index

Real3_int **r2Real3_int**() **const**

Return packed *r* and *i1* as Real3_int

real4 **r2Real4**() **const**
Helps writing particles back to *real4* array.

Return packed *r* and *i1* as *real4*

Real3_int **u2Real3_int**() **const**

Return packed *u* and *i2* as Real3_int

real4 **u2Real4**() **const**
Helps writing particles back to *real4* array.

Return packed *u* and *i2* as *real4*

void **write2Real4**(real4 **pos*, real4 **vel*, int *pid*) **const**
Helps writing particles back to *real4* arrays.

Parameters

- `pos`: destination array that contains positions
- `vel`: destination array that contains velocities
- `pid`: particle index

void **mark**()
mark the particle; this will erase its position information

bool **isMarked**() **const**

Return `true` if the particle has been marked

Public Members

real3 **r**
position

integer **i1**
lower part of particle id

real3 **u**
velocity

integer **i2** = {0}
higher part of particle id

struct Force
Structure that holds force as *real4* (to reduce number of load/store instructions)

Due to performance reasons it should be aligned to 16 bytes boundary. The integer field is not reserved for anything at the moment

Public Functions

Force()
default constructor, does NOT initialize anything

Force(**const** real3 *vecPart*, int *intPart*)
Construct a *Force* from a vector part and an integer part.

Force(**const** real4 *f4*)
Construct a force from `real4`.

The 4th component will be reinterpreted as an integer (not converted)

real4 **toReal4**() **const**

Return packed real part + integer part as *real4*

Public Members

real3 **f**
Force value.

integer **i**
extra integer variable (unused)

struct Stress
Store a symmetric stess tensor in 3 dimensions.

Since it is symmetric, only 6 components are needed (diagonal and upper part

Public Members

real **xx**
: x diagonal term

real **xy**
: xy upper term

real **xz**
: xz upper term

real **yy**
: y diagonal term

real **yz**
: yz upper term

real **zz**
: z diagonal term

struct COMandExtent
: Contains the rigid object center of mass and bounding box Used e.g.

to decide which domain the objects belong to and what particles / cells are close to it

Public Members

real3 **com**
: center of mass

real3 **low**
: lower corner of the bounding box

real3 **high**
: upper corner of the bounding box

struct ComQ
: Contains coordinates of the center of mass and orientation of an object Used to initialize object positions.

Public Members

real3 **r**
: object position

real4 **q**
: quaternion that represents the orientation

18.6 Domain

In Mirheo, the simulation domain has a rectangular shape subdivided in equal subdomains. Each simulation rank is mapped to a single subdomain in a cartesian way. Furthermore, we distinguish the global coordinates (that are the same for all ranks) from the local coordinates (different from one subdomain to another). The *mirheo::DomainInfo* utility class provides a description of the domain, subdomain and a mapping between the coordinates of these two entities.

API

DomainInfo `mirheo::createDomainInfo`(MPI_Comm *cartComm*, real3 *globalSize*)
Construct a *DomainInfo*.

Return The *DomainInfo*

Parameters

- `cartComm`: A cartesian MPI communicator of the simulation
- `globalSize`: The size of the whole simulation domain

struct DomainInfo
Describes the simulation global and local domains, with a mapping between the two.

The simulation domain is a rectangular box. It is splitted into smaller rectangles, one by simulation rank. Each of these subdomains have a local system of coordinates, centered at the center of these rectangular boxes. The global system of coordinate has the lowest corner of the domain at (0,0,0).

Public Functions

real3 **local2global**(real3 *x*) **const**
Convert local coordinates to global coordinates.

Return The position `x` expressed in global coordinates

Parameters

- `x`: The local coordinates in the current subdomain

real3 **global2local**(real3 *x*) **const**
Convert global coordinates to local coordinates.

Return The position `x` expressed in local coordinates

Parameters

- `x`: The global coordinates in the simulation domain

template <**typename** RealType3>
bool **inSubDomain**(RealType3 *xg*) **const**
Checks if the global coordinates `xg` are inside the current subdomain.

Return `true` if `xg` is inside the current subdomain, `false` otherwise

Parameters

- `xg`: The global coordinates in the simulation domain

Public Members

real3 **globalSize**
Size of the whole simulation domain.

real3 **globalStart**
coordinates of the lower corner of the local domain, in global coordinates

real3 **localSize**
: size of the sub domain in the current rank.

18.7 Exchangers

A set of classes responsible to:

- exchange ghost paticles between neighbouring ranks
- redistribite particles accross ranks

The implementation is split into two parts:

- *exchanger classes*, that are responsible to pack and unpack the data from `mirheo::ParticleVector` to buffers (see also *Packers*).
- *communication engines*, that communicate the buffers created by the exchangers between ranks. The user must instantiate one engine per exchanger.

Exchanger classes

Different kind of exchange are implemented in Mirheo:

- Redistribution: the data is migrated from one rank to another (see `mirheo::ParticleRedistributor` and `mirheo::ObjectRedistributor`)
- Ghost particles: the data is copied from one rank to possibly multiple ones (see `mirheo::ParticleHaloExchanger`, `mirheo::ObjectHaloExchanger` and `mirheo::ObjectExtraExchanger`)
- Reverse exchange: data is copied from possibly multiple ranks to another. This can be used to gather e.g. the forces computed on ghost particles, and therefore is related to the ghost particles exchangers. (see `mirheo::ObjectReverseExchanger`)

In general, the process consists in:

1. Create a map from particle/object to buffer(s) (this step might be unnecessary for e.g. `mirheo::ObjectExtraExchanger` and `mirheo::ObjectReverseExchanger`)
2. Pack the data into the send buffers according to the map
3. The *communication engines* communicate the data to the recv buffers (not the exchangers job)
4. Unpack the data from recv buffers to a local container.

Interface

class Exchanger
: Pack and unpack `ParticleVector` objects for exchange.

  The user should register one (or more) *ExchangeEntity* objects that represent the data to exchange. The functions interface functions can then be called in the correct order to pack and unpack the data.

  Designed to be used with an *ExchangeEngine*.

  Subclassed by *mirheo::ObjectExtraExchanger*, *mirheo::ObjectHaloExchanger*, *mirheo::ObjectRedistributor*, *mirheo::ObjectReverseExchanger*, *mirheo::ParticleHaloExchanger*, *mirheo::ParticleRedistributor*

Public Functions

void **addExchangeEntity**(std::unique_ptr<*ExchangeEntity*> &&*e*)
register an *ExchangeEntity* in this exchanger.

Parameters

- `e`: The *ExchangeEntity* object to register. Will pass ownership.

ExchangeEntity ***getExchangeEntity**(size_t *id*)

Return *ExchangeEntity* with the given id (0 <= id < *getNumExchangeEntities()*).

const *ExchangeEntity* ***getExchangeEntity**(size_t *id*) **const**
see *getExchangeEntity()*

size_t **getNumExchangeEntities**() **const**

Return The number of registered *ExchangeEntity*.

virtual void **prepareSizes**(size_t *id*, cudaStream_t *stream*) = 0
Compute the sizes of the data to be communicated in the given *ExchangeEntity*.

After this call, the `send.sizes`, `send.sizeBytes`, `send.offsets` and `send.offsetsBytes` of the *ExchangeEntity* are available on the CPU.

Parameters

- `id`: The index of the concerned *ExchangeEntity*
- `stream`: Execution stream

virtual void **prepareData**(size_t *id*, cudaStream_t *stream*) = 0
Pack the data managed by the given *ExchangeEntity*.

Note Must be executed after *prepareSizes()*

Parameters

- `id`: The index of the concerned *ExchangeEntity*
- `stream`: Execution stream

virtual void **combineAndUploadData**(size_t *id*, cudaStream_t *stream*) = 0
Unpack the received data.

After this call, the `recv.sizes`, `recv.sizeBytes`, `recv.offsets` and `recv.offsetsBytes` of the *ExchangeEntity* must be available on the CPU and GPU before this call. Furthermore, the recv buffers must already be on the device memory.

Parameters

- `id`: The index of the concerned *ExchangeEntity*
- `stream`: Execution stream

Note Must be executed after *prepareData()*

virtual bool **needExchange** (size_t *id*) = 0
Stats if the data of an *ExchangeEntity* needs to be exchanged.

If the *ParticleVector* didn't change since the last exchange, there is no need to run the exchange again. This function controls such behaviour.

Return `true` if exchange is required, `false` otherwise

Parameters

- `id`: The index of the concerned *ExchangeEntity*

Derived classes

class ParticleRedistributor : **public** *mirheo*::*Exchanger*
Pack and unpack data for particle redistribution.

The redistribution consists in moving (not copying) the particles from one rank to the other. It affects all particles that have left the current subdomain. The redistribution is accelerated by making use of the cell-lists of the *ParticleVector*. This allows to check only the particles that are on the boundary cells; However, this assumes that only those particles leave the domain.

Public Functions

ParticleRedistributor()
default constructor

void **attach** (*ParticleVector* **pv*, *CellList* **cl*)
Add a *ParticleVector* to the redistribution.

Multiple *ParticleVector* objects can be attached to the same redistribution object.

Parameters

- `pv`: The *ParticleVector* to attach
- `cl`: The associated cell-list of `pv`.

class ObjectRedistributor : **public** *mirheo*::*Exchanger*
Pack and unpack data for object redistribution.

As opposed to particles, objects must be redistributed as a whole for two reasons:

- the particles of one object must stay into a contiguous chunk in memory
- the objects might have associated data per object (or per bisegments for rods)

The redistribution consists in moving (not copying) the object data from one rank to the other. It affects all objects that have left the current subdomain (an object belongs to a subdomain if its center of mass is inside).

Public Functions

ObjectRedistributor()
default constructor

void **attach** (*ObjectVector* **ov*)
Add an *ObjectVector* to the redistribution.

Multiple *ObjectVector* objects can be attached to the same redistribution object.

Parameters

- `ov`: The *ObjectVector* to attach

class ParticleHaloExchanger : **public** *mirheo*::*Exchanger*

Pack and unpack data for halo particles exchange.

The halo exchange consists in copying an image of all particles that are within one cut-off radius away to the neighbouring ranks. This leaves the original *ParticleVector* local data untouched. The result of this operation is stored in the halo *LocalParticleVector*.

The halo exchange is accelerated by making use of the associated *CellList* of the *ParticleVector*.

Public Functions

ParticleHaloExchanger()

default constructor

void **attach**(*ParticleVector* *pv, *CellList* *cl, **const** std::vector<std::string> &*extraChannelNames*)

Add a *ParticleVector* for halo exchange.

Multiple *ParticleVector* objects can be attached to the same halo exchanger.

Parameters

- `pv`: The *ParticleVector* to attach
- `cl`: The associated cell-list of `pv`
- `extraChannelNames`: The list of channels to exchange (additionally to the default positions and velocities)

class ObjectHaloExchanger : **public** *mirheo*::*Exchanger*

Pack and unpack data for halo object exchange.

The halo exchange consists in copying an image of all objects with bounding box that is within one cut-off radius away to the neighbouring ranks. This leaves the original *ObjectVector* local data untouched. The result of this operation is stored in the halo *LocalObjectVector*.

This is needed only when the full object is needed on the neighbour ranks (e.g. `Bouncer` or *ObjectBelongingChecker*).

Public Functions

ObjectHaloExchanger()

default constructor

void **attach**(*ObjectVector* *ov, real *rc*, **const** std::vector<std::string> &*extraChannelNames*)

Add a *ObjectVector* for halo exchange.

Multiple *ObjectVector* objects can be attached to the same halo exchanger.

Parameters

- `ov`: The *ObjectVector* to attach
- `rc`: The required cut-off radius
- `extraChannelNames`: The list of channels to exchange (additionally to the default positions and velocities)

PinnedBuffer<int> &**getSendOffsets**(size_t *id*)

Return send offset within the send buffer (in number of elements) of the given ov

PinnedBuffer<int> &**getRecvOffsets**(size_t *id*)

Return recv offset within the send buffer (in number of elements) of the given ov

DeviceBuffer<MapEntry> &**getMap**(size_t *id*)

Return The map from *LocalObjectVector* to send buffer ids

class ObjectExtraExchanger : **public** *mirheo*::*Exchanger*

Pack and unpack extra data for halo object exchange.

This class only exchanges the additional data (not e.g. the default particle's positions and velocities). It uses the packing map from an external *ObjectHaloExchanger*. The attached *ObjectVector* objects must be the same as the ones in the external *ObjectHaloExchanger* (and in the same order).

See *ObjectHaloExchanger*

Public Functions

ObjectExtraExchanger(*ObjectHaloExchanger* **entangledHaloExchanger*)

Construct a *ObjectExtraExchanger*.

Parameters

- `entangledHaloExchanger`: The object that will contain the packing maps.

void **attach**(*ObjectVector* **ov*, **const** std::vector<std::string> &*extraChannelNames*)

Add a *ObjectVector* for halo exchange.

Parameters

- `ov`: The *ObjectVector* to attach
- `extraChannelNames`: The list of channels to exchange

class ObjectReverseExchanger : **public** *mirheo*::*Exchanger*

Pack and unpack data from ghost particles back to the original bulk data.

The ghost particles data must come from a *ObjectHaloExchanger* object. The attached *ObjectVector* objects must be the same as the ones in the external *ObjectHaloExchanger* (and in the same order).

Public Functions

ObjectReverseExchanger(*ObjectHaloExchanger* **entangledHaloExchanger*)

Construct a *ObjectReverseExchanger*.

Parameters

- `entangledHaloExchanger`: The object that will create the ghost particles.

void **attach**(*ObjectVector* **ov*, std::vector<std::string> *channelNames*)

Add an *ObjectVector* for reverse halo exchange.

Parameters

- `ov`: The *ObjectVector* to attach
- `channelNames`: The list of channels to send back

Exchange Entity

Helper classes responsible to hold the buffers of the packed data to be communicated.

struct BufferOffsetsSizesWrap

A device-compatible structure that holds buffer information for packing / unpacking data.

In general, there is one buffer per source/destination rank. The current implementation uses a single array that contains all buffers in a contiguous way. The offsetsBytes values (one per buffer) state where each buffer start within the array.

Public Functions

char ***getBuffer**(int *bufId*)

Return buffer with id `bufId`

Public Members

int **nBuffers**

number of buffers

char ***buffer**

device data pointer to the array containing all buffers

int ***offsets**

device array of size *nBuffers*+1 with i-th buffer start index (in number of elements)

int ***sizes**

device array of size *nBuffers* with i-th buffer size (in number of elements)

size_t ***offsetsBytes**

device array of size *nBuffers*+1 with i-th buffer start index (in number of bytes)

struct BufferInfos

Structure held on the host only that contains pack/unpack buffers and their sizes/offsets (see *BufferOffsetsSizesWrap*).

Public Functions

void **clearAllSizes**(cudaStream_t *stream*)

set sizes and sizesBytes to zero on host and device

void **resizeInfos**(int *nBuffers*)

resize the size and offset buffers to support a given number of buffers

void **uploadInfosToDevice**(cudaStream_t *stream*)

upload all size and offset information from host to device

char ***getBufferDevPtr**(int *bufId*)

Return The device pointer to the buffer with the given index

Public Members

PinnedBuffer<int> **sizes**
number of elements in each buffer

PinnedBuffer<int> **offsets**
prefix sum of the above

PinnedBuffer<size_t> **sizesBytes**
number of bytes per buffer

PinnedBuffer<size_t> **offsetsBytes**
start of each buffer (in bytes) within the array

PinnedBuffer<char> **buffer**
all buffers in contiguous memory.

std::vector<MPI_Request> **requests**
send or recv requests associated to each buffer; only relevant for *MPIExchangeEngine*

class ExchangeEntity
Manages communication data per *ParticleVector*.

Each *ExchangeEntity* holds send and recv *BufferInfos* object for a given *ParticleVector*.

Public Functions

ExchangeEntity(std::string *name*, int *uniqueId*, *ParticlePacker* *packer)
Construct an *ExchangeEntity* object.

Parameters

- `name`: The name of the Corresponding *ParticleVector*
- `uniqueId`: A positive integer. This must be unique when a collection of *ExchangeEntity* objects is registered in a single *Exchanger*.
- `packer`: The class used to pack/unpack the data into buffers

void **computeRecvOffsets**()
Compute the recv offsets and offsetsBytes on the host.

Note the recv sizes must be available on the host.

void **computeSendOffsets**()
Compute the send offsets and offsetsBytes on the host.

Note the send sizes must be available on the host.

void **computeSendOffsets_Dev2Dev**(cudaStream_t *stream*)
Compute the send offsets and offsetBytes on the device and download all sizes and offsets on the host.

Note The send sizes must be available on the device

void **resizeSendBuf**()
resize the internal send buffers; requires send offsetsBytes to be available on the host

void **resizeRecvBuf**()
resize the internal recv buffers; requires recv offsetsBytes to be available on the host

int **getUniqueId**() **const**

Return the unique id

BufferOffsetsSizesWrap **wrapSendData**()

Return a *BufferOffsetsSizesWrap* from the send *BufferInfos*

BufferOffsetsSizesWrap **wrapRecvData**()

Return a *BufferOffsetsSizesWrap* from the recv *BufferInfos*

const std::string &**getName**() **const**

Return the name of the attached *ParticleVector*

const char ***getCName**() **const**

Return the name of the attached *ParticleVector* in c-style string

Public Members

const int **nBuffers** = fragment_mapping::numFragments
equal to number of neighbours + 1 (for bulk)

const int **bulkId** = fragment_mapping::bulkId
The index of the bulk buffer.

BufferInfos **send**
buffers for the send data

BufferInfos **recv**
buffers for the recv data

std::vector<int> **recvRequestIdxs**
only relevant for *MPIExchangeEngine*

Communication engines

Interface

class ExchangeEngine
Base communication engine class.

Responsible to communicate the data managed by an `Exchanger` between different subdomains. The communication is split into two parts so that asynchronous communication can be used. Every *init()* call must have a single *finalize()* call that follows.

Subclassed by *mirheo::MPIExchangeEngine*, *mirheo::SingleNodeExchangeEngine*

Public Functions

ExchangeEngine(std::unique_ptr<*Exchanger*> &&*exchanger*)
Construct a communication engine.

Parameters

- `exchanger`: The *`Exchanger`* object that will prepare the data to communicate. The ownership of exchanger is transfered to the engine.

virtual void **init**(cudaStream_t *stream*) = 0
Initialize the communication.

The data packing from the exchanger happens in this step.

Parameters

- `stream`: Execution stream used to prepare / download the data

virtual void **finalize**(cudaStream_t *stream*) = 0
Finalize the communication.

Must follow a pending *init()* call. The data unpacking from the exchanger happens in this step.

Parameters

- `stream`: Execution stream used to upload / unpack the data

Derived classes

class MPIExchangeEngine : **public** *mirheo*::*ExchangeEngine*
Engine implementing asynchronous MPI communication.

The pipeline is as follows:

- *init()* prepares the data into buffers, exchange the sizes, allocate recv buffers and post the asynchronous communication calls.
- *finalize()* waits for the communication to finish and unpacks the data.

Public Functions

MPIExchangeEngine(std::unique_ptr<*Exchanger*> &&*exchanger*, MPI_Comm *comm*, bool *gpuAwareMPI*)
Construct a *MPIExchangeEngine*.

Parameters

- `exchanger`: The class responsible to pack and unpack the data.
- `comm`: The cartesian communicator that represents the simulation domain.
- `gpuAwareMPI`: `true` to enable RDMA implementation. Only works if the MPI library has this feature implemented.

void **init**(cudaStream_t *stream*)
Initialize the communication.

The data packing from the exchanger happens in this step.

Parameters

- `stream`: Execution stream used to prepare / download the data

void **finalize**(cudaStream_t *stream*)
Finalize the communication.

Must follow a pending *init()* call. The data unpacking from the exchanger happens in this step.

Parameters

- `stream`: Execution stream used to upload / unpack the data

class SingleNodeExchangeEngine : **public** *mirheo*::*ExchangeEngine*
Special engine optimized for single node simulations.

Instead of communicating thedata through MPI, the send and recv buffers are simply swapped.

Public Functions

SingleNodeExchangeEngine(std::unique_ptr<*Exchanger*> &&*exchanger*)
Construct a *SingleNodeExchangeEngine*.

Parameters

- `exchanger`: The class responsible to pack and unpack the data.

void **init**(cudaStream_t *stream*)
Initialize the communication.

The data packing from the exchanger happens in this step.

Parameters

- `stream`: Execution stream used to prepare / download the data

void **finalize**(cudaStream_t *stream*)
Finalize the communication.

Must follow a pending *init()* call. The data unpacking from the exchanger happens in this step.

Parameters

- `stream`: Execution stream used to upload / unpack the data

18.8 Field

Interface

class FieldDeviceHandler
a device-compatible structure that represents a scalar field

Subclassed by *mirheo::Field*

Public Functions

real **operator()**(real3 *x*) **const**
Evaluate the field at a given position.

> **Warning:** The position must be inside the subdomain enlarged with a given margin (see c Field)

Return The scalar value at `x`

Parameters

- `x`: The position, in local coordinates

class Field : **public** *mirheo*::*FieldDeviceHandler*, **public** *mirheo*::*MirSimulationObject*

Driver class used to create a *FieldDeviceHandler*.

Subclassed by *mirheo::FieldFromFile*, *mirheo::FieldFromFunction*

Public Functions

Field(**const** *MirState* **state*, std::string *name*, real3 *h*, real3 *margin*)

Construct a `Field` object.

Parameters

- `state`: The global state of the system
- `name`: The name of the field object
- `h`: the grid size
- `margin`: Additional margin to store in each rank

Field(*Field*&&)

move constructor

const *FieldDeviceHandler* &**handler**() **const**

Return The handler that can be used on the device

virtual void **setup**(**const** MPI_Comm &*comm*) = 0

Prepare the internal state of the `Field`.

Must be called before *handler()*.

Parameters

- `comm`: The cartesian communicator of the domain.

Derived classes

class FieldFromFile : **public** *mirheo*::*Field*

a `Field` that can be initialized from a file

Public Functions

FieldFromFile(**const** *MirState* **state*, std::string *name*, std::string *fieldFileName*, real3 *h*, real3 *margin*)

Construct a *FieldFromFile* object.

The format of the file is custom. It is a single file that contains a header followed by the data grid data in binary format. The header is composed of two lines in ASCII format:

- domain size (3 floating point numbers)
- number of grid points (3 integers)

Parameters

- `state`: The global state of the system
- `name`: The name of the field object
- `fieldFileName`: The input file name
- `h`: the grid size
- `margin`: Additional margin to store in each rank

The data is an array that contains all grid values (x is the fast running index).

FieldFromFile(*FieldFromFile*&&)
move constructor

void **setup**(**const** MPI_Comm &*comm*)
Prepare the internal state of the *Field*.

Must be called before *handler()*.

Parameters

- `comm`: The cartesian communicator of the domain.

class FieldFromFunction : **public** *mirheo*::*Field*
a *Field* that can be initialized from FieldFunction

Public Functions

FieldFromFunction(**const** *MirState* **state*, std::string *name*, FieldFunction *func*, real3 *h*, real3 *margin*)
Construct a *FieldFromFunction* object.

The scalar values will be discretized and stored on the grid. This can be useful as one can have a general scalar field configured on the host (e.g. from python) but usable on the device.

Parameters

- `state`: The global state of the system
- `name`: The name of the field object
- `func`: The scalar field function
- `h`: the grid size
- `margin`: Additional margin to store in each rank

FieldFromFunction(*FieldFromFunction*&&)
move constructor

void **setup**(**const** MPI_Comm &*comm*)
Prepare the internal state of the *Field*.

Must be called before *handler()*.

Parameters

- `comm`: The cartesian communicator of the domain.

Utilities

template <**typename** FieldHandler>
real3 *mirheo*::**computeGradient**(**const** FieldHandler &*field*, real3 *x*, real *h*)
compute the gradient of a scalar field using finite differences on the device

Return The approximation of the gradient of `field` at `x`

Template Parameters

- `FieldHandler`: Type of device handler describing the field. Must contain parenthesis operator

Parameters

- `field`: The functor that describes the continuous scalar field
- `x`: The position at which to compute the gradient
- `h`: The step size used to compute the gradient

18.9 Initial Conditions

See also *the user interface*.

Base class

class InitialConditions
Initializer for objects in group PVs.

ICs are temporary objects and do not need name or checkpoint/restart mechanism. The *exec()* member function is called by the *Simulation* when the *ParticleVector* is registered.

Subclassed by *mirheo::FromArrayIC*, *mirheo::MembraneIC*, mirheo::RandomChainsIC, *mirheo::RestartIC*, *mirheo::RigidIC*, *mirheo::RodIC*, mirheo::StraightChainsIC, *mirheo::UniformFilteredIC*, *mirheo::UniformIC*, *mirheo::UniformSphereIC*

Public Functions

virtual void **exec**(**const** MPI_Comm &*comm*, *ParticleVector* **pv*, cudaStream_t *stream*) = 0
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

Derived classes

class RestartIC : **public** *mirheo*::*InitialConditions*
Initialize a *ParticleVector* from a checkpoint file.

Will call the restart() member function of the given *ParticleVector*.

Public Functions

RestartIC(**const** std::string &*path*)
Construct a *RestartIC* object.

Parameters

- `path`: The directory containing the restart files.

void **exec**(**const** MPI_Comm &*comm*, *ParticleVector* **pv*, cudaStream_t *stream*)
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

class UniformIC : **public** *mirheo*::*InitialConditions*
Fill the domain with uniform number density.

Initialize particles uniformly with the given number density on the whole domain. The domain considered is that of the *ParticleVector*. *ObjectVector* objects are not supported.

Public Functions

UniformIC(real *numDensity*)
Construct a *UniformIC* object.

Parameters

- `numDensity`: Number density of the particles to initialize

void **exec**(**const** MPI_Comm &*comm*, *ParticleVector* **pv*, cudaStream_t *stream*)
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

class UniformSphereIC : **public** *mirheo*::*InitialConditions*
Fill the domain with uniform number density in a given ball.

Initialize particles uniformly with the given number density inside or outside a ball. The domain considered is that of the *ParticleVector*. *ObjectVector* objects are not supported.

Public Functions

UniformSphereIC (real *numDensity*, real3 *center*, real *radius*, bool *inside*)
Construct a *UniformSphereIC* object.

Parameters

- `numDensity`: Number density of the particles to initialize
- `center`: Center of the ball
- `radius`: Radius of the ball
- `inside`: The particles will be inside the ball if set to `true`, outside otherwise.

void **exec** (**const** MPI_Comm &*comm*, *ParticleVector* **pv*, cudaStream_t *stream*)
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

class UniformFilteredIC : **public** *mirheo*::*InitialConditions*
Fill the domain with uniform number density in a given region.

Initialize particles uniformly with the given number density on a specified region of the domain. The region is specified by a filter functor. The domain considered is that of the *ParticleVector*. *ObjectVector* objects are not supported.

Public Functions

UniformFilteredIC (real *numDensity*, PositionFilter *filter*)
Construct a *UniformFilteredIC* object.

Parameters

- `numDensity`: Number density of the particles to initialize
- `filter`: Indicator function that maps a position of the domain to a boolean value. It returns `true` if the position is inside the region.

void **exec** (**const** MPI_Comm &*comm*, *ParticleVector* **pv*, cudaStream_t *stream*)
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

class FromArrayIC : **public** *mirheo*::*InitialConditions*
Initialize particles to given positions and velocities.

ObjectVector objects are not supported.

Public Functions

FromArrayIC(**const** std::vector<real3> &*pos*, **const** std::vector<real3> &*vel*)
Construct a *FromArrayIC* object.

Parameters

- `pos`: list of initial positions in global coordinates. The size will determine the maximum number of particles. Positions outside the domain are filtered out.
- `vel`: list of initial velocities. Must have the same size as `pos`.

void **exec**(**const** MPI_Comm &*comm*, *ParticleVector* **pv*, cudaStream_t *stream*)
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

class RigidIC : **public** *mirheo*::*InitialConditions*
Initialize *RigidObjectVector* objects.

Initialize rigid objects from center of mass positions, orientations and frozen particles.

Public Functions

RigidIC(**const** std::vector<*ComQ*> &*comQ*, **const** std::string &*xyzfname*)
Construct a *RigidIC* object.

This method will die if the file does not exist.

Parameters

- `comQ`: List of (position, orientation) corresponding to each object. The size of the list is the number of rigid objects that will be initialized.
- `xyzfname`: The name of a file in xyz format. It contains the list of coordinates of the frozen particles (in the object frame of reference).

RigidIC(**const** std::vector<*ComQ*> &*comQ*, **const** std::vector<real3> &*coords*)
Construct a *RigidIC* object.

Parameters

- `comQ`: List of (position, orientation) corresponding to each object. The size of the list is the number of rigid objects that will be initialized.
- `coords`: List of positions of the frozen particles of one object, in the object frame of reference.

RigidIC(**const** std::vector<*ComQ*> &*comQ*, **const** std::vector<real3> &*coords*, **const** std::vector<real3> &*comVelocities*)
Construct a *RigidIC* object.

Parameters

- `comQ`: List of (position, orientation) corresponding to each object. The size of the list is the number of rigid objects that will be initialized.
- `coords`: List of positions of the frozen particles of one object, in the object frame of reference.
- `comVelocities`: List of velocities of the velocities of the objects center of mass. Must have the same size as `comQ`.

void **exec**(**const** MPI_Comm &*comm*, *ParticleVector* **pv*, cudaStream_t *stream*)
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

class MembraneIC : **public** *mirheo*::*InitialConditions*
Initialize *MembraneVector* objects.

Initialize membrane objects from center of mass positions and orientations.

Subclassed by *mirheo::MembraneWithTypeIdsIC*

Public Functions

MembraneIC(**const** std::vector<*ComQ*> &*comQ*, real *globalScale* = 1.0)
Construct a *MembraneIC* object.

Parameters

- `comQ`: List of (position, orientation) corresponding to each object. The size of the list is the number of membrane objects that will be initialized.
- `globalScale`: scale the membranes by this scale when placing the initial vertices.

void **exec**(**const** MPI_Comm &*comm*, *ParticleVector* **pv*, cudaStream_t *stream*)
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

class MembraneWithTypeIdsIC : **public** *mirheo*::*MembraneIC*
Initialize *MembraneVector* objects with a typeId.

See *MembraneIC*. Attach an additional typeId field to each membrane. This is useful to have different membrane forces without having many *MembraneVector* objects.

Public Functions

MembraneWithTypeIdsIC(**const** std::vector<*ComQ*> &*comQ*, **const** std::vector<int> &*typeIds*, real *globalScale* = 1.0)
Construct a *MembraneWithTypeIdsIC* object.

Parameters

- `comQ`: List of (position, orientation) corresponding to each object. The size of the list is the number of membrane objects that will be initialized.
- `typeIds`: List of type Ids. must have the same size as `comQ`.
- `globalScale`: scale the membranes by this scale when placing the initial vertices.

void **exec**(**const** MPI_Comm &*comm*, *ParticleVector* *pv, cudaStream_t *stream*)
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

class RodIC : **public** *mirheo*::*InitialConditions*
Initialize *RodVector* objects.

All rods will have the same torsion and centerline in there frame of reference. Each rod has a specific center of mass and orientation.

Public Types

using MappingFunc3D = std::function<real3 (real) >
a map from [0,1] to R^3

using MappingFunc1D = std::function<real (real) >
a map from [0,1] to R

Public Functions

RodIC(**const** std::vector<*ComQ*> &*comQ*, *MappingFunc3D* *centerLine*, *MappingFunc1D* *torsion*, real *a*, real3 *initialMaterialFrame* = *DefaultFrame*)
Construct a *RodIC* object.

Parameters

- `comQ`: list of center of mass and orientation of each rod. This will determine the number of rods. The rods with center of mass outside of the domain will be discarded.
- `centerLine`: Function describing the centerline in the frame of reference of the rod
- `torsion`: Function describing the torsion along the centerline.
- `a`: The width of the rod (the cross particles are separated by `a`).
- `initialMaterialFrame`: If set, this describes the orientation of the local material frame at the start of the rod (in the object frame of reference). If not set, this is chosen arbitrarily.

void **exec**(**const** MPI_Comm &*comm*, *ParticleVector* **pv*, cudaStream_t *stream*)
Initialize a given *ParticleVector*.

Parameters

- `comm`: A Cartesian MPI communicator from the simulation tasks
- `pv`: The resulting *ParticleVector* to be initialized (on chip data)
- `stream`: cuda stream

Public Static Attributes

const real **Default**
default real value, used to pass default parameters

const real3 **DefaultFrame**
default orientation, used to pass default parameters

18.10 Integrators

See also *the user interface*.

Base class

class Integrator : **public** *mirheo*::*MirSimulationObject*
Advance *ParticleVector* objects in time.

`Integrator` objects are responsible to advance the state of *ParticleVector* objects on the device. After executed, the *CellList* of the *ParticleVector* object might be outdated; in that case, the `Integrator` is invalidates the current cell-lists, halo and redistribution status on the *ParticleVector*.

Subclassed by *mirheo::IntegratorConstOmega*, *mirheo::IntegratorMinimize*, *mirheo::IntegratorOscillate*, *mirheo::IntegratorSubStep*, *mirheo::IntegratorSubStepShardlowSweep*, *mirheo::IntegratorTranslate*, *mirheo::IntegratorVV< ForcingTerm >*, *mirheo::IntegratorVVRigid*

Public Functions

Integrator(**const** *MirState* **state*, **const** std::string &*name*)
Construct a `Integrator` object.

Parameters

- `state`: The global state of the system. The time step and domain used during the execution are passed through this object.
- `name`: The name of the integrator.

virtual void **setPrerequisites**(*ParticleVector* **pv*)
Setup conditions on the ParticledVector.

Set specific properties to pv that will be modified during *execute()*. Default: ask nothing. Must be called before *execute()* with the same pv.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.

virtual void **execute**(*ParticleVector* *pv, cudaStream_t *stream*) = 0
Advance the ParticledVector for one time step.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.
- `stream`: The stream used for execution.

Derived classes

class IntegratorConstOmega : **public** *mirheo*::*Integrator*
Rotate *ParticleVector* objects with a constant angular velocity.

The center of rotation is defined in the global coordinate system.

Public Functions

IntegratorConstOmega(**const** *MirState* *state, **const** std::string &*name*, real3 *center*, real3 *omega*)
Construct a *IntegratorConstOmega* object.

Parameters

- `state`: The global state of the system. The time step and domain used during the execution are passed through this object.
- `name`: The name of the integrator.
- `center`: The center of rotation, in global coordinates system.
- `omega`: The angular velocity of rotation.

void **execute**(*ParticleVector* *pv, cudaStream_t *stream*)
Advance the ParticledVector for one time step.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.
- `stream`: The stream used for execution.

class IntegratorMinimize : **public** *mirheo*::*Integrator*
Energy minimization integrator.

Updates positions using a force-based gradient descent, without affecting or reading velocities.

Public Functions

IntegratorMinimize(**const** *MirState* *state, **const** std::string &*name*, real *maxDisplacement*)

Parameters

- `state`: The global state of the system. The time step and domain used during the execution are passed through this object.

- `name`: The name of the integrator.
- `maxDisplacement`: Maximum particle displacement per time step.

void **execute**(*ParticleVector* **pv*, cudaStream_t *stream*)
Advance the ParticledVector for one time step.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.
- `stream`: The stream used for execution.

class IntegratorOscillate : **public** *mirheo*::*Integrator*
Restrict *ParticleVector* velocities to a sine wave.

Set velocities to follow a sine wave:

$$v(t) = v \cos\left(\frac{2\pi t}{T}\right)$$

The positions are integrated with forwards euler from the above velocities.

