HOOMD-blue Documentation

Release 2.5.0

The Regents of the University of Michigan

Feb 06, 2019

Concepts

Welcome to the reference documentation for HOOMD-blue!

The HOOMD examples and tutorials complement this documentation. Read the HOOMD-blue tutorial online

On laptops/workstations, you can install stable binaries with conda. If you haven't already, download and install miniconda. Then add the conda-forge channel and install HOOMD-blue:

```
$ conda config --add channels conda-forge
$ conda install hoomd
```

If you have already installed hoomd in conda, you can upgrade to the latest version:

```
$ conda update --all
```

On clusters, use singularity containers or compile HOOMD from source so that you are using the right MPI version to take advantage of the high performance network. Your cluster may also require a specific version of CUDA.

CHAPTER 1

Compiling HOOMD-blue

1.1 Software Prerequisites

HOOMD-blue requires a number of prerequisite software packages and libraries.

- **Required:**
  - Git >= 1.7.0
  - Python >= 2.7
  - numpy >= 1.7
  - CMake >= 2.8.0
  - C++ 11 capable compiler (tested with gcc 4.8, 4.9, 5.4, 6.4, 7.0, 8.0, clang 3.8, 5.0, 6.0)
- **Optional:**
  - NVIDIA CUDA Toolkit >= 8.0
  - Intel Threaded Building Blocks >= 4.3
  - MPI (tested with OpenMPI, MVAPICH)
  - LLVM >= 3.6, <= 7.0.0
- **Useful developer tools**
  - Doxygen >= 1.8.5

1.1.1 Software prerequisites on clusters

Most cluster administrators provide versions of Git, Python, NumPy, MPI, and CUDA as modules. You will need to consult the documentation or ask the system administrators for instructions to load the appropriate modules.

1.1.2 Prerequisites on workstations

On a workstation, use the system's package manager to install all of the prerequisites. Some Linux distributions separate `-dev` and normal packages, you need the development packages to build hoomd.

Mac OS systems do not come with a package manager or the necessary prerequisites. Use [macports](https://www.macports.org/) or [homebrew](https://brew.sh/) to install them. You will need to install XCode (free) through the Mac app store to supply the C++ compiler.

1.1.3 Installing prerequisites with conda

Caution: *Not recommended.* Conda is very useful as a delivery platform for stable binaries, but there are many pitfalls when using it to provide development prerequisites.

Despite this warning: many users wish to use conda to those provide development prerequisites. There are a few additional steps required to build hoomd against a conda software stack, as you must ensure that all libraries (mpi, python, etc...) are linked from the conda environment. First, install miniconda. Then, uninstall the hoomd binaries if you have them installed and install the prerequisite libraries and tools:

```
# if using linux
conda install sphinx git mpich2 numpy cmake pkg-config
# if using mac
conda install sphinx git numpy cmake pkg-config
```

Check the CMake configuration to ensure that it finds python, numpy, and MPI from within the conda installation. If any of these library or include files reference directories other than your conda environment, you will need to set the appropriate setting for `PYTHON_EXECUTABLE`, etc...

Note: The `mpich2` package is not available on Mac. Without it, HOOMD will build without MPI support.

Warning: On Mac OS, installing gcc with conda is not sufficient to build HOOMD. Update XCode to the latest version using the Mac OS app store.

1.2 Compile HOOMD-blue

Set the environment variable SOFTWARE_ROOT to the location you wish to install HOOMD:

```
$ export SOFTWARE_ROOT=/path/to/prefix
```

Clone the git repository to get the source:

```
$ git clone --recursive https://bitbucket.org/glotzer/hoomd-blue
```

By default, the *maint* branch will be checked out. This branch includes all bug fixes since the last stable release. HOOMD-blue uses submodules, you the `--recursive` option to clone instructs git to fetch all of the submodules. When you later update this git repository with `git pull`, run `git submodule update` update all of the submodules.