Public Functions

IntegratorOscillate(**const** *MirState* **state*, **const** std::string &*name*, real3 *vel*, real *period*)

Parameters

- `state`: The global state of the system. The time step and domain used during the execution are passed through this object.
- `name`: The name of the integrator.
- `vel`: Velocity magnitude.
- `period`: The time taken for one oscillation.

void **execute**(*ParticleVector* **pv*, cudaStream_t *stream*)
Advance the ParticledVector for one time step.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.
- `stream`: The stream used for execution.

class IntegratorVVRigid : **public** *mirheo*::*Integrator*
Integrate *RigidObjectVector* objects given torque and force.

Advance the RigidMotion and the frozen particles of the RigidObjectVector objects. The particles of each object are given the velocities corresponding to the rigid object motion.

Public Functions

IntegratorVVRigid(**const** *MirState* **state*, **const** std::string &*name*)

Parameters

- `state`: The global state of the system. The time step and domain used during the execution are passed through this object.

- `name`: The name of the integrator.

void **`setPrerequisites`**(*ParticleVector* **pv*)
Setup conditions on the ParticledVector.

Set specific properties to pv that will be modified during *execute()*. Default: ask nothing. Must be called before *execute()* with the same pv.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.

void **`execute`**(*ParticleVector* **pv*, cudaStream_t *stream*)
Advance the ParticledVector for one time step.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.
- `stream`: The stream used for execution.

`class IntegratorSubStepShardlowSweep` : **`public`** *mirheo*::*Integrator*
Advance one *MembraneVector* associated with internal forces with smaller time step, similar to IntgratorSubStep.

We distinguish slow forces, which are computed outside of this class, from fast forces, computed only inside this class. Each time step given by the simulation is split into n sub time steps. Each of these sub time step advances the object using the non updated slow forces and the updated fast forces n times using the Shardlow method.

This was motivated by the separation of time scale of membrane viscosity (fast forces) and solvent viscosity (slow forces) in blood.

> **Warning:** The fast forces should NOT be registered in the c Simulation. Otherwise, it will be executed twice (by the simulation and by this class).

Public Functions

`IntegratorSubStepShardlowSweep`(**`const`** *MirState* **state*, **`const`** std::string &*name*, int *substeps*, *BaseMembraneInteraction* **fastForces*, real *gammaC*, real *kBT*, int *nsweeps*)
construct a *IntegratorSubStepShardlowSweep* object.

Parameters

- `state`: The global state of the system. The time step and domain used during the execution are passed through this object.
- `name`: The name of the integrator.
- `substeps`: Number of sub steps. Must be at least 1.
- `fastForces`: Internal interactions executed at each sub step.
- `gammaC`: The dissipation coefficient.
- `kBT`: The temperature in energy units.
- `nsweeps`: The number of iterations spent for each viscous update. Must be at least 1.

void **execute**(*ParticleVector* *pv, cudaStream_t *stream*)
Advance the ParticledVector for one time step.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.
- `stream`: The stream used for execution.

void **setPrerequisites**(*ParticleVector* *pv)
Setup conditions on the ParticledVector.

Set specific properties to pv that will be modified during *execute()*. Default: ask nothing. Must be called before *execute()* with the same pv.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.

class IntegratorSubStep : **public** *mirheo*::*Integrator*
Advance one *ObjectVector* associated with internal forces with smaller time step.

We distinguish slow forces, which are computed outside of this class, from fast forces, computed only inside this class. Each time step given by the simulation is split into n sub time steps. Each of these sub time step advances the object using the non updated slow forces and the updated fast forces n times.

This was motivated by the separation of time scale of membrane viscosity (fast forces) and solvent viscosity (slow forces) in blood.

Warning: The fast forces should NOT be registered in the c Simulation. Otherwise, it will be executed twice (by the simulation and by this class).

Public Functions

IntegratorSubStep(**const** *MirState* *state, **const** std::string &*name*, int *substeps*, **const** std::vector<*Interaction* *> &*fastForces*)
construct a *IntegratorSubStep* object.

This constructor will die if the fast forces need to exchange ghost particles with other ranks.

Parameters

- `state`: The global state of the system. The time step and domain used during the execution are passed through this object.
- `name`: The name of the integrator.
- `substeps`: Number of sub steps
- `fastForces`: Internal interactions executed at each sub step.

void **execute**(*ParticleVector* *pv, cudaStream_t *stream*)
Advance the ParticledVector for one time step.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.
- `stream`: The stream used for execution.

void **setPrerequisites**(*ParticleVector* *pv)
Setup conditions on the ParticledVector.

Set specific properties to pv that will be modified during *execute()*. Default: ask nothing. Must be called before *execute()* with the same pv.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.

class IntegratorTranslate : **public** *mirheo*::*Integrator*
Restrict *ParticleVector* velocities to a constant.

The positions are integrated with forwards euler with a constant velocity.

Public Functions

IntegratorTranslate(**const** *MirState* *state, **const** std::string &name, real3 *vel*)

Parameters

- `state`: The global state of the system. The time step and domain used during the execution are passed through this object.
- `name`: The name of the integrator.
- `vel`: Velocity magnitude.

void **execute**(*ParticleVector* *pv, cudaStream_t *stream*)
Advance the ParticledVector for one time step.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.
- `stream`: The stream used for execution.

template <**class** ForcingTerm>
class IntegratorVV : **public** *mirheo*::*Integrator*
Advance individual particles with Velocity-Verlet scheme.

Template Parameters

- `ForcingTerm`: a functor that can add additional force to the particles depending on their position

Public Functions

IntegratorVV(**const** *MirState* *state, **const** std::string &name, ForcingTerm *forcingTerm*)

Parameters

- `state`: The global state of the system. The time step and domain used during the execution are passed through this object.
- `name`: The name of the integrator.
- `forcingTerm`: Additional force added to the particles.

void **execute**(*ParticleVector* *pv, cudaStream_t *stream*)
Advance the ParticledVector for one time step.

Parameters

- `pv`: The *ParticleVector* that will be advanced in time.
- `stream`: The stream used for execution.

Forcing terms

The forcing terms must follow the same interface. Currently implemented forcing terms:

`class ForcingTermNone`
No forcing term.

Public Functions

void **`setup`**(*ParticleVector* *pv, real *t*)
Initialize internal state.

This method must be called at every time step

Parameters

- `pv`: the *ParticleVector* that will be updated
- `t`: Current simulation time

real3 **`operator()`**(real3 *original*, *Particle* *p*) **`const`**
Add the additional force to the current one on a particle.

Return The total force that must be applied to the particle

Parameters

- `original`: Original force acting on the particle
- `p`: *Particle* on which to apply the additional force

`class ForcingTermConstDP`
Apply a constant force independently of the position.

Public Functions

`ForcingTermConstDP`(real3 *extraForce*)
Construct a *ForcingTermConstDP* object.

Parameters

- `extraForce`: The force to add to every particle

void **`setup`**(*ParticleVector* *pv, real *t*)
Initialize internal state.

This method must be called at every time step.

Parameters

- `pv`: the *ParticleVector* that will be updated
- `t`: Current simulation time

real3 **operator()** (real3 *original*, *Particle* *p*) **const**
Add the additional force to the current one on a particle.

Return The total force that must be applied to the particle

Parameters

- `original`: Original force acting on the particle
- `p`: *Particle* on which to apply the additional force

class ForcingTermPeriodicPoiseuille
Apply equal but opposite forces in two halves of the global domain.

$$f_x = \begin{cases} F, & y_p > L_y/2 \\ -F, & y_p \leqslant L_y/2 \end{cases}$$

Similarly, if the force is parallel to the y axis, its sign will depend on z; parallel to z it will depend on x.

Public Types

enum Direction
Encode directions.

Values:

x

y

z

Public Functions

ForcingTermPeriodicPoiseuille (real *magnitude*, *Direction* *dir*)
Construct a *ForcingTermPeriodicPoiseuille* object.

Parameters

- `magnitude`: force magnitude to be applied.
- `dir`: The force will be applied parallel to the specified axis.

void **setup** (*ParticleVector* **pv*, real *t*)
Initialize internal state.

This method must be called at every time step.

Parameters

- `pv`: the *ParticleVector* that will be updated
- `t`: Current simulation time

real3 **operator()** (real3 *original*, *Particle* *p*) **const**
Add the additional force to the current one on a particle.

Return The total force that must be applied to the particle

Parameters

- `original`: Original force acting on the particle
- `p`: *Particle* on which to apply the additional force

18.11 Interactions

Interface

class Interaction : **public** *mirheo*::*MirSimulationObject*

Compute forces from particle interactions.

We distinguish two kinds of interactions (see Stage enum type):

1. Intermediate ones, that do not compute any force, but compute intermediate quantities (e.g. densities in SDPD).
2. Final ones, that compute forces (and possibly other quantities, e.g. stresses).

Subclassed by *mirheo::BaseMembraneInteraction*, *mirheo::BasePairwiseInteraction*, *mirheo::BaseRodInteraction*, mirheo::ChainInteraction, *mirheo::ObjectBindingInteraction*, *mirheo::ObjectRodBindingInteraction*

Public Types

enum Stage

Describes the stage of an interaction.

Values:

Intermediate

Final

using ActivePredicate = std::function<bool()>

Used to specify if a channel is active or not.

If a channel is inactive, the *Interaction* object can tell the simulation via this function object that the concerned channel does not need to be exchanged.

Typically, this can store the simulation state and be active only at given time intervals. The most common case is to be always active.

Public Functions

Interaction(**const** *MirState* **state*, std::string *name*)

Constructs a `Interaction` object.

Parameters

- `state`: The global state of the system
- `name`: The name of the interaction

virtual void **setPrerequisites**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2)

Add needed properties to the given ParticleVectors for future interactions.

Must be called before any other method of this class.

Parameters

- `pv1`: One *ParticleVector* of the interaction
- `pv2`: The other *ParticleVector* of that will interact
- `cl1`: *CellList* of pv1
- `cl2`: *CellList* of pv2

virtual void **local**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2, cudaStream_t *stream*) = 0

Compute interactions between bulk particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. The order of pv1 and pv2 may change the performance of the interactions.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

virtual void **halo**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2, cudaStream_t *stream*) = 0

Compute interactions between bulk particles and halo particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. In general, the following interactions will be computed: pv1->*halo()* <> pv2->*local()* and pv2->*halo()* <> pv1->*local()*.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

virtual bool **isSelfObjectInteraction**() **const**

This is useful to know if we need exchange / cell-lists for that interaction. Example: membrane interactions are internal, all particles of a membrane are always on the same rank thus it does not need halo particles.

Return boolean describing if the interaction is an internal interaction.

virtual *Stage* **getStage**() **const**

returns the Stage corresponding to this interaction.

virtual std::vector<*InteractionChannel*> **getInputChannels**() **const**
Returns which channels are required as input.

Positions and velocities are always required and are not listed here; Only other channels must be specified here.

virtual std::vector<*InteractionChannel*> **getOutputChannels**() **const**
Returns which channels are those output by the interactions.

virtual std::optional<real> **getCutoffRadius**() **const**

Return the cut-off radius of the interaction; std::nullopt if there is no cutoff.

Public Static Attributes

const *ActivePredicate* **alwaysActive**
a predicate that always returns true.

struct InteractionChannel
A simple structure used to describe which channels are active.

Public Members

std::string **name**
the name of the channel

ActivePredicate **active**
the activity of the channel

Object binding

class ObjectBindingInteraction : **public** *mirheo*::*Interaction*
Compute binding interaction used to attach two *ParticleVector* together.

The interaction has the form of that of a linear spring with constant kBound.

Public Functions

ObjectBindingInteraction(**const** *MirState* **state*, std::string *name*, real *kBound*, std::vector<int2> *pairs*)
Construct an *ObjectBindingInteraction* interaction.

Parameters

- `state`: The global state of the system.
- `name`: The name of the interaction.
- `kBound`: The force coefficient (spring constant).
- `pairs`: The list of pairs of particles that will interact with each other. A pair contains the global ids of the first *ParticleVector* (first entry) and the second *ParticleVector* (second entry).

void **local**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2, cudaStream_t *stream*)

Compute interactions between bulk particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. The order of pv1 and pv2 may change the performance of the interactions.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

void **halo**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2, cudaStream_t *stream*)

Compute interactions between bulk particles and halo particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. In general, the following interactions will be computed: pv1->*halo()* <> pv2->*local()* and pv2->*halo()* <> pv1->*local()*.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

Membrane Interactions

Base class

This is the visible class that is output of the factory function.

class BaseMembraneInteraction : **public** *mirheo*::*Interaction*

Base class that represents membrane interactions.

This kind of interactions does not require any cell-lists and is always a "self-interaction", hence the halo interaction does not do anything. This must be used only with *MembraneVector* objects.

Subclassed by *mirheo::MembraneInteraction< TriangleInteraction, DihedralInteraction, Filter >*

Public Functions

BaseMembraneInteraction(**const** *MirState* *state, **const** std::string &*name*)

Construct a *BaseMembraneInteraction* object.

Parameters

- `state`: The global state of the system

- `name`: The name of the interaction

void **`setPrerequisites`**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2)
Set the required channels to the concerned *ParticleVector* that will participate in the interactions.

This method will fail if pv1 is not a *MembraneVector* or if pv1 is not the same as pv2.

Parameters

- `pv1`: The conserned data that will participate in the interactions.
- `pv2`: The conserned data that will participate in the interactions.
- `cl1`: Unused
- `cl2`: Unused

void **`halo`**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2, cudaStream_t *stream*)
Compute interactions between bulk particles and halo particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. In general, the following interactions will be computed: pv1->*halo()* <> pv2->*local()* and pv2->*halo()* <> pv1->*local()*.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

bool **`isSelfObjectInteraction`**() **`const`**

This is useful to know if we need exchange / cell-lists for that interaction. Example: membrane interactions are internal, all particles of a membrane are always on the same rank thus it does not need halo particles.

Return boolean describing if the interaction is an internal interaction.

Implementation

The factory instantiates one of this templated class. See *Triangle Kernels*, *Dihedral Kernels* and *Filters* for possible template parameters.

`template <class` TriangleInteraction, **`class`** DihedralInteraction, **`class`** Filter>
`class MembraneInteraction` : **`public`** *mirheo*::*BaseMembraneInteraction*
Generic implementation of membrane forces.

Template Parameters

- `TriangleInteraction`: Describes what forces are applied to triangles
- `DihedralInteraction`: Describes what forces are applied to dihedrals
- `Filter`: This allows to apply the interactions only to a subset of membranes

Public Functions

MembraneInteraction(**const** *MirState* **state*, std::string *name*, CommonMembraneParameters *parameters*, **typename** TriangleInteraction::ParametersType *triangleParams*, **typename** DihedralInteraction::ParametersType *dihedralParams*, real *initLengthFraction*, real *growUntil*, Filter *filter*, long *seed* = 42424242)

Construct a *MembraneInteraction* object.

More information can be found on `growUntil` in _scaleFromTime().

Parameters

- `state`: The global state of the system
- `name`: The name of the interaction
- `parameters`: The common parameters from all kernel forces
- `triangleParams`: Parameters that contain the parameters of the triangle forces kernel
- `dihedralParams`: Parameters that contain the parameters of the dihedral forces kernel
- `initLengthFraction`: The membrane will grow from this fraction of its size to its full size in `growUntil` time
- `growUntil`: The membrane will grow from `initLengthFraction` fraction of its size to its full size in this amount of time
- `filter`: Describes which membranes to apply the interactions
- `seed`: Random seed for rng

void **local**(*ParticleVector* **pv1*, *ParticleVector* **pv2*, *CellList* **cl1*, *CellList* **cl2*, cudaStream_t *stream*)

Compute interactions between bulk particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. The order of pv1 and pv2 may change the performance of the interactions.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

void **setPrerequisites**(*ParticleVector* **pv1*, *ParticleVector* **pv2*, *CellList* **cl1*, *CellList* **cl2*)

Set the required channels to the concerned *ParticleVector* that will participate in the interactions.

This method will fail if pv1 is not a *MembraneVector* or if pv1 is not the same as pv2.

Parameters

- `pv1`: The conserned data that will participate in the interactions.
- `pv2`: The conserned data that will participate in the interactions.
- `cl1`: Unused
- `cl2`: Unused

void **checkpoint**(MPI_Comm *comm*, **const** std::string &*path*, int *checkPointId*)
Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

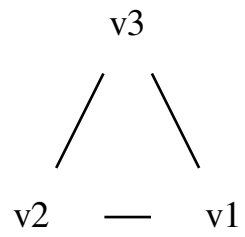
void **restart**(MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

Triangle Kernels

Each thread is mapped to one vertex *v1* and loops over all adjacent triangles labeled as follows:



The output of the kernel is the forces of a given dihedral on *v1*. The forces on *v2* and *v3* from the same dihedral are computed by the thread mapped on *v2* and *v3*, respectively.

template <StressFreeState *stressFreeState*>
class TriangleLimForce
Compute shear energy on a given triangle with the Lim model.

Template Parameters

- `stressFreeState`: States if there is a stress free mesh associated with the interaction

Public Functions

TriangleLimForce(ParametersType *p*, **const** *Mesh* **mesh*, mReal *lscale*)
Construct the functor.

Parameters

- `p`: The parameters of the model
- `mesh`: Triangle mesh information
- `lscale`: Scaling length factor, applied to all parameters

void **`applyLengthScalingFactor`**(mReal *lscale*)
Scale length-dependent parameters.

EquilibriumTriangleDesc **`getEquilibriumDesc`**(**`const`** *MembraneMeshView* &*mesh*, int *i0*, int *i1*) **`const`**
Get the reference triangle information.

Return The reference triangle information.

Parameters

- `mesh`: *Mesh* view that contains the reference mesh. Only used when stressFreeState is Active.
- `i0`: Index (in the adjacent vertex ids space, see `Mesh`) of the first adjacent vertex
- `i1`: Index (in the adjacent vertex ids space, see `Mesh`) of the second adjacent vertex

mReal3 **`operator()`**(mReal3 *v1*, mReal3 *v2*, mReal3 *v3*, EquilibriumTriangleDesc *eq*) **`const`**
Compute the triangle force on `v1`.

See Developer docs for more details.

Return The triangle force acting on `v1`

Parameters

- `v1`: vertex 1
- `v2`: vertex 2
- `v3`: vertex 3
- `eq`: The reference triangle information

`template` <StressFreeState *stressFreeState*>
`class TriangleWLCForce`
Compute shear energy on a given triangle with the Lim model.

Template Parameters

- `stressFreeState`: States if there is a stress free mesh associated with the interaction

Public Functions

`TriangleWLCForce`(ParametersType *p*, **`const`** *Mesh* **mesh*, mReal *lscale*)
Construct the functor.

Parameters

- `p`: The parameters of the model
- `mesh`: Triangle mesh information
- `lscale`: Scaling length factor, applied to all parameters

void **applyLengthScalingFactor**(mReal *lscale*)
Scale length-dependent parameters.

EquilibriumTriangleDesc **getEquilibriumDesc**(**const** *MembraneMeshView* &*mesh*, int *i0*, int *i1*) **const**
Get the reference triangle information.

Return The reference triangle information.

Parameters

- `mesh`: *Mesh* view that contains the reference mesh. Only used when stressFreeState is Active.
- `i0`: Index (in the adjacent vertex ids space, see `Mesh`) of the first adjacent vertex
- `i1`: Index (in the adjacent vertex ids space, see `Mesh`) of the second adjacent vertex

mReal3 **operator()**(mReal3 *v1*, mReal3 *v2*, mReal3 *v3*, EquilibriumTriangleDesc *eq*) **const**
Compute the triangle force on `v1`.

See Developer docs for more details.

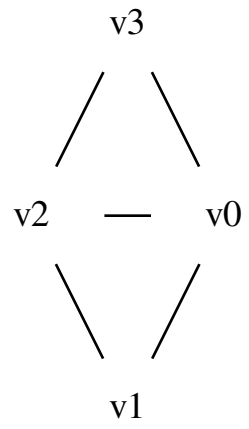
Return The triangle force acting on `v1`

Parameters

- `v1`: vertex 1
- `v2`: vertex 2
- `v3`: vertex 3
- `eq`: The reference triangle information

Dihedral Kernels

Each thread is mapped to one vertex *v0* and loops over all adjacent dihedrals labeled as follows:



The output of the kernel is the forces of a given dihedral on *v0* and *v1*. The forces on *v2* and *v3* from the same dihedral are computed by the thread mapped on *v3*.

class DihedralJuelicher : **public** *mirheo*::*VertexFetcherWithMeanCurvatures*
Compute bending forces from the extended Juelicher model.

Public Types

using ParametersType = JuelicherBendingParameters
Type of parameters that describe the kernel.

Public Functions

DihedralJuelicher(*ParametersType* *p*, mReal *lscale*)
Initialize the functor.

Parameters

- `p`: The parameters of the functor
- `lscale`: Scaling length factor, applied to all parameters

void **applyLengthScalingFactor**(mReal *lscale*)
Scale length-dependent parameters.

void **computeInternalCommonQuantities**(**const** ViewType &*view*, int *rbcId*)
Precompute internal values that are common to all vertices in the cell.

Parameters

- `view`: The view that contains the required object channels

- `rbcId`: The index of the membrane in the `view`

mReal3 **operator()**(VertexType *v0*, VertexType *v1*, VertexType *v2*, VertexType *v3*, mReal3 &*f1*) **const**
Compute the dihedral forces.

See Developer docs for more details.

Return The dihedral force acting on `v0`

Parameters

- `v0`: vertex 0
- `v1`: vertex 1
- `v2`: vertex 2
- `v3`: vertex 3
- `f1`: force acting on `v1`; this method will add (not set) the dihedral force to that quantity.

class DihedralKantor : **public** *mirheo*::*VertexFetcher*
Compute bending forces from the Kantor model.

Public Types

using ParametersType = KantorBendingParameters
Type of parameters that describe the kernel.

Public Functions

DihedralKantor(*ParametersType p*, mReal *lscale*)
Initialize the functor.

Parameters

- `p`: The parameters of the functor
- `lscale`: Scaling length factor, applied to all parameters

void **applyLengthScalingFactor**(mReal *lscale*)
Scale length-dependent parameters.

void **computeInternalCommonQuantities**(**const** ViewType &*view*, int *rbcId*)
Precompute internal values that are common to all vertices in the cell.

mReal3 **operator()**(VertexType *v0*, VertexType *v1*, VertexType *v2*, VertexType *v3*, mReal3 &*f1*) **const**
Compute the dihedral forces.

See Developer docs for more details.

Return The dihedral force acting on `v0`

Parameters

- `v0`: vertex 0
- `v1`: vertex 1
- `v2`: vertex 2

- `v3`: vertex 3
- `f1`: force acting on `v1`; this method will add (not set) the dihedral force to that quantity.

Filters

The membrane interactions can be applied to only a subset of the given *`mirheo::MembraneVector`*. This can be convenient to have different interaction parameters for different membranes with the same mesh topology. Furthermore, reducing the numper of *`mirheo::ParticleVector`* is beneficial for performance (less interaction kernel launches so overhead for e.g. FSI).

`class FilterKeepAll`
Filter that keeps all the membranes.

Public Functions

void **`setPrerequisites`**(*MembraneVector* *mv) **`const`**
set required properties to `mv`

void **`setup`**(*MembraneVector* *mv)
Set internal state of the object.

bool **`inWhiteList`**(long *membraneId*) **`const`**
States if the given membrane must be kept or not.

Return `true` if the membrane should be kept, `false` otherwise.

Parameters

- `membraneId`: The index of the membrane to keep or not

`class FilterKeepByTypeId`
Keep membranes that have a given `typeId`.

The `typeId` of each membrane is stored in the object channel `ChannelNames::membraneTypeId`.

Public Functions

`FilterKeepByTypeId`(int *whiteListTypeId*)
Construct *FilterKeepByTypeId* that wil keep only the membranes with type id `whiteListTypeId`.

Parameters

- `whiteListTypeId`: The type id of the membranes to keep

void **`setPrerequisites`**(*MembraneVector* *mv) **`const`**
set required properties to `mv`

Parameters

- `mv`: The *MembraneVector* taht will be used

void **setup**(*MembraneVector* *mv)
Set internal state of the object.

This must be called after every change of `mv` *DataManager*

Parameters

- `mv`: The *MembraneVector* tahat will be used

bool **inWhiteList**(long *membraneId*) **const**
States if the given membrane must be kept or not.

Return `true` if the membrane should be kept, `false` otherwise.

Parameters

- `membraneId`: The index of the membrane to keep or not

Fetchers

Fetchers are used to load generic data that is needed for kernel computation. In most cases, only vertex coordinates are sufficient (see *`mirheo::VertexFetcher`*). Additional data attached to each vertex may be needed, such as mean curvature in e.g. *`mirheo::DihedralJuelicher`* (see *`mirheo::VertexFetcherWithMeanCurvatures`*).

class VertexFetcher
Fetch a vertex for a given view.

Subclassed by *mirheo::DihedralKantor*, *mirheo::VertexFetcherWithMeanCurvatures*

Public Types

using VertexType = mReal3
info contained in the fetched data

using ViewType = *OVview*
compatible view

Public Functions

VertexType **fetchVertex**(**const** *ViewType* &*view*, int *i*) **const**
fetch a vertex coordinates from a view

Return The vertex coordinates

Parameters

- `view`: The view from which to fetch the vertex
- `i`: The index of the vertex in `view`

class VertexFetcherWithMeanCurvatures : **public** *mirheo*::*VertexFetcher*
Fetch vertex coordinates and mean curvature for a given view.

Subclassed by *mirheo::DihedralJuelicher*

Public Types

using VertexType = *VertexWithMeanCurvature*
info contained in the fetched data

using ViewType = *OVviewWithJuelicherQuants*
compatible view

Public Functions

VertexType **fetchVertex**(**const** *ViewType* &*view*, int *i*) **const**
fetch a vertex coordinates and its mean curvature from a view

Return The vertex coordinates

Parameters

- `view`: The view from which to fetch the vertex
- `i`: The index of the vertex in `view`

struct VertexWithMeanCurvature
holds vertex coordinates and mean curvature

Public Members

mReal3 **r**
vertex coordinates

mReal **H**
mean curvature

Object-Rod binding

This is experimental.

class ObjectRodBindingInteraction : **public** *mirheo*::*Interaction*
Compute elastic interaction used to attach a rod to a *RigidObjectVector* entity.

Public Functions

ObjectRodBindingInteraction(**const** *MirState* **state*, std::string *name*, real *torque*, real3 *relAnchor*, real *kBound*)
Construct an *ObjectRodBindingInteraction* interaction.

Parameters

- `state`: The global state of the system
- `name`: The name of the interaction
- `torque`: Torque applied from the rigid objet to the rod
- `relAnchor`: position of attachement with respect to the rigid object
- `kBound`: The elastic constant for the binding

void **setPrerequisites**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2)
Add needed properties to the given ParticleVectors for future interactions.

Must be called before any other method of this class.

Parameters

- `pv1`: One *ParticleVector* of the interaction
- `pv2`: The other *ParticleVector* of that will interact
- `cl1`: *CellList* of pv1
- `cl2`: *CellList* of pv2

void **local**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2, cudaStream_t *stream*)
Compute interactions between bulk particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. The order of pv1 and pv2 may change the performance of the interactions.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

void **halo**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2, cudaStream_t *stream*)
Compute interactions between bulk particles and halo particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. In general, the following interactions will be computed: pv1->*halo()* <> pv2->*local()* and pv2->*halo()* <> pv1->*local()*.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

Pairwise Interactions

Base class

This is the visible class that is output of the factory function.

class BasePairwiseInteraction : **public** *mirheo*::*Interaction*
Base class for short-range symmetric pairwise interactions.

Subclassed by *mirheo::PairwiseInteraction< PairwiseKernel >*, *mirheo::PairwiseInteractionWithStress< PairwiseKernel >*, *mirheo::PairwiseInteraction< mirheo::PairwiseKernel >*, *mirheo::PairwiseInteraction< mirheo::PairwiseStressWrapper< mirheo::PairwiseKernel > >*

Public Functions

BasePairwiseInteraction(**const** *MirState* *state, **const** std::string &name, real rc)
Construct a base pairwise interaction from parameters.

Parameters

- `state`: The global state of the system.
- `name`: The name of the interaction.
- `rc`: The cutoff radius of the interaction. Must be positive and smaller than the sub-domain size.

std::optional<real> **getCutoffRadius**() **const**

Return the cut-off radius of the pairwise interaction.

Implementation

The factory instantiates one of this templated class. See below for the requirements on the kernels.

template <class *PairwiseKernel*>
class PairwiseInteraction : **public** *mirheo*::*BasePairwiseInteraction*
Short-range symmetric pairwise interactions.

See the pairwise interaction entry of the developer documentation for the interface requirements of the kernel.

Template Parameters

- *PairwiseKernel*: The functor that describes the interaction between two particles (interaction kernel).

Public Functions

PairwiseInteraction(**const** *MirState* *state, **const** std::string &name, real rc, KernelParams pairParams, long seed = 42424242)
Construct a *PairwiseInteraction* object.

Parameters

- `state`: The global state of the system
- `name`: The name of the interaction
- `rc`: The cut-off radius of the interaction
- `pairParams`: The parameters used to construct the interaction kernel
- `seed`: used to initialize random number generator (needed to construct some interaction kernels).

void **setPrerequisites**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2)
Add needed properties to the given ParticleVectors for future interactions.

Must be called before any other method of this class.

Parameters

- `pv1`: One *ParticleVector* of the interaction
- `pv2`: The other *ParticleVector* of that will interact
- `cl1`: *CellList* of pv1
- `cl2`: *CellList* of pv2

void **`local`**(*ParticleVector* **pv1*, *ParticleVector* **pv2*, *CellList* **cl1*, *CellList* **cl2*, cudaStream_t *stream*)

Compute interactions between bulk particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. The order of pv1 and pv2 may change the performance of the interactions.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

void **`halo`**(*ParticleVector* **pv1*, *ParticleVector* **pv2*, *CellList* **cl1*, *CellList* **cl2*, cudaStream_t *stream*)

Compute interactions between bulk particles and halo particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. In general, the following interactions will be computed: pv1->*halo()* <> pv2->*local()* and pv2->*halo()* <> pv1->*local()*.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

Stage **`getStage`**`()` **`const`**

returns the Stage corresponding to this interaction.

std::vector<InteractionChannel> **`getInputChannels`**`()` **`const`**

Returns which channels are required as input.

Positions and velocities are always required and are not listed here; Only other channels must be specified here.

std::vector<InteractionChannel> **`getOutputChannels`**`()` **`const`**

Returns which channels are those output by the interactions.

void **`checkpoint`**(MPI_Comm *comm*, **`const`** std::string &*path*, int *checkPointId*)

Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

void **restart**(MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

Public Static Functions

static std::string **getTypeName**()

Return A string that describes the type of this object

A specific class can be used to compute addtionally the stresses of a given interaction.

template <**class** *PairwiseKernel*>
class PairwiseInteractionWithStress : **public** *mirheo*::*BasePairwiseInteraction*
Short-range symmetric pairwise interactions with stress output.

This object manages two interaction: one with stress, which is used every stressPeriod time, and one with no stress wrapper, that is used the rest of the time. This is motivated by the fact that stresses are not needed for the simulation but rather for post processing; thus the stresses may not need to be computed at every time step.

Template Parameters

- `PairwiseKernel`: The functor that describes the interaction between two particles (interaction kernel).

Public Functions

PairwiseInteractionWithStress(**const** *MirState* *state, **const** std::string &*name*, real *rc*, real *stressPeriod*, KernelParams *pairParams*, long *seed* = 42424242)
Construct a *PairwiseInteractionWithStress* object.

Parameters

- `state`: The global state of the system
- `name`: The name of the interaction
- `rc`: The cut-off radius of the interaction
- `stressPeriod`: The simulation time between two stress computation
- `pairParams`: The parameters used to construct the interaction kernel
- `seed`: used to initialize random number generator (needed to construct some interaction kernels).

void **setPrerequisites**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2)
Add needed properties to the given ParticleVectors for future interactions.

Must be called before any other method of this class.

Parameters

- `pv1`: One *ParticleVector* of the interaction
- `pv2`: The other *ParticleVector* of that will interact
- `cl1`: *CellList* of pv1
- `cl2`: *CellList* of pv2

void **`local`**(*ParticleVector* **pv1*, *ParticleVector* **pv2*, *CellList* **cl1*, *CellList* **cl2*, cudaStream_t *stream*)

Compute interactions between bulk particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. The order of pv1 and pv2 may change the performance of the interactions.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

void **`halo`**(*ParticleVector* **pv1*, *ParticleVector* **pv2*, *CellList* **cl1*, *CellList* **cl2*, cudaStream_t *stream*)

Compute interactions between bulk particles and halo particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. In general, the following interactions will be computed: pv1->*halo()* <> pv2->*local()* and pv2->*halo()* <> pv1->*local()*.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

std::vector<InteractionChannel> **`getInputChannels`**() **`const`**

Returns which channels are required as input.

Positions and velocities are always required and are not listed here; Only other channels must be specified here.

std::vector<InteractionChannel> **`getOutputChannels`**() **`const`**

Returns which channels are those output by the interactions.

void **`checkpoint`**(MPI_Comm *comm*, **`const`** std::string &*path*, int *checkPointId*)

Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

- `checkPointId`: The id of the dump.

void **restart** (MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

Public Static Functions

static std::string **getTypeName** ()

Return A string that describes the type of this object

Kernels

Interface

The *mirheo::PairwiseInteraction* takes a functor that describes a pairwise interaction. This functor may be splitted into two parts:

- a handler, that must be usable on the device.
- a manager, that may store extra information on the host. For simple interactions, this can be the same as the handler class.

The interface of the functor must follow the following requirements:

1. Define a view type to be passed (e.g. *mirheo::PVview*) as well as a particle type to be fetched and the parameter struct used for initialization:

```
using ViewType = <particle vector view type>
using ParticleType = <particle type>
using HandlerType = <type passed to GPU>
using ParamsType = <struct that contains the parameters of this functor>
```

2. A generic constructor from the `ParamsType` parameters:

```
PairwiseKernelType(real rc, const ParamsType& p, real dt, long seed=42424242);
```

3. Setup function (on Host, for manager only)

```
void setup(LocalParticleVector* lpv1, LocalParticleVector* lpv2, CellList* cl1,␣
↪CellList* cl2, const MirState *state);
```

4. Handler function (on Host, for manager only)

```
const HandlerType& handler() const;
```

5. Interaction function (output must match with accumulator, see below) (on GPU)

```
__D__ <OutputType> operator()(const ParticleType dst, int dstId, const␣
↪ParticleType src, int srcId) const;
```

6. *Accumulator* initializer (on GPU)

```
__D__ <Accumulator> getZeroedAccumulator() const;
```

7. Fetch functions (see in *fetchers.h* or see the *docs*):

```
__D__ ParticleType read(const ViewType& view, int id) const;
__D__ ParticleType readNoCache(const ViewType& view, int id) const;

__D__ void readCoordinates(ParticleType& p, const ViewType& view, int id) const;
__D__ void readExtraData(ParticleType& p, const ViewType& view, int id) const;
```

8. Interacting checker to discard pairs not within cutoff:

```
__D__ bool withinCutoff(const ParticleType& src, const ParticleType& dst) const;
```

9. Position getter from generic particle type:

```
__D__ real3 getPosition(const ParticleType& p) const;
```

Note: To implement a new kernel, the following must be done: - satisfy the above interface - add a corresponding parameter in parameters.h - add it to the variant in parameters.h - if necessary, add type traits specialization in type_traits.h

---

This is the interface for the host calls:

class PairwiseKernel

Interface of host methods required for a pairwise kernel.