Configure:

```
$ cd hoomd-blue
$ mkdir build
$ cd build
$ cmake ../ -DCMAKE_INSTALL_PREFIX=${SOFTWARE_ROOT}/lib/python
```

By default, HOOMD configures a *Release* optimized build type for a generic CPU architecture and with no optional libraries. Specify `-DCMAKE_CXX_FLAGS=-march=native -DCMAKE_C_FLAGS=-march=native` (or whatever is the appropriate option for your compiler) to enable optimizations specific to your CPU. Specify `-DENABLE_CUDA=ON` to compile code for the GPU (requires CUDA) and `-DENABLE_MPI=ON` to enable parallel simulations with MPI. See the build options section below for a full list of options:

```
$ cmake ../ -DCMAKE_INSTALL_PREFIX=${SOFTWARE_ROOT}/lib/python -DCMAKE_CXX_FLAGS=-
↪march=native -DCMAKE_C_FLAGS=-march=native -DENABLE_CUDA=ON -DENABLE_MPI=ON
```

Compile:

```
$ make -j4
```

Run:

```
$ make test
```

to test your build. If you built with CUDA support, you need a GPU for all tests to pass.

Attention: On a cluster, run `make test` within a job on a GPU compute node.

To install a stable version for general use, run:

```
make install
```

Then set your PYTHONPATH so that python can find hoomd:

```
export PYTHONPATH=$PYTHONPATH:${SOFTWARE_ROOT}/lib/python
```

1.3 Build options

Here is a list of all the build options that can be changed by CMake. To changes these settings, cd to your *build* directory and run:

```
$ ccmake .
```

After changing an option, press *c* to configure then press *g* to generate. The makefile/IDE project is now updated with the newly selected options. Alternately, you can set these parameters on the command line with cmake:

```
cmake $HOME/devel/hoomd -DENABLE_CUDA=on
```

Options that specify library versions only take effect on a clean invocation of cmake. To set these options, first remove *CMakeCache.txt* and then run cmake and specify these options on the command line:

- **PYTHON_EXECUTABLE - Specify which python to build against. Example: /usr/bin/python2.**
  - Default: `python3` or `python` detected on `$PATH`
- **CUDA_TOOLKIT_ROOT_DIR - Specify the root direction of the CUDA installation.**

  - Default: location of `nvcc` detected on `$PATH`

- **MPI_HOME (env var) - Specify the location where MPI is installed.**
  - Default: location of `mpicc` detected on the `$PATH`

Other option changes take effect at any time. These can be set from within *ccmake* or on the command line:

- **CMAKE_INSTALL_PREFIX** - Directory to install the hoomd python module. All files will be under ${CMAKE_INSTALL_PREFIX}/hoomd
- **BUILD_CGCMM** - Enables building the cgcmm component
- **BUILD_DEPRECATED** - Enables building the deprecated component
- **BUILD_HPMC** - Enables building the hpmc component.
- **BUILD_MD** - Enables building the md component
- **BUILD_METAL** - Enables building the metal component
- **BUILD_TESTING** - Enables the compilation of unit tests
- **CMAKE_BUILD_TYPE - sets the build type (case sensitive)**
  - **Debug** - Compiles debug information into the library and executables. Enables asserts to check for programming mistakes. HOOMD-blue will run slow when compiled in Debug mode, but problems are easier to identify.
  - **RelWithDebInfo** - Compiles with optimizations and debug symbols. Useful for profiling benchmarks.
  - **Release** - (default) All compiler optimizations are enabled and asserts are removed. Recommended for production builds: required for any benchmarking.
- **ENABLE_CUDA** - Enable compiling of the GPU accelerated computations using CUDA. Defaults *on* if the CUDA toolkit is found. Defaults *off* if the CUDA toolkit is not found.
- **ENABLE_DOXYGEN** - enables the generation of developer documentation (Defaults *off*)
- **SINGLE_PRECISION - Controls precision**
  - When set to **ON**, all calculations are performed in single precision.
  - When set to **OFF**, all calculations are performed in double precision.
- **ENABLE_HPMC_MIXED_PRECISION - Controls mixed precision in the hpmc component. When on, single precision is** in expensive shape overlap checks.
- **ENABLE_MPI - Enable multi-processor/GPU simulations using MPI**
  - When set to **ON** (default if any MPI library is found automatically by CMake), multi-GPU simulations are supported
  - When set to **OFF**, HOOMD always runs in single-GPU mode
- **ENABLE_MPI_CUDA - Enable CUDA-aware MPI library support**
  - Requires a MPI library with CUDA support to be installed
  - When set to **ON** (default if a CUDA-aware MPI library is detected), HOOMD-blue will make use of the capability of the MPI library to accelerate CUDA-buffer transfers
  - When set to **OFF**, standard MPI calls will be used
  - *Warning:* Manually setting this feature to ON when the MPI library does not support CUDA may result in a crash of HOOMD-blue

- **ENABLE_TBB - Enable support for Intel's Threading Building Blocks (TBB)**
  - Requires TBB to be installed
  - When set to **ON**, HOOMD will use TBB to speed up calculations in some classes on multiple CPU cores
- **UPDATE_SUBMODULES** - When ON (the default), execute `git submodule update --init` whenever cmake runs.
- **COPY_HEADERS** - When ON (OFF is default), copy header files into the build directory to make it a valid plugin build source

These options control CUDA compilation:

- **CUDA_ARCH_LIST** - A semicolon separated list of GPU architecture to compile in.
- **NVCC_FLAGS** - Allows additional flags to be passed to nvcc.

CHAPTER 2

Command line options

2.1 Overview

Arguments are processed in *hoomd.context.initialize()*. Call *hoomd.context.initialize()* immediately after importing hoomd so that the requested MPI and GPU options can be initialized as early as possible.

There are two ways to specify arguments.

1. On the command line: `python script.py [options]`:

```
import hoomd
hoomd.context.initialize()
```

2. Within your script:

```
import hoomd
hoomd.context.initialize("[options]")
```

With no arguments, *hoomd.context.initialize()* will attempt to parse **all** arguments from the command line, whether it understands them or not. When you pass a string, it ignores the command line (`sys.argv`) and parses the given string as if it were issued on the command line. In jupyter notebooks, use `context.initialize("")` to avoid errors from jupyter specific command line arguments.

2.2 Options

- **no options given**

  hoomd will automatically detect the fastest GPU and run on it, or fall back on the CPU if no GPU is found.

- **-h, –help**

  print a description of all the command line options

- **–mode**={cpu | gpu}

  force hoomd to run either on the cpu or gpu

- **–gpu**=#

  specify the GPU id or comma-separated list of GPUs (with NVLINK) that hoomd will use. Implies `--mode=gpu`.

- **–ignore-display-gpu**

  prevent hoomd from using any GPU that is attached to a display

- **–minimize-cpu-usage**

  minimize the CPU usage of hoomd when it runs on a GPU at reduced performance

- **–gpu_error_checking**

  enable error checks after every GPU kernel call

- **–notice-level**=#

  specifies the level of notice messages to print

- **–msg-file**=filename

  specifies a file to write messages (the file is overwritten)

- **–single-mpi**

  allow single-threaded HOOMD builds in MPI jobs

- **–user**

  user options

- ***MPI only options***
  - **–nx**=#

    Number of domains along the x-direction

  - **–ny**=#

    Number of domains along the y-direction

  - **–nz**=#

    Number of domains along the z-direction

  - **–linear**

    Force a slab (1D) decomposition along the z-direction

  - **–nrank**=#

    Number of ranks per partition

  - **–shared-msg-file**=prefix

    specifies the prefix of files to write per-partition output to (filename: *prefix.<partition_id>*)

- ***Option available only when compiled with TBB support***
  - **–nthreads**=#

    Number of TBB threads to use, by default use all CPUs in the system

2.3 Detailed description

2.3.1 Control hoomd execution

HOOMD-blue can run on the CPU or the GPU. To control which, set the `--mode` option on the script command line. Valid settings are `cpu` and `gpu`:

```
python script.py --mode=cpu
```

When `--mode` is set to `gpu` and no other options are specified, hoomd will choose a GPU automatically. It will prioritize the GPU choice based on speed and whether it is attached to a display. Unless you take steps to configure your system (see below), then running a second instance of HOOMD-blue will place it on the same GPU as the first. HOOMD-blue will run correctly with more than one simulation on a GPU as long as there is enough memory, but at reduced performance.