Subclassed by *mirheo::PairwiseDensity< DensityKernel >*, *mirheo::PairwiseDPD*, *mirheo::PairwiseLJ*, *mirheo::PairwiseMDPD*, *mirheo::PairwiseMorse< Awareness >*, *mirheo::PairwiseNorandomDPD*, *mirheo::PairwiseRepulsiveLJ< Awareness >*, *mirheo::PairwiseSDPD< PressureEOS, DensityKernel >*, *mirheo::PairwiseStressWrapper< mirheo::PairwiseKernel >*

Public Functions

virtual void **setup**(*LocalParticleVector* **lpv1*, *LocalParticleVector* **lpv2*, *CellList* **cl1*, *CellList* **cl2*, **const** *MirState* **state*)

setup the internal state of the functor

virtual void **writeState**(std::ofstream &*fout*)

write internal state to a stream

virtual bool **readState**(std::ifstream &*fin*)

restore internal state from a stream

The rest is directly implemented in the kernels, as no virtual functions are allowed on the device.

Implemented kernels

template <**typename** DensityKernel>
class PairwiseDensity : **public** *mirheo*::*PairwiseKernel*, **public** *mirheo*::*ParticleFetcher*

Compute number density from pairwise kernel.

Template Parameters

- `DensityKernel`: The kernel used to evaluate the number density

Public Functions

`PairwiseDensity`(real *rc*, DensityKernel *densityKernel*)
construct from density kernel

`PairwiseDensity`(real *rc*, **`const`** ParamsType &*p*, long *seed* = 42424242)
generic constructor

real **`operator()`**(**`const`** ParticleType *dst*, int *dstId*, **`const`** ParticleType *src*, int *srcId*) **`const`**
evaluate the number density contribution of this pair

DensityAccumulator **`getZeroedAccumulator`**() **`const`**
initialize the accumulator

`const` HandlerType &**`handler`**() **`const`**
get the handler that can be used on device

`class PairwiseDPDHandler` : **`public`** *mirheo*::*ParticleFetcher*
a GPU compatible functor that computes DPD interactions

Subclassed by *mirheo::PairwiseDPD*

Public Types

`using ViewType` = *PVview*
compatible view type

`using ParticleType` = *Particle*
compatible particle type

Public Functions

`PairwiseDPDHandler`(real *rc*, real *a*, real *gamma*, real *power*)
constructor

real3 **`operator()`**(**`const`** *ParticleType* *dst*, int *dstId*, **`const`** *ParticleType* *src*, int *srcId*) **`const`**
evaluate the force

ForceAccumulator **`getZeroedAccumulator`**() **`const`**
initialize accumulator

`class PairwiseDPD` : **`public`** *mirheo*::*PairwiseKernel*, **`public`** *mirheo*::*PairwiseDPDHandler*
Helper class that constructs *PairwiseDPDHandler*.

Public Types

`using HandlerType` = *PairwiseDPDHandler*
handler type corresponding to this object

`using ParamsType` = DPDParams
parameters that are used to create this object

Public Functions

PairwiseDPD(real *rc*, real *a*, real *gamma*, real *kBT*, real *power*, long *seed* = 42424242)
Constructor.

PairwiseDPD(real *rc*, **const** *ParamsType* &*p*, long *seed* = 42424242)
Generic constructor.

const *HandlerType* &**handler**() **const**
get the handler that can be used on device

void **setup**(*LocalParticleVector* **lpv1*, *LocalParticleVector* **lpv2*, *CellList* **cl1*, *CellList* **cl2*, **const** *MirState* **state*)
setup the internal state of the functor

void **writeState**(std::ofstream &*fout*)
write internal state to a stream

bool **readState**(std::ifstream &*fin*)
restore internal state from a stream

Public Static Functions

static std::string **getTypeName**()

Return type name string

class PairwiseLJ : **public** *mirheo*::*PairwiseKernel*, **public** *mirheo*::*ParticleFetcher*
Compute Lennard-Jones forces on the device.

Public Types

using ViewType = *PVview*
Compatible view type.

using ParticleType = *Particle*
Compatible particle type.

using HandlerType = *PairwiseLJ*
Corresponding handler.

using ParamsType = LJParams
Corresponding parameters type.

Public Functions

PairwiseLJ(real *rc*, real *epsilon*, real *sigma*)
Constructor.

PairwiseLJ(real *rc*, **const** *ParamsType* &*p*, long *seed* = 42424242)
Generic constructor.

real3 **operator()**(*ParticleType* *dst*, int, *ParticleType* *src*, int) **const**
Evaluate the force.

ForceAccumulator **getZeroedAccumulator**() **const**
initialize accumulator

const *HandlerType* &**handler**() **const**
get the handler that can be used on device

Public Static Functions

static std::string **getTypeName**()

Return type name string

template <**class** Awareness>
class **PairwiseMorse** : **public** *mirheo*::*PairwiseKernel*, **public** *mirheo*::*ParticleFetcher*
Compute Morse forces on the device.

Template Parameters

- `Awareness`: A functor that describes which particles pairs interact.

Public Functions

PairwiseMorse(real *rc*, real *De*, real *r0*, real *beta*, Awareness *awareness*)
Constructor.

PairwiseMorse(real *rc*, **const** ParamsType &*p*, long *seed*)
Generic constructor.

real3 **operator()**(ParticleType *dst*, int *dstId*, ParticleType *src*, int *srcId*) **const**
Evaluate the force.

ForceAccumulator **getZeroedAccumulator**() **const**
initialize accumulator

const HandlerType &**handler**() **const**
get the handler that can be used on device

void **setup**(*LocalParticleVector* **lpv1*, *LocalParticleVector* **lpv2*, *CellList* **cl1*, *CellList* **cl2*, **const** *MirState* **state*)
setup the internal state of the functor

Public Static Functions

static std::string **getTypeName**()

Return type name string

class **PairwiseMDPDHandler** : **public** *mirheo*::*ParticleFetcherWithDensity*
a GPU compatible functor that computes MDPD interactions

Subclassed by *mirheo::PairwiseMDPD*

Public Types

using ViewType = *PVviewWithDensities*
compatible view type

using ParticleType = ParticleWithDensity
compatible particle type

Public Functions

PairwiseMDPDHandler(real *rc*, real *rd*, real *a*, real *b*, real *gamma*, real *power*)
constructor

real3 **operator()**(**const** *ParticleType* *dst*, int *dstId*, **const** *ParticleType* *src*, int *srcId*) **const**
evaluate the force

ForceAccumulator **getZeroedAccumulator**() **const**
initialize accumulator

class PairwiseMDPD : **public** *mirheo*::*PairwiseKernel*, **public** *mirheo*::*PairwiseMDPDHandler*
Helper class that constructs *PairwiseMDPDHandler*.

Public Types

using HandlerType = *PairwiseMDPDHandler*
handler type corresponding to this object

using ParamsType = MDPDParams
parameters that are used to create this object

Public Functions

PairwiseMDPD(real *rc*, real *rd*, real *a*, real *b*, real *gamma*, real *kBT*, real *power*, long *seed* = 42424242)
Constructor.

PairwiseMDPD(real *rc*, **const** *ParamsType* &*p*, long *seed* = 42424242)
Generic constructor.

const *HandlerType* &**handler**() **const**
get the handler that can be used on device

void **setup**(*LocalParticleVector* **lpv1*, *LocalParticleVector* **lpv2*, *CellList* **cl1*, *CellList* **cl2*, **const** *MirState* **state*)
setup the internal state of the functor

void **writeState**(std::ofstream &*fout*)
write internal state to a stream

bool **readState**(std::ifstream &*fin*)
restore internal state from a stream

Public Static Functions

static std::string **getTypeName**()

Return type name string

class PairwiseNorandomDPD : **public** *mirheo*::*PairwiseKernel*, **public** *mirheo*::*ParticleFetcher*

a GPU compatible functor that computes DPD interactions without fluctuations.

Used in unit tests

Public Types

using ViewType = *PVview*

compatible view type

using ParticleType = *Particle*

compatible particle type

using HandlerType = *PairwiseNorandomDPD*

handler type corresponding to this object

using ParamsType = NoRandomDPDParams

parameters that are used to create this object

Public Functions

PairwiseNorandomDPD(real *rc*, real *a*, real *gamma*, real *kBT*, real *power*)

constructor

PairwiseNorandomDPD(real *rc*, **const** *ParamsType* &*p*, long *seed* = 42424242)

Generic constructor.

real3 **operator()**(**const** *ParticleType* *dst*, int *dstId*, **const** *ParticleType* *src*, int *srcId*) **const**

evaluate the force

ForceAccumulator **getZeroedAccumulator**() **const**

initialize accumulator

const *HandlerType* &**handler**() **const**

get the handler that can be used on device

void **setup**(*LocalParticleVector* **lpv1*, *LocalParticleVector* **lpv2*, *CellList* **cl1*, *CellList* **cl2*, **const** *MirState* **state*)

setup the internal state of the functor

Public Static Functions

static std::string **getTypeName**()

Return type name string

template <**class** Awareness>
class PairwiseRepulsiveLJ : **public** *mirheo*::*PairwiseKernel*

Kernel for repulsive LJ forces.

Subclassed by mirheo::PairwiseGrowingRepulsiveLJ< Awareness >

Public Functions

PairwiseRepulsiveLJ(real *rc*, real *epsilon*, real *sigma*, real *maxForce*, Awareness *awareness*)
Constructor.

PairwiseRepulsiveLJ(real *rc*, **const** ParamsType &*p*, long *seed* = 42424242)
Generic constructor.

const HandlerType &**handler**() **const**
get the handler that can be used on device

void **setup**(*LocalParticleVector* *lpv1, *LocalParticleVector* *lpv2, *CellList* *cl1, *CellList* *cl2, **const** *MirState* *state)
setup the internal state of the functor

template <**typename** PressureEOS, **typename** DensityKernel>
class PairwiseSDPDHandler : **public** *mirheo*::*ParticleFetcherWithDensityAndMass*
Compute smooth dissipative particle dynamics forces on the device.

Template Parameters

- `PressureEos`: The equation of state
- `DensityJKernel`: The kernel used to compute the density

Subclassed by *mirheo::PairwiseSDPD< PressureEOS, DensityKernel >*

Public Functions

PairwiseSDPDHandler(real *rc*, PressureEOS *pressure*, DensityKernel *densityKernel*, real *viscosity*)
Constructor.

real3 **operator()**(**const** ParticleType *dst*, int *dstId*, **const** ParticleType *src*, int *srcId*) **const**
evaluate the force

ForceAccumulator **getZeroedAccumulator**() **const**
initialize the accumulator

template <**typename** PressureEOS, **typename** DensityKernel>
class PairwiseSDPD : **public** *mirheo*::*PairwiseKernel*, **public** *mirheo*::*PairwiseSDPDHandler*<PressureEOS, DensityKernel
Helper class to create *PairwiseSDPDHandler* from host.

Template Parameters

- `PressureEos`: The equation of state
- `DensityJKernel`: The kernel used to compute the number density

Public Functions

PairwiseSDPD(real *rc*, PressureEOS *pressure*, DensityKernel *densityKernel*, real *viscosity*, real *kBT*, long *seed* = 42424242)
Constructor.

PairwiseSDPD(real *rc*, **const** ParamsType &*p*, long *seed* = 42424242)
Generic constructor.

const HandlerType &**handler**() **const**
get the handler that can be used on device

void **setup**(*LocalParticleVector* *lpv1, *LocalParticleVector* *lpv2, *CellList* *cl1, *CellList* *cl2, **const** *MirState* *state)
setup the internal state of the functor

void **writeState**(std::ofstream &*fout*)
write internal state to a stream

bool **readState**(std::ifstream &*fin*)
restore internal state from a stream

The above kernels that output a force can be wrapped by the stress wrapper:

template <**typename** BasicPairwiseForceHandler>
class **PairwiseStressWrapperHandler** : **public** BasicPairwiseForceHandler
Compute force and stress from a pairwise force kernel.

Template Parameters

- `BasicPairwiseForceHandler`: The underlying pairwise interaction handler (must output a force)

Public Functions

PairwiseStressWrapperHandler(BasicPairwiseForceHandler *basicForceHandler*)
Constructor.

__device__ *ForceStress* **operator()**(**const** ParticleType *dst*, int *dstId*, **const** ParticleType *src*, int *srcId*) **const**
Evaluate the force and the stress.

ForceStressAccumulator<BasicViewType> **getZeroedAccumulator**() **const**
Initialize the accumulator.

template <typename *BasicPairwiseForce*>
class **PairwiseStressWrapper** : **public** BasicPairwiseForce
Create *PairwiseStressWrapperHandler* from host.

Template Parameters

- `BasicPairwiseForceHandler`: The underlying pairwise interaction (must output a force)

Public Functions

PairwiseStressWrapper(BasicPairwiseForce *basicForce*)
Constructor.

PairwiseStressWrapper(real *rc*, **const** ParamsType &*p*, long *seed* = 42424242)
Generic constructor.

void **setup**(*LocalParticleVector* *lpv1, *LocalParticleVector* *lpv2, *CellList* *cl1, *CellList* *cl2, **const** *MirState* *state)
setup internal state

const HandlerType &**handler**() **const**
get the handler that can be used on device

Fetchers

Fetchers are used to load the correct data needed by the pairwise kernels (e.g. the *mirheo::PairwiseRepulsiveLJ* kernel needs only the positions while the *mirheo::PairwiseSDPD* kernel needs also velocities and number densities).

class ParticleFetcher
fetch position, velocity and global id

Subclassed by *mirheo::PairwiseDensity< DensityKernel >*, *mirheo::PairwiseDPDHandler*, *mirheo::PairwiseLJ*, *mirheo::PairwiseMorse< Awareness >*, *mirheo::PairwiseNorandomDPD*, mirheo::PairwiseRepulsiveLJHandler< Awareness >, *mirheo::ParticleFetcherWithDensity*

Public Types

using ViewType = *PVview*
compatible view type

using ParticleType = *Particle*
compatible particle type

Public Functions

ParticleFetcher(real *rc*)

Parameters

- `rc`: cut-off radius

ParticleType **read**(**const** *ViewType* &*view*, int *id*) **const**
fetch the particle information

Return *Particle* information

Parameters

- `view`: The view pointing to the data
- `id`: The particle index

ParticleType **readNoCache**(**const** *ViewType* &*view*, int *id*) **const**
read particle information directly from the global memory (without going through the L1 or L2 cache) This may be beneficial if one want to maximize the cahe usage on a concurrent stream

void **readCoordinates**(*ParticleType* &*p*, **const** *ViewType* &*view*, int *id*) **const**
read the coordinates only (used for the first pass on the neighbors, discard cut-off radius)

void **readExtraData**(*ParticleType* &*p*, **const** *ViewType* &*view*, int *id*) **const**
read the additional data contained in the particle (other than coordinates)

bool **withinCutoff**(**const** *ParticleType* &*src*, **const** *ParticleType* &*dst*) **const**

Return `true` if the particles `src` and `dst` are within the cut-off radius distance; `false` otherwise.

real3 **getPosition**(**const** *ParticleType* &*p*) **const**
Generic converter from the ParticleType type to the common `real3` coordinates.

int64_t **getId**(**const** *ParticleType* &*p*) **const**

Return Global id of the particle

class ParticleFetcherWithDensity : **public** *mirheo*::*ParticleFetcher*

fetch positions, velocities and number densities

Subclassed by *mirheo::PairwiseMDPDHandler*, *mirheo::ParticleFetcherWithDensityAndMass*

Public Types

using ViewType = *PVviewWithDensities*

compatible view type

using ParticleType = *ParticleWithDensity*

compatible particle type

Public Functions

ParticleFetcherWithDensity(real *rc*)

Parameters

- `rc`: cut-off radius

ParticleType **read**(**const** *ViewType* &*view*, int *id*) **const**

read full particle information

ParticleType **readNoCache**(**const** *ViewType* &*view*, int *id*) **const**

read full particle information through global memory

void **readCoordinates**(*ParticleType* &*p*, **const** *ViewType* &*view*, int *id*) **const**

read particle coordinates only

void **readExtraData**(*ParticleType* &*p*, **const** *ViewType* &*view*, int *id*) **const**

read velocity and number density of the particle

bool **withinCutoff**(**const** *ParticleType* &*src*, **const** *ParticleType* &*dst*) **const**

Return `true` if `src` and `dst` are within a cut-off radius distance; `false` otherwise

real3 **getPosition**(**const** *ParticleType* &*p*) **const**

fetch position from the generic particle structure

int64_t **getId**(**const** *ParticleType* &*p*) **const**

Return Global id of the particle

struct ParticleWithDensity

contains position, global index, velocity and number density of a particle

Public Members

Particle **p**

positions, global id, velocity

real **d**

number density

class ParticleFetcherWithDensityAndMass : **public** *mirheo*::*ParticleFetcherWithDensity*

fetch that reads positions, velocities, number densities and mass

Subclassed by *mirheo::PairwiseSDPDHandler< PressureEOS, DensityKernel >*

Public Types

using ViewType = *PVviewWithDensities*

Compatible view type.

using ParticleType = *ParticleWithDensityAndMass*

Compatible particle type.

Public Functions

ParticleFetcherWithDensityAndMass(real *rc*)

Parameters

- `rc`: The cut-off radius

ParticleType **read**(**const** *ViewType* &*view*, int *id*) **const**

read full particle information

ParticleType **readNoCache**(**const** *ViewType* &*view*, int *id*) **const**

read full particle information through global memory

void **readCoordinates**(*ParticleType* &*p*, **const** *ViewType* &*view*, int *id*) **const**

read particle coordinates only

void **readExtraData**(*ParticleType* &*p*, **const** *ViewType* &*view*, int *id*) **const**

read velocity, number density and mass of the particle

bool **withinCutoff**(**const** *ParticleType* &*src*, **const** *ParticleType* &*dst*) **const**

Return `true` if `src` and `dst` are within a cut-off radius distance; `false` otherwise

real3 **getPosition**(**const** *ParticleType* &*p*) **const**

fetch position from the generic particle structure

int64_t **getId**(**const** *ParticleType* &*p*) **const**

Return Global id of the particle

struct ParticleWithDensityAndMass

contains position, velocity, global id, number density and mass of a particle

Public Members

Particle **p**

position, global id, velocity

real **d**

number density

real **m**

mass

Accumulators

Every *interaction kernel* must initialize an accumulator that is used to add its output quantity. Depending on the kernel, that quantity may be of different type, and may behave in a different way (e.g. forces and stresses are different).

It must satisfy the following interface requirements (in the following, we denote the type of the local variable as `LType` and the *view type* as `ViewType`):

1. A default constructor which initializes the internal local variable
2. Atomic accumulator from local value to destination view:

```
__D__ void atomicAddToDst(LType, ViewType&, int id) const;
```

3. Atomic accumulator from local value to source view:

```
__D__ inline void atomicAddToSrc(LType, ViewType&, int id) const;
```

4. Accessor of accumulated value:

```
__D__ inline LType get() const;
```

5. Function to add a value to the accumulator (from output of pairwise kernel):

```
__D__ inline void add(LType);
```

The following accumulators are currently implemented:

class DensityAccumulator
Accumulate densities on device.

Public Functions

DensityAccumulator()
Initialize the *DensityAccumulator*.

void **atomicAddToDst**(real *d*, *PVviewWithDensities* &*view*, int *id*) **const**
Atomically add density `d` to the destination `view` at id `id`.

Parameters

- `d`: The value to add
- `view`: The destination container
- `id`: destination index in `view`

void **atomicAddToSrc**(real *d*, *PVviewWithDensities* &*view*, int *id*) **const**
Atomically add density `d` to the source `view` at id `id`.

Parameters

- `d`: The value to add
- `view`: The destination container
- `id`: destination index in `view`

real **get**() **const**

Return the internal accumulated density

void **add**(real *d*)
add `d` to the internal density

class ForceAccumulator
Accumulate forces on device.

Public Functions

ForceAccumulator()
Initialize the *ForceAccumulator*.

void **atomicAddToDst**(real3 *f*, *PVview* &*view*, int *id*) **const**
Atomically add the force `f` to the destination `view` at id `id`.

Parameters

- `f`: The force, directed from src to dst
- `view`: The destination container
- `id`: destination index in `view`

void **atomicAddToSrc**(real3 *f*, *PVview* &*view*, int *id*) **const**
Atomically add the force `f` to the source `view` at id `id`.

Parameters

- `f`: The force, directed from src to dst
- `view`: The destination container
- `id`: destination index in `view`

real3 **get**() **const**

Return the internal accumulated force

void **add**(real3 *f*)
add `f` to the internal force

struct ForceStress
Holds force and stress together.

Public Members

real3 **force**
force value

Stress **stress**
stress value

template <typename BasicView**>**
class ForceStressAccumulator
Accumulate *ForceStress* structure on device.

Template Parameters

- `BasicView`: The view type without stress, to enforce the use of the stress view wrapper

Public Functions

`ForceStressAccumulator``()`
Initialize the *ForceStressAccumulator*.

void **`atomicAddToDst`**`(`**`const`** *ForceStress* *&fs*, *PVviewWithStresses*<BasicView> *&view*, int *id*`)` **`const`**
Atomically add the force and stress `fs` to the destination `view` at id `id`.

Parameters

- `fs`: The force, directed from src to dst, and the corresponding stress
- `view`: The destination container
- `id`: destination index in `view`

void **`atomicAddToSrc`**`(`**`const`** *ForceStress* *&fs*, *PVviewWithStresses*<BasicView> *&view*, int *id*`)` **`const`**
Atomically add the force and stress `fs` to the source `view` at id `id`.

Parameters

- `fs`: The force, directed from src to dst, and the corresponding stress
- `view`: The destination container
- `id`: destination index in `view`

ForceStress **`get`**`()` **`const`**

Return the internal accumulated force and stress

void **`add`**`(`**`const`** *ForceStress* *&fs*`)`
add `fs` to the internal force

Rod Interactions

Base class

This is the visible class that is output of the factory function.

`class` **`BaseRodInteraction`** : **`public`** *mirheo*::*Interaction*
Base class to manage rod interactions.

Rod interactions must be used with a *RodVector*. They are internal forces, meaning that *halo()* does not compute anything.

Subclassed by *mirheo::RodInteraction< Nstates, StateParameters >*

Public Functions

BaseRodInteraction(**const** *MirState* *state, **const** std::string &name)
Construct a `BaseRodInteraction`.

Parameters

- `state`: The global state of the system
- `name`: Name of the interaction

void **halo**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2, cudaStream_t stream)
Compute interactions between bulk particles and halo particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. In general, the following interactions will be computed: pv1->*halo()* <> pv2->*local()* and pv2->*halo()* <> pv1->*local()*.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

bool **isSelfObjectInteraction**() **const**

This is useful to know if we need exchange / cell-lists for that interaction. Example: membrane interactions are internal, all particles of a membrane are always on the same rank thus it does not need halo particles.

Return boolean describing if the interaction is an internal interaction.

Implementation

The factory instantiates one of this templated class.

template <int *Nstates*, **class** StateParameters>
class RodInteraction : **public** *mirheo*::*BaseRodInteraction*
Generic implementation of rod forces.

Template Parameters

- `Nstates`: Number of polymorphic states
- `StateParameters`: parameters associated to the polymorphic state model

Public Functions

RodInteraction(**const** *MirState* *state, **const** std::string &name, RodParameters *parameters*, StateParameters *stateParameters*, bool *saveEnergies*)
Construct a *RodInteraction* object.

Parameters

- `state`: The global state of the system
- `name`: The name of the interaction
- `parameters`: The common parameters from all kernel forces
- `stateParameters`: Parameters related to polymorphic states transition
- `saveEnergies`: `true` if the user wants to also compute the energies. In this case, energies will be saved in the `channel_names::energies` bisegment channel.

void **`setPrerequisites`**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2)

Add needed properties to the given ParticleVectors for future interactions.

Must be called before any other method of this class.

Parameters

- `pv1`: One *ParticleVector* of the interaction
- `pv2`: The other *ParticleVector* of that will interact
- `cl1`: *CellList* of pv1
- `cl2`: *CellList* of pv2

void **`local`**(*ParticleVector* *pv1, *ParticleVector* *pv2, *CellList* *cl1, *CellList* *cl2, cudaStream_t *stream*)

Compute interactions between bulk particles.

The result of the interaction is **added** to the corresponding channel of the *ParticleVector*. The order of pv1 and pv2 may change the performance of the interactions.

Parameters

- `pv1`: first interacting *ParticleVector*
- `pv2`: second interacting *ParticleVector*. If it is the same as the pv1, self interactions will be computed.
- `cl1`: cell-list built for the appropriate cut-off radius for pv1
- `cl2`: cell-list built for the appropriate cut-off radius for pv2
- `stream`: Execution stream

Kernels

The following support structure is used to compute the elastic energy:

`template` <int *Nstates*>
`struct BiSegment`

Helper class to compute elastic forces and energy on a bisegment.

Template Parameters

- `Nstates`: Number of polymorphic states

Public Functions

__device__ **BiSegment**(**const** *RVview* &*view*, int *start*)
Fetch bisegment data and prepare helper quantities.

__device__ rReal3 **applyGrad0Bicur**(**const** rReal3 &*v*) **const**
compute gradient of the bicurvature w.r.t. r0 times v

__device__ rReal3 **applyGrad2Bicur**(**const** rReal3 &*v*) **const**
compute gradient of the bicurvature w.r.t. r0 times v

__device__ void **computeBendingForces**(int *state*, **const** GPU_RodBiSegmentParameters<Nstates> &*params*, rReal3 &*fr0*, rReal3 &*fr2*, rReal3 &*fpm0*, rReal3 &*fpm1*) **const**
compute the bending forces acting on the bisegment particles

This metho will add bending foces to the given variables. The other forces (on e.g. r1) can be computed by the symmetric nature of the model.

Parameters

- `state`: Polymorphic state
- `params`: Elastic forces parameters
- `fr0`: *Force* acting on r0
- `fr2`: *Force* acting on r2
- `fpm0`: *Force* acting on pm0
- `fpm1`: *Force* acting on pm1

__device__ void **computeTwistForces**(int *state*, **const** GPU_RodBiSegmentParameters<Nstates> &*params*, rReal3 &*fr0*, rReal3 &*fr2*, rReal3 &*fpm0*, rReal3 &*fpm1*) **const**
compute the torsion forces acting on the bisegment particles

This metho will add twist foces to the given variables. The other forces (on e.g. r1) can be computed by the symmetric nature of the model.

Parameters

- `state`: Polymorphic state
- `params`: Elastic forces parameters
- `fr0`: *Force* acting on r0
- `fr2`: *Force* acting on r2
- `fpm0`: *Force* acting on pm0
- `fpm1`: *Force* acting on pm1

__device__ void **computeCurvatures**(rReal2 &*kappa0*, rReal2 &*kappa1*) **const**
Compute the curvatures along the material frames on each segment.

Parameters

- `kappa0`: Curvature on the first segment
- `kappa1`: Curvature on the second segment

__device__ void **computeTorsion**(rReal &*tau*) **const**
Compute the torsion along the bisegment.

Parameters

- `tau`: Torsion

__device__ void **computeCurvaturesGradients**(rReal3 &*gradr0x*, rReal3 &*gradr0y*, rReal3 &*gradr2x*, rReal3 &*gradr2y*, rReal3 &*gradpm0x*, rReal3 &*gradpm0y*, rReal3 &*gradpm1x*, rReal3 &*gradpm1y*) **const**
compute gradients of curvature term w.r.t. particle positions (see drivers)

__device__ void **computeTorsionGradients**(rReal3 &*gradr0*, rReal3 &*gradr2*, rReal3 &*gradpm0*, rReal3 &*gradpm1*) **const**
compute gradients of torsion term w.r.t. particle positions (see drivers)

__device__ rReal **computeEnergy**(int *state*, **const** GPU_RodBiSegmentParameters<Nstates> &*params*) **const**
Compute the energy of the bisegment.

Public Members

rReal3 **e0**
first segment

rReal3 **e1**
second segment

rReal3 **t0**
first segment direction

rReal3 **t1**
second segment direction

rReal3 **dp0**
first material frame direction

rReal3 **dp1**
second material frame direction

rReal3 **bicur**
bicurvature

rReal **bicurFactor**
helper scalar to compute bicurvature

rReal **e0inv**
1 / length of first segment

rReal **e1inv**
1 / length of second segment

rReal **linv**
1 / l

rReal **l**
average of the lengths of the two segments

Utils

Parameter wrap

This class is used to facilitate parameters read.

class ParametersWrap

A tool to transform a map from string keys to variant parameters.

The input map is typically an input from the python interface.

Public Types

using VarParam = std::variant<real, std::vector<real>, std::vector<real2>, std::string, bool>

A variant that contains the possible types to represent parameters.

using MapParams = std::map<std::string, *VarParam*>

Represents the map from parameter names to parameter values.

Public Functions

ParametersWrap(**const** *MapParams* &*params*)

Construct a *ParametersWrap* object from a MapParams.

template <**typename** T>
bool **exists**(**const** std::string &*key*)

Check if a parameter of a given type and name exists in the map.

Return true if T and key match, false otherwise.

Template Parameters

- `T`: The type of the parameter

Parameters

- `key`: The name of the parameter to check

void **checkAllRead**() **const**

Die if some keys were not read (see *read()*)

template <**typename** T>
T **read**(**const** std::string &*key*)

Fetch a parameter value for a given key.

On success, this method will also mark internally the parameter as read. This allows to check if some parameters were never used (see *checkAllRead()*).

Template Parameters

- `T`: the type of the parameter to read.

Parameters

- `key`: the parameter name to read.

This method dies if key does not exist or if T is the wrong type.

StepRandomGen

class StepRandomGen

A random number generator that generates a different number at every time step but returns the same number while the time step is not updated.

Used to generate independant random numbers at every time step. Several calls at the same time step will return the same random number. This is used to keep the interactionssymmetric accross ranks (pairwise particle halo interactions are computed twice, once on each rank. The random seed must therefore be the same and only depend on the time step, not the rank).

Public Functions

StepRandomGen(long *seed*)

construct a *StepRandomGen*

Parameters

- `seed`: The random seed.

real **generate**(**const** *MirState* **state*)

Generates a random number from the current state.

Return a random number uniformly distributed on [0.001, 1].

Parameters

- `state`: The currenst state that contains time step info.

Friends

std::ofstream &**operator<<**(std::ofstream &*stream*, **const** StepRandomGen &*gen*)

serialization helper

std::ifstream &**operator>>**(std::ifstream &*stream*, StepRandomGen &*gen*)

deserialization helper

18.12 Logger

Example of a log entry:

```
15:10:35:639  Rank 0000    INFO at /Mirheo/src/mirheo/core/logger.cpp:54 Compiled␣
↪with maximum debug level 10
15:10:35:640  Rank 0000    INFO at /Mirheo/src/mirheo/core/logger.cpp:56 Debug level␣
↪requested 3, set to 3
15:10:35:640  Rank 0000    INFO at /Mirheo/src/mirheo/core/mirheo.cpp:110 Program␣
↪started, splitting communicator
15:10:35:684  Rank 0000    INFO at /Mirheo/src/mirheo/core/mirheo.cpp:58 Detected 1␣
↪ranks per node, my intra-node ID will be 0
15:10:35:717  Rank 0000    INFO at /Mirheo/src/mirheo/core/mirheo.cpp:65 Found 1 GPUs␣
↪per node, will use GPU 0
```

API

class Logger

logging functionality with MPI support.

Each MPI process writes to its own file, prefixing messages with time stamps so that later the information may be combined and sorted. Filenames have the following pattern, NNNNN is the MPI rank with leading zeros: `<common_name>_NNNNN.log`

Debug level governs which messages to log will be printed (a higher level will dump more log messages).

Every logging call has an importance level associated with it, which is compared against the governing debug level, e.g. debug() importance is 4 and error() importance is 1.

```
Logger logger;
```

has to be defined in one the objective file (typically the one that contains main()). Prior to any logging the method *init()* must be called.

Public Functions

Logger()

Initialize the logger.

Experimental: logger can be automatically set to output to *stdout* by settings the `MIRHEO_LOGGER_AUTO_STDOUT` environment variable to a non-zero value. This is useful in multi-library contexts where multiple loggers may be created.

void **init**(MPI_Comm *comm*, **const** std::string &*filename*, int *debugLvl* = -1)

Setup the logger object.

Must be called before any logging method.

Parameters

- `comm`: MPI communicator that contains all ranks that will use the logger. If set to MPI_COMM_NULL, the logger does not require MPI to be initialized.
- `filename`: log files will be prefixed with *filename:* e.g. *filename_<rank_with_leading_zeros>.log*
- `debugLvl`: debug level or -1 to use the default value

void **init**(MPI_Comm *comm*, *FileWrapper* *fout*, int *debugLvl* = -1)

Setup the logger object to write to a given file.

Parameters

- `comm`: MPI communicator that contains all ranks that will use the logger. If set to MPI_COMM_NULL, the logger does not require MPI to be initialized.
- `fout`: file handler, must be open, typically *stdout* or *stderr*
- `debugLvl`: debug leve or -1 to use the default value

int **getDebugLvl**() **const**

return The current debug level

void **setDebugLvl**(int *debugLvl*)

set the debug level

Parameters

- `debugLvl`: debug level

void **log**(**const** char *_key_, **const** char *_filename_, int _line_, **const** char *_pattern_, ...) **const**
Main logging function.

Construct and dump a log entry with time prefix, importance string, filename and line number, and the message itself.

This function is not supposed to be called directly, use appropriate macros instead, e.g. say(), error(), debug().

> **Warning:** When the debug level is higher or equal to the c *flushThreshold_* member variable (default 8), every message is flushed to disk immediately. This may increase the runtime significantly and only recommended to debug crashes.

Parameters

- `key`: The importance string, e.g. LOG or WARN
- `filename`: name of the current source file
- `line`: line number in the current source file
- `pattern`: message pattern to be passed to *printf*

void **_die**(**const** char *_filename_, int _line_, **const** char *_fmt_, ...) **const**
Calls *log()* and kills the application on a fatal error.

Print stack trace, error message, close the file and abort. See *log()* for parameters.

void **_CUDA_die**(**const** char *_filename_, int _line_, cudaError_t _code_) **const**
Calls *_die()* with the error message corresponding to the given CUDA error code.

Parameters

- `filename`: name of the current source file
- `line`: line number in the current source file
- `code`: CUDA error code (returned by a CUDA call)

void **_MPI_die**(**const** char *_filename_, int _line_, int _code_) **const**
Calls *_die()* with the error message corresponding to the given MPI error code.

Parameters

- `filename`: name of the current source file
- `line`: line number in the current source file
- `code`: MPI error code (returned by an MPI call)

void **_CUDA_Check**(**const** char *_filename_, **const** int _line_, cudaError_t _code_) **const**
check a CUDA error call and call *_CUDA_die()* in case of error

void **_MPI_Check**(**const** char *_filename_, **const** int _line_, **const** int _code_) **const**
check an MPI error call and call *_MPI_die()* in case of error

18.13 Marching Cubes

API

struct Triangle
simple tructure that represents a triangle in 3D

Public Members

real3 **a**
vertex 0

real3 **b**
vertex 1

real3 **c**
vertex 2

void *mirheo*::marching_cubes::**computeTriangles**(*DomainInfo domain*, real3 *resolution*, **const** ImplicitSurfaceFunction *&surface*, std::vector<*Triangle*> *&triangles*)
Create an explicit surface (triangles) from implicit surface (scalar field) using marching cubes.

Parameters

- `domain`: Domain information
- `resolution`: the number of grid points in each direction
- `surface`: The scalar field that represents implicitly the surface (0 levelset)
- `triangles`: The explicit surface representation

18.14 Mesh

Represent explicit surfaces on the device with triangle mesh. This was designed for close surfaces.