You can select the GPU on which to run using the `--gpu` command line option:

```
python script.py --gpu=1
```

Note: `--gpu` implies `--mode=gpu`. To find out which id is assigned to each GPU in your system, download the CUDA SDK for your system from http://www.nvidia.com/object/cuda_get.html and run the *deviceQuery* sample.

If you run a script without any options:

```
python script.py
```

hoomd first checks if there are any GPUs in the system. If it finds one or more, it makes the same automatic choice described previously. If none are found, it runs on the CPU.

2.3.2 Multi-GPU (and multi-CPU) execution with MPI

HOOMD-blue uses MPI domain decomposition for parallel execution. Execute python with `mpirun`, `mpiexec`, or whatever the appropriate launcher is on your system. For more information, see *MPI domain decomposition*:

```
mpirun -n 8 python script.py
```

All command line options apply to MPI execution in the same way as single process runs.

When `n > 1` and no explicit GPU is specified, HOOMD uses the the local MPI rank to assign GPUs to ranks on each node. This is the default behavior and works on most cluster schedulers.

2.3.3 Multi-GPU execution with NVLINK

You can run HOOMD on multiple GPUs in the same compute node that are connected with NVLINK. To find out if your node supports it, run:

```
nvidia-smi -m topo
```

If the GPUs *are* connected by NVLINK, launch HOOMD with:

```
python script.py --gpu=0,1,2
```

to execute on GPUs 0,1 and 2. For multi-GPU execution it is required that all GPUs have the same compute capability >= 6.0. Not all kernels are currently NVLINK enabled; performance may depend on the subset of features used.

Multi-GPU execution with NVLINK may be combined with MPI parallel execution (see above). It is especially beneficial when further decomposition of the domain using MPI is not feasible or slower, but speed-ups are still possible.

2.3.4 Automatic free GPU selection

You can configure your system for HOOMD-blue to choose free GPUs automatically when each instance is run. To utilize this capability, the system administrator (root) must first use the `nvidia-smi` utility to enable the compute-exclusive mode on all GPUs in the system. With this mode enabled, running hoomd with no options or with the `--mode=gpu` option will result in an automatic choice of the first free GPU from the prioritized list.

The compute-exclusive mode allows *only* a **single CUDA application** to run on each GPU. If you have 4 compute-exclusive GPUs available in the system, executing a fifth instance of hoomd with `python script.py` will result in the error: `***Error! no CUDA-capable device is available`.

Most compute clusters do not support automatic free GPU selection. Instead the schedulers pin jobs to specific GPUs and bind the host processes to attached cores. In this case, HOOMD uses the rank-based GPU selection described above. HOOMD only applies exclusive mode automatic GPU selection when built without MPI support (ENABLE_MPI=off) or executing on a single rank.

2.3.5 Minimize the CPU usage of HOOMD-blue

When hoomd is running on a GPU, it uses 100% of one CPU core by default. This CPU usage can be decreased significantly by specifying the `--minimize-cpu-usage` command line option:

```
python script.py --minimize-cpu-usage
```

Enabling this option incurs a 10% overall performance reduction, but the CPU usage of hoomd is reduced to only 10% of a single CPU core.

2.3.6 Prevent HOOMD-blue from running on the display GPU

Running hoomd on the display GPU works just fine, but it does moderately slow the simulation and causes the display to lag. If you wish to prevent hoomd from running on the display, add the `--ignore-display-gpu` command line flag:

```
python script.py --ignore-display-gpu
```

2.3.7 Enable error checking on the GPU

Detailed error checking is off by default to enable the best performance. If you have trouble that appears to be caused by the failure of a calculation to run on the GPU, you should run with GPU error checking enabled to check for any errors returned by the GPU.