Internal structure

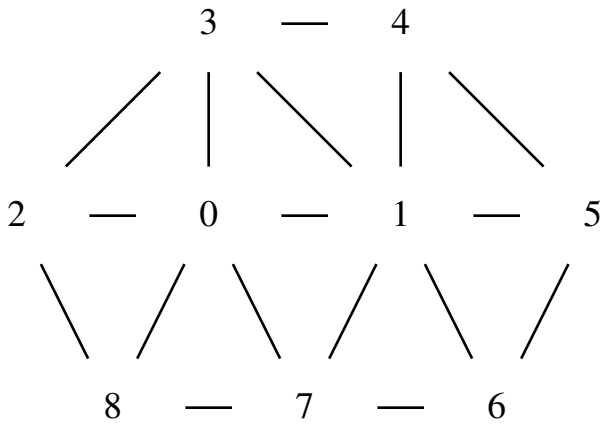
A *mirheo::Mesh* is composed of an array of vertices (it contains the coordinates of each vertex) and a list of faces. Each entry of the faces is composed of three indices that correspond to the vertices in the corresponding triangle.

A *mirheo::MembraneMesh* contains also adjacent information. This is a mapping from one vertex index to the indices of all adjacent vertices (that share an edge with the input vertex).

Example: In the following mesh, suppose that the maximum degree is `maxDegree = 7`, the adjacent lists have the entries:

```
1 7 8 2 3 * * 0 3 4 5 6 7 * ...
```

The first part is the ordered list of adjacent vertices of vertex `0` (the `*` indicates that the entry will not be used). The second part corresponds to vertex `1`. The first entry in each list is arbitrary, only the order is important. The list of adjacent vertices of vertex `i` starts at `i * maxDegree`.



API

Host classes

class Mesh

A triangle mesh structure.

The topology is represented by a list of faces (three vertex indices per face).

Subclassed by *mirheo::MembraneMesh*

Public Functions

Mesh()

Default constructor. no vertex and faces.

Mesh(**const** std::string &*fileName*)

Construct a `Mesh` from a off file.

Parameters

- `fileName`: The name of the file (contains the extension).

Mesh(**const** std::tuple<std::vector<real3>, std::vector<int3>> &*mesh*)

Construct a `Mesh` from a list of vertices and faces.

Mesh(**const** std::vector<real3> &*vertices*, **const** std::vector<int3> &*faces*)

Construct a `Mesh` from a list of vertices and faces.

Mesh(*Mesh*&&)

move constructor

Mesh &**operator=**(*Mesh*&&)
move assignment operator

int **getNtriangles**() **const**

Return the number of faces

int **getNvertices**() **const**

Return the number of vertices

int **getMaxDegree**() **const**

Return the maximum valence of all vertices

const *PinnedBuffer*<real4> &**getVertices**() **const**

Return the list of vertices

const *PinnedBuffer*<int3> &**getFaces**() **const**

Return the list of faces

class MembraneMesh : **public** *mirheo*::*Mesh*

A triangle mesh with face connectivity, adjacent vertices and geometric precomputed values.

This class was designed to assist *MembraneInteraction*.

A stress-free state can be associated to the mesh. The precomputed geometric quantities that are stored in the object are computed from the stress free state.

Additionally to the list of faces (

See *Mesh*), this class contains a list of adjacent vertices for each vertex. The list is stored in a single array, each vertex having a contiguous chunk of length maxDegree. See developer docs for more information.

Public Functions

MembraneMesh()
construct an empty mesh

MembraneMesh(**const** std::string &*initialMesh*)
Construct a *MembraneMesh* from an off file.

Note The stress free state will be the one given by `initialMesh`

Parameters

- `initialMesh`: File (in off format) that contains the mesh information.

MembraneMesh(**const** std::string &*initialMesh*, **const** std::string &*stressFreeMesh*)
Construct a *MembraneMesh* from an off file.

Note `initialMesh` and `stressFreeMesh` must have the same topology.

Parameters

- `initialMesh`: File (in off format) that contains the mesh information.
- `stressFreeMesh`: File (in off format) that contains the stress free state of the mesh.

MembraneMesh(**const** std::vector<real3> &*vertices*, **const** std::vector<int3> &*faces*)
Construct a *MembraneMesh* from a list of vertices and faces.

Note The stress free state is the same as the current mesh.

Parameters

- `vertices`: The vertex coordinates of the mesh
- `faces`: List of faces that contains the vertex indices.

MembraneMesh(**const** std::vector<real3> &*vertices*, **const** std::vector<real3> &*stressFreeVertices*, **const** std::vector<int3> &*faces*)
Construct a *MembraneMesh* from a list of vertices and faces.

Parameters

- `vertices`: The vertex coordinates of the mesh
- `stressFreeVertices`: The vertex coordinates that represent the stress free state.
- `faces`: List of faces that contains the vertex indices.

MembraneMesh(*MembraneMesh*&&)
move constructor

MembraneMesh &**operator=**(*MembraneMesh*&&)
move assignment operator

const *PinnedBuffer*<int> &**getAdjacents**() **const**

Return The adjacency list of each vertex

const *PinnedBuffer*<int> &**getDegrees**() **const**

Return The degree of each vertex

class MeshDistinctEdgeSets
Stores sets of edges that share the same colors as computed by computeEdgeColors().

This allows to work on edges in parallel with no race conditions.

Public Functions

MeshDistinctEdgeSets(**const** *MembraneMesh* **mesh*)
Construct a *MeshDistinctEdgeSets*.

Parameters

- `mesh`: The input mesh with adjacency lists.

int **numColors**() **const**

Return the number of colors in the associated mesh.

const *PinnedBuffer*<int2> &**edgeSet**(int *color*) **const**

Return The list of edges (vertex indices pairs) that have the given color.

Views

struct MeshView

A device-compatible structure that represents a triangle mesh topology (.

See *Mesh*)

Subclassed by *mirheo::MembraneMeshView*

Public Functions

MeshView(**const** *Mesh* *m)

Construct a *MeshView* from a *Mesh*.

Public Members

int **nvertices**

number of vertices

int **ntriangles**

number of faces

int3 ***triangles**

list of faces

struct MembraneMeshView : **public** *mirheo*::*MeshView*

A device-compatible structure that represents a data stored in a *MembraneMesh* additionally to its topology.

Public Functions

MembraneMeshView(**const** *MembraneMesh* *m)

Construct a *MembraneMeshView* from a *MembraneMesh* object.

Public Members

int **maxDegree**

maximum degree of all vertices

int ***adjacent**

lists of adjacent vertices

int ***degrees**

degree of each vertex

real ***initialLengths**

lengths of edges in the stress-free state

real ***initialAreas**

areas of each face in the stress-free state

real ***initialDotProducts**

do products between adjacent edges in the stress-free state

18.15 Mirheo Objects

All Mirheo objects must derive from this base class:

class MirObject

Base class for all the objects of *Mirheo*.

Subclassed by *mirheo::MirSimulationObject*, *mirheo::Postprocess*, *mirheo::PostprocessPlugin*, *mirheo::Simulation*

Public Functions

MirObject(**const** std::string &*name*)

Construct a *MirObject* object.

Parameters

- `name`: Name of the object.

const std::string &**getName**() **const**

Return the name of the object.

const char ***getCName**() **const**

Return the name of the object in c style. Useful for printf.

virtual void **checkpoint**(MPI_Comm *comm*, **const** std::string &*path*, int *checkPointId*)

Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

virtual void **restart**(MPI_Comm *comm*, **const** std::string &*path*)

Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

std::string **createCheckpointName**(**const** std::string &*path*, **const** std::string &*identifier*, **const** std::string &*extension*) **const**

Helper function to create file name for checkpoint/restart.

Return The file name.

Parameters

- `path`: The checkpoint/restart directory.
- `identifier`: An additional identifier, ignored if empty.
- `extension`: File extension.

std::string **createCheckpointNameWithId**(**const** std::string &*path*, **const** std::string &*identifier*, **const** std::string &*extension*, int *checkpointId*) **const**

Helper function to create file name for checkpoint/restart with a given Id.

Return The file name.

Parameters

- `path`: The checkpoint/restart directory.
- `identifier`: An additional identifier, ignored if empty.
- `extension`: File extension.
- `checkpointId`: Dump Id.

void **createCheckpointSymlink**(MPI_Comm *comm*, **const** std::string &*path*, **const** std::string &*identifier*, **const** std::string &*extension*, int *checkpointId*) **const**

Create a symlink with a name with no id to the file with a given id.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The checkpoint/restart directory.
- `identifier`: An additional identifier, ignored if empty.
- `extension`: File extension.
- `checkpointId`: Dump Id.

class MirSimulationObject : **public** *mirheo*::*MirObject*

Base class for the objects of *Mirheo* simulation task.

Contains global information common to all objects.

Subclassed by *mirheo::Bouncer*, *mirheo::Field*, *mirheo::Integrator*, *mirheo::Interaction*, *mirheo::ObjectBelongingChecker*, *mirheo::ParticleVector*, *mirheo::SimulationPlugin*, *mirheo::Wall*

Public Functions

MirSimulationObject(**const** *MirState* ***state*, **const** std::string &*name*)

Construct a *MirSimulationObject* object.

Parameters

- `name`: Name of the object.
- `state`: State of the simulation.

const *MirState* ****getState**() **const**

Return the simulation state.

virtual void **setState**(**const** *MirState* ***state*)

Set the simulation state.

18.16 Mirheo State

This class provides a global context to all Mirheo objects of the simulation.

class MirState
Global quantities accessible by all simulation objects in *Mirheo*.

Public Types

using TimeType = double
type used to store time information

using StepType = long long
type to store time step information

Public Functions

MirState(*DomainInfo domain*, real *dt* = (real)*InvalidDt*)
Construct a *MirState* object.

Parameters

- `domain`: The *DomainInfo* of the simulation
- `dt`: *Simulation* time step

void **checkpoint**(MPI_Comm *comm*, std::string *path*)
Save internal state to file.

Parameters

- `comm`: MPI comm of the simulation
- `path`: The directory in which to save the file

void **restart**(MPI_Comm *comm*, std::string *path*)
Load internal state from file.

Parameters

- `comm`: MPI comm of the simulation
- `path`: The directory from which to load the file

real **getDt**() **const**
Get the current time step dt.

Accessible only during *Mirheo::run*.

void **setDt**(real *dt*)
Set the time step dt.

Parameters

- `dt`: time step duration

Public Members

DomainInfo **domain**
Global *DomainInfo*.

TimeType **currentTime**
Current simulation time.

StepType **currentStep**
Current simulation step.

Public Static Attributes

constexpr real **InvalidDt** = -1
Special value used to initialize invalid dt.

18.17 Mirheo

The main coordinator class

API

class Mirheo
Coordinator class for a full simulation.

Manages and splits work between `Simulation` and `Postprocess` ranks.

Public Functions

Mirheo(int3 *nranks3D*, real3 *globalDomainSize*, LogInfo *logInfo*, CheckpointInfo *checkpointInfo*, real *maxObjHalfLength*, bool *gpuAwareMPI* = false)
Construct a `Mirheo` object using MPI_COMM_WORLD.

The product of `nranks3D` must be equal to the number of available ranks (or hals if postprocess is used)

Note MPI will be initialized internally. If this constructor is used, the destructor will also finalize MPI.

Parameters

- `nranks3D`: Number of ranks along each cartesian direction.
- `globalDomainSize`: The full domain dimensions in length units. Must be positive.
- `logInfo`: Information about logging
- `checkpointInfo`: Information about checkpoint
- `maxObjHalfLength`: Half of the maximum length of all objects.
- `gpuAwareMPI`: `true` to use RDMA (must be compile with a MPI version that supports it)

Mirheo(MPI_Comm *comm*, int3 *nranks3D*, real3 *globalDomainSize*, LogInfo *logInfo*, CheckpointInfo *checkpointInfo*, real *maxObjHalfLength*, bool *gpuAwareMPI* = false)
Construct a `Mirheo` object using a given communicator.

Note MPI will be NOT be initialized. If this constructor is used, the destructor will NOT finalize MPI.

void **restart** (std::string *folder* = "restart/")
reset the internal state from a checkpoint folder

MPI_Comm **getWorldComm**() **const**

Return the world communicator

bool **isComputeTask**() **const**

Return `true` if the current rank is a *Simulation* rank

bool **isMasterTask**() **const**

Return `true` if the current rank is the root (i.e. rank = 0)

bool **isSimulationMasterTask**() **const**

Return `true` if the current rank is the root within the simulation communicator

bool **isPostprocessMasterTask**() **const**

Return `true` if the current rank is the root within the postprocess communicator

void **startProfiler**()
start profiling for nvvp

void **stopProfiler**()
stop profiling for nvvp

void **dumpDependencyGraphToGraphML**(**const** std::string &*fname*, bool *current*) **const**
dump the task dependency of the simulation in graphML format.

Parameters

- `fname`: The file name to dump the graph to (without extension).
- `current`: if `true`, will only dump the current tasks; otherwise, will dump all possible ones.

void **run**(*MirState*::*StepType* *niters*, real *dt*)
advance the system for a given number of time steps

Parameters

- `niters`: number of interations
- `dt`: time step duration

void **registerParticleVector**(**const** std::shared_ptr<*ParticleVector*> &*pv*, **const** std::shared_ptr<*InitialConditions*> &*ic*)
register a *ParticleVector* in the simulation and initialize it with the gien *InitialConditions*.

Parameters

- `pv`: The *ParticleVector* to register
- `ic`: The *InitialConditions* that will be applied to `pv` when registered

void **registerInteraction**(**const** std::shared_ptr<*Interaction*> &*interaction*)
register an *Interaction*

See *setInteraction()*.

Parameters

- `interaction`: the *`Interaction`* to register.

void **`registerIntegrator`**(**`const`** std::shared_ptr<*Integrator*> &*integrator*)
register an *`Integrator`*

See *setIntegrator()*.

Parameters

- `integrator`: the *`Integrator`* to register.

void **`registerWall`**(**`const`** std::shared_ptr<*Wall*> &*wall*, int *checkEvery* = 0)
register a *`Wall`*

Parameters

- `wall`: The *`Wall`* to register
- `checkEvery`: The particles that will bounce against this wall will be checked (inside/outside log info) every this number of time steps. 0 means no check.

void **`registerBouncer`**(**`const`** std::shared_ptr<*Bouncer*> &*bouncer*)
register a *`Bouncer`*

See *setBouncer()*.

Parameters

- `bouncer`: the *`Bouncer`* to register.

void **`registerPlugins`**(**`const`** std::shared_ptr<*SimulationPlugin*> &*simPlugin*, **`const`** std::shared_ptr<*PostprocessPlugin*> &*postPlugin*)
register a *SimulationPlugin*

Parameters

- `simPlugin`: the *SimulationPlugin* to register (only relevant if the current rank is a compute task).
- `postPlugin`: the *PostprocessPlugin* to register (only relevant if the current rank is a postprocess task).

void **`registerPlugins`**(**`const`** PairPlugin &*plugins*)
More generic version of *registerPlugins()*

void **`registerObjectBelongingChecker`**(**`const`** std::shared_ptr<*ObjectBelongingChecker*> &*checker*, *ObjectVector* *ov)
register a *ObjectBelongingChecker*

See *applyObjectBelongingChecker()*

Parameters

- `checker`: the *ObjectBelongingChecker* to register.
- `ov`: the associated *ObjectVector* (must be registered).

void **deregisterIntegrator**(*Integrator* *integrator)
deregister an `Integrator`

See *registerIntegrator()*.

Parameters

- `integrator`: the `Integrator` to deregister.

void **deregisterPlugins**(*SimulationPlugin* *simPlugin, *PostprocessPlugin* *postPlugin)
deregister a *Plugin*

Parameters

- `simPlugin`: the *SimulationPlugin* to deregister (only relevant if the current rank is a compute task).
- `postPlugin`: the *PostprocessPlugin* to deregister (only relevant if the current rank is a postprocess task).

void **setIntegrator**(*Integrator* *integrator, *ParticleVector* *pv)
Assign a registered `Integrator` to a registered *ParticleVector*.

Parameters

- `integrator`: The registered integrator (will die if it was not registered)
- `pv`: The registered *ParticleVector* (will die if it was not registered)

void **setInteraction**(*Interaction* *interaction, *ParticleVector* *pv1, *ParticleVector* *pv2)
Assign two registered `Interaction` to two registered *ParticleVector* objects.

This was designed to handle *PairwiseInteraction*, which needs up to two *ParticleVector*. For self interaction cases (such as *MembraneInteraction*), `pv1` and `pv2` must be the same.

Parameters

- `interaction`: The registered interaction (will die if it is not registered)
- `pv1`: The first registered *ParticleVector* (will die if it is not registered)
- `pv2`: The second registered *ParticleVector* (will die if it is not registered)

void **setBouncer**(*Bouncer* *bouncer, *ObjectVector* *ov, *ParticleVector* *pv)
Assign a registered `Bouncer` to registered *ObjectVector* and *ParticleVector*.

Parameters

- `bouncer`: The registered bouncer (will die if it is not registered)
- `ov`: The registered *ObjectVector* that contains the surface to bounce on (will die if it is not registered)
- `pv`: The registered *ParticleVector* to bounce (will die if it is not registered)

void **setWallBounce**(*Wall* *wall, *ParticleVector* *pv, real *maximumPartTravel* = 0.25f)
Set a registered *ParticleVector* to bounce on a registered `Wall`.

Parameters

- `wall`: The registered wall (will die if it is not registered)

- `pv`: The registered *ParticleVector* (will die if it is not registered)
- `maximumPartTravel`: Performance parameter. See *Wall* for more information.

MirState ***getState**()

Return the global state of the system

const *MirState* ***getState**() **const**

Return the global state of the system (const version)

Simulation ***getSimulation**()

Return the *Simulation* object; `nullptr` on postprocess tasks.

const *Simulation* ***getSimulation**() **const**
see *getSimulation()*; const version

std::shared_ptr<*MirState*> **getMirState**()
see *getMirState()*; shared_ptr version

void **dumpWalls2XDMF**(std::vector<std::shared_ptr<*Wall*>> *walls*, real3 *h*, **const** std::string &*filename*)
Compute the SDF field from the given walls and dump it to a file in xmf+h5 format.

Parameters

- `walls`: List of *Wall* objects. The union of these walls will be dumped.
- `h`: The grid spacing
- `filename`: The base name of the dumped files (without extension)

double **computeVolumeInsideWalls**(std::vector<std::shared_ptr<*Wall*>> *walls*, long *nSamplesPerRank* = 100000)
Compute the volume inside the geometry formed by the given walls with simple Monte-Carlo integration.

Return The Monte-Carlo estimate of the volume

Parameters

- `walls`: List of *Wall* objects. The union of these walls form the geometry.
- `nSamplesPerRank`: The number of Monte-Carlo samples per rank

std::shared_ptr<*ParticleVector*> **makeFrozenWallParticles**(std::string *pvName*, std::vector<std::shared_ptr<*Wall*>> *walls*, std::vector<std::shared_ptr<*Interaction*>> *interactions*, std::shared_ptr<*Integrator*> *integrator*, real *numDensity*, real *mass*, real *dt*, int *nsteps*)
Create a layer of frozen particles inside the given walls.

This will run a simulation of "bulk" particles and select the particles that are inside the effective cut-off radius of the given list of interactions.

Return The frozen particles

Parameters

- `pvName`: The name of the frozen *ParticleVector* that will be created
- `walls`: The list of registered walls that need frozen particles
- `interactions`: List of interactions (not necessarily registered) that will be used to equilibrate the particles
- `integrator`: *Integrator* object used to equilibrate the particles
- `numDensity`: The number density used to initialize the particles
- `mass`: The mass of one particle
- `dt`: Equilibration time step
- `nsteps`: Number of equilibration steps

std::shared_ptr<*ParticleVector*> **makeFrozenRigidParticles**(std::shared_ptr<*ObjectBelongingChecker*> *checker*, std::shared_ptr<*ObjectVector*> *shape*, std::shared_ptr<*InitialConditions*> *icShape*, std::vector<std::shared_ptr<*Interaction*>> *interactions*, std::shared_ptr<*Integrator*> *integrator*, real *numDensity*, real *mass*, real *dt*, int *nsteps*)

Create frozen particles inside the given objects.

This will run a simulation of "bulk" particles and select the particles that are inside `shape`.

Return The frozen particles, with name "inside_" + name of `shape`

Note For now, the output *ParticleVector* has mass 1.0.

Parameters

- `checker`: The *ObjectBelongingChecker* to split inside particles
- `shape`: The *ObjectVector* that will be used to define inside particles
- `icShape`: The *InitialConditions* object used to set the objects positions
- `interactions`: List of interactions (not necessarily registered) that will be used to equilibrate the particles
- `integrator`: *Integrator* object used to equilibrate the particles
- `numDensity`: The number density used to initialize the particles
- `mass`: The mass of one particle
- `dt`: Equilibration time step
- `nsteps`: Number of equilibration steps

std::shared_ptr<*ParticleVector*> **applyObjectBelongingChecker**(*ObjectBelongingChecker* **checker*, *ParticleVector* **pv*, int *checkEvery*, std::string *inside* = "", std::string *outside* = "")

Enable a registered *ObjectBelongingChecker* to split particles of a registered *ParticleVector*.

`inside` or `outside` can take the reserved value "none", in which case the corresponding particles will be deleted. Furthermore, exactly one of `inside` and `outside` must be the same as `pv`.

Parameters

- `checker`: The *ObjectBelongingChecker* (will die if it is not registered)
- `pv`: The registered *ParticleVector* that must be split (will die if it is not registered)
- `checkEvery`: The particle split will be performed every this amount of time steps.
- `inside`: Name of the *ParticleVector* that will contain the particles of `pv` that are inside the objects. See below for more information.
- `outside`: Name of the *ParticleVector* that will contain the particles of `pv` that are outside the objects. See below for more information.

If `inside` or `outside` has the name of a *ParticleVector* that is not registered, this call will create an empty *ParticleVector* with the given name and register it in the *Simulation*. Otherwise the already registered *ParticleVector* will be used.

void **logCompileOptions**() **const**
print the list of all compile options and their current value in the logs

18.18 Object Belonging checkers

See also *the user interface*.

Base class

class ObjectBelongingChecker : **public** *mirheo*::*MirSimulationObject*
Mark or split particles which are inside of a given *ObjectVector*.

The user must call *setup()* exactly once before any call of *checkInner()* or *splitByBelonging()*.

Subclassed by *mirheo::ObjectVectorBelongingChecker*

Public Functions

ObjectBelongingChecker(**const** *MirState* **state*, **const** std::string &*name*)
Construct a *ObjectBelongingChecker* object.

Parameters

- `state`: *Simulation* state.
- `name`: Name of the bouncer.

virtual void **splitByBelonging**(*ParticleVector* **src*, *ParticleVector* **pvIn*, *ParticleVector* **pvOut*, cudaStream_t *stream*) = 0
Split a *ParticleVector* into inside and outside particles.

The `pvIn` and `pvOut` *ParticleVector* can be set to `nullptr`, in which case they will be ignored. If `pvIn` and `src` point to the same object, `pvIn` will contain only inside particles of `src` in the end. Otherwise, `pvIn` will contain its original particles additionally to the inside particles of `src`. If `pvOut` and `src` point to the same object, `pvOut` will contain only outside particles of `src` in the end. Otherwise, `pvOut` will contain its original particles additionally to the outside particles of `src`.

Parameters

- `src`: The particles to split.
- `pvIn`: Buffer that will contain the inside particles.
- `pvOut`: Buffer that will contain the outside particles.
- `stream`: Stream used for the execution.

This method will die if the type of `pvIn`, `pvOut` and `src` have a different type.

Must be called after *setup()*.

virtual void **checkInner**(*ParticleVector* *pv, *CellList* *cl, cudaStream_t *stream*) = 0
Prints number of inside and outside particles in the log as a `Info` entry.

Additionally, this will compute the inside/outside tags of the particles and store it inside this object instance.

Parameters

- `pv`: The particles to check.
- `cl`: Cell lists of pv.
- `stream`: Stream used for execution.

Must be called after *setup()*.

virtual void **setup**(*ObjectVector* *ov) = 0
Register the *ObjectVector* that defines inside and outside.

Parameters

- `ov`: The *ObjectVector* to register.

virtual std::vector<std::string> **getChannelsToBeExchanged**() **const**
Return the channels of the registered *ObjectVector* to be exchanged before splitting.

virtual *ObjectVector* ***getObjectVector**() = 0
Return the registered *ObjectVector*.

Derived classes

class ObjectVectorBelongingChecker : **public** *mirheo*::*ObjectBelongingChecker*
ObjectBelongingChecker base implementation.

Subclassed by *mirheo::MeshBelongingChecker*, *mirheo::RodBelongingChecker*, *mirheo::ShapeBelongingChecker< Shape >*

Public Functions

ObjectVectorBelongingChecker(**const** *MirState* *state, **const** std::string &*name*)
Construct a *ObjectVectorBelongingChecker* object.

Parameters

- `state`: *Simulation* state.
- `name`: Name of the bouncer.

void **splitByBelonging**(*ParticleVector* *src, *ParticleVector* *pvIn, *ParticleVector* *pvOut, cudaStream_t *stream*)

Split a *ParticleVector* into inside and outside particles.

The `pvIn` and `pvOut` *ParticleVector* can be set to `nullptr`, in which case they will be ignored. If `pvIn` and `src` point to the same object, `pvIn` will contain only inside particles of `src` in the end. Otherwise, `pvIn` will contain its original particles additionally to the inside particles of `src`. If `pvOut` and `src` point to the same object, `pvOut` will contain only outside particles of `src` in the end. Otherwise, `pvOut` will contain its original particles additionally to the outside particles of `src`.

Parameters

- `src`: The particles to split.
- `pvIn`: Buffer that will contain the inside particles.
- `pvOut`: Buffer that will contain the outside particles.
- `stream`: Stream used for the execution.

This method will die if the type of `pvIn`, `pvOut` and `src` have a different type.

Must be called after *setup()*.

void **checkInner**(*ParticleVector* *pv, *CellList* *cl, cudaStream_t *stream*)

Prints number of inside and outside particles in the log as a `Info` entry.

Additionally, this will compute the inside/outside tags of the particles and store it inside this object instance.

Parameters

- `pv`: The particles to check.
- `cl`: Cell lists of pv.
- `stream`: Stream used for execution.

Must be called after *setup()*.

void **setup**(*ObjectVector* *ov)

Register the *ObjectVector* that defines inside and outside.

Parameters

- `ov`: The *ObjectVector* to register.

std::vector<std::string> **getChannelsToBeExchanged**() **const**

Return the channels of the registered *ObjectVector* to be exchanged before splitting.

ObjectVector ***getObjectVector**()

Return the registered *ObjectVector*.

class MeshBelongingChecker : **public** *mirheo*::*ObjectVectorBelongingChecker*

Check in/out status of particles against an *ObjectVector* with a triangle mesh.

class RodBelongingChecker : **public** *mirheo*::*ObjectVectorBelongingChecker*

Check in/out status of particles against a RodObjectVector.

Public Functions

RodBelongingChecker(**const** *MirState* *state, **const** std::string &name, real *radius*)
Construct a *RodBelongingChecker* object.

Parameters

- `state`: *Simulation* state.
- `name`: Name of the bouncer.
- `radius`: The radius of the rod. Must be positive.

template <**class** Shape>
class ShapeBelongingChecker : **public** *mirheo*::*ObjectVectorBelongingChecker*
Check in/out status of particles against a *RigidShapedObjectVector*.

Template Parameters

- `Shape`: The AnalyticShape that represent the shape of the objects.

18.19 Plugins

Interface

class Plugin
Base class to represent a *Plugin*.

Plugins are functionalities that are not required to run a simulation. Each plugin must have a *SimulationPlugin* derived class, and, optionally, a compatible *PostprocessPlugin* derived class. The latter is used to perform potentially expensive work asynchronously while the simulation is running (e.g. I/O).

Subclassed by *mirheo::PostprocessPlugin*, *mirheo::SimulationPlugin*

Public Functions

Plugin()
default constructor

virtual void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

void **setTag**(int *tag*)
Set the tag that will be used internally to communicate between *SimulationPlugin* and a *PostprocessPlugin*.

Must be called before any other methods.

Parameters

- `tag`: The tag, must be unique (all plugins using the same intercommunicator must have a different tag, see _setup())

class SimulationPlugin : **public** *mirheo*::*Plugin*, **public** *mirheo*::*MirSimulationObject*
Base class for the simulation side of a `Plugin`.

A simulation plugin is able to modify the state of the simulation. Depending on its action, one of the "hooks" (e.g. *beforeCellLists()*) must be overriden (by default they do not do anything).

If a plugin needs reference to objects held by the simulation, it must be saved in its internal structure at *setup()* time.

Subclassed by *mirheo::AddForcePlugin*, mirheo::AddFourRollMillForcePlugin, *mirheo::AddTorquePlugin*, *mirheo::AnchorParticlesPlugin*, *mirheo::Average3D*, *mirheo::BerendsenThermostatPlugin*, *mirheo::DensityControlPlugin*, *mirheo::ExchangePVSFluxPlanePlugin*, *mirheo::ExternalMagneticTorquePlugin*, *mirheo::ForceSaverPlugin*, *mirheo::ImposeProfilePlugin*, *mirheo::ImposeVelocityPlugin*, *mirheo::MagneticDipoleInteractionsPlugin*, mirheo::MembraneExtraForcePlugin, *mirheo::MeshPlugin*, *mirheo::MsdPlugin*, *mirheo::ObjStatsPlugin*, *mirheo::OutletPlugin*, *mirheo::ParticleChannelAveragerPlugin*, *mirheo::ParticleChannelSaverPlugin*, *mirheo::ParticleCheckerPlugin*, *mirheo::ParticleDisplacementPlugin*, *mirheo::ParticleDragPlugin*, *mirheo::ParticleSenderPlugin*, *mirheo::PinObjectPlugin*, *mirheo::PinRodExtremityPlugin*, *mirheo::RdfPlugin*, *mirheo::SimulationStats*, *mirheo::SimulationVelocityControl*, *mirheo::TemperaturizePlugin*, *mirheo::VacfPlugin*, *mirheo::VelocityInletPlugin*, *mirheo::VirialPressurePlugin*, *mirheo::WallForceCollectorPlugin*, *mirheo::WallRepulsionPlugin*, *mirheo::XYZPlugin*

Public Functions

SimulationPlugin(**const** *MirState* **state*, **const** std::string &*name*)
Construct a *SimulationPlugin*.

Parameters

- `state`: The global simulation state
- `name`: the name of the plugin (must be the same as that of the postprocess plugin)

virtual bool **needPostproc**() = 0

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

virtual void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

virtual void **beforeCellLists**(cudaStream_t *stream*)
hook before building the cell lists

virtual void **beforeForces**(cudaStream_t *stream*)
hook before computing the forces and after the cell lists are created

virtual void **beforeIntegration**(cudaStream_t *stream*)
hook before integrating the particle vectors but after the forces are computed

virtual void **afterIntegration**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

virtual void **beforeParticleDistribution**(cudaStream_t *stream*)
hook before redistributing *ParticleVector* objects and after bounce

virtual void **serializeAndSend**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

virtual void **finalize**()
hook that happens once at the end of the simulation loop

class PostprocessPlugin : **public** *mirheo*::*Plugin*, **public** *mirheo*::*MirObject*
Base class for the postprocess side of a `Plugin`.

A postprocess plugin can only receive information from its associated *SimulationPlugin*. The use of such class is to wait for a message and then deserialize it (where additional actions may be performed, such as I/O).

Subclassed by *mirheo::AnchorParticlesStatsPlugin*, *mirheo::MeshDumper*, *mirheo::MsdDumper*, *mirheo::ObjStatsDumper*, *mirheo::ParticleDumperPlugin*, *mirheo::PostprocessDensityControl*, *mirheo::PostprocessStats*, *mirheo::PostprocessVelocityControl*, *mirheo::RdfDump*, mirheo::ReportPinObjectPlugin, *mirheo::UniformCartesianDumper*, *mirheo::VacfDumper*, *mirheo::VirialPressureDumper*, *mirheo::WallForceDumperPlugin*, *mirheo::XYZDumper*

Public Functions

PostprocessPlugin(**const** std::string &*name*)
Construct a *PostprocessPlugin*.

Parameters

- `name`: the name of the plugin (must be the same as that of the associated simulation plugin)

virtual void **setup**(**const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *PostprocessPlugin*.

This method must be called before any other function call.

Parameters

- `comm`: Contains all postprocess ranks
- `interComm`: used to communicate with the simulation ranks

void **recv**()
Post an asynchronous receive request to get a message from the associated *SimulationPlugin*.

MPI_Request **waitData**()
wait for the completion of the asynchronous receive request. Must be called after *recv()* and before *deserialize()*.

virtual void **deserialize**() = 0
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

List of plugins

Dump Plugins

These plugins do not modify the state of the simulation. They can be used to dump selected parts of the state of the simulation to the disk.

class Average3D : **public** *mirheo*::*SimulationPlugin*

Average particles quantities into spacial bins over a cartesian grid and average it over time.

Useful to compute e.g. velocity or density profiles. The number density is always computed inside each bin, as it is used to compute the averages. Other quantities must be specified by giving the channel names.

This plugin should be used with *UniformCartesianDumper* on the postprocessing side.

Cannot be used with multiple invocations of `Mirheo.run`.

Subclassed by *mirheo::AverageRelative3D*

Public Types

enum ChannelType

Specify the form of a channel data.

Values:

Scalar

Vector_real3

Vector_real4

Tensor6

None

Public Functions

Average3D(**const** *MirState* **state*, std::string *name*, std::vector<std::string> *pvNames*, std::vector<std::string> *channelNames*, int *sampleEvery*, int *dumpEvery*, real3 *binSize*)

Create an `Average3D` object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: The list of names of the *ParticleVector* that will be used when averaging.
- `channelNames`: The list of particle data channels to average. Will die if the channel does not exist.
- `sampleEvery`: Compute spatial averages every this number of time steps.
- `dumpEvery`: Compute time averages and send to the postprocess side every this number of time steps.
- `binSize`: Size of one spatial bin along the three axes.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

void **afterIntegration**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **serializeAndSend**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

struct HostChannelsInfo
A helper structure that contains grid average info for all required channels.

Public Members

int **n**
The number of channels (excluding number density).

std::vector<std::string> **names**
List of channel names.

PinnedBuffer<*ChannelType*> **types**
List of channel data forms.

PinnedBuffer<real *> **averagePtrs**
List of averages of each channel.

PinnedBuffer<real *> **dataPtrs**
List of data to average, for each channel.

std::vector<*DeviceBuffer*<real>> **average**
data container for the averages, for each channel.

class AverageRelative3D : **public** *mirheo*::*Average3D*
Perform the same task as *AverageRelative3D* on a grid that moves relatively to a given object's center of mass in a *RigidObjectVector*.

Cannot be used with multiple invocations of *Mirheo.run*.

Public Functions

AverageRelative3D(**const** *MirState* *state, std::string *name*, std::vector<std::string> *pvNames*, std::vector<std::string> *channelNames*, int *sampleEvery*, int *dumpEvery*, real3 *binSize*, std::string *relativeOVname*, int *relativeID*)

Create an *AverageRelative3D* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: The list of names of the *ParticleVector* that will be used when averaging.
- `channelNames`: The list of particle data channels to average. Will die if the channel does not exist.
- `sampleEvery`: Compute spatial averages every this number of time steps.
- `dumpEvery`: Compute time averages and send to the postprocess side every this number of time steps.
- `binSize`: Size of one spatial bin along the three axes.
- `relativeOVname`: Name of the *RigidObjectVector* that contains the reference object.
- `relativeID`: Index of the reference object within the *RigidObjectVector*.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)

setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **afterIntegration**(cudaStream_t *stream*)

hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **serializeAndSend**(cudaStream_t *stream*)

Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class UniformCartesianDumper : **public** *mirheo*::*PostprocessPlugin*

Postprocessing side of `Average3D` or `AverageRelative3D`.