To do this, run the script with the `--gpu_error_checking` command line option:

```
python script.py --gpu_error_checking
```

2.3.8 Control message output

You can adjust the level of messages written to `sys.stdout` by a running hoomd script. Set the notice level to a high value to help debug where problems occur. Or set it to a low number to suppress messages. Set it to 0 to remove all notices (warnings and errors are still output):

```
python script.py --notice-level=10
```

All messages (notices, warnings, and errors) can be redirected to a file. The file is overwritten:

```
python script.py --msg-file=messages.out
```

In MPI simulations, messages can be aggregated per partition. To write output for partition 0,1,.. in files `messages.0`, `messages.1`, etc., use:

```
mpirun python script.py --shared-msg-file=messages
```

2.3.9 Set the MPI domain decomposition

When no MPI options are specified, HOOMD uses a minimum surface area selection of the domain decomposition strategy:

```
mpirun -n 8 python script.py
# 2x2x2 domain
```

The linear option forces HOOMD-blue to use a 1D slab domain decomposition, which may be faster than a 3D decomposition when running jobs on a single node:

```
mpirun -n 4 python script.py --linear
# 1x1x4 domain
```

You can also override the automatic choices completely:

```
mpirun -n 4 python script.py --nx=1 --ny=2 --nz=2
# 1x2x2 domain
```

You can group multiple MPI ranks into partitions, to simulate independent replicas:

```
mpirun -n 12 python script.py --nrank=3
```

This sub-divides the total of 12 MPI ranks into four independent partitions, with to which 3 GPUs each are assigned.

2.3.10 User options

User defined options may be passed to a job script via `--user` and retrieved by calling *hoomd.option.get_user()*. For example, if hoomd is executed with:

```
python script.py --gpu=2 --ignore-display-gpu --user="--N=5 --rho=0.5"
```

then *hoomd.option.get_user()* will return `['--N=5', '--rho=0.5']`, which is a format suitable for processing by standard tools such as `optparse`.

2.3.11 Execution with CPU threads (Intel TBB support)

Some classes in HOOMD support CPU threads using Intel's Threading Building Blocks (TBB). TBB can speed up the calculation considerably, depending on the number of CPU cores available in the system. If HOOMD was compiled with support for TBB, the number of threads can be set. On the command line, this is done using:

```
python script.py --mode=cpu --nthreads=20
```

Alternatively, the same option can be passed to *hoomd.context.initialize()*, and the number of threads can be updated any time using *hoomd.option.set_num_threads()* . If no number of threads is specified, TBB by default uses all CPUs in the system. For compatibility with OpenMP, HOOMD also honors a value set in the environment variable **OMP_NUM_THREADS**.

CHAPTER 3

Units

HOOMD-blue stores and computes all values in a system of generic, fully self-consistent set of units. No conversion factors need to be applied to values at every step. For example, a value with units of force comes from dividing energy by distance.

3.1 Fundamental Units

The three fundamental units are:

- distance - $\mathcal{D}$
- energy - $\mathcal{E}$
- mass - $\mathcal{M}$

All other units that appear in HOOMD-blue are derived from these. Values can be converted into any other system of units by assigning the desired units to $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{M}$ and then multiplying by the appropriate conversion factors.

The standard *Lennard-Jones* symbols σ and ϵ are intentionally not referred to here. When you assign a value to ϵ in hoomd, for example, you are assigning it in units of energy: $\epsilon = 5\mathcal{E}$. ϵ is **NOT** the unit of energy - it is a value with units of energy.

3.2 Temperature (thermal energy)

HOOMD-blue accepts all temperature inputs and provides all temperature output values in units of energy: kT, where k is Boltzmann's constant. When using physical units, the value k_B is determined by the choices for distance, energy, and mass. In reduced units, one usually reports the value $T^* = \frac{kT}{\mathcal{E}}$.

Most of the argument inputs in HOOMD take the argument name `kT` to make it explicit. A few areas of the code may still refer to this as `temperature`.

3.3 Charge

The unit of charge used in HOOMD-blue is also reduced, but is not represented using just the 3 fundamental units - the permittivity of free space ε_0 is also present. The units of charge are: $(4\pi\varepsilon_0\mathcal{D}\mathcal{E})^{1/2}$. Divide a given charge by this quantity to convert it into an input value for HOOMD-blue.

3.4 Common derived units

Here are some commonly used derived units:

- time - $\tau = \sqrt{\frac{\mathcal{M}\mathcal{D}^2}{\mathcal{E}}}$
- volume - $\mathcal{D}^3$
- velocity - $\frac{\mathcal{D}}{\tau}$
- momentum - $\mathcal{M}\frac{\mathcal{D}}{\tau}$
- acceleration - $\frac{\mathcal{D}}{\tau^2}$
- force - $\frac{\mathcal{E}}{\mathcal{D}}$
- pressure - $\frac{\mathcal{E}}{\mathcal{D}^3}$

3.5 Example physical units

There are many possible choices of physical units that one can assign. One common choice is:

- distance - $\mathcal{D} = \mathrm{nm}$
- energy - $\mathcal{E} = \mathrm{kJ/mol}$
- mass - $\mathcal{M} = \mathrm{amu}$

Derived units / values in this system:

- time - picoseconds
- velocity - nm/picosecond
- k = 0.00831445986144858 kJ/mol/Kelvin

CHAPTER 4

Periodic boundary conditions

4.1 Introduction

All simulations executed in HOOMD-blue occur in a triclinic simulation box with periodic boundary conditions in all three directions. A triclinic box is defined by six values: the extents L_x, L_y and L_z of the box in the three directions, and three tilt factors xy, xz and yz.

The parameter matrix $\mathbf{h}$ is defined in terms of the lattice vectors $\vec{a}_1$, $\vec{a}_2$ and $\vec{a}_3$:

$$\mathbf{h} \equiv (\vec{a}_1, \vec{a}_2, \vec{a}_3)$$

By convention, the first lattice vector $\vec{a}_1$ is parallel to the unit vector $\vec{e}_x = (1, 0, 0)$. The tilt factor xy indicates how the second lattice vector $\vec{a}_2$ is tilted with respect to the first one. It specifies many units along the x-direction correspond to one unit of the second lattice vector. Similarly, xz and yz indicate the tilt of the third lattice vector $\vec{a}_3$ with respect to the first and second lattice vector.

4.2 Definitions and formulas for the cell parameter matrix

The full cell parameter matrix is:

$$\mathbf{h} = \begin{pmatrix} L_x & xyL_y & xzL_z \\ 0 & L_y & yzL_z \\ 0 & 0 & L_z \end{pmatrix}$$

The tilt factors xy, xz and yz are dimensionless. The relationships between the tilt factors and the box angles α, β and γ are as follows:

$$\begin{aligned} \cos\gamma \equiv \cos(\angle\vec{a}_1, \vec{a}_2) &= \frac{xy}{\sqrt{1+xy^2}} \\ \cos\beta \equiv \cos(\angle\vec{a}_1, \vec{a}_3) &= \frac{xz}{\sqrt{1+xz^2+yz^2}} \\ \cos\alpha \equiv \cos(\angle\vec{a}_2, \vec{a}_3) &= \frac{xy \cdot xz + yz}{\sqrt{1+xy^2}\sqrt{1+xz^2+yz^2}} \end{aligned}$$

Given an arbitrarily oriented lattice with box vectors $\vec{v}_1, \vec{v}_2, \vec{v}_3$, the HOOMD-blue box parameters for the rotated box can be found as follows.