Dump uniform grid data to xmf + hdf5 format.

Public Functions

UniformCartesianDumper(std::string *name*, std::string *path*)
Create a *UniformCartesianDumper*.

Parameters

- `name`: The name of the plugin.
- `path`: The files will be dumped to `pathXXXXX.[xmf,h5]`, where `XXXXX` is the time stamp.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

XDMF::*Channel* **getChannelOrDie**(std::string *chname*) **const**
Get the average channel data.

Return The channel data.

Parameters

- `chname`: The name of the channel.

std::vector<int> **getLocalResolution**() **const**
Get the grid size in the local domain.

Return An array with 3 entries, contains the number of grid points along each direction.

class MeshPlugin : **public** *mirheo*::*SimulationPlugin*
Send mesh information of an object for dump to *MeshDumper* postprocess plugin.

Public Functions

MeshPlugin(**const** *MirState* **state*, std::string *name*, std::string *ovName*, int *dumpEvery*)
Create a *MeshPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `ovName`: The name of the *ObjectVector* that has a mesh to dump.
- `dumpEvery`: Will dump the mesh every this number of timesteps.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`beforeForces`**(cudaStream_t *stream*)
hook before computing the forces and after the cell lists are created

void **`serializeAndSend`**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

`class MeshDumper` : **`public`** *mirheo*::*PostprocessPlugin*
Postprocess side of *MeshPlugin*.

Receives mesh info and dump it to ply format.

Public Functions

`MeshDumper`(std::string *name*, std::string *path*)
Create a *MeshDumper* object.

Parameters

- `name`: The name of the plugin.
- `path`: The files will be dumped to `path-XXXXX.ply`.

void **`deserialize`**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **`setup`**(**`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *PostprocessPlugin*.

This method must be called before any other function call.

Parameters

- `comm`: Contains all postprocess ranks
- `interComm`: used to communicate with the simulation ranks

`class ParticleSenderPlugin` : **`public`** *mirheo*::*SimulationPlugin*
Send particle data to *ParticleDumperPlugin*.

Subclassed by *mirheo::ParticleWithMeshSenderPlugin*, mirheo::ParticleWithPolylinesSenderPlugin

Public Functions

ParticleSenderPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, int *dumpEvery*, **const** std::vector<std::string> &*channelNames*)

Create a *ParticleSenderPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to dump.
- `dumpEvery`: Send the data to the postprocess side every this number of steps.
- `channelNames`: The list of channels to send, additionally to the default positions, velocities and global ids.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)

setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **handshake**()

Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

void **beforeForces**(cudaStream_t *stream*)

hook before computing the forces and after the cell lists are created

void **serializeAndSend**(cudaStream_t *stream*)

Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

Public Members

std::string **pvName_**

name of the *ParticleVector* to dump.

ParticleVector ***pv_**

pointer to the *ParticleVector* to dump.

std::vector<char> **sendBuffer_**
Buffer used to send the data to the postprocess side.

class ParticleDumperPlugin : **public** *mirheo*::*PostprocessPlugin*
Postprocess side of *ParticleSenderPlugin*.

Dump particles data to xmf + hdf5 format.

Subclassed by *mirheo::ParticleWithMeshDumperPlugin*, mirheo::ParticleWithPolylinesDumperPlugin

Public Functions

ParticleDumperPlugin(std::string *name*, std::string *path*)
Create a *ParticleDumperPlugin* object.

Parameters

- `name`: The name of the plugin.
- `path`: *Particle* data will be dumped to `pathXXXXX.[xmf,h5]`.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

class ParticleWithMeshSenderPlugin : **public** *mirheo*::*ParticleSenderPlugin*
Send particle data to *ParticleWithMeshSenderPlugin*.

Does the same as *ParticleSenderPlugin* with additional *Mesh* connectivity information. This is compatible only with *ObjectVector*.

Public Functions

ParticleWithMeshSenderPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, int *dumpEvery*, **const** std::vector<std::string> &*channelNames*)
Create a *ParticleWithMeshSenderPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to dump.
- `dumpEvery`: Send the data to the postprocess side every this number of steps.
- `channelNames`: The list of channels to send, additionally to the default positions, velocities and global ids.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

class ParticleWithMeshDumperPlugin : **public** *mirheo*::*ParticleDumperPlugin*
Postprocess side of *ParticleWithMeshSenderPlugin*.

Dump particles data with connectivity to xmf + hdf5 format.

Public Functions

ParticleWithMeshDumperPlugin(std::string *name*, std::string *path*)
Create a *ParticleWithMeshDumperPlugin* object.

Parameters

- `name`: The name of the plugin.
- `path`: Data will be dumped to `pathXXXXX.[xmf,h5]`.

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

class XYZPlugin : **public** *mirheo*::*SimulationPlugin*
Send particle positions to *XYZDumper*.

Public Functions

XYZPlugin(**const** *MirState* *state, std::string *name*, std::string *pvName*, int *dumpEvery*)
Create a *XYZPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to dump.
- `dumpEvery`: Send the data to the postprocess side every this number of steps.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeForces**(cudaStream_t *stream*)
hook before computing the forces and after the cell lists are created

void **serializeAndSend**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class XYZDumper : **public** *mirheo*::*PostprocessPlugin*
Postprocess side of *XYZPlugin*.

Dump the particle positions to simple .xyz format.

Public Functions

XYZDumper(std::string *name*, std::string *path*)
Create a *XYZDumper* object.

Parameters

- `name`: The name of the plugin.
- `path`: Data will be dumped to `pathXXXXX.xyz`.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **setup**(**const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *PostprocessPlugin*.

This method must be called before any other function call.

Parameters

- `comm`: Contains all postprocess ranks
- `interComm`: used to communicate with the simulation ranks

Statistics and In-situ analysis Plugins

These plugins do not modify the state of the simulation. They are used to measure properties of the simulation that can be processed directly at runtime. Their output is generally much lighter than dump plugins. The prefered format is csv, to allow clean postprocessing from e.g. python.

class MsdPlugin : **public** *mirheo*::*SimulationPlugin*

Compute the mean squared distance (MSD) of a given *ParticleVector*.

The MSD is computed every dumpEvery steps on the time interval [startTime, endTime].

Each particle stores the total displacement from startTime. To compute this, it also stores its position at each step.

Public Functions

MsdPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, *MirState*::*TimeType* *startTime*, *MirState*::*TimeType* *endTime*, int *dumpEvery*)

Create a *MsdPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* from which to measure the MSD.
- `startTime`: MSD will use this time as origin.
- `endTime`: The MSD will be reported only on [startTime, endTime].
- `dumpEvery`: Will send the MSD to the postprocess side every this number of steps, only during the valid time interval.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)

setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **afterIntegration**(cudaStream_t *stream*)

hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **serializeAndSend**(cudaStream_t *stream*)

Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class MsdDumper : **public** *mirheo*::*PostprocessPlugin*
Postprocess side of *MsdPlugin*.

Dumps the VACF in a csv file.

Public Functions

MsdDumper(std::string *name*, std::string *path*)
Create a *MsdDumper* object.

Parameters

- `name`: The name of the plugin.
- `path`: The folder that will contain the vacf csv file.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **setup**(**const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *PostprocessPlugin*.

This method must be called before any other function call.

Parameters

- `comm`: Contains all postprocess ranks
- `interComm`: used to communicate with the simulation ranks

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

class ObjStatsPlugin : **public** *mirheo*::*SimulationPlugin*
Send object information to *ObjStatsDumper*.

Used to track the center of mass, linear and angular velocities, orintation, forces and torques of an *ObjectVector*.

Public Functions

ObjStatsPlugin(**const** *MirState* **state*, std::string *name*, std::string *ovName*, int *dumpEvery*)
Create a *ObjStatsPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.

- `ovName`: The name of the *ObjectVector* to extract the information from.
- `dumpEvery`: Send the information to the postprocess side every this number of steps.

void **`setup`**(*Simulation* **simulation*, **`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`afterIntegration`**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **`serializeAndSend`**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

void **`handshake`**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

`class ObjStatsDumper` : **`public`** *mirheo*::*PostprocessPlugin*
Postprocess side of *ObjStatsPlugin*.

Dump object information to a csv file.

Public Functions

`ObjStatsDumper`(std::string *name*, std::string *filename*)
Create a *ObjStatsDumper* object.

Parameters

- `name`: The name of the plugin.
- `filename`: The name of the csv file to dump to.

void **`deserialize`**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **`setup`**(**`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *PostprocessPlugin*.

This method must be called before any other function call.

Parameters

- `comm`: Contains all postprocess ranks
- `interComm`: used to communicate with the simulation ranks

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

void **checkpoint**(MPI_Comm *comm*, **const** std::string &*path*, int *checkPointId*)
Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

void **restart**(MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

class RdfPlugin : **public** *mirheo*::*SimulationPlugin*
Measure the radial distribution function (RDF) of a *ParticleVector*.

The RDF is estimated periodically from ensemble averages. See *RdfDump* for the I/O.

Public Functions

RdfPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, real *maxDist*, int *nbins*, int *computeEvery*)
Create a *RdfPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* from which to measure the RDF.
- `maxDist`: The RDF will be measured on [0, maxDist].
- `nbins`: The number of bins in the interval [0, maxDist].
- `computeEvery`: The number of time steps between two RDF evaluations and dump.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`afterIntegration`**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **`serializeAndSend`**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

`class RdfDump` : **`public`** *mirheo*::*PostprocessPlugin*
Postprocess side of *RdfPlugin*.

Dump the RDF to a csv file.

Public Functions

`RdfDump`(std::string *name*, std::string *basename*)
Create a *RdfDump* object.

Parameters

- `name`: The name of the plugin.
- `basename`: The RDF will be dumped to `basenameXXXXX.csv`.

void **`setup`**(**`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *PostprocessPlugin*.

This method must be called before any other function call.

Parameters

- `comm`: Contains all postprocess ranks
- `interComm`: used to communicate with the simulation ranks

void **`deserialize`**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

`class SimulationStats` : **`public`** *mirheo*::*SimulationPlugin*
Collect global statistics of the simulation and send it to the postprocess ranks.

Compute total linear momentum and estimate of temperature. Furthermore, measures average wall time of time steps.

Public Functions

SimulationStats(**const** *MirState* **state*, std::string *name*, int *every*, std::vector<std::string> *pvNames*)
Create a *SimulationPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `every`: Compute the statistics every this number of steps.
- `pvNames`: List of names of the pvs to compute statistics from. If empty, will take all the pvs in the simulation.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **afterIntegration**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **serializeAndSend**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class PostprocessStats : **public** *mirheo*::*PostprocessPlugin*
Dump the stats sent by *SimulationStats* to a csv file and to the console output.

Public Functions

PostprocessStats(std::string *name*, std::string *filename* = std::string())
Construct a *PostprocessStats* plugin.

Parameters

- `name`: The name of the plugin.
- `filename`: The csv file name that will be dumped.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **checkpoint**(MPI_Comm *comm*, **const** std::string &*path*, int *checkPointId*)
Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

void **restart**(MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

class VacfPlugin : **public** *mirheo*::*SimulationPlugin*
Compute the velocity autocorrelation function (VACF) of a given *ParticleVector*.

The VACF is computed every dumpEvery steps on the time interval [startTime, endTime].

Public Functions

VacfPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, *MirState*::*TimeType* *startTime*, *MirState*::*TimeType* *endTime*, int *dumpEvery*)
Create a *VacfPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* from which to measure the VACF.
- `startTime`: VACF will use this time as origin.
- `endTime`: The VACF will be reported only on [startTime, endTime].
- `dumpEvery`: Will send the VACF to the postprocess side every this number of steps, only during the valid time interval.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks

- `interComm`: used to communicate with the postprocess ranks

void **afterIntegration**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **serializeAndSend**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class VacfDumper : **public** *mirheo*::*PostprocessPlugin*
Postprocess side of *VacfPlugin*.

Dumps the VACF in a csv file.

Public Functions

VacfDumper(std::string *name*, std::string *path*)
Create a *VacfDumper* object.

Parameters

- `name`: The name of the plugin.
- `path`: The folder that will contain the vacf csv file.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **setup**(**const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *PostprocessPlugin*.

This method must be called before any other function call.

Parameters

- `comm`: Contains all postprocess ranks
- `interComm`: used to communicate with the simulation ranks

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

class VirialPressurePlugin : **public** *mirheo*::*SimulationPlugin*
Compute the pressure in a given region from the virial theorem and send it to the *VirialPressureDumper*.

Public Functions

VirialPressurePlugin(**const** *MirState* *state, std::string *name*, std::string *pvName*, FieldFunction *func*, real3 *h*, int *dumpEvery*)
Create a *VirialPressurePlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to add the particles to.
- `func`: The scalar field is negative in the region of interest and positive outside.
- `h`: The grid size used to discretize the field.
- `dumpEvery`: Will compute and send the pressure every this number of steps.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **afterIntegration**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **serializeAndSend**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class VirialPressureDumper : **public** *mirheo*::*PostprocessPlugin*
Postprocess side of *VirialPressurePlugin*.

Recieves and dump the virial pressure.

Public Functions

VirialPressureDumper(std::string *name*, std::string *path*)
Create a *VirialPressureDumper*.

Parameters

- `name`: The name of the plugin.
- `path`: The csv file to which the data will be dumped.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **setup**(**const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *PostprocessPlugin*.

This method must be called before any other function call.

Parameters

- `comm`: Contains all postprocess ranks
- `interComm`: used to communicate with the simulation ranks

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

class WallForceCollectorPlugin : **public** *mirheo*::*SimulationPlugin*
Compute the force exerted by particles on the walls.

It has two contributions:

- Interactio forces with frozen particles
- bounce-back.

Public Functions

WallForceCollectorPlugin(**const** *MirState* **state*, std::string *name*, std::string *wallName*, std::string *frozenPvName*, int *sampleEvery*, int *dumpEvery*)
Create a *WallForceCollectorPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `wallName`: The name of the *Wall* to collect the forces from.
- `frozenPvName`: The name of the frozen *ParticleVector* assigned to the wall.
- `sampleEvery`: Compute forces every this number of steps, and average it in time.
- `dumpEvery`: Send the average forces to the postprocessing side every this number of steps.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **afterIntegration**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **serializeAndSend**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class WallForceDumperPlugin : **public** *mirheo*::*PostprocessPlugin*
Postprocess side of *WallForceCollectorPlugin*.

Dump the forces to a txt file.

Public Functions

WallForceDumperPlugin(std::string *name*, std::string *filename*, bool *detailedDump*)
Create a *WallForceDumperPlugin*.

Parameters

- `name`: The name of the plugin.
- `filename`: The file to dump the stats to.
- `detailedDump`: If `true`, the file will contain the bounce contribution and particle interactions contributions instead of the sum.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

Modifier plugins

These plugins add more functionalities to the simulation.

class AddForcePlugin : **public** *mirheo*::*SimulationPlugin*
Add a constant force to every particle of a given *ParticleVector* at every time step.

The force is added at the beforeForce() stage.

Public Functions

AddForcePlugin(**const** *MirState* *state, **const** std::string &name, **const** std::string &pvName, real3 force)
Create a *AddForcePlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to which the force should be applied.
- `force`: The force to apply.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeForces**(cudaStream_t stream)
hook before computing the forces and after the cell lists are created

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class AddTorquePlugin : **public** *mirheo*::*SimulationPlugin*
Add a constant torque to every object of a given *RigidObjectVector* at every time step.

The torque is added at the beforeForce() stage.

Public Functions

AddTorquePlugin(**const** *MirState* *state, **const** std::string &name, **const** std::string &rovName, real3 torque)
Create a *AddTorquePlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `rovName`: The name of the *RigidObjectVector* to which the torque should be applied.
- `torque`: The torque to apply.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeForces**(cudaStream_t *stream*)
hook before computing the forces and after the cell lists are created

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class AnchorParticlesPlugin : **public** *mirheo*::*SimulationPlugin*
Add constraints on the positions and velocities of given particles of a *ParticleVector*.

The forces required to keep the particles along the given constrains are recorded and reported via *AnchorParticlesStatsPlugin*.

Note This should not be used with *RigidObjectVector*.

Note This was designed to work with ObjectVectors containing a single object, on a single rank. Using a plain *ParticleVector* might not work since particles will be reordered.

Public Functions

AnchorParticlesPlugin(**const** *MirState* *state, std::string *name*, std::string *pvName*, FuncTime3D *positions*, FuncTime3D *velocities*, std::vector<int> *pids*, int *reportEvery*)
Create a *AnchorParticlesPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* that contains the particles of interest.
- `positions`: The constrains on the positions.
- `velocities`: The constrains on the velocities.
- `pids`: The concerned particle ids (starting from 0). See the restrictions in the class docs.
- `reportEvery`: Statistics (forces) will be sent to the *AnchorParticlesStatsPlugin* every this number of steps.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`afterIntegration`**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **`serializeAndSend`**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

void **`handshake`**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

`class AnchorParticlesStatsPlugin` : **`public`** *mirheo*::*PostprocessPlugin*
Postprocessing side of *AnchorParticlesPlugin*.

Reports the forces required to achieve the constrains in a csv file.

Public Functions

`AnchorParticlesStatsPlugin`(std::string *name*, std::string *path*)
Create a *AnchorParticlesStatsPlugin* object.

Parameters

- `name`: The name of the plugin.
- `path`: The directory to which the stats will be dumped. Will create a single file `path/<pv_name>.csv`.

void **`deserialize`**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **`setup`**(**`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *PostprocessPlugin*.

This method must be called before any other function call.

Parameters

- `comm`: Contains all postprocess ranks
- `interComm`: used to communicate with the simulation ranks

void **handshake**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

class BerendsenThermostatPlugin : **public** *mirheo*::*SimulationPlugin*
Apply Berendsen thermostat to the given particles.

Public Functions

BerendsenThermostatPlugin(**const** *MirState* **state*, std::string *name*, std::vector<std::string> *pvNames*, real *kBT*, real *tau*, bool *increaseIfLower*)
Create a *BerendsenThermostatPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: The list of names of the concerned *ParticleVector* s.
- `kBT`: The target temperature, in energy units.
- `tau`: The relaxation time.
- `increaseIfLower`: Whether to increase the temperature if it's lower than the target temperature.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **afterIntegration**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class DensityControlPlugin : **public** *mirheo*::*SimulationPlugin*
Apply forces on particles in order to keep the number density constant within layers in a field.

The layers are determined by the level sets of the field. Forces are perpendicular to these layers; their magnitude is computed from PID controllers.

Cannot be used with multiple invocations of *`Mirheo.run`*.

Public Types

using RegionFunc = std::function<real (real3)>
functor that describes the region in terms of level sets.

Public Functions

DensityControlPlugin(**const** *MirState* **state*, std::string *name*, std::vector<std::string> *pvNames*, real *targetDensity*, *RegionFunc* *region*, real3 *resolution*, real *levelLo*, real *levelHi*, real *levelSpace*, real *Kp*, real *Ki*, real *Kd*, int *tuneEvery*, int *dumpEvery*, int *sampleEvery*)
Create a *DensityControlPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: The names of the *ParticleVector* that have the target density..
- `targetDensity`: The target number density.
- `region`: The field used to partition the space.
- `resolution`: The grid spacing used to discretized `region`
- `levelLo`: The minimum level set of the region to control.
- `levelHi`: The maximum level set of the region to control.
- `levelSpace`: Determines the difference between 2 consecutive layers in the partition of space.
- `Kp`: "Proportional" coefficient of the PID.
- `Ki`: "Integral" coefficient of the PID.
- `Kd`: "Derivative" coefficient of the PID.
- `tuneEvery`: Update th PID controllers every this number of steps.
- `dumpEvery`: Dump statistics every this number of steps. See also *PostprocessDensityControl*.
- `sampleEvery`: Sample statistics every this number of steps. Used by PIDs.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeForces**(cudaStream_t *stream*)
hook before computing the forces and after the cell lists are created

void **serializeAndSend**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

void **checkpoint**(MPI_Comm *comm*, **const** std::string &*path*, int *checkPointId*)
Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

void **restart**(MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

struct LevelBounds
Helper structure to partition the space.

Public Members

real **lo**
Smallest level set.

real **hi**
Largest level set.

real **space**
Difference between two level sets.

class ExternalMagneticTorquePlugin : **public** *mirheo*::*SimulationPlugin*
Apply a magnetic torque on given a *RigidObjectVector*.

Public Types

using UniformMagneticFunc = std::function<real3(real)>
Time varying uniform field.

Public Functions

ExternalMagneticTorquePlugin(**const** *MirState* *state, std::string *name*, std::string *rovName*, real3 *moment*, *UniformMagneticFunc* *magneticFunction*)
Create a *ExternalMagneticTorquePlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `rovName`: The name of the *RigidObjectVector* to apply the torque to.
- `moment`: The constant magnetic moment of one object, in its frame of reference.
- `magneticFunction`: The external uniform magnetic field which possibly varies in time.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeForces**(cudaStream_t *stream*)
hook before computing the forces and after the cell lists are created

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class PostprocessDensityControl : **public** *mirheo*::*PostprocessPlugin*
Postprocessing side of *DensityControlPlugin*.

Dumps the density and force in each layer of the space partition.

Public Functions

PostprocessDensityControl(std::string *name*, std::string *filename*)
Create a *PostprocessDensityControl* object.

Parameters

- `name`: The name of the plugin.
- `filename`: The txt file that will contain the density and corresponding force magnitudes in each layer.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

class ParticleDisplacementPlugin : **public** *mirheo*::*SimulationPlugin*

Compute the dispacement of particles between a given number of time steps.

Public Functions

ParticleDisplacementPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, int *updateEvery*)

Create a *ParticleDisplacementPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the concerned *ParticleVector*.
- `updateEvery`: The number of steps between two steps used to compute the displacement.

void **afterIntegration**(cudaStream_t *stream*)

hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)

setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class ExchangePVSFluxPlanePlugin : **public** *mirheo*::*SimulationPlugin*

Transfer particles from one *ParticleVector* to another when they cross a given plane.

Public Functions

ExchangePVSFluxPlanePlugin(**const** *MirState* **state*, std::string *name*, std::string *pv1Name*, std::string *pv2Name*, real4 *plane*)

Create a *ExchangePVSFluxPlanePlugin* object.

The particle has crossed the plane if a *x + b * y + c * z + d goes from negative to positive.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pv1Name`: The name of the source *ParticleVector*. Only particles from this *ParticleVector* are transfered.

- `pv2Name`: The name of the destination *ParticleVector*.
- `plane`: Coefficients of the plane to be crossed, (a, b, c, d).

void **`setup`**(*Simulation* **simulation*, **`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`beforeCellLists`**(cudaStream_t *stream*)
hook before building the cell lists

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

`class ForceSaverPlugin` : **`public`** *mirheo*::*SimulationPlugin*
Copies the forces of a given *ParticleVector* to a new channel at every time step.

This allows to dump the forces since they are reset to zero at every time step.

Public Functions

`ForceSaverPlugin`(**`const`** *MirState* **state*, std::string *name*, std::string *pvName*)
Create a *ForceSaverPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to save forces from and to.

void **`setup`**(*Simulation* **simulation*, **`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

void **beforeIntegration**(cudaStream_t *stream*)
hook before integrating the particle vectors but after the forces are computed

class ImposeProfilePlugin : **public** *mirheo*::*SimulationPlugin*
Set the velocity to a given one in a box.

The velocity is set to a constant plus a random velocity that has a Maxwell distribution.

Public Functions

ImposeProfilePlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, real3 *low*, real3 *high*, real3 *targetVel*, real *kBT*)
Create a *ImposeProfilePlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to modify.
- `low`: Lower coordinates of the region of interest.
- `high`: Upper coordinates of the region of interest.
- `targetVel`: The constant part of the new velocity.
- `kBT`: Temperature used to draw the velocity from the maxwellian distribution.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

void **afterIntegration**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

class ImposeVelocityPlugin : **public** *mirheo*::*SimulationPlugin*
Add a constant to the velocity of particles in a given region such that it matches a given average.

Public Functions

ImposeVelocityPlugin(**const** *MirState* **state*, std::string *name*, std::vector<std::string> *pvNames*, real3 *low*, real3 *high*, real3 *targetVel*, int *every*)
Create a *ImposeVelocityPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: The name of the (list of) *ParticleVector* to modify.
- `low`: Lower coordinates of the region of interest.
- `high`: Upper coordinates of the region of interest.
- `targetVel`: The target average velocity in the region.
- `every`: Correct the velocity every this number of time steps.

void **`setup`**(*Simulation* **simulation*, **`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`afterIntegration`**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

void **`setTargetVelocity`**(real3 *v*)
Change the target velocity to a new value.

Parameters

- `v`: The new target velocity.

`class MagneticDipoleInteractionsPlugin` : **`public`** *mirheo*::*SimulationPlugin*
Compute the magnetic dipole-dipole forces and torques induced by the interactions between rigid objects that have a magnetic moment.

Public Functions

`MagneticDipoleInteractionsPlugin`(**`const`** *MirState* **state*, std::string *name*, std::string *rovName*, real3 *moment*, real *mu0*)
Create a *MagneticDipoleInteractionsPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `rovName`: The name of the *RigidObjectVector* interacting.

- `moment`: The constant magnetic moment of one object, in its frame of reference.
- `mu0`: The magnetic permeability of the medium.

void **`setup`**(*Simulation* **simulation*, **`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`beforeCellLists`**(cudaStream_t *stream*)
hook before building the cell lists

void **`beforeForces`**(cudaStream_t *stream*)
hook before computing the forces and after the cell lists are created

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

`class OutletPlugin` : **`public`** *mirheo*::*SimulationPlugin*
Base class for outlet Plugins.

Outlet plugins delete particles of given a *ParticleVector* list in a region.

Subclassed by *mirheo::PlaneOutletPlugin*, *mirheo::RegionOutletPlugin*

Public Functions

`OutletPlugin`(**`const`** *MirState* **state*, std::string *name*, std::vector<std::string> *pvNames*)
Create a *OutletPlugin*.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: List of names of the *ParticleVector* that the outlet will be applied to.

void **`setup`**(*Simulation* **simulation*, **`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class PlaneOutletPlugin : **public** *mirheo*::*OutletPlugin*

Delete all particles that cross a given plane.

Public Functions

PlaneOutletPlugin(**const** *MirState* **state*, std::string *name*, std::vector<std::string> *pvNames*, real4 *plane*)

Create a *PlaneOutletPlugin*.

A particle crosses the plane if a*x + b*y + c*z + d goes from nbegative to postive accross one time step.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: List of names of the *ParticleVector* that the outlet will be applied to.
- `plane`: Coefficients (a, b, c, d) of the plane.

void **beforeCellLists**(cudaStream_t *stream*)

hook before building the cell lists

class RegionOutletPlugin : **public** *mirheo*::*OutletPlugin*

Delete all particles in a given region, defined implicitly by a field.

A particle is considered inside the region if the given field is negative at the particle's position.

Subclassed by *mirheo::DensityOutletPlugin*, *mirheo::RateOutletPlugin*

Public Types

using RegionFunc = std::function<real(real3)>

A scalar field to represent inside (negative) / outside (positive) region.

Public Functions

RegionOutletPlugin(**const** *MirState* **state*, std::string *name*, std::vector<std::string> *pvNames*, *RegionFunc* *region*, real3 *resolution*)

Create a *RegionOutletPlugin*.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: List of names of the *ParticleVector* that the outlet will be applied to.
- `region`: The field that describes the region. This will be sampled on a uniform grid and uploaded to the GPU.
- `resolution`: The grid space used to discretize `region`.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class DensityOutletPlugin : **public** *mirheo*::*RegionOutletPlugin*
Delete particles located in a given region if the number density is higher than a target one.

Public Functions

DensityOutletPlugin(**const** *MirState* *state, std::string *name*, std::vector<std::string> *pvNames*, real *numberDensity*, RegionFunc *region*, real3 *resolution*)
Create a *DensityOutletPlugin*.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: List of names of the *ParticleVector* that the outlet will be applied to.
- `numberDensity`: The target number density.
- `region`: The field that describes the region. This will be sampled on a uniform grid and uploaded to the GPU.
- `resolution`: The grid space used to discretize `region`.

void **beforeCellLists**(cudaStream_t *stream*)
hook before building the cell lists

class RateOutletPlugin : **public** *mirheo*::*RegionOutletPlugin*
Delete particles located in a given region at a given rate.

Public Functions

RateOutletPlugin(**const** *MirState* *state, std::string *name*, std::vector<std::string> *pvNames*, real *rate*, RegionFunc *region*, real3 *resolution*)
Create a *RateOutletPlugin*.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.

- `pvNames`: List of names of the *ParticleVector* that the outlet will be applied to.
- `rate`: The rate of deletion of particles.
- `region`: The field that describes the region. This will be sampled on a uniform grid and uploaded to the GPU.
- `resolution`: The grid space used to discretize `region`.

void **`beforeCellLists`**(cudaStream_t *stream*)
hook before building the cell lists

`class ParticleChannelAveragerPlugin` : **`public`** *mirheo*::*SimulationPlugin*
Average over time a particle vector channel.

Public Functions

`ParticleChannelAveragerPlugin`(**`const`** *MirState* **state*, std::string *name*, std::string *pvName*, std::string *channelName*, std::string *averageName*, real *updateEvery*)
Create a *ParticleChannelAveragerPlugin*.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector*.
- `channelName`: The name of the channel to average. Will fail if it does not exist.
- `averageName`: The name of the new channel, that will contain the time-averaged quantity..
- `updateEvery`: Will reset the averaged channel every this number of steps. Must be positive.

void **`beforeIntegration`**(cudaStream_t *stream*)
hook before integrating the particle vectors but after the forces are computed

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

void **`setup`**(*Simulation* **simulation*, **`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

`class ParticleChannelSaverPlugin` : **`public`** *mirheo*::*SimulationPlugin*
Copies a given channel to another one that will "stick" to the particle vector.

This is useful to collect statistics on non permanent quantities (e.g. stresses).

Public Functions

ParticleChannelSaverPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, std::string *channelName*, std::string *savedName*)
Create a *ParticleChannelSaverPlugin*.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector*.
- `channelName`: The name of the channel to save at every time step. Will fail if it does not exist.
- `savedName`: The name of the new channel.

void **beforeIntegration**(cudaStream_t *stream*)
hook before integrating the particle vectors but after the forces are computed

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

class ParticleDragPlugin : **public** *mirheo*::*SimulationPlugin*
Apply a drag force proportional to the velocity of every particle in a *ParticleVector*.

Public Functions

ParticleDragPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, real *drag*)
Create a *ParticleDragPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to which the force should be applied.
- `drag`: The drag coefficient applied to each particle.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeForces**(cudaStream_t *stream*)
hook before computing the forces and after the cell lists are created

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class PinObjectPlugin : **public** *mirheo*::*SimulationPlugin*
Add constraints on objects of an *ObjectVector*.

This modifies the velocities and forces on the objects in order to satisfy the given constraints on linear and angular velocities.

This plugin also collects statistics on the required forces and torques used to maintain the constraints. This may be useful to e.g. measure the drag around objects. See ReportPinObjectPlugin.

Public Functions

PinObjectPlugin(**const** *MirState* *state, std::string *name*, std::string *ovName*, real3 *translation*, real3 *rotation*, int *reportEvery*)
Create a *PinObjectPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `ovName`: The name of the *ObjectVector* that will be subjected of the constraints.
- `translation`: The target linear velocity. Components set to Unrestricted will not be constrained.
- `rotation`: The target angular velocity. Components set to Unrestricted will not be constrained.
- `reportEvery`: Send forces and torques stats to the postprocess side every this number of time steps.

void **setup**(*Simulation* *simulation, **const** MPI_Comm &comm, **const** MPI_Comm &interComm)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.

- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`beforeIntegration`**(cudaStream_t *stream*)
hook before integrating the particle vectors but after the forces are computed

void **`afterIntegration`**(cudaStream_t *stream*)
hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **`serializeAndSend`**(cudaStream_t *stream*)
Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

void **`handshake`**()
Used to communicate initial information between a *SimulationPlugin* and a *PostprocessPlugin*.

Does not do anything by default.

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

Public Static Attributes

`constexpr` real **`Unrestricted`** = std::numeric_limits<real>::infinity()
Special value reserved to represent unrestricted components.

`class PinRodExtremityPlugin` : **`public`** *mirheo*::*SimulationPlugin*
Add alignment force on a rod segment.

Public Functions

`PinRodExtremityPlugin`(**`const`** *MirState* **state*, std::string *name*, std::string *rvName*, int *segmentId*, real *fmagn*, real3 *targetDirection*)
Create a *PinRodExtremityPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `rvName`: The name of the *RodVector* to which the force should be applied.
- `segmentId`: The segment that will be constrained.
- `fmagn`: The force coefficient.
- `targetDirection`: The target direction.

void **`setup`**(*Simulation* **simulation*, **`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`beforeIntegration`**(cudaStream_t *stream*)
hook before integrating the particle vectors but after the forces are computed

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

`class TemperaturizePlugin` : **`public`** *mirheo*::*SimulationPlugin*
Add or set maxwellian drawn velocities to the particles of a given *ParticleVector*.

Public Functions

`TemperaturizePlugin`(**`const`** *MirState* **state*, std::string *name*, std::string *pvName*, real *kBT*, bool *keepVelocity*)
Create a *TemperaturizePlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to modify.
- `kBT`: Target temperature.
- `keepVelocity`: Wether to add or reset the velocities.

void **`setup`**(*Simulation* **simulation*, **`const`** MPI_Comm &*comm*, **`const`** MPI_Comm &*interComm*)
setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **`beforeForces`**(cudaStream_t *stream*)
hook before computing the forces and after the cell lists are created

bool **`needPostproc`**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class SimulationVelocityControl : **public** *mirheo*::*SimulationPlugin*

Apply a force in a box region to all particles.

The force is controled by a PID controller that has a target mean velocity in that same region.

Public Functions

SimulationVelocityControl(**const** *MirState* **state*, std::string *name*, std::vector<std::string> *pvNames*, real3 *low*, real3 *high*, int *sampleEvery*, int *tuneEvery*, int *dumpEvery*, real3 *targetVel*, real *Kp*, real *Ki*, real *Kd*)

Create a *SimulationVelocityControl* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvNames`: The list of names of the *ParticleVector* to control.
- `low`: The lower coordinates of the control region.
- `high`: The upper coordinates of the control region.
- `sampleEvery`: Sample the velocity average every this number of steps.
- `tuneEvery`: Update the PID controller every this number of steps.
- `dumpEvery`: Send statistics of the PID to the postprocess plugin every this number of steps.
- `targetVel`: The target mean velocity in the region of interest.
- `Kp`: "Proportional" coefficient of the PID.
- `Ki`: "Integral" coefficient of the PID.
- `Kd`: "Derivative" coefficient of the PID.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)

setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeForces**(cudaStream_t *stream*)

hook before computing the forces and after the cell lists are created

void **afterIntegration**(cudaStream_t *stream*)

hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

void **serializeAndSend**(cudaStream_t *stream*)

Pack and send data to the postprocess rank.

Happens between *beforeForces()* and *beforeIntegration()*.

Note This may happens while computing the forces.

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

void **checkpoint**(MPI_Comm *comm*, **const** std::string &*path*, int *checkPointId*)
Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

void **restart**(MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

class PostprocessVelocityControl : **public** *mirheo*::*PostprocessPlugin*
Postprocess side of *SimulationVelocityControl*.

Receives and dump the PID stats to a csv file.

Public Functions

PostprocessVelocityControl(std::string *name*, std::string *filename*)
Create a *SimulationVelocityControl* object.