$$
\begin{aligned}
L_x &= v_1 \\
a_{2x} &= \frac{\vec{v}_1 \cdot \vec{v}_2}{v_1} \\
L_y &= \sqrt{v_2^2 - a_{2x}^2} \\
xy &= \frac{a_{2x}}{L_y} \\
L_z &= \vec{v}_3 \cdot \frac{\vec{v}_1 \times \vec{v}_2}{|\vec{v}_1 \times \vec{v}_2|} \\
a_{3x} &= \frac{\vec{v}_1 \cdot \vec{v}_3}{v_1} \\
xz &= \frac{a_{3x}}{L_z} \\
yz &= \frac{\vec{v}_2 \cdot \vec{v}_3 - a_{2x} a_{3x}}{L_y L_z}
\end{aligned}
$$

Example:

```
# boxMatrix contains an arbitrarily oriented right-handed box matrix.
v[0] = boxMatrix[:,0]
v[1] = boxMatrix[:,1]
v[2] = boxMatrix[:,2]
Lx = numpy.sqrt(numpy.dot(v[0], v[0]))
a2x = numpy.dot(v[0], v[1]) / Lx
Ly = numpy.sqrt(numpy.dot(v[1],v[1]) - a2x*a2x)
xy = a2x / Ly
v0xv1 = numpy.cross(v[0], v[1])
v0xv1mag = numpy.sqrt(numpy.dot(v0xv1, v0xv1))
Lz = numpy.dot(v[2], v0xv1) / v0xv1mag
a3x = numpy.dot(v[0], v[2]) / Lx
xz = a3x / Lz
yz = (numpy.dot(v[1],v[2]) - a2x*a3x) / (Ly*Lz)
```

4.3 Initializing a system with a triclinic box

You can specify all parameters of a triclinic box in a GSD file.

You can also pass a *hoomd.data.boxdim* argument to the constructor of any initialization method. Here is an example for *hoomd.deprecated.init.create_random()*:

```
init.create_random(box=data.boxdim(L=18, xy=0.1, xz=0.2, yz=0.3), N=1000))
```

This creates a triclinic box with edges of length 18, and tilt factors $xy = 0.1$, $xz = 0.2$ and $yz = 0.3$.

You can also specify a 2D box to any of the initialization methods:

```
init.create_random(N=1000, box=data.boxdim(xy=1.0, volume=2000, dimensions=2), min_
↪dist=1.0)
```

4.4 Change the simulation box

The triclinic unit cell can be updated in various ways.

4.4.1 Resizing the box

The simulation box can be gradually resized during a simulation run using *hoomd.update.box_resize*.

To update the tilt factors continuously during the simulation (shearing the simulation box with **Lees-Edwards** boundary conditions), use:

```
update.box_resize(xy = variant.linear_interp([(0,0), (1e6, 1)]))
```

This command applies shear in the xy -plane so that the angle between the x and y-directions changes continuously from 0 to $45°$ during 10^6 time steps.

hoomd.update.box_resize can change any or all of the six box parameters.

4.4.2 NPT or NPH integration

In a constant pressure ensemble, the box is updated every time step, according to the anisotropic stresses in the system. This is supported by:

- *hoomd.md.integrate.npt*
- *hoomd.md.integrate.nph*

CHAPTER 5

Rotational degrees of freedom

5.1 Overview

HOOMD-blue natively supports the integration of rotational degrees of freedom. Every particle in a hoomd simulation may have rotational degrees of freedom. When any torque-producing potential or constraint is defined in the system, integrators automatically integrate both the rotational and translational degrees of freedom of the system. Anisotropic integration can also be explicitly enabled or disabled through the `aniso` argument of *hoomd.md.integrate.mode_standard*. *hoomd.md.pair.gb*, *hoomd.dem*, *hoomd.md.constrain.rigid* are examples of potentials and constraints that produce torques on particles.

The integrators detect what rotational degrees of freedom exist per particle. Each particle has a diagonal moment of inertia tensor that specifies the moment of inertia about the 3 principle axes in the particle's local reference frame. Integrators only operate on rotational degrees of freedom about axes where the moment of inertia is non-zero. Ensure that you set appropriate moments of inertia for all particles that have them in the system.