Parameters

- `name`: The name of the plugin.
- `filename`: The csv file to which the statistics will be dumped.

void **deserialize**()
Perform the action implemented by the plugin using the data received from the *SimulationPlugin*.

void **checkpoint**(MPI_Comm *comm*, **const** std::string &*path*, int *checkPointId*)
Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

void **restart**(MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

class VelocityInletPlugin : **public** *mirheo*::*SimulationPlugin*

Add particles to a given *ParticleVector*.

The particles are injected on a given surface at a given influx rate.

Public Types

using ImplicitSurfaceFunc = std::function<real (real3)>

Representation of a surface from a scalar field.

using VelocityFieldFunc = std::function<real3 (real3)>

Velocity field used to describe the inflow.

Public Functions

VelocityInletPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, *ImplicitSurfaceFunc* *implicitSurface*, *VelocityFieldFunc* *velocityField*, real3 *resolution*, real *numberDensity*, real *kBT*)

Create a *VelocityInletPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* to add the particles to.
- `implicitSurface`: The scalar field that has the desired surface as zero level set.
- `velocityField`: The inflow velocity. Only relevant on the surface.
- `resolution`: Grid size used to sample the fields.
- `numberDensity`: The target number density of injection.
- `kBT`: The temperature of the injected particles.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)

setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeCellLists**(cudaStream_t *stream*)

hook before building the cell lists

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

class WallRepulsionPlugin : **public** *mirheo*::*SimulationPlugin*

Add a force that pushes particles away from the wall surfaces.

The magnitude of the force decreases linearly down to zero at a given distance h. Furthermore, the force can be capped.

Public Functions

WallRepulsionPlugin(**const** *MirState* **state*, std::string *name*, std::string *pvName*, std::string *wallName*, real *C*, real *h*, real *maxForce*)

Create a *WallRepulsionPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `pvName`: The name of the *ParticleVector* that will be subject to the force.
- `wallName`: The name of the *Wall*.
- `C`: *Force* coefficient.
- `h`: *Force* maximum distance.
- `maxForce`: Maximum force magnitude.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)

setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeIntegration**(cudaStream_t *stream*)

hook before integrating the particle vectors but after the forces are computed

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

Debugging plugins

class ParticleCheckerPlugin : **public** *mirheo*::*SimulationPlugin*

Check the validity of all *ParticleVector* in the simulation:

- Check that the positions are within reasonable bounds.
- Check that the velocities are within reasonable bounds.
- Check that forces do not contain NaN or Inf values.

If either of the above is not satisfied, the plugin will make the code die with an informative error.

Public Types

enum Info

Encode error type if there is any.

Values:

Ok

Out

Nan

Public Functions

ParticleCheckerPlugin(**const** *MirState* **state*, std::string *name*, int *checkEvery*)

Create a *ParticleCheckerPlugin* object.

Parameters

- `state`: The global state of the simulation.
- `name`: The name of the plugin.
- `checkEvery`: Will check the states of particles every this number of steps.

void **setup**(*Simulation* **simulation*, **const** MPI_Comm &*comm*, **const** MPI_Comm &*interComm*)

setup the internal state of the *SimulationPlugin*.

This method must be called before any of the hooks of the plugin. This is the place to fetch reference to objects held by the simulation.

Parameters

- `simulation`: The simulation to which the plugin is registered.
- `comm`: Contains all simulation ranks
- `interComm`: used to communicate with the postprocess ranks

void **beforeIntegration**(cudaStream_t *stream*)

hook before integrating the particle vectors but after the forces are computed

void **afterIntegration**(cudaStream_t *stream*)

hook after the *ObjectVector* objects are integrated but before redistribution and bounce back

bool **needPostproc**()

Return `true` if this plugin needs a postprocess side; `false` otherwise.

Note The plugin can have a postprocess side but not need it.

struct Status
Helper to encode problematic particles.

Public Members

int **id**
The Index of the potential problematic particle.

Info **info**
What is problematic.

Utils

Common helper classes and functions used by plugins.

template <typename *ControlType*>
class PidControl
PID controller class.

Template Parameters

- `ControlType`: The Datatype of the scalar to control.

Public Functions

PidControl(ControlType *initError*, real *Kp*, real *Ki*, real *Kd*)
Initialize the PID.

Parameters

- `initError`: The initial difference between the current state and the target.
- `Kp`: The proportional coefficient.
- `Ki`: The integral coefficient.
- `Kd`: The derivative coefficient.

ControlType **update**(ControlType *error*)
Update the internal state of the PID controller.

Return The control variable value.

Parameters

- `error`: The difference between the current state and the target.

Friends

std::ofstream &**operator<<** (std::ofstream &*stream*, **const** PidControl<ControlType> &*pid*)
Serialize a controller into a stream.

Return The stream.

Parameters

- `stream`: The stream that will contain the serialized data.
- `pid`: The current state to serialize.

std::ifstream &**operator>>** (std::ifstream &*stream*, PidControl<ControlType> &*pid*)
Deserialize a controller from a stream.

Return The stream.

Parameters

- `stream`: The stream that contains the serialized data.
- `pid`: The deserialized state.

void *mirheo*::**writeXYZ** (MPI_Comm *comm*, std::string *fname*, **const** real4 **positions*, int *np*)
Dump positions to a file in xyz format using MPI IO.

Parameters

- `comm`: The MPI communicator.
- `fname`: The name of the target file.
- `positions`: Array of positions xyz_.
- `np`: Local number of particles.

bool *mirheo*::**isTimeEvery** (**const** *MirState* **state*, int *dumpEvery*)
Check if a dump should occur at the current time step.

Return `true` if the current step is a dump time; `false` otherwise.

Parameters

- `state`: The current state of the simulation.
- `dumpEvery`: The number of steps between two dumps.

MirState::*StepType* *mirheo*::**getTimeStamp** (**const** *MirState* **state*, int *dumpEvery*)
Get the dump stamp from current time and dump frequency.

Return The dump stamp.

Parameters

- `state`: The current state of the simulation.
- `dumpEvery`: The number of steps between two dumps.

class SimpleSerializer

Helper class To serialize and deserialize data.

This is used to communicate data between simulation and postprocess plugins.

Only POD types and std::vectors/HostBuffers/PinnedBuffers of POD and std::strings are supported.

Public Static Functions

static int **totSize**()

Return The default total size of one element.

template <typename Arg**>**
static int **totSize**(**const** Arg &*arg*)

The total size of one element.

Return The size in bytes of the element.

Template Parameters

- `Arg`: The type of the element

Parameters

- `arg`: The element instance.

template <typename Arg, typename... *OthArgs***>**
static int **totSize**(**const** Arg &*arg*, **const** OthArgs&... *othArgs*)

The total size of one element.

Return The size in bytes of all elements.

Template Parameters

- `Arg`: The type of the element
- `OthArgs`: The types of the element other elements

Parameters

- `arg`: The element instance.
- `othArgs`: The other element instances.

template <typename... *Args*>
static void **serialize**(std::vector<char> &*buf*, **const** Args&... *args*)

Serialize multiple elements into a buffer.

The buffer will be allocated to the correct size.

Template Parameters

- `Args`: The types the elements to serialize.

Parameters

- `args`: The elements to serialize.
- `buf`: The buffer that will contain the serialized data.

template <typename... *Args*>

static void **deserialize**(**const** std::vector<char> &*buf*, Args&... *args*)
Deserialize multiple elements from a buffer.

Template Parameters

- `Args`: The types the elements to deserialize.

Parameters

- `args`: The deserialized elements.
- `buf`: The buffer that contains the serialized data.

template <typename... *Args*>
static void **serialize**(char **to*, **const** Args&... *args*)
Serialize multiple elements into a buffer.

The buffer will **NOT** be allocated to the correct size.

Template Parameters

- `Args`: The types the elements to serialize.

Parameters

- `args`: The elements to serialize.
- `to`: The buffer that will contain the serialized data. Must be sufficiently large.

template <typename... *Args*>
static void **deserialize**(**const** char **from*, Args&... *args*)
Deserialize multiple elements from a buffer.

Template Parameters

- `Args`: The types the elements to deserialize.

Parameters

- `args`: The deserialized elements.
- `from`: The buffer that contains the serialized data.

18.20 Postproc

class Postprocess : *mirheo*::*MirObject*
Manage post processing tasks (see `Plugin`) related to a `Simulation`.

There must be exactly one `Postprocess` rank per `Simulation` rank or no `Postprocess` rank at all. All `Plugin` objects must be registered and set before calling *init()* and *run()*. This can be instantiated on ranks that have no access to GPUs.

The *run()* method consists in waiting for messages incoming from the simulation ranks and execute the registered plugins functions with that data.

Public Functions

Postprocess(MPI_Comm &*comm*, MPI_Comm &*interComm*, **const** CheckpointInfo &*checkpointInfo*)
Construct a `Postprocess` object.

Parameters

- `comm`: a communicator that holds all postprocessing ranks.
- `interComm`: An inter communicator to communicate with the *`Simulation`* ranks.
- `checkpointInfo`: Checkpoint configuratoin.

void **`registerPlugin`**(std::shared_ptr<*PostprocessPlugin*> *plugin*, int *tag*)
Register a plugin to this object.

Note The *SimulationPlugin* counterpart of the registered *PostprocessPlugin* must be registered on the simulation side.

Parameters

- `plugin`: The plugin to register
- `tag`: a tag that is unique for each registered plugin

void **`deregisterPlugin`**(*PostprocessPlugin* **plugin*)
Deregister a postprocess plugin.

Note The *SimulationPlugin* counterpart of the deregistered *PostprocessPlugin* must also be deregistered. An exception is thrown if the plugin is not found.

Parameters

- `plugin`: The plugin to deregister

void **`init`**()
Setup all registered plugins. Must be called before *run()*

void **`run`**()
Start the postprocess. Will run until a termination notification is sent by the simulation.

void **`restart`**(**`const`** std::string &*folder*)
Restore the state from checkpoint information.

Parameters

- `folder`: The path containing the checkpoint files

void **`checkpoint`**(int *checkpointId*)
Dump the state of all postprocess plugins to the checkpoint folder.

Parameters

- `checkpointId`: The index of the dump, used to name the files.

18.21 Packers

Packers are used to store data of a set of registered channels from a *`mirheo::DataManager`* into a single buffer and vice-versa. They are used to redistribute and exchange data accross neighbouring ranks efficiently. This allows to send single MPI messages instead of one message per channel.

Generic Packer

This is the base packer class. All packers contain generic packers that are used to pack different kind of data (such as particle or object data).

struct GenericPackerHandler

A device-friendly structure that is used to pack and unpack multiple channels into a single buffer.

Additionally to being packed and unpacked, the data can be shifted. This facilitate the exchange and redistribute operations.

The packed channels are structured in a single buffer containing:

1. The first channel data
2. padding
3. The second channel data
4. padding
5. ...

Hence, the number of elements must be known in advance before packing. This is generally not a limitation, as memory must be allocated before packing.

Subclassed by *mirheo::GenericPacker*

Public Functions

size_t **pack**(int *srcId*, int *dstId*, char **dstBuffer*, int *numElements*) **const**

Fetch one datum from the registered channels and pack it into a buffer.

Return The size (in bytes) taken by the packed data (numElements elements)

Parameters

- `srcId`: Index of the datum to fetch from registered channel space (in number of elements).
- `dstId`: Index of the datum to store in dstBuffer space (in number of elements).
- `dstBuffer`: Destination buffer
- `numElements`: Total number of elements that will be packed in the buffer.

size_t **packShift**(int *srcId*, int *dstId*, char **dstBuffer*, int *numElements*, real3 *shift*) **const**

Fetch one datum from the registered channels, shift it (if applicable) and pack it into the buffer.

Only channels with active shift will be shifted.

Return The size (in bytes) taken by the packed data (numElements elements)

Parameters

- `srcId`: Index of the datum to fetch from registered channel space (in number of elements).
- `dstId`: Index of the datum to store in dstBuffer space (in number of elements).
- `dstBuffer`: Destination buffer
- `numElements`: Total number of elements that will be packed in the buffer.
- `shift`: The coordinate shift

size_t **unpack** (int *srcId*, int *dstId*, **const** char **srcBuffer*, int *numElements*) **const**
Unpack one datum from the buffer and store it in the registered channels.

Return The size (in bytes) taken by the packed data (numElements elements)

Parameters

- `srcId`: Index of the datum to fetch from the buffer (in number of elements).
- `dstId`: Index of the datum to store in the registered channels (in number of elements).
- `srcBuffer`: Source buffer that contains packed data.
- `numElements`: Total number of elements that are packed in the buffer.

size_t **unpackAtomicAddNonZero** (int *srcId*, int *dstId*, **const** char **srcBuffer*, int *numElements*) **const**
Unpack one datum from the buffer and add it to the registered channels atomically.

Return The size (in bytes) taken by the packed data (numElements elements)

Parameters

- `srcId`: Index of the datum to fetch from the buffer (in number of elements).
- `dstId`: Index of the datum to add to the registered channels (in number of elements).
- `srcBuffer`: Source buffer that contains packed data.
- `numElements`: Total number of elements that are packed in the buffer.

size_t **unpackShift** (int *srcId*, int *dstId*, **const** char **srcBuffer*, int *numElements*, real3 *shift*) **const**
Unpack and shift one datum from the buffer and store it in the registered channels.

Return The size (in bytes) taken by the packed data (numElements elements)

Parameters

- `srcId`: Index of the datum to fetch from the buffer (in number of elements).
- `dstId`: Index of the datum to store into the registered channels (in number of elements).
- `srcBuffer`: Source buffer that contains packed data.
- `numElements`: Total number of elements that are packed in the buffer.
- `shift`: The coordinate shift

void **copyTo** (*GenericPackerHandler* &*dst*, int *srcId*, int *dstId*) **const**
Copy one datum from the registered channels to the registered channels of another *GenericPackerHandler*.

Parameters

- `dst`: The other *GenericPackerHandler* that will receive the new datum.
- `srcId`: Index of the datum to fetch from the registered channels (in number of elements).
- `dstId`: Index of the datum to store into the dst registered channels (in number of elements).

size_t **getSizeBytes**(int *numElements*) **const**
Get the size (in bytes) of the buffer that can hold the packed data of numElements elements from all registered channels.

This must be used to allocate the buffer size. Because of padding, the size is not simply the sum of sizes of all elements.

Return The size (in bytes) of the buffer.

Parameters

- `numElements`: The number of elements that the buffer must contain once packed.

Public Static Attributes

constexpr size_t **alignment** = getPaddedSize<char>(1)
Alignment sufficient for all types used in channels.

Useful for external codes that operate with *Mirheo*'s packing functions.

class GenericPacker : *mirheo*::*GenericPackerHandler*
This class is used to construct *GenericPackerHandler*, to be passed to the device.

Public Functions

void **updateChannels**(*DataManager* *&dataManager*, PackPredicate *&predicate*, cudaStream_t *stream*)
Register all channels of a *DataManager* satisfying a predicate.

All previously registered channels will be removed before adding those described above.

Parameters

- `dataManager`: The object that contains the channels to register
- `predicate`: The filter (white list) that is used to select the channels to register, based on their description and names
- `stream`: The stream used to transfer the data on the device

GenericPackerHandler &**handler**()
Get a handler that can be used on the device.

size_t **getSizeBytes**(int *numElements*) **const**
see *GenericPackerHandler::getSizeBytes()*.

Particles Packer

struct ParticlePackerHandler
A packer specific to particle data only.

The user can use the internal generic packer directly.

Subclassed by *mirheo::ObjectPackerHandler*

Public Functions

size_t **getSizeBytes**(int *numElements*) **const**
Get the reuired size (in bytes) of the buffer to hold the packed data.

Public Members

GenericPackerHandler **particles**
The packer responsible for the particles.

class ParticlePacker
Helper class to construct a *ParticlePackerHandler*.

Subclassed by *mirheo::ObjectPacker*

Public Functions

ParticlePacker(PackPredicate *predicate*)
Construct a *ParticlePacker*.

Parameters

- `predicate`: The channel filter that will be used to select the channels to be registered.

virtual void **update**(*LocalParticleVector* **lpv*, cudaStream_t *stream*)
Register the channels of a *LocalParticleVector* that meet the predicate requirements.

Parameters

- `lpv`: The *LocalParticleVector* that holds the channels to be registered.
- `stream`: The stream used to transfer the channels information to the device.

ParticlePackerHandler **handler**()
get a handler usable on device

virtual size_t **getSizeBytes**(int *numElements*) **const**
Get the reuired size (in bytes) of the buffer to hold all the packed data.

Objects Packer

struct ObjectPackerHandler : **public** *mirheo*::*ParticlePackerHandler*
A packer specific to objects.

Will store both particle and object data into a single buffer.

Subclassed by *mirheo::RodPackerHandler*

Public Functions

size_t **getSizeBytes**(int *numElements*) **const**
Get the reuired size (in bytes) of the buffer to hold the packed data.

__device__ size_t **blockPack**(int *numElements*, char **buffer*, int *srcObjId*, int *dstObjId*) **const**

Fetch a full object from the registered channels and pack it into the buffer.

This method must be called by one CUDA block per object.

Return The size (in bytes) taken by the packed data (numElements objects). Only relevant for thread with Id 0.

Parameters

- `numElements`: Number of objects that will be packed in the buffer.
- `buffer`: Destination buffer that will hold the packed object
- `srcObjId`: The index of the object to fetch from registered channels
- `dstObjId`: The index of the object to store into the buffer

__device__ size_t **blockPackShift**(int *numElements*, char **buffer*, int *srcObjId*, int *dstObjId*, real3 *shift*) **const**

Fetch a full object from the registered channels, shift it and pack it into the buffer.

This method must be called by one CUDA block per object.

Return The size (in bytes) taken by the packed data (numElements objects). Only relevant for thread with Id 0.

Parameters

- `numElements`: Number of objects that will be packed in the buffer.
- `buffer`: Destination buffer that will hold the packed object
- `srcObjId`: The index of the object to fetch from registered channels
- `dstObjId`: The index of the object to store into the buffer
- `shift`: The coordnate shift

__device__ size_t **blockUnpack**(int *numElements*, **const** char **buffer*, int *srcObjId*, int *dstObjId*) **const**

Unpack a full object from the buffer and store it into the registered channels.

This method must be called by one CUDA block per object.

Return The size (in bytes) taken by the packed data (numElements objects). Only relevant for thread with Id 0.

Parameters

- `numElements`: Number of objects that will be packed in the buffer.
- `buffer`: Buffer that holds the packed object
- `srcObjId`: The index of the object to fetch from the buffer
- `dstObjId`: The index of the object to store into the registered channels

__device__ size_t **blockUnpackAddNonZero**(int *numElements*, **const** char **buffer*, int *srcObjId*, int *dstObjId*) **const**

Unpack a full object from the buffer and add it to the registered channels.

This method must be called by one CUDA block per object.

Return The size (in bytes) taken by the packed data (numElements objects). Only relevant for thread with Id 0.

Parameters

- `numElements`: Number of objects that will be packed in the buffer.
- `buffer`: Buffer that holds the packed object
- `srcObjId`: The index of the object to fetch from the buffer
- `dstObjId`: The index of the object to store into the registered channels

__device__ size_t **`blockUnpackShift`** (int *numElements*, **`const`** char **buffer*, int *srcObjId*, int *dstObjId*, real3 *shift*) **`const`**

Unpack a full object from the buffer, shift it and store it into the registered channels.

This method must be called by one CUDA block per object.

Return The size (in bytes) taken by the packed data (numElements objects). Only relevant for thread with Id 0.

Parameters

- `numElements`: Number of objects that will be packed in the buffer.
- `buffer`: Buffer that holds the packed object
- `srcObjId`: The index of the object to fetch from the buffer
- `dstObjId`: The index of the object to store into the registered channels
- `shift`: Coordinates shift

__device__ void **`blockCopyParticlesTo`** (*ParticlePackerHandler* &*dst*, int *srcObjId*, int *dstPartIdOffset*) **`const`**

Copy the particle data of a full object from registered channels into the registered channels of a *ParticlePackerHandler*.

This method must be called by one CUDA block per object.

Parameters

- `dst`: The destination *ParticlePackerHandler*
- `srcObjId`: The index of the object to fetch from the registered channels
- `dstPartIdOffset`: The index of the first particle in the destination *ParticlePackerHandler*

__device__ void **`blockCopyTo`** (*ObjectPackerHandler* &*dst*, int *srcObjId*, int *dstObjId*) **`const`**

Copy a full object from registered channels into the registered channels of a *ObjectPackerHandler*.

This method must be called by one CUDA block per object.

Parameters

- `dst`: The destination *ObjectPackerHandler*
- `srcObjId`: The index of the object to fetch from the registered channels
- `dstObjId`: The index of the object to store in the destination *ObjectPackerHandler*

Public Members

int **`objSize`**

number of particles per object

GenericPackerHandler **`objects`**

packer responsible for the object data

class ObjectPacker : **public** *mirheo*::*ParticlePacker*

Helper class to construct a *ObjectPackerHandler*.

Subclassed by *mirheo::RodPacker*

Public Functions

ObjectPacker(PackPredicate *predicate*)

Construct a *ObjectPacker*.

Parameters

- `predicate`: The channel filter that will be used to select the channels to be registered.

void **update**(*LocalParticleVector* **lpv*, cudaStream_t *stream*)

Register the channels of a *LocalParticleVector* that meet the predicate requirements.

Parameters

- `lpv`: The *LocalParticleVector* that holds the channels to be registered.
- `stream`: The stream used to transfer the channels information to the device.

ObjectPackerHandler **handler**()

get a handler usable on device

size_t **getSizeBytes**(int *numElements*) **const**

Get the reuired size (in bytes) of the buffer to hold all the packed data.

Rods Packer

struct RodPackerHandler : **public** *mirheo*::*ObjectPackerHandler*

A packer specific to rods.

Will store particle, object and bisegment data into a single buffer.

Public Functions

size_t **getSizeBytes**(int *numElements*) **const**

Get the reuired size (in bytes) of the buffer to hold the packed data.

__device__ size_t **blockPack**(int *numElements*, char **buffer*, int *srcObjId*, int *dstObjId*) **const**

Fetch a full rod from the registered channels and pack it into the buffer.

This method must be called by one CUDA block per object.

Return The size (in bytes) taken by the packed data (numElements rods). Only relevant for thread with Id 0.

Parameters

- `numElements`: Number of rods that will be packed in the buffer.
- `buffer`: Destination buffer that will hold the packed rod
- `srcObjId`: The index of the rod to fetch from registered channels

- `dstObjId`: The index of the rod to store into the buffer

__device__ size_t **blockPackShift**(int *numElements*, char **buffer*, int *srcObjId*, int *dstObjId*, real3 *shift*) **const**

Fetch a full rod from the registered channels, shift it and pack it into the buffer.

This method must be called by one CUDA block per object.

Return The size (in bytes) taken by the packed data (numElements rods). Only relevant for thread with Id 0.

Parameters

- `numElements`: Number of rods that will be packed in the buffer.
- `buffer`: Destination buffer that will hold the packed rod
- `srcObjId`: The index of the rod to fetch from registered channels
- `dstObjId`: The index of the rod to store into the buffer
- `shift`: The coordnate shift

__device__ size_t **blockUnpack**(int *numElements*, **const** char **buffer*, int *srcObjId*, int *dstObjId*) **const**

Unpack a full rod from the buffer and store it into the registered channels.

This method must be called by one CUDA block per object.

Return The size (in bytes) taken by the packed data (numElements objects). Only relevant for thread with Id 0.

Parameters

- `numElements`: Number of rods that will be packed in the buffer.
- `buffer`: Buffer that holds the packed rod
- `srcObjId`: The index of the rod to fetch from the buffer
- `dstObjId`: The index of the rod to store into the registered channels

__device__ size_t **blockUnpackAddNonZero**(int *numElements*, **const** char **buffer*, int *srcObjId*, int *dstObjId*) **const**

Unpack a full rod from the buffer and add it into the registered channels.

This method must be called by one CUDA block per object.

Return The size (in bytes) taken by the packed data (numElements objects). Only relevant for thread with Id 0.

Parameters

- `numElements`: Number of rods that will be packed in the buffer.
- `buffer`: Buffer that holds the packed rod
- `srcObjId`: The index of the rod to fetch from the buffer
- `dstObjId`: The index of the rod to store into the registered channels

Public Members

int **nBisegments**

number of bisegment per rod

GenericPackerHandler **bisegments**
packer responsible for the bisegment data

class RodPacker : **public** *mirheo*::*ObjectPacker*
Helper class to construct a *RodPackerHandler*.

Public Functions

RodPacker(PackPredicate *predicate*)
Construct a *RodPacker*.

Parameters

- `predicate`: The channel filter that will be used to select the channels to be registered.

void **update**(*LocalParticleVector* *lpv, cudaStream_t *stream*)
Register the channels of a *LocalParticleVector* that meet the predicate requirements.

Parameters

- `lpv`: The *LocalParticleVector* that holds the channels to be registered.
- `stream`: The stream used to transfer the channels information to the device.

RodPackerHandler **handler**()
get a handler usable on device

size_t **getSizeBytes**(int *numElements*) **const**
Get the reuired size (in bytes) of the buffer to hold all the packed data.

18.22 Particle Vectors

See also *the user interface*.

Particle Vectors

class ParticleVector : **public** *mirheo*::*MirSimulationObject*
Base particles container.

Holds two *LocalParticleVector*: local and halo. The local one contains the data present in the current subdomain. The halo one is used to exchange particle data with the neighboring ranks.

By default, contains positions, velocities, forces and global ids.

Subclassed by *mirheo::ObjectVector*

Unnamed Group

void **setCoordinates_vector**(**const** std::vector<real3> &*coordinates*)
Python getters / setters Use default blocking stream.

Public Functions

ParticleVector(**const** *MirState* *state, **const** std::string &name, real mass, int n = 0)
Construct a *ParticleVector*.

Parameters

- `state`: The simulation state
- `name`: Name of the pv
- `mass`: Mass of one particle
- `n`: Number of particles

LocalParticleVector ***local**()
get the local *LocalParticleVector*

LocalParticleVector ***halo**()
get the halo *LocalParticleVector*

LocalParticleVector ***get**(ParticleVectorLocality *locality*)
get the *LocalParticleVector* corresponding to a given locality

Parameters

- `locality`: local or halo

const *LocalParticleVector* ***local**() **const**
get the local *LocalParticleVector*

const *LocalParticleVector* ***halo**() **const**
get the halo *LocalParticleVector*

void **checkpoint**(MPI_Comm *comm*, **const** std::string &*path*, int *checkPointId*)
Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

void **restart**(MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

template <**typename** T>
void **requireDataPerParticle**(**const** std::string &*name*, *DataManager*::PersistenceMode *persistence*, *DataManager*::ShiftMode *shift* = *DataManager*::ShiftMode::None)
Add a new channel to hold additional data per particle.

Template Parameters

- `T`: The type of data to add

Parameters

- `name`: channel name
- `persistence`: If the data should stich to the particles or not when exchanged
- `shift`: If the data needs to be shifted when exchanged

real **getMassPerParticle**() **const**
get the particle mass

Public Members

bool **haloValid** = {false}
true if the halo is up to date

bool **redistValid** = {false}
true if the particles are redistributed

int **cellListStamp** = {0}
stamp that keep track if the cell list is up to date

class ObjectVector : **public** *mirheo*::*ParticleVector*
Base objects container.

Holds two *LocalObjectVector*: local and halo.

Subclassed by mirheo::ChainVector, *mirheo::MembraneVector*, *mirheo::RigidObjectVector*, *mirheo::RodVector*

Public Functions

ObjectVector(**const** *MirState* **state*, **const** std::string &*name*, real *mass*, int *objSize*, int *nObjects* = 0)
Construct a *ObjectVector*.

Parameters

- `state`: The simulation state
- `name`: Name of the pv
- `mass`: Mass of one particle
- `objSize`: Number of particles per object
- `nObjects`: Number of objects

void **findExtentAndCOM**(cudaStream_t *stream*, ParticleVectorLocality *locality*)
Compute Extents and center of mass of each object in the given *LocalObjectVector*.

Parameters

- `stream`: The stream to execute the kernel on.
- `locality`: Specify which *LocalObjectVector* to compute the data

LocalObjectVector ***local**()
get local *LocalObjectVector*

LocalObjectVector ***halo**()
get halo *LocalObjectVector*

LocalObjectVector ***get**(ParticleVectorLocality *locality*)
get *LocalObjectVector* from locality

void **checkpoint**(MPI_Comm *comm*, **const** std::string &*path*, int *checkPointId*)
Save the state of the object on disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.
- `checkPointId`: The id of the dump.

void **restart**(MPI_Comm *comm*, **const** std::string &*path*)
Load the state of the object from the disk.

Parameters

- `comm`: MPI communicator to perform the I/O.
- `path`: The directory path to store the object state.

template <typename T>
void **requireDataPerObject**(**const** std::string &*name*, *DataManager*::PersistenceMode *persistence*, *DataManager*::ShiftMode *shift* = *DataManager*::ShiftMode::None)
Add a new channel to hold additional data per object.

Template Parameters

- `T`: The type of data to add

Parameters

- `name`: channel name
- `persistence`: If the data should stich to the objects or not when exchanging
- `shift`: If the data needs to be shifted when exchanged

int **getObjectSize**() **const**
get number of particles per object

Public Members

std::shared_ptr<*Mesh*> **mesh**
Triangle mesh that can be used to represent the object surface.

class RigidObjectVector : **public** *mirheo*::*ObjectVector*
Rigid objects container.

Holds two *LocalRigidObjectVector*: local and halo.

Subclassed by *mirheo::RigidShapedObjectVector< Shape >*

Public Functions

RigidObjectVector(**const** *MirState* *state, **const** std::string &name, real *partMass*, real3 *J*, **const** int *objSize*, std::shared_ptr<*Mesh*> *mesh*, **const** int *nObjects* = 0)
Construct a *RigidObjectVector*.

Parameters

- `state`: The simulation state
- `name`: Name of the pv
- `partMass`: Mass of one frozen particle
- `J`: Diagonal entries of the inertia tensor, which must be diagonal.
- `objSize`: Number of particles per object
- `mesh`: *Mesh* representing the surface of the object.
- `nObjects`: Number of objects

LocalRigidObjectVector ***local**()
get local *LocalRigidObjectVector*

LocalRigidObjectVector ***halo**()
get halo *LocalRigidObjectVector*

LocalRigidObjectVector ***get**(ParticleVectorLocality *locality*)
get *LocalRigidObjectVector* from locality

real3 **getInertialTensor**() **const**
get diagonal entries of the inertia tensor

Public Members

PinnedBuffer<real4> **initialPositions**
Coordinates of the frozen particles in the frame of reference of the object.

template <class *Shape*>
class RigidShapedObjectVector : **public** *mirheo*::*RigidObjectVector*
RigidObjectVector with analytic shape instead of triangle mesh.

Template Parameters

- `Shape`: The analytic shape that represents the object surface in its frame of reference

Public Functions

RigidShapedObjectVector(**const** *MirState* *state, **const** std::string &name, real *mass*, int *objSize*, Shape *shape*, int *nObjects* = 0)
Construct a *RigidShapedObjectVector*.

Parameters

- `state`: The simulation state
- `name`: Name of the pv

- `mass`: Mass of one frozen particle
- `objSize`: Number of particles per object
- `shape`: The shape that represents the surface of the object
- `nObjects`: Number of objects

RigidShapedObjectVector(**const** *MirState* **state*, **const** std::string &*name*, real *mass*, int *objSize*, Shape *shape*, std::shared_ptr<*Mesh*> *mesh*, int *nObjects* = 0)
Construct a *RigidShapedObjectVector*.

Note: The mesh is used only for visualization purpose

Parameters

- `state`: The simulation state
- `name`: Name of the pv
- `mass`: Mass of one frozen particle
- `objSize`: Number of particles per object
- `shape`: The shape that represents the surface of the object
- `mesh`: The mesh that represents the surface, should not used in the simulation.
- `nObjects`: Number of objects

const Shape &**getShape**() **const**
get the handler that represent the shape of the objects

class RodVector : **public** *mirheo*::*ObjectVector*
Rod objects container.

Holds two *LocalRodVector*: local and halo.

Public Functions

RodVector(**const** *MirState* **state*, **const** std::string &*name*, real *mass*, int *nSegments*, int *nObjects* = 0)
Construct a *ObjectVector*.

Parameters

- `state`: The simulation state
- `name`: Name of the pv
- `mass`: Mass of one particle
- `nSegments`: Number of segments per rod
- `nObjects`: Number of rods

LocalRodVector ***local**()
get local *LocalRodVector*

LocalRodVector ***halo**()
get halo *LocalRodVector*

LocalRodVector ***get**(ParticleVectorLocality *locality*)
get *LocalRodVector* from locality

template <typename T>
void **requireDataPerBisegment**(**const** std::string &*name*, *DataManager*::PersistenceMode *persistence*, *DataManager*::ShiftMode *shift* = *DataManager*::ShiftMode::None)
Add a new channel to hold additional data per bisegment.

Template Parameters

- `T`: The type of data to add

Parameters

- `name`: channel name
- `persistence`: If the data should stich to the object or not when exchanging
- `shift`: If the data needs to be shifted when exchanged

class MembraneVector : **public** *mirheo*::*ObjectVector*
Represent a set of membranes.

Each membrane is composed of the same connectivity (stored in mesh) and number of vertices. The particles data correspond to the vertices of the membranes.

Public Functions

MembraneVector(**const** *MirState* **state*, **const** std::string &*name*, real *mass*, std::shared_ptr<*MembraneMesh*> *mptr*, int *nObjects* = 0)
Construct a *MembraneVector*.

Parameters

- `state`: The simulation state
- `name`: Name of the pv
- `mass`: Mass of one particle
- `mptr`: Triangle mesh which stores the connectivity of a single membrane
- `nObjects`: Number of objects

Local Particle Vectors

class LocalParticleVector
Particles container.

This is used to represent local or halo particles in *ParticleVector*.

Subclassed by *mirheo::LocalObjectVector*

Public Functions

LocalParticleVector(*ParticleVector* *pv, int np = 0)
Construct a *LocalParticleVector*.

Parameters

- pv: Pointer to the parent *ParticleVector*.
- np: Number of particles.

int **size**() **const**
return the number of particles

virtual void **resize**(int *n*, cudaStream_t *stream*)
resize the container, preserving the data.

Parameters

- n: new number of particles
- stream: that is used to copy data

virtual void **resize_anew**(int *n*)
resize the container, without preserving the data.

Parameters

- n: new number of particles

PinnedBuffer<*Force*> &**forces**()
get forces container reference

PinnedBuffer<real4> &**positions**()
get positions container reference

PinnedBuffer<real4> &**velocities**()
get velocities container reference

virtual void **computeGlobalIds**(MPI_Comm *comm*, cudaStream_t *stream*)
Set a unique Id for each particle in the simulation.

The ids are stored in the channel ChannelNames::globalIds.

Parameters

- comm: MPI communicator of the simulation
- stream: Stream used to transfer data between host and device

ParticleVector ***parent**()
get parent *ParticleVector*

const *ParticleVector* ***parent**() **const**
get parent *ParticleVector*

Public Members

DataManager **dataPerParticle**
Contains all particle channels.

Friends

void **swap**(LocalParticleVector&, LocalParticleVector&)
swap two *LocalParticleVector*

class LocalObjectVector : **public** *mirheo*::*LocalParticleVector*
Objects container.

This is used to represent local or halo objects in *ObjectVector*. An object is a chunk of particles, each chunk with the same number of particles within an *ObjectVector*. Additionally, data can be attached to each of those chunks.

Subclassed by *mirheo::LocalRigidObjectVector*, *mirheo::LocalRodVector*

Public Functions

LocalObjectVector(*ParticleVector* *pv, int *objSize*, int *nObjects* = 0)
Construct a *LocalParticleVector*.

Parameters

- `pv`: Parent *ObjectVector*
- `objSize`: Number of particles per object
- `nObjects`: Number of objects

void **resize**(int *n*, cudaStream_t *stream*)
resize the container, preserving the data.

Parameters

- `n`: new number of particles
- `stream`: that is used to copy data

void **resize_anew**(int *n*)
resize the container, without preserving the data.

Parameters

- `n`: new number of particles

void **computeGlobalIds**(MPI_Comm *comm*, cudaStream_t *stream*)
Set a unique Id for each particle in the simulation.

The ids are stored in the channel ChannelNames::globalIds.

Parameters

- `comm`: MPI communicator of the simulation
- `stream`: Stream used to transfer data between host and device

virtual *PinnedBuffer*<real4> ***getMeshVertices**(cudaStream_t *stream*)
get positions of the mesh vertices

virtual *PinnedBuffer*<real4> ***getOldMeshVertices**(cudaStream_t *stream*)
get positions of the old mesh vertices

virtual *PinnedBuffer*<*Force*> ***getMeshForces**(cudaStream_t *stream*)
get forces on the mesh vertices

int **getObjectSize**() **const**
get number of particles per object

int **getNumObjects**() **const**
get number of objects

Public Members

DataManager **dataPerObject**
contains object data

Friends

void **swap**(LocalObjectVector&, LocalObjectVector&)
swap two *LocalObjectVector*

class LocalRigidObjectVector : **public** *mirheo*::*LocalObjectVector*
Rigid objects container.

This is used to represent local or halo objects in *RigidObjectVector*. A rigid object is composed of frozen particles inside a volume that is represented by a triangle mesh. There is then two sets of particles: mesh vertices and frozen particles. The frozen particles are stored in the particle data manager, while the mesh particles are stored in additional buffers.

Additionally, each rigid object has a RigidMotion datum associated that fully describes its state.

Public Functions

LocalRigidObjectVector(*ParticleVector* **pv*, int *objSize*, int *nObjects* = 0)
Construct a *LocalRigidObjectVector*.

Parameters

- `pv`: Parent *RigidObjectVector*
- `objSize`: Number of frozen particles per object
- `nObjects`: Number of objects

PinnedBuffer<real4> ***getMeshVertices**(cudaStream_t *stream*)
get positions of the mesh vertices

PinnedBuffer<real4> ***getOldMeshVertices**(cudaStream_t *stream*)
get positions of the old mesh vertices

PinnedBuffer<*Force*> ***getMeshForces**(cudaStream_t *stream*)
get forces on the mesh vertices

void **clearRigidForces**(cudaStream_t *stream*)
set forces in rigid motions to zero

class LocalRodVector : **public** *mirheo*::*LocalObjectVector*
Rod container.

This is used to represent local or halo rods in *RodVector*. A rod is a chunk of particles connected implicitly in segments with additional 4 particles per edge. The number of particles per rod is then 5*n + 1 if n is the number of segments. Each object (called a rod) within a *LocalRodVector* has the same number of particles. Additionally to particle and object, data can be attached to each bisegment.

Public Functions

LocalRodVector(*ParticleVector* *pv, int *objSize*, int *nObjects* = 0)
Construct a *LocalRodVector*.

Parameters

- `pv`: Parent *RodVector*
- `objSize`: Number of particles per object
- `nObjects`: Number of objects

void **resize**(int *n*, cudaStream_t *stream*)
resize the container, preserving the data.

Parameters

- `n`: new number of particles
- `stream`: that is used to copy data

void **resize_anew**(int *n*)
resize the container, without preserving the data.

Parameters

- `n`: new number of particles

int **getNumSegmentsPerRod**() **const**
get the number of segment per rod

Public Members

DataManager **dataPerBisegment**
contains bisegment data

Views

struct PVview
GPU-compatible struct that contains particle basic data.

Contains particle positions, velocities, forces, and mass info.

Subclassed by *mirheo::OVview*, *mirheo::PVviewWithDensities*, *mirheo::PVviewWithOldParticles*

Public Types

using PVType = *ParticleVector*
Particle Vector compatible type.

using LPVType = *LocalParticleVector*
Local *Particle* Vector compatible type.

Public Functions

PVview(*ParticleVector* *pv, *LocalParticleVector* *lpv)
Construct a `PVview`.

Parameters

- `pv`: The *ParticleVector* that the view represents
- `lpv`: The *LocalParticleVector* that the view represents

real4 **readPosition**(int *id*) **const**
fetch position from given particle index

real4 **readVelocity**(int *id*) **const**
fetch velocity from given particle index

void **readPosition**(*Particle* &*p*, int *id*) **const**
fetch position from given particle index and store it into p

void **readVelocity**(*Particle* &*p*, int *id*) **const**
fetch velocity from given particle index and store it into p

Particle **readParticle**(int *id*) **const**
fetch particle from given particle index

real4 **readPositionNoCache**(int *id*) **const**
fetch position from given particle index without going through the L1/L2 cache This can be useful to reduce the cache pressure on concurrent kernels

Particle **readParticleNoCache**(int *id*) **const**
fetch particle from given particle index without going through the L1/L2 cache This can be useful to reduce the cache pressure on concurrent kernels

void **writePosition**(int *id*, **const** real4 &*r*)
Store position at the given particle id.

void **writeVelocity**(int *id*, **const** real4 &*u*)
Store velocity at the given particle id.

void **writeParticle**(int *id*, **const** *Particle* &*p*)
Store particle at the given particle id.

Public Members

int **size** = {0}
number of particles

real4 ***positions** = {nullptr}
particle positions in local coordinates

real4 ***velocities** = {nullptr}
particle velocities

real4 ***forces** = {nullptr}
particle forces

real **mass** = {0._r}
mass of one particle

real **invMass** = {0._r}
1 / mass

struct PVviewWithOldParticles : **public** *mirheo*::*PVview*
PVview with additionally positions from previous time steps

Public Functions

PVviewWithOldParticles(*ParticleVector* **pv*, *LocalParticleVector* **lpv*)
Construct a *PVviewWithOldParticles*.

Note: if pv does not have old positions channel, this will be ignored and oldPositions will be set to nullptr.

Parameters

- `pv`: The *ParticleVector* that the view represents
- `lpv`: The *LocalParticleVector* that the view represents

real3 **readOldPosition**(int *id*) **const**
fetch positions at previous time step

Public Members

real4 ***oldPositions** = {nullptr}
particle positions from previous time steps

struct PVviewWithDensities : **public** *mirheo*::*PVview*
PVview with additionally densities data

Public Functions

PVviewWithDensities(*ParticleVector* **pv*, *LocalParticleVector* **lpv*)
Construct a *PVviewWithOldParticles*.

Warning: The pv must hold a density channel.

Parameters

- `pv`: The *ParticleVector* that the view represents

- `lpv`: The *LocalParticleVector* that the view represents

Public Members

real ***densities** = {nullptr}
: particle densities

template <typename *BasicView*>
struct PVviewWithStresses : **public** BasicView
A View with additional stress info.

Template Parameters

- `BasicView`: The pv view to extend with stresses

Public Functions

PVviewWithStresses(PVType **pv*, LPVType **lpv*)
: Construct a *PVviewWithStresses*.

> **Warning:** The pv must hold a stress per particle channel.

Parameters

- `pv`: The *ParticleVector* that the view represents
- `lpv`: The *LocalParticleVector* that the view represents

Public Members

Stress ***stresses** = {nullptr}
: stresses per particle

struct OVview : **public** *mirheo*::*PVview*
A *PVview* with additionally basic object data.

Contains object ids, object extents.

Subclassed by *mirheo::OVviewWithAreaVolume*, *mirheo::OVviewWithNewOldVertices*, *mirheo::ROVview*, *mirheo::RVview*

Public Types

using PVType = *ObjectVector*
: *Particle* Vector compatible type.

using LPVType = *LocalObjectVector*
: Local *Particle* Vector compatible type.

Public Functions

OVview (*ObjectVector* *ov, *LocalObjectVector* *lov)
Construct a `OVview`.

Parameters

- `ov`: The *ObjectVector* that the view represents
- `lov`: The *LocalObjectVector* that the view represents

Public Members

int **nObjects** = {0}
number of objects

int **objSize** = {0}
number of particles per object

real **objMass** = {0._r}
mass of one object

real **invObjMass** = {0._r}
1 / objMass

COMandExtent ***comAndExtents** = {nullptr}
center of mass and extents of the objects

int64_t ***ids** = {nullptr}
global ids of objects

struct OVviewWithAreaVolume : **public** *mirheo*::*OVview*
A `OVview` with additionally area and volume information.

Subclassed by *mirheo::OVviewWithJuelicherQuants*

Public Functions

OVviewWithAreaVolume (*ObjectVector* *ov, *LocalObjectVector* *lov)
Construct a *OVviewWithAreaVolume*.

> **Warning:** The ov must hold a areaVolumes channel.

Parameters

- `ov`: The *ObjectVector* that the view represents
- `lov`: The *LocalObjectVector* that the view represents

Public Members

real2 ***area_volumes** = {nullptr}
area and volume per object

struct OVviewWithJuelicherQuants : **public** *mirheo*::*OVviewWithAreaVolume*
A `OVviewWithAreaVolume` with additional curvature related quantities.

Public Functions

OVviewWithJuelicherQuants(*ObjectVector* *ov, *LocalObjectVector* *lov)
Construct a *OVviewWithJuelicherQuants*.

> **Warning:** The ov must hold areaVolumes and lenThetaTot object channels and vertex areas, meanCurvatures particle channels.

Parameters

- `ov`: The *ObjectVector* that the view represents
- `lov`: The *LocalObjectVector* that the view represents

Public Members

real ***vertexAreas** = {nullptr}
area per vertex (defined on a triangle mesh)

real ***vertexMeanCurvatures** = {nullptr}
mean curvature vertex (defined on a triangle mesh)

real ***lenThetaTot** = {nullptr}
helper quantity to compute Juelicher bending energy

struct OVviewWithNewOldVertices : **public** *mirheo*::*OVview*
A *OVview* with additionally vertices information.

Public Functions

OVviewWithNewOldVertices(*ObjectVector* *ov, *LocalObjectVector* *lov, cudaStream_t *stream*)
Construct a *OVviewWithNewOldVertices*.

Parameters

- `ov`: The *ObjectVector* that the view represents
- `lov`: The *LocalObjectVector* that the view represents
- `stream`: Stream used to create mesh vertices if not already present

Public Members

real4 ***vertices** = {nullptr}
vertex positions

real4 ***old_vertices** = {nullptr}
vertex positions at previous time step

real4 ***vertexForces** = {nullptr}
vertex forces

int **nvertices** = {0}
number of vertices

struct ROVview : **public** *mirheo*::*OVview*

A *OVview* with additional rigid object infos.

Subclassed by *mirheo::ROVviewWithOldMotion*, *mirheo::RSOVview< Shape >*

Public Functions

ROVview(*RigidObjectVector* *rov, *LocalRigidObjectVector* *lrov)

Construct a *ROVview*.

Parameters

- `rov`: The *RigidObjectVector* that the view represents
- `lrov`: The *LocalRigidObjectVector* that the view represents

Public Members

RigidMotion ***motions** = {nullptr}

rigid object states

real3 **J** = {0._r, 0._r, 0._r}

diagonal entries of inertia tensor

real3 **J_1** = {0._r ,0._r, 0._r}

diagonal entries of the inverse inertia tensor

struct ROVviewWithOldMotion : **public** *mirheo*::*ROVview*

A *OVview* with additional rigid object info from previous time step.

Public Functions

ROVviewWithOldMotion(*RigidObjectVector* *rov, *LocalRigidObjectVector* *lrov)

Construct a *ROVview*.

> **Warning:** The rov must hold old motions channel.

Parameters

- `rov`: The *RigidObjectVector* that the view represents
- `lrov`: The *LocalRigidObjectVector* that the view represents

Public Members

RigidMotion ***old_motions** = {nullptr}

rigid object states at previous time step

template <class Shape>
struct RSOVview : **public** *mirheo*::*ROVview*

A *ROVview* with additional analytic shape infos.

Template Parameters

- `Shape`: the analytical shape that represents the object shape

Subclassed by *mirheo::RSOVviewWithOldMotion< Shape >*

Public Functions

RSOVview(*RigidShapedObjectVector*<Shape> **rsov*, *LocalRigidObjectVector* **lrov*)
Construct a *RSOVview*.

Parameters

- `rsov`: The *RigidShapedObjectVector* that the view represents
- `lrov`: The *LocalRigidObjectVector* that the view represents

Public Members

Shape **shape**
Represents the object shape.

template <**class** Shape>
struct **RSOVviewWithOldMotion** : **public** *mirheo*::*RSOVview*<Shape>
A *RSOVview* with additional rigid object info from previous time step.

Template Parameters

- `Shape`: the analytical shape that represents the object shape

Public Functions

RSOVviewWithOldMotion(*RigidShapedObjectVector*<Shape> **rsov*, *LocalRigidObjectVector* **lrov*)
Construct a *RSOVview*.

> **Warning:** The rov must hold old motions channel.

Parameters

- `rsov`: The *RigidShapedObjectVector* that the view represents
- `lrov`: The *LocalRigidObjectVector* that the view represents

Public Members

RigidMotion ***old_motions** = {nullptr}
rigid object states at previous time step

struct **RVview** : **public** *mirheo*::*OVview*
A *OVview* with additional rod object infos.

Subclassed by *mirheo::RVviewWithOldParticles*

Public Functions

RVview (*RodVector* *rv, *LocalRodVector* *lrv)
Construct a `RVview`.

Parameters

- `rv`: The *RodVector* that the view represents
- `lrv`: The *LocalRodVector* that the view represents

Public Members

int **nSegments** = {0}
number of segments per rod

int ***states** = {nullptr}
polymorphic states per bisegment

real ***energies** = {nullptr}
energies per bisegment

struct RVviewWithOldParticles : **public** *mirheo*::*RVview*
A `RVview` with additional particles from previous time step.

Public Functions

RVviewWithOldParticles (*RodVector* *rv, *LocalRodVector* *lrv)
Construct a `RVview`.

Parameters

- `rv`: The *RodVector* that the view represents
- `lrv`: The *LocalRodVector* that the view represents

Public Members

real4 ***oldPositions** = {nullptr}
positions o the particles at previous time step

Data Manager

This is the building block to create particle vectors.

class DataManager
Container for multiple channels on device and host.

Used by *ParticleVector* and *ObjectVector* to hold data per particle and per object correspondingly. All channels are stored as *PinnedBuffer*, which allows to easily transfer the data between host and device. Channels can hold data of types listed in VarPinnedBufferPtr variant. See *ChannelDescription* for the description of one channel.

Unnamed Group

DataManager(**const** *DataManager* &*b*)
copy and move constructors

Unnamed Group

ChannelDescription ***getChannelDesc**(**const** std::string &*name*)
Get channel from its name or nullptr if it is not found.

Unnamed Group

ChannelDescription &**getChannelDescOrDie**(**const** std::string &*name*)
Get channel from its name or die if it is not found.

Public Types

using NamedChannelDesc = std::pair<std::string, **const** *ChannelDescription* *>
The full description of a channel, contains its name and description.

Public Functions

void **copyChannelMap**(**const** *DataManager*&)
Copy channel names and their types from a given *DataManager*.

Does not copy data or resize buffers. New buffers are empty.

template <**typename** T>
void **createData**(**const** std::string &*name*, int *size* = 0)
Allocate a new channel.

This method will die if a channel with different type but same name already exists. If a channel with the same name and same type exists, this method will not allocate a new channel.

Template Parameters

- `T`: datatype of the buffer element. `sizeof(T)` should be compatible with VarPinnedBufferPtr

Parameters

- `name`: buffer name
- `size`: resize buffer to `size` elements

void **setPersistenceMode**(**const** std::string &*name*, PersistenceMode *persistence*)
Set the persistence mode of the data.

This method will die if the required name does not exist.

> **Warning:** This method can only increase the persistence. If the channel is already persistent, this method can not set its persistent mode to None.

Parameters

- `name`: The name of the channel to modify
- `persistence`: Persistence mode to add to the channel.

void **setShiftMode**(**const** std::string &*name*, ShiftMode *shift*)
Set the shift mode of the data.

This method will die if the required name does not exist.

> **Warning:** This method can only increase the shift mode. If the channel already needs shift, this method can not set its shift mode to None.

Parameters

- `name`: The name of the channel to modify
- `shift`: Shift mode to add to the channel.

GPUcontainer ***getGenericData**(**const** std::string &*name*)
Get gpu buffer by name.

This method will die if the required name does not exist.

Return pointer to *`GPUcontainer`* corresponding to the given name

Parameters

- `name`: buffer name

template <**typename** T>
PinnedBuffer<T> ***getData**(**const** std::string &*name*)
Get buffer by name.

This method will die if the required name does not exist or if T is of the wrong type.

Return pointer to PinnedBuffer<T> corresponding to the given name

Parameters

- `name`: buffer name

Template Parameters

- `T`: type of the element of the *PinnedBuffer*

void ***getGenericPtr**(**const** std::string &*name*)
Get device buffer pointer regardless of its type.

This method will die if the required name does not exist.

Return pointer to device data held by the corresponding *PinnedBuffer*

Parameters

- `name`: buffer name

bool **checkChannelExists**(**const** std::string &*name*) **const**
`true` if channel with given `name` exists, `false` otherwise

const std::vector<*NamedChannelDesc*> &**getSortedChannels**() **const**

Return vector of channels sorted (descending) by size of their elements (and then name)

bool **checkPersistence**(**const** std::string &*name*) **const**

Return `true` if the channel is persistent

void **resize**(int *n*, cudaStream_t *stream*)
Resize all the channels and preserve their data.

void **resize_anew**(int *n*)
Resize all the channels without preserving the data.

Friends

void **swap**(DataManager &*a*, DataManager &*b*)
swap two *DataManager*

struct ChannelDescription
Holds information and data of a single channel.

A channel has a type, persistence mode and shift mode.

Public Functions

bool **needShift**() **const**
returns true if the channel's data needs to be shifted when exchanged or redistributed.

Public Members

std::unique_ptr<*GPUcontainer*> **container**
The data stored in the channel. Internally stored as a *PinnedBuffer*.

VarPinnedBufferPtr **varDataPtr**
Pointer to container that holds the correct type.

PersistenceMode **persistence** = {PersistenceMode::None}
The persistence mode of the channel.

ShiftMode **shift** = {ShiftMode::None}
The shift mode of the channel.

18.23 Rigid

Rigid body representation and tools

Rigid state structure

Tools

void *mirheo*::rigid_operations::**collectRigidForces**(**const** *ROVview* &*view*, cudaStream_t *stream*)
Reduce the forces contained in the particles to the force and torque variable of the RigidMotion objects.

enum *mirheo*::rigid_operations::**ApplyTo**
controls to what quantities to apply the

Values:

PositionsOnly

PositionsAndVelocities

void *mirheo*::rigid_operations::**applyRigidMotion**(**const** *ROVview* &*view*, **const** *PinnedBuffer*<real4> &*initialPositions*, *ApplyTo action*, cudaStream_t *stream*)

Set the positions (and optionally velocities, according to the rigid motions.

Note The size of `initialPositions` must be the same as the object sizes described by `view`

Parameters

- `view`: The view that contains the input RigidMotion and output particles
- `initialPositions`: The positions of the particles in the frame of reference of the object
- `action`: Apply the rigid motion to positions or positions and velocities
- `stream`: execution stream

void *mirheo*::rigid_operations::**clearRigidForcesFromMotions**(**const** *ROVview* &*view*, cudaStream_t *stream*)

set the force and torques of the RigidMotion objects to zero

18.24 Simulation

class Simulation : **protected** *mirheo*::*MirObject*

Manage and combine all *MirObject* objects to run a simulation.

All *MirObject* objects must be registered and set before calling *run()*.

This must be instantiated only by ranks that have access to a GPU. Optionally, this class can communicate with a *Postprocess* one held on a different rank. This option is used for `Plugins`.

Public Functions

Simulation(**const** MPI_Comm &*cartComm*, **const** MPI_Comm &*interComm*, *MirState* **state*, CheckpointInfo *checkpointInfo*, real *maxObjHalfLength*, bool *gpuAwareMPI* = false)

Construct an empty *Simulation* object.

Parameters

- `cartComm`: a cartesian communicator that holds all ranks of the simulation.
- `interComm`: An inter communicator to communicate with the *Postprocess* ranks.
- `state`: The global state of the simulation. Does not pass ownership.
- `checkpointInfo`: Configuration of checkpoint
- `maxObjHalfLength`: Half of the maximum length of all objects.
- `gpuAwareMPI`: Performance parameter that controls if communication can be performed through RDMA.

void **restart**(**const** std::string &*folder*)

restore the simulation state from a folder that contains all restart files

void **checkpoint**()
Dump the whole simulation state to the checkpoint folder and advance the checkpoint ID.

void **registerParticleVector**(std::shared_ptr<*ParticleVector*> *pv*,
std::shared_ptr<*InitialConditions*> *ic*)
register a *ParticleVector* and initialize it with the gien *InitialConditions*.

Parameters

- `pv`: The *ParticleVector* to register
- `ic`: The *InitialConditions* that will be applied to `pv` when registered

void **registerWall**(std::shared_ptr<*Wall*> *wall*, int *checkEvery* = 0)
register a *Wall*

Parameters

- `wall`: The *Wall* to register
- `checkEvery`: The particles that will bounce against this wall will be checked (inside/outside log info) every this number of time steps. 0 means no check.

void **registerInteraction**(std::shared_ptr<*Interaction*> *interaction*)
register an *Interaction*

See *setInteraction()*.

Parameters

- `interaction`: the *Interaction* to register.

void **registerIntegrator**(std::shared_ptr<*Integrator*> *integrator*)
register an *Integrator*

See *setIntegrator()*.

Parameters

- `integrator`: the *Integrator* to register.

void **registerBouncer**(std::shared_ptr<*Bouncer*> *bouncer*)
register a *Bouncer*

See *setBouncer()*.

Parameters

- `bouncer`: the *Bouncer* to register.

void **registerPlugin**(std::shared_ptr<*SimulationPlugin*> *plugin*, int *tag*)
register a *SimulationPlugin*

Note If there is a *Postprocess* rank, it might need to register the corrsponding *PostprocessPlugin*.

Parameters

- `plugin`: the *SimulationPlugin* to register.
- `tag`: A unique tag per plugin, used by MPI communications. Must be different for every plugin.

void **registerObjectBelongingChecker**(std::shared_ptr<*ObjectBelongingChecker*> *checker*)
register a *ObjectBelongingChecker*

See *applyObjectBelongingChecker()*

Parameters

- `checker`: the *ObjectBelongingChecker* to register.

void **deregisterIntegrator**(*Integrator* **integrator*)
deregister an `Integrator`

See *registerIntegrator()*.

Parameters

- `integrator`: the `Integrator` to deregister.

void **deregisterPlugin**(*SimulationPlugin* **plugin*)
deregister a *SimulationPlugin*

Note If there is a `Postprocess` rank, the corresponding *PostprocessPlugin* must also be deregistered.

Parameters

- `plugin`: the *SimulationPlugin* to deregister.

void **setIntegrator**(**const** std::string &*integratorName*, **const** std::string &*pvName*)
Assign a registered `Integrator` to a registered *ParticleVector*.

Parameters

- `integratorName`: Name of the registered integrator (will die if it does not exist)
- `pvName`: Name of the registered *ParticleVector* (will die if it does not exist)

void **setInteraction**(**const** std::string &*interactionName*, **const** std::string &*pv1Name*, **const** std::string &*pv2Name*)
Assign two registered `Interaction` to two registered *ParticleVector* objects.

This was designed to handle *PairwiseInteraction*, which needs up to two *ParticleVector*. For self interaction cases (such as *MembraneInteraction*), `pv1Name` and `pv2Name` must be the same.

Parameters

- `interactionName`: Name of the registered interaction (will die if it does not exist)
- `pv1Name`: Name of the first registered *ParticleVector* (will die if it does not exist)
- `pv2Name`: Name of the second registered *ParticleVector* (will die if it does not exist)

void **setBouncer**(**const** std::string &*bouncerName*, **const** std::string &*objName*, **const** std::string &*pvName*)
Assign a registered `Bouncer` to registered *ObjectVector* and *ParticleVector*.

Parameters

- `bouncerName`: Name of the registered bouncer (will die if it does not exist)
- `objName`: Name of the registered *ObjectVector* that contains the surface to bounce on (will die if it does not exist)

- `pvName`: Name of the registered *ParticleVector* to bounce (will die if it does not exist)

void **setWallBounce**(**const** std::string &*wallName*, **const** std::string &*pvName*, real *maximumPartTravel*)
Set a registered *ParticleVector* to bounce on a registered `Wall`.

Parameters

- `wallName`: Name of the registered wall (will die if it does not exist)
- `pvName`: Name of the registered *ParticleVector* (will die if it does not exist)
- `maximumPartTravel`: Performance parameter. See `Wall` for more information.

void **setObjectBelongingChecker**(**const** std::string &*checkerName*, **const** std::string &*objName*)
Associate a registered *ObjectBelongingChecker* to a registered *ObjectVector*.

Note this is required before calling *applyObjectBelongingChecker()*

Parameters

- `checkerName`: Name of the registered *ObjectBelongingChecker* (will die if it does not exist)
- `objName`: Name of the registered *ObjectVector* (will die if it does not exist)

void **applyObjectBelongingChecker**(**const** std::string &*checkerName*, **const** std::string &*source*, **const** std::string &*inside*, **const** std::string &*outside*, int *checkEvery*)
Enable a registered *ObjectBelongingChecker* to split particles of a registered *ParticleVector*.

`inside` or `outside` can take the reserved value "none", in which case the corresponding particles will be deleted. Furthermore, exactly one of `inside` and `outside` must be the same as `source`.

Parameters

- `checkerName`: The name of the *ObjectBelongingChecker*. Must be associated to an *ObjectVector* with *setObjectBelongingChecker()* (will die if it does not exist)
- `source`: The registered *ParticleVector* that must be split (will die if it does not exist)
- `inside`: Name of the *ParticleVector* that will contain the particles of `source` that are inside the objects. See below for more information.
- `outside`: Name of the *ParticleVector* that will contain the particles of `source` that are outside the objects. See below for more information.
- `checkEvery`: The particle split will be performed every this amount of time steps.

If `inside` or `outside` has the name of a *ParticleVector* that is not registered, this call will create an empty *ParticleVector* with the given name and register it in the `Simulation`. Otherwise the already registered *ParticleVector* will be used.

void **init**()
setup all the simulation tasks from the registered objects and their relation. Must be called after all the register and set methods.

void **run**(*MirState*::*StepType* *nsteps*)
advance the system for a given number of time steps. Must be called after *init()*

void **notifyPostProcess**(int *tag*, int *msg*) **const**
Send a tagged message to the *Postprocess* rank.

This is useful to pass special messages, e.g. termination or checkpoint.

std::vector<*ParticleVector* *> **getParticleVectors**() **const**

Return a list of all *ParticleVector* registered objects

ParticleVector ***getPVbyName**(**const** std::string &*name*) **const**

Return *ParticleVector* with given name if found, `nullptr` otherwise

ParticleVector ***getPVbyNameOrDie**(**const** std::string &*name*) **const**

Return *ParticleVector* with given name if found, die otherwise

ObjectVector ***getOVbyName**(**const** std::string &*name*) **const**

Return *ObjectVector* with given name if found, `nullptr` otherwise

ObjectVector ***getOVbyNameOrDie**(**const** std::string &*name*) **const**

Return *ObjectVector* with given name if found, die otherwise

std::shared_ptr<*ParticleVector*> **getSharedPVbyName**(**const** std::string &*name*) **const**

Return *ParticleVector* with the given name if found, `nullptr` otherwise

Wall ***getWallByNameOrDie**(**const** std::string &*name*) **const**

Return *Wall* with the given name if found, die otherwise

CellList ***gelCellList**(*ParticleVector* **pv*) **const**

This method will die if `pv` was not registered

Return the *CellList* associated to the given *ParticleVector*, nullptr if there is none

Parameters

- `pv`: The registered *ParticleVector*

void **startProfiler**() **const**
start the cuda profiler; used for nvprof

void **stopProfiler**() **const**
end the cuda profiler; used for nvprof

MPI_Comm **getCartComm**() **const**

Return the cartesian communicator of the *Simulation*

int3 **getRank3D**() **const**

Return the coordinates in the cartesian communicator of the current rank

int3 **getNRanks3D**() **const**

Return the dimensions of the cartesian communicator

real **getCurrentDt**() **const**

Return The current time step

real **getCurrentTime**() **const**

Return The current simulation time

real **getMaxEffectiveCutoff**() **const**

This takes into account the intermediate interactions, e.g. in SDPD this will corrspond to the cutoff used for the density + the one from the SDPD kernel. Useful e.g. to decide the widh of frozen particles in walls.

Return The largest cut-off radius of all "full" force computation.

void **dumpDependencyGraphToGraphML**(**const** std::string &*fname*, bool *current*) **const**
dump the task dependency of the simulation in graphML format.

Parameters

- `fname`: The file name to dump the graph to (without extension).
- `current`: if `true`, will only dump the current tasks; otherwise, will dump all possible ones.

18.25 Task Scheduler

Becaus of the high number of tasks to execute and their *complex dependencies*, Mirheo uses a *`mirheo::TaskScheduler`* that takes care of executing all these tasks on concurrent streams. The synchronization is therefore hidden in this class.

API

class TaskScheduler
CUDA-aware task scheduler.

Manages task dependencies and run them concurrently on different CUDA streams. This is designed to be run in a time stepping scheme, e.g. all the tasks of a single time step must be described here before calling the *run()* method repetitively.

Public Types

using TaskID = int
Represents the unique id of a task.

using Function = std::function<void(cudaStream_t)>
Represents the function performed by a task. Will be executed on the given stream.

Public Functions

TaskScheduler()
Default constructor.

TaskID **createTask**(**const** std::string &*label*)
Create and register an empty task named `label`.

This method will die if a task with the given label already exists.

Return the task id associated with the new task

Parameters

- `label`: The name of the task

TaskID **getTaskId**(**const** std::string &*label*) **const**
Retrieve the task id of the task with a given label.

Return the task id if it exists, or `invalidTaskId` if it doesn't

Parameters

- `label`: The name of the task

TaskID **getTaskIdOrDie**(**const** std::string &*label*)
Retrieve the task id of the task with a given label.

This method will die if no registered task has the given label

Return the task id

Parameters

- `label`: The name of the task

void **addTask**(*TaskID id*, *Function task*, int *execEvery* = 1)
Add a function to execute to the given task.

Multiple functions can be added in a single task. The order of execution of these functions is the order in which they were added. This method will fail if the required task does not exist.

Parameters

- `id`: Task Id
- `task`: The function to execute
- `execEvery`: Execute his function every this number of calls of *run()*.

void **addDependency**(*TaskID id*, std::vector<*TaskID*> *before*, std::vector<*TaskID*> *after*)
add dependencies around a given task

Parameters

- `id`: The task that must be executed before `before` and after `after`
- `before`: the list of tasks that must be executed after the task with id `id`
- `after`: the list of tasks that must be executed before the task with id `id`

void **setHighPriority**(*TaskID id*)
Set the execution of a task to high priority.

Parameters

- `id`: The task id

void **compile**()
Prepare the internal state so that the scheduler can perform execution of all tasks.

No other calls related to task creation / modification / dependencies must be performed after calling this function.

void **run**()
Execute the tasks in the order required by the given dependencies and priorities.

Must be called after *compile()*.

void **dumpGraphToGraphML**(**const** std::string *&fname*) **const**
Dump a representation of the tasks and their dependencies in graphML format.

Parameters

- `fname`: The file name to dump the graph to (without extension).

void **forceExec**(*TaskID id*, cudaStream_t *stream*)
Execute a given task on a given stream.

Parameters

- `id`: the task to execute
- `stream`: The stream to execute the task

Public Static Attributes

constexpr *TaskID* **invalidTaskId** = {static_cast<*TaskID*>(-1)}
Special task id value to represent invalid tasks.

18.26 Types

Each channel in *`mirheo::DataManager`* can have one of the types listed in the following xmacro:

MIRHEO_TYPE_TABLE__(OP, SEP)
xmacro that contains the list of type available for data channels.

Must contain POd structures that are compatible with device code.

Host variants

template <**class** T>
struct DataTypeWrapper
A simple structure to store a c type.

This is useful with a variant and visitor pattern.

Template Parameters

- `T`: The type to wrap

The `mirheo::TypeDescriptor` variant contains a type of *`mirheo::DataTypeWrapper`* that is in the type list.

Device variants

The `mirheo::CudaVarPtr` variant contains a pointer of a type that is in the type list.

Utils

std::string *mirheo*::**typeDescriptorToString**(**const** TypeDescriptor &*desc*)
Convert a TypeDescriptor variant to the string that represents the type.

Return The string that correspond to the type (e.g. int gives "int")

Parameters

- `desc`: The variant of *DataTypeWrapper*

TypeDescriptor *mirheo*::**stringToTypeDescriptor**(**const** std::string &*str*)
reverse operation of *typeDescriptorToString()*.

This method will die if `str` does not correspond to any type in the type list.

Return a variant that contains the *DataTypeWrapper* with the correct type.

Parameters

- `str`: The string representation of the type (e.g. "int" for int)

18.27 Utils

FileWrapper

class FileWrapper
Wrapper for c-stype FILE with RAII.

Public Types

enum SpecialStream
Used to construct special stream handlers for cout and cerr.

Values:

Cout

Cerr

enum Status
return status when opening the files

Values:

Success

Failed

Public Functions

FileWrapper()
default constructor

FileWrapper(**const** std::string &*fname*, **const** std::string &*mode*)
Construct a *FileWrapper* and tries to open the file `fname` in `mode` mode.

This method will die if the file was not found

Parameters

- `fname`: The name of the file to open
- `mode`: The open mode, e.g. "r" for read mode (see docs of std::fopen)

`FileWrapper`(*SpecialStream stream*, bool *forceFlushOnClose*)
Construct a *FileWrapper* for console output.

Note See also *open(SpecialStream, bool)*)

Parameters

- `stream`: The `SpecialStream` to dump to.
- `forceFlushOnClose`: If `true`, flushes to the stream when the object is closed.

`FileWrapper`(*FileWrapper*&&)
move constructor

FileWrapper &**`operator=`**(*FileWrapper*&&)
move assignment

Status **`open`**(**`const`** std::string &*fname*, **`const`** std::string &*mode*)
Open a file in a given mode.

Return Status::Success if the file was open succesfully, Status::Failed otherwise

Parameters

- `fname`: The name of the file to open
- `mode`: The open mode, e.g. "r" for read mode (see docs of std::fopen)

Status **`open`**(*SpecialStream stream*, bool *forceFlushOnClose*)
Set the wrapper to write to a special stream.

Return success status

Parameters

- `stream`: stdout or stderr
- `forceFlushOnClose`: If set to `true`, the buffer will be flushed when *close()* is called.

FILE ***`get`**()

Return the C-style file handler

void **`close`**()
Close the current handler.

This does not need to be called manually unless reopening a new file, since it will be called in the destructor.

If the handler was pointing to a file, the file is close. If the handler was pointing to a special stream (cout, cerr), fflush may be called (see forceFlushOnClose parameter in *open(SpecialStream, bool)*) but the stream is not closed. If the handler did not point to anything, nothing happens.

void **`fread`**(void **ptr*, size_t *size*, size_t *count*)
Wrapper around std::fread. Throws an exception if reading failed.

Folders

A set of functions to manipulate paths and create folders.

std::string *mirheo*::**createStrZeroPadded**(long long *i*, int *zeroPadding* = 5)
Create a string representing an integer with 0 padding.

If `zeroPadding` is too small, this method will die. Example: `createStrZeroPadded(42, 5)` gives "00042"

Return the string representation of `i` with padded zeros

Parameters

- `i`: The integer to print (must non negative)
- `zeroPadding`: The total number of characters

std::vector<std::string> *mirheo*::**splitByDelim**(std::string *str*, char *delim* = ',')
Split a string according to a delimiter character.

e.g. `splitByDelim`("string_to_split", '_') -> {"string", "to", "split"}

Return The list of substrings (without the delimiter)

Parameters

- `str`: The input sequence of characters
- `delim`: The delimiter

std::string *mirheo*::**makePath**(std::string *path*)
append '/' at the end of `path` if it doesn t have it already

Return The path with a trailing separator

Parameters

- `path`: The path to work with

std::string *mirheo*::**getParentPath**(std::string *path*)
Get the parent folder of the given filename.

If the input is a path (it ends with a '/'), the output is the same as the input. If the input is just a filename with no '/', this function returns an empty string.

Return The parent folder.

Parameters

- `path`: The filename containing a path

std::string *mirheo*::**getBaseName**(std::string *path*)
remove the path from the given filename.

Return the filename only without any prepended folder

Parameters

- `path`: The filename with full relative or absolute path

std::string *mirheo*::**joinPaths**(**const** std::string &*A*, **const** std::string &*B*)
Concatenate two paths A and B.

Return A/B Adds a '/' between `A` and `B` if `A` is non-empty and if it doesn't already end with '/'.

Parameters

- `A`: first part of the full path
- `B`: last part of the full path

bool *mirheo*::**createFoldersCollective**(**const** MPI_Comm &*comm*, std::string *path*)

Create a folder.

The operation is collective. This means that all ranks in the `comm` must call it. The returned value is accessible by all ranks.

Return `true` if the operation was successful, `false` otherwise

Parameters

- `comm`: The communicator used to decide which rank creates the folder
- `path`: the folder to create

Quaternion

template <class *Real*>
class Quaternion

Quaternion representation with basic operations.

See also:

- http://www.iri.upc.edu/people/jsola/JoanSola/objectes/notes/kinematics.pdf
- https://arxiv.org/pdf/0811.2889.pdf

Template Parameters

- `Real`: The precision to be used. Must be a scalar real number type (e.g. float, double).

Public Functions

Quaternion(**const** *Quaternion* &*q*)

copy constructor

Quaternion &**operator=**(**const** *Quaternion* &*q*)

assignment operator

template <class T>
operator Quaternion<T>() **const**

conversion to different precision

operator float4() **const**

conversion to float4 (real part will be stored first, followed by the vector part)

operator double4() **const**

conversion to double4 (real part will be stored first, followed by the vector part)

Real **realPart**() **const**

Return the real part of the quaternion

Real3 **vectorPart**() **const**

Return the vector part of the quaternion

Quaternion<Real> **conjugate**() **const**

Return the conjugate of the quaternion

Real **norm**() **const**

Return the norm of the quaternion

Quaternion &**normalize**()
Normalize the current quaternion. Must be non zero.

Quaternion **normalized**() **const**

Return A normalized copy of this *Quaternion*

Quaternion &**operator+=**(**const** *Quaternion* &*q*)
Add a quaternion to the current one.

Quaternion &**operator-=**(**const** *Quaternion* &*q*)
Subtract a quaternion to the current one.

Quaternion &**operator*=**(Real *a*)
Scale the current quaternion.

Quaternion &**operator*=**(**const** *Quaternion* &*q*)
Multiply the current quaternion with another with *Quaternion* multiplication and store the result in this object.

Real3 **rotate**(Real3 *v*) **const**

Return The input vector rotated by the current quaternion

Real3 **inverseRotate**(Real3 *v*) **const**

Return The input vector rotated by the current quaternion inverse

Quaternion **timeDerivative**(Real3 *omega*) **const**

Return The time derivative of the given angular velocity, useful for time integration of rigid objects

Public Members

Real **w**
real part

Real **x**
vector part, x

Real **y**
vector part, y

Real **z**
vector part, z

Public Static Functions

static *Quaternion* **createFromComponents**(Real *w*, Real *x*, Real *y*, Real *z*)
Create a `Quaternion` from components.

static *Quaternion* **createFromComponents** (Real *w*, Real3 *v*)
Create a `Quaternion` from real part and vector part.

static *Quaternion* **createFromComponents** (Real4 *v*)
Create a `Quaternion` from components.

static *Quaternion* **pureVector** (Real3 *v*)
Create a pure vector `Quaternion`.

static *Quaternion* **createFromRotation** (Real *angle*, Real3 *axis*)
Create a `Quaternion` that represents the rotation around an axis with a given angle.

Parameters

- `angle`: The angle (in radians) of the rotation
- `axis`: The axis of rotation, must be non zero (or nan will be returned)

static *Quaternion* **createFromVectors** (Real3 *from*, Real3 *to*)
Create a `Quaternion` that represents the "shortest" rotation between two vectors.

The vectors must be non zero.

Parameters

- `from`: The origin vector
- `to`: The vector obtained by applying the rotation to `from`

Friends

Quaternion **operator+** (Quaternion *q1*, **const** Quaternion &*q2*)

Return The sum of 2 quaternions

Quaternion **operator-** (Quaternion *q1*, **const** Quaternion &*q2*)

Return The difference of 2 quaternions

Quaternion **operator*** (Real *a*, Quaternion *q*)

Return The scalar multiplication of a quaternion

Quaternion **operator*** (Quaternion *q*, Real *a*)

Return The scalar multiplication of a quaternion

Quaternion **operator*** (**const** Quaternion &*q1*, **const** Quaternion &*q2*)

Return The quaternion product of 2 quaternions

18.28 Walls

See also *the user interface*.

Base classes

class Wall : **public** *mirheo*::*MirSimulationObject*
Physical boundaries of the simulation.

A wall is composed of its surface. Additionally, frozen particles can be created from that surface and attached to the wall.

Subclassed by *mirheo::SDFBasedWall*

Public Functions

Wall(**const** *MirState* *state, **const** std::string &name)
Construct a `Wall`.

Parameters

- `state`: The simulation state.
- `name`: The name of the wall.

virtual void **setup**(MPI_Comm &comm) = 0
Initialize the wall internal state.

This must be called before any other wall operations that involve its surface.

Parameters

- `comm`: The MPI Cartesian communicator of the simulation.

virtual void **attachFrozen**(*ParticleVector* *pv) = 0
Set frozen particles to the wall.

The frozen particles may be modified in the operation (velocities set to the wall's one).

Parameters

- `pv`: The frozen particles.

virtual void **removeInner**(*ParticleVector* *pv) = 0
Remove particles inside the walls.

If pv is an *ObjectVector*, any object with at least one particle will be removed by this operation.

Parameters

- `pv`: *ParticleVector* to remove the particles from.

virtual void **attach**(*ParticleVector* *pv, *CellList* *cl, real maximumPartTravel) = 0
Register a *ParticleVector* that needs to be bounced from the wall.

Multiple *ParticleVector* can be registered by calling this method several times. The parameter maximumPartTravel is used for performance, lower leading to higher performances. Note that if it is too low, some particles may be ignored and not bounced and end up inside the walls (see *bounce()*).

Parameters

- `pv`: The particles to be bounced. Will be ignored if it is the same as the frozen particles.
- `cl`: Cell lists corresponding to pv.
- `maximumPartTravel`: The estimated maximum distance traveled by one particle over a single time step.

virtual void **detachAllCellLists**() = 0
Clean up all information regarding cell lists generated by `attach`.

virtual void **bounce**(cudaStream_t *stream*) = 0
Bounce the particles attached to the wall.

The particles that are bounced must be registered previously exactly once with *attach()*.

Parameters

- `stream`: The stream to execute the bounce operation on.

virtual void **setPrerequisites**(*ParticleVector* **pv*)
Set properties needed by the particles to be bounced.

Must be called just after *setup()* and before any *bounce()*. Default: ask nothing.

Parameters

- `pv`: Particles to add additional properties to.

virtual void **check**(cudaStream_t *stream*) = 0
Counts number of particles inside the walls and report it in the logs.

The particles that are counted must be previously attached to the walls by calling *attach()*.

Parameters

- `stream`: The stream to execute the check operation on.

class SDFBasedWall : **public** *mirheo*::*Wall*
Wall with surface represented via a signed distance function (SDF).

The surface of the wall is the zero level set of its SDF. The SDF has positive values **outside** the simulation domain (called inside the walls), and is negative **inside** the simulation domain.

Subclassed by *mirheo::SimpleStationaryWall< InsideWallChecker >*

Public Functions

virtual void **sdfPerParticle**(*LocalParticleVector* **lpv*, *GPUcontainer* **sdfs*, *GPUcontainer* **gradients*, real *gradientThreshold*, cudaStream_t *stream*) = 0
Compute the wall SDF at particles positions.

Parameters

- `lpv`: Input particles.
- `sdfs`: Values of the SDF at the particle positions.
- `gradients`: Gradients of the SDF at the particle positions. Can be disabled by passing a nullptr.
- `gradientThreshold`: Compute gradients for particles that are only within that distance. Irrelevant if gradients is nullptr.
- `stream`: The stream to execute the operation on.

virtual void **sdfPerPosition**(*GPUcontainer* **positions*, *GPUcontainer* **sdfs*, cudaStream_t *stream*) = 0
Compute the wall SDF at given positions.

Parameters

- `positions`: Input positions.
- `sdfs`: Values of the SDF at the given positions.
- `stream`: The stream to execute the operation on.

virtual void **sdfOnGrid**(real3 *gridH*, *GPUcontainer* **sdfs*, cudaStream_t *stream*) = 0
Compute the wall SDF on a uniform grid.

This method will resize the sdfs container internally.

Parameters

- `gridH`: grid spacing.
- `sdfs`: Values of the SDF at the grid nodes positions.
- `stream`: The stream to execute the operation on.

virtual *PinnedBuffer*<double3> ***getCurrentBounceForce**() = 0
Get accumulated force of particles on the wall at the previous *bounce()* operation..

Derived classes

template <**class** InsideWallChecker>
class **SimpleStationaryWall** : **public** *mirheo*::*SDFBasedWall*
SDF based *Wall* with zero velocity boundary conditions.

Template Parameters

- `InsideWallChecker`: *Wall* shape representation.

Subclassed by *mirheo::WallWithVelocity< InsideWallChecker, VelocityField >*

Public Functions

SimpleStationaryWall(**const** *MirState* **state*, **const** std::string &*name*, InsideWallChecker &&*insideWallChecker*)
Construct a *SimpleStationaryWall* object.

Parameters

- `state`: The simulation state.
- `name`: The wall name.
- `insideWallChecker`: A functor that represents the wall surface (see stationary_walls/).

void **setup**(MPI_Comm &*comm*)
Initialize the wall internal state.

This must be called before any other wall operations that involve its surface.

Parameters

- `comm`: The MPI Cartesian communicator of the simulation.

void **setPrerequisites**(*ParticleVector* **pv*)
Set properties needed by the particles to be bounced.

Must be called just after *setup()* and before any *bounce()*. Default: ask nothing.

Parameters

- `pv`: Particles to add additional properties to.

void **attachFrozen**(*ParticleVector* **pv*)
Set frozen particles to the wall.

The frozen particles may be modified in the operation (velocities set to the wall's one).

Parameters

- `pv`: The frozen particles.

void **removeInner**(*ParticleVector* **pv*)
Remove particles inside the walls.

If pv is an *ObjectVector*, any object with at least one particle will be removed by this operation.

Parameters

- `pv`: *ParticleVector* to remove the particles from.

void **attach**(*ParticleVector* **pv*, *CellList* **cl*, real *maximumPartTravel*)
Register a *ParticleVector* that needs to be bounced from the wall.

Multiple *ParticleVector* can be registered by calling this method several times. The parameter maximumPartTravel is used for performance, lower leading to higher performances. Note that if it is too low, some particles may be ignored and not bounced and end up inside the walls (see *bounce()*).

Parameters

- `pv`: The particles to be bounced. Will be ignored if it is the same as the frozen particles.
- `cl`: Cell lists corresponding to pv.
- `maximumPartTravel`: The estimated maximum distance traveled by one particle over a single time step.

void **detachAllCellLists**()
Clean up all information regarding cell lists generated by `attach`.

void **bounce**(cudaStream_t *stream*)
Bounce the particles attached to the wall.

The particles that are bounced must be registered previously exactly once with *attach()*.

Parameters

- `stream`: The stream to execute the bounce operation on.

void **check**(cudaStream_t *stream*)
Counts number of particles inside the walls and report it in the logs.

The particles that are counted must be previously attached to the walls by calling *attach()*.

Parameters

- `stream`: The stream to execute the check operation on.

void **sdfPerParticle**(*LocalParticleVector* *lpv, *GPUcontainer* *sdfs, *GPUcontainer* *gradients, real *gradientThreshold*, cudaStream_t *stream*)
Compute the wall SDF at particles positions.

Parameters

- `lpv`: Input particles.
- `sdfs`: Values of the SDF at the particle positions.
- `gradients`: Gradients of the SDF at the particle positions. Can be disabled by passing a nullptr.
- `gradientThreshold`: Compute gradients for particles that are only within that distance. Irrelevant if gradients is nullptr.
- `stream`: The stream to execute the operation on.

void **sdfPerPosition**(*GPUcontainer* *positions, *GPUcontainer* *sdfs, cudaStream_t *stream*)
Compute the wall SDF at given positions.

Parameters

- `positions`: Input positions.
- `sdfs`: Values of the SDF at the given positions.
- `stream`: The stream to execute the operation on.

void **sdfOnGrid**(real3 *gridH*, *GPUcontainer* *sdfs, cudaStream_t *stream*)
Compute the wall SDF on a uniform grid.

This method will resize the sdfs container internally.

Parameters

- `gridH`: grid spacing.
- `sdfs`: Values of the SDF at the grid nodes positions.
- `stream`: The stream to execute the operation on.

InsideWallChecker &**getChecker**()
get a reference of the wall surfae representation.

PinnedBuffer<double3> ***getCurrentBounceForce**()
Get accumulated force of particles on the wall at the previous *bounce()* operation..

template <**class** InsideWallChecker, **class** VelocityField>
class **WallWithVelocity** : **public** *mirheo*::*SimpleStationaryWall*<InsideWallChecker>
SDF-based wall with non zero velocity boundary conditions.

Template Parameters

- `InsideWallChecker`: *Wall* shape representation.
- `VelocityField`: *Wall* velocity representation.

Public Functions

WallWithVelocity(**const** *MirState* *state, **const** std::string &name, InsideWallChecker &&insideWallChecker, VelocityField &&velField)
Construct a *WallWithVelocity* object.

Parameters

- `state`: The simulation state.
- `name`: The wall name.
- `insideWallChecker`: A functor that represents the wall surface (see stationary_walls/).
- `velField`: A functor that represents the wall velocity (see velocity_field/).

void **setup**(MPI_Comm &comm)
Initialize the wall internal state.

This must be called before any other wall operations that involve its surface.

Parameters

- `comm`: The MPI Cartesian communicator of the simulation.

void **attachFrozen**(*ParticleVector* *pv)
Set frozen particles to the wall.

The frozen particles may be modified in the operation (velocities set to the wall's one).

Parameters

- `pv`: The frozen particles.

void **bounce**(cudaStream_t stream)
Bounce the particles attached to the wall.

The particles that are bounced must be registered previously exactly once with *attach()*.

Parameters

- `stream`: The stream to execute the bounce operation on.

Wall shapes

class StationaryWallBox
Represents a box shape.

Public Functions

StationaryWallBox(real3 lo, real3 hi, bool inside)
Construct a *StationaryWallBox*.

Parameters

- `lo`: Lower bounds of the box (in global coordinates).
- `hi`: Upper bounds of the box (in global coordinates).
- `inside`: Domain is inside the box if set to `true`.

void **setup**(MPI_Comm &*comm*, *DomainInfo* *domain*)
Synchronize internal state with simulation.

Parameters

- `comm`: MPI carthesia communicator
- `domain`: Domain info

const *StationaryWallBox* &**handler**() **const**
Get a handler of the shape representation usable on the device.

real **operator()**(real3 *r*) **const**
Get the SDF of the current shape at a given position.

Return The SDF value

Parameters

- `r`: position in local coordinates

class StationaryWallCylinder
Represents a cylinder along one of the main axes.

Public Types

enum Direction
Represents the direction of the main axis of the cylinder.

Values:

x

y

z

Public Functions

StationaryWallCylinder(real2 *center*, real *radius*, *Direction* *dir*, bool *inside*)
Construct a *StationaryWallCylinder*.

Parameters

- `center`: Center of the cylinder in global coordinates in the plane perpendicular to the direction
- `radius`: Radius of the cylinder
- `dir`: The direction of the main axis.
- `inside`: Domain is inside the cylinder if set to `true`.

void **setup**(MPI_Comm &*comm*, *DomainInfo* *domain*)
Synchronize internal state with simulation.

Parameters

- `comm`: MPI carthesia communicator

- `domain`: Domain info

const *StationaryWallCylinder* &**handler**() **const**
Get a handler of the shape representation usable on the device.

real **operator()**(real3 *r*) **const**
Get the SDF of the current shape at a given position.

Return The SDF value

Parameters

- `r`: position in local coordinates

class StationaryWallPlane
Represents a planar wall.

Public Functions

StationaryWallPlane(real3 *normal*, real3 *pointThrough*)
Construct a *StationaryWallPlane*.

Parameters

- `normal`: Normal of the wall, pointing inside the walls.
- `pointThrough`: One point inside the plane, in global coordinates.

void **setup**(MPI_Comm &*comm*, *DomainInfo* *domain*)
Synchronize internal state with simulation.

Parameters

- `comm`: MPI carthesia communicator
- `domain`: Domain info

const *StationaryWallPlane* &**handler**() **const**
Get a handler of the shape representation usable on the device.

real **operator()**(real3 *r*) **const**
Get the SDF of the current shape at a given position.

Return The SDF value

Parameters

- `r`: position in local coordinates

class StationaryWallSDF
Represent an arbitrary SDF field on a grid.

Public Functions

StationaryWallSDF(**const** *MirState* **state*, std::string *sdfFileName*, real3 *sdfH*, real3 *margin*)
Construct a *StationaryWallSDF* from a file.

Parameters

- `state`: *Simulation* state
- `sdfFileName`: The input file name
- `sdfH`: The grid spacing
- `margin`: Additional margin to store in each rank; useful to bounce-back local particles.

StationaryWallSDF(*StationaryWallSDF*&&)
Move ctor.

void **setup**(MPI_Comm &*comm*, *DomainInfo* *domain*)
Synchronize internal state with simulation.

Parameters

- `comm`: MPI carthesia communicator
- `domain`: Domain info

const *FieldDeviceHandler* &**handler**() **const**
Get a handler of the shape representation usable on the device.

class StationaryWallSphere
Represents a sphere shape.

Public Functions

StationaryWallSphere(real3 *center*, real *radius*, bool *inside*)
Construct a *StationaryWallSphere*.

Parameters

- `center`: Center of the sphere in global coordinates
- `radius`: Radius of the sphere
- `inside`: Domain is inside the box if set to `true`.

void **setup**(MPI_Comm &*comm*, *DomainInfo* *domain*)
Synchronize internal state with simulation.

Parameters

- `comm`: MPI carthesia communicator
- `domain`: Domain info

const *StationaryWallSphere* &**handler**() **const**
Get a handler of the shape representation usable on the device.

real **operator()** (real3 *r*) **const**
Get the SDF of the current shape at a given position.

Return The SDF value

Parameters

- `r`: position in local coordinates

Velocity fields

class VelocityFieldNone
Zero velocity field.

Public Functions

void **setup** (real *t*, *DomainInfo* *domain*)
to fit the interface

const *VelocityFieldNone* &**handler**() **const**
get a handler that can be used on device

real3 **operator()** (real3 *r*) **const**
Evaluate the velocity field at a given position.

Return The velocity value

Parameters

- `r`: The position in local coordinates

class VelocityFieldOscillate
Oscillating velocity field in time.

$$\mathbf{v}(t) = \cos\frac{2\pi t}{T}\mathbf{v},$$

where T is the period.

Public Functions

VelocityFieldOscillate (real3 *vel*, real *period*)
Construct a *VelocityFieldOscillate* object.

Parameters

- `vel`: The maximum velocity vector
- `period`: Oscillating period in simulation time. Fails if negative.

void **setup** (real *t*, *DomainInfo* *domain*)
Synchronize with simulation state.

Must be called at every time step.

Parameters

- `t`: *Simulation* time.
- `domain`: domain info.

const *VelocityFieldOscillate* &**handler**`()` **const**
get a handler that can be used on the device.

real3 **operator()**(real3 *r*) **const**
Evaluate the velocity field at a given position.

Return The velocity value

Parameters

- `r`: The position in local coordinates

class VelocityFieldRotate
Rotating velocity field (constant in time).

The field is defined by a center and an angular velocity:

$$\mathbf{v}(\mathbf{r}) = \omega \times (\mathbf{r} - \mathbf{c})$$

Public Functions

VelocityFieldRotate(real3 *omega*, real3 *center*)
Construct a *VelocityFieldRotate* object.

Parameters

- `omega`: The angular velocity
- `center`: Center of rotation in global coordinates

void **setup**(real *t*, *DomainInfo* *domain*)
Synchronize with simulation state.

Must be called at every time step.

Parameters

- `t`: *Simulation* time.
- `domain`: domain info.

const *VelocityFieldRotate* &**handler**`()` **const**
get a handler that can be used on the device.

real3 **operator()**(real3 *r*) **const**
Evaluate the velocity field at a given position.

Return The velocity value

Parameters

- `r`: The position in local coordinates

class VelocityFieldTranslate
Constant velocity field.

Public Functions

VelocityFieldTranslate(real3 *vel*)
Construct a *VelocityFieldTranslate*.

Parameters

- `vel`: The constant velocity

void **setup**(real *t*, *DomainInfo* *domain*)
to fir the interface

const *VelocityFieldTranslate* &**handler**() **const**
get a handler that can be used on the device

real3 **operator()**(real3 *r*) **const**
Evaluate the velocity field at a given position.

Return The velocity value

Parameters

- `r`: The position in local coordinates

18.29 XDMF

A set of classes and functions to write/read data to/from xdmf + hdf5 files format.

VertexChannelsData *mirheo*::*XDMF*::**readVertexData**(**const** std::string &*filename*, MPI_Comm *comm*, int *chunkSize*)
Read particle data from a pair of xmf+hdf5 files.

Return The read data (on the local rank)

Parameters

- `filename`: the xdmf file name (with extension)
- `comm`: The communicator used in the I/O process
- `chunkSize`: The smallest piece that processors can split

Grids

mirheo::XDMF::Grid objects are used to represent the geometry of the data that will be dumped.

Interface

class GridDims
Interface to represent the dimensions of the geometry data.

Subclassed by mirheo::XDMF::VertexGrid::VertexGridDims

Public Functions

virtual std::vector<hsize_t> **getLocalSize**() **const** = 0
number of elements in the current subdomain

virtual std::vector<hsize_t> **getGlobalSize**() **const** = 0
number of elements in the whole domain

virtual std::vector<hsize_t> **getOffsets**() **const** = 0
start indices in the current subdomain

bool **localEmpty**() **const**

Return `true` if there is no data in the current subdomain

bool **globalEmpty**() **const**

Return `true` if there is no data in the whole domain

int **getDims**() **const**

Return The current dimension of the data (e.g. 3D for uniform grids, 1D for particles)

class Grid
Interface to represent The geometry of channels to dump.

Subclassed by *mirheo::XDMF::UniformGrid*, *mirheo::XDMF::VertexGrid*

Public Functions

virtual const *GridDims* ***getGridDims**() **const** = 0

Return the *GridDims* that describes the data dimensions

virtual std::string **getCentering**() **const** = 0

Return A string describing (for *XDMF*) data location (e.g. "Node" or "Cell")

virtual void **writeToHDF5**(hid_t *file_id*, MPI_Comm *comm*) **const** = 0
Dump the geometry description to hdf5 file.

Parameters

- `file_id`: The hdf5 file description
- `comm`: MPI communicator that was used to open the file

virtual pugi::xml_node **writeToXMF**(pugi::xml_node *node*, std::string *h5filename*) **const** = 0
Dump the geometry description to xdmf file.

Parameters

- `node`: The xml node that will store the geometry information
- `h5filename`: name of the hdf5 file that will contain the data

virtual void **readFromXMF**(**const** pugi::xml_node &*node*, std::string &*h5filename*) = 0
read the geometry info contained in the xdmf file

Note must be called before *splitReadAccess()*

Parameters

- `node`: The xmf data
- `h5filename`: The name of the associated hdf5 file

virtual void **splitReadAccess**(MPI_Comm *comm*, int *chunkSize* = 1) = 0
Set the number of elements to read for the current subdomain.

Note must be called after *readFromXMF()*

Parameters

- `comm`: Communicator that will be used to read the hdf5 file
- `chunkSize`: For particles, this affects the number of particles to keep together on a single rank. Useful for objects.

virtual void **readFromHDF5**(hid_t *file_id*, MPI_Comm *comm*) = 0
Read the geometry data contained in the hdf5 file.

Note must be called after *splitReadAccess()*

Parameters

- `file_id`: The hdf5 file reference
- `comm`: MPI communicator used in the I/O

Implementation

class UniformGrid : **public** *mirheo*::*XDMF*::*Grid*
Representation of a uniform grid geometry.

Each subdomain has the same number of grid points in every direction.

Public Functions

UniformGrid(int3 *localSize*, real3 *h*, MPI_Comm *cartComm*)
construct a *UniformGrid* object

Note all these parameters must be the same on every rank

Parameters

- `localSize`: The dimensions of the grid per rank
- `h`: grid spacing
- `cartComm`: The cartesian communicator that will be used for I/O

const *GridDims* ***getGridDims**() **const**

Return the *GridDims* that describes the data dimensions

std::string **getCentering**() **const**

Return A string describing (for *XDMF*) data location (e.g. “Node” or “Cell”)

void **writeToHDF5** (hid_t *file_id*, MPI_Comm *comm*) **const**
Dump the geometry description to hdf5 file.

Parameters

- `file_id`: The hdf5 file description
- `comm`: MPI communicator that was used to open the file

pugi::xml_node **writeToXMF** (pugi::xml_node *node*, std::string *h5filename*) **const**
Dump the geometry description to xdmf file.

Parameters

- `node`: The xml node that will store the geometry information
- `h5filename`: name of the hdf5 file that will contain the data

void **readFromXMF** (**const** pugi::xml_node &*node*, std::string &*h5filename*)
read the geometry info contained in the xdmf file

Note must be called before *splitReadAccess()*

Parameters

- `node`: The xmf data
- `h5filename`: The name of the associated hdf5 file

void **splitReadAccess** (MPI_Comm *comm*, int *chunkSize* = 1)
Set the number of elements to read for the current subdomain.

Note must be called after *readFromXMF()*

Parameters

- `comm`: Communicator that will be used to read the hdf5 file
- `chunkSize`: For particles, this affects the number of particles to keep together on a single rank. Useful for objects.

void **readFromHDF5** (hid_t *file_id*, MPI_Comm *comm*)
Read the geometry data contained in the hdf5 file.

Note must be called after *splitReadAccess()*

Parameters

- `file_id`: The hdf5 file reference
- `comm`: MPI communicator used in the I/O

class VertexGrid : **public** *mirheo*::*XDMF*::*Grid*
Representation of particles geometry.

Each rank contains the positions of the particles in GLOBAL coordinates.

Subclassed by mirheo::XDMF::PolylineMeshGrid, *mirheo::XDMF::TriangleMeshGrid*

Public Functions

VertexGrid(std::shared_ptr<std::vector<real3>> *positions*, MPI_Comm *comm*)
Construct a *VertexGrid* object.

Note The `positions` are passed as a shared pointer so that this class is able to either allocate its own memory or can share it with someone else

Parameters

- `positions`: The positions of the particles in the current subdomain, in global coordinates
- `comm`: The communicator that will be used for I/O

const *GridDims* ***getGridDims**() **const**

Return the *GridDims* that describes the data dimensions

std::string **getCentering**() **const**

Return A string describing (for *XDMF*) data location (e.g. "Node" or "Cell")

void **writeToHDF5**(hid_t *file_id*, MPI_Comm *comm*) **const**
Dump the geometry description to hdf5 file.

Parameters

- `file_id`: The hdf5 file description
- `comm`: MPI communicator that was used to open the file

pugi::xml_node **writeToXMF**(pugi::xml_node *node*, std::string *h5filename*) **const**
Dump the geometry description to xdmf file.

Parameters

- `node`: The xml node that will store the geometry information
- `h5filename`: name of the hdf5 file that will contain the data

void **readFromXMF**(**const** pugi::xml_node &*node*, std::string &*h5filename*)
read the geometry info contained in the xdmf file

Note must be called before *splitReadAccess()*

Parameters

- `node`: The xmf data
- `h5filename`: The name of the associated hdf5 file

void **splitReadAccess**(MPI_Comm *comm*, int *chunkSize* = 1)
Set the number of elements to read for the current subdomain.

Note must be called after *readFromXMF()*

Parameters

- `comm`: Communicator that will be used to read the hdf5 file

- `chunkSize`: For particles, this affects the number of particles to keep together on a single rank. Useful for objects.

void **readFromHDF5**(hid_t *file_id*, MPI_Comm *comm*)
Read the geometry data contained in the hdf5 file.

Note must be called after *splitReadAccess()*

Parameters

- `file_id`: The hdf5 file reference
- `comm`: MPI communicator used in the I/O

class TriangleMeshGrid : **public** *mirheo*::*XDMF*::*VertexGrid*
Representation of triangle mesh geometry.

This is a *VertexGrid* associated with the additional connectivity (list of triangle faces). The vertices are stored in global coordinates and the connectivity also stores indices in global coordinates.

Public Functions

TriangleMeshGrid(std::shared_ptr<std::vector<real3>> *positions*, std::shared_ptr<std::vector<int3>> *triangles*, MPI_Comm *comm*)
Construct a *TriangleMeshGrid* object.

Parameters

- `positions`: The positions of the particles in the current subdomain, in global coordinates
- `triangles`: The list of faces in the current subdomain (global indices)
- `comm`: The communicator that will be used for I/O

void **writeToHDF5**(hid_t *file_id*, MPI_Comm *comm*) **const**
Dump the geometry description to hdf5 file.

Parameters

- `file_id`: The hdf5 file description
- `comm`: MPI communicator that was used to open the file

Channel

namespace mirheo
Common namespace for all *Mirheo* code.

namespace XDMF
namespace for all functions related to I/O with *XDMF* + hdf5

Functions

std::string **dataFormToXDMFAttribute**(*Channel*::*DataForm* *dataForm*)
Return the xdmf-compatible string that describes the *Channel::DataForm*

int **dataFormToNcomponents**(*Channel*::*DataForm* *dataForm*)
Return the number of components in the *Channel::DataForm*

std::string **dataFormToDescription**(*Channel*::*DataForm* *dataForm*)
Return a unique string that describes the *Channel::DataForm* (two different may map to the same xdmf attribute)

Channel::*DataForm* **descriptionToDataForm**(**const** std::string &*str*)
reverse of *dataFormToDescription()*

decltype(H5T_NATIVE_FLOAT) **numberTypeToHDF5type**(*Channel*::NumberType *nt*)
Return the HDF5-compatible description of the given *Channel::NumberType* data type

std::string **numberTypeToString**(*Channel*::NumberType *nt*)
Return the xdmf-compatible string corresponding to the given *Channel::NumberType* data type

int **numberTypeToPrecision**(*Channel*::NumberType *nt*)
Return the size in bytes of the type represented by the given *Channel::NumberType* data type

Channel::NumberType **infoToNumberType**(**const** std::string &*str*, int *precision*)
reverse of *numberTypeToString()* and *numberTypeToPrecision()*

struct Channel
#include <*channel.h*> Describes one array of data to be dumped or read.

Public Types

enum NumberType
The type of the data contained in one element.

Values:

Float

Double

Int

Int64

enum NeedShift
If the data depends on the coordinates.

Values:

True

False

using DataForm = std::variant<Scalar, Vector, Tensor6, Tensor9, *Quaternion*, Triangle, Vector4, RigidMotion, *Poly*
The topology of one element in the channel.

Public Functions

int **nComponents**() **const**
Number of component in each element (e.g. Vector has 3)

int **precision**() **const**
Number of bytes of each component in one element.

Public Members

std::string **name**
Name of the channel.

void ***data**
pointer to the data that needs to be dumped

DataForm **dataForm**
topology of one element

NumberType **numberType**
data type (enum version)

TypeDescriptor **type**
data type (variant version)

NeedShift **needShift**
wether the data depends on the coordinates or not

struct Polyline
#include <channel.h> Sequence of positions on a chain.

Public Members

int **numVertices**
Number of vertices fora each polyline.

References

[alexeev2020] Alexeev, Dmitry, et al. "Mirheo: High-performance mesoscale simulations for microfluidics." Computer Physics Communications 254 (2020): 107298.

[economides2021] Economides, Athena, et al. "Hierarchical Bayesian Uncertainty Quantification for a Model of the Red Blood Cell." Physical Review Applied 15.3 (2021): 034062.

[amoudruz2021] Amoudruz, Lucas, and Petros Koumoutsakos. "Independent Control and Path Planning of Microswimmers with a Uniform Magnetic Field." Advanced Intelligent Systems (2021): 2100183.

[Fedosov2010] Fedosov, D. A.; Caswell, B. & Karniadakis, G. E. A multiscale red blood cell model with accurate mechanics, rheology, and dynamics Biophysical journal, Elsevier, 2010, 98, 2215-2225

[kantor1987] Kantor, Y. & Nelson, D. R. Phase transitions in flexible polymeric surfaces Physical Review A, APS, 1987, 36, 4020

[Juelicher1996] Juelicher, Frank, and Reinhard Lipowsky. Shape transformations of vesicles with intramembrane domains. Physical Review E 53.3 (1996): 2670.

[Bian2020] Bian, Xin, Sergey Litvinov, and Petros Koumoutsakos. Bending models of lipid bilayer membranes: Spontaneous curvature and area-difference elasticity. Computer Methods in Applied Mechanics and Engineering 359 (2020): 112758.

[Lim2008] Lim HW, Gerald, Michael Wortis, and Ranjan Mukhopadhyay. Red blood cell shapes and shape transformations: newtonian mechanics of a composite membrane: sections 2.1–2.4. Soft Matter: Lipid Bilayers and Red Blood Cells 4 (2008): 83-139.

[Groot1997] Groot, R. D., & Warren, P. B. (1997). Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulations. J. Chem. Phys., 107(11), 4423-4435. *doi <https://doi.org/10.1063/1.474784>*

[Warren2003] Warren, P. B. "Vapor-liquid coexistence in many-body dissipative particle dynamics." Physical Review E 68.6 (2003): 066702.

[Hu2006] Hu, X. Y., and N. A. Adams. "Angular-momentum conservative smoothed particle dynamics for incompressible viscous flows." Physics of Fluids 18.10 (2006): 101702.

[Bian2012] Bian, Xin, et al. "Multiscale modeling of particle in suspension with smoothed dissipative particle dynamics." Physics of Fluids 24.1 (2012): 012002.

[bergou2008] Bergou, M.; Wardetzky, M.; Robinson, S.; Audoly, B. & Grinspun, E. Discrete elastic rods ACM transactions on graphics (TOG), 2008, 27, 63

Python Module Index

m

mmirheo.BelongingCheckers, 49
mmirheo.Bouncers, 63
mmirheo.InitialConditions, 45
mmirheo.Integrators, 51
mmirheo.Interactions, 54
mmirheo.ParticleVectors, 35
mmirheo.Plugins, 72
mmirheo.Utils, 85
mmirheo.Walls, 66