Particles have a number of properties related to rotation accessible using the particle data API (*hoomd.data*):

- orientation - Quaternion to rotate the particle from its base orientation to its current orientation, in the order $(real, imag_x, imag_y, imag_z)$
- angular_momentum - Conjugate quaternion representing the particle's angular momentum
- moment_inertia - principal moments of inertia (I_{xx}, I_{yy}, I_{zz})
- net_torque - net torque on the particle in the global reference frame

GSD files store the orientation, moment of inertia, and angular momentum of particles.

5.2 Quaternions for angular momentum

Particle angular momenta are stored in quaternion form as defined in Kamberaj 2005 : the angular momentum quaternion $\mathbf{P}$ is defined with respect to the orientation quaternion of the particle $\mathbf{q}$ and the angular momentum of the particle,

lifted into pure imaginary quaternion form $\mathbf{S}^{(4)}$ as:

$$\mathbf{P} = 2\mathbf{q} \times \mathbf{S}^{(4)}$$

in other words, the angular momentum vector $\vec{S}$ with respect to the principal axis of the particle is

$$\vec{S} = \frac{1}{2} im(\mathbf{q}^* \times \mathbf{P})$$

where $\mathbf{q}^*$ is the conjugate of the particle's orientation quaternion and $\times$ is quaternion multiplication.

CHAPTER 6

Neighbor lists

6.1 Overview

Neighbor lists accelerate the search for pairs of atoms that are within a certain *cutoff radius* of each other. They are most commonly used in `hoomd.md.pair` to accelerate the calculation of pair forces between atoms. This significantly reduces the number of pairwise distances that are evaluated, which is $O(N^2)$ if all possible pairs are checked. A small *buffer radius* (skin layer) `r_buff` is typically added to the cutoff radius so that the neighbor list can be computed less frequently. The neighbor list must only be rebuilt any time a particle diffuses `r_buff/2`. However, increasing `r_buff` also increases the number of particles that are included in the neighbor list, which slows down the pair force evaluation. A balance can be obtained between the two by optimizing `r_buff`.

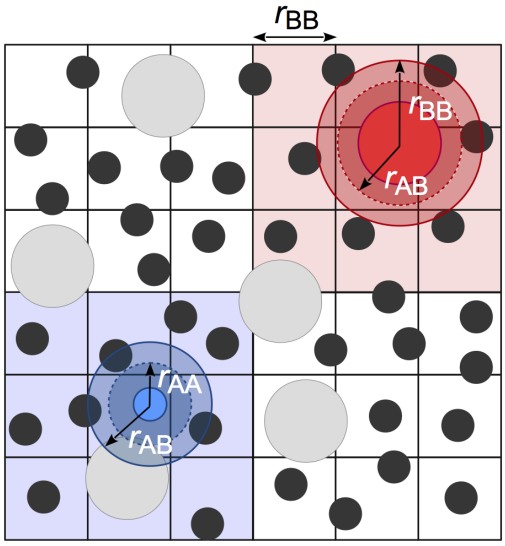
A simple neighbor list is built by checking all possible pairs of atoms periodically, which makes the overall algorithm $O(N^2)$. The neighbor list can be computed more efficiently using an *acceleration structure* which further reduces the complexity of the problem. There are three accelerators implemented in HOOMD-blue:

- *Cell list*
- *Stenciled cell list*
- *LBVH tree*

More details for each can be found below and in M.P. Howard et al. 2016. Each neighbor list style has its own advantages and disadvantages that the user should consider on a case-by-case basis.

6.2 Cell list

The cell-list neighbor list (`hoomd.md.nlist.cell`) spatially sorts particles into bins that are sized by the **largest** cutoff radius of all pair potentials attached to the neighbor list. For example, in the figure below, there are small A particles and large B particles. The bin size is based on the cutoff radius of the largest particles r_{BB}. To find neighbors, each particle searches the 27 cells that are adjacent to its cell, which are shaded around each particle. Binning particles is *O(N)*, and so neighbor search from the cell list is also *O(N)*.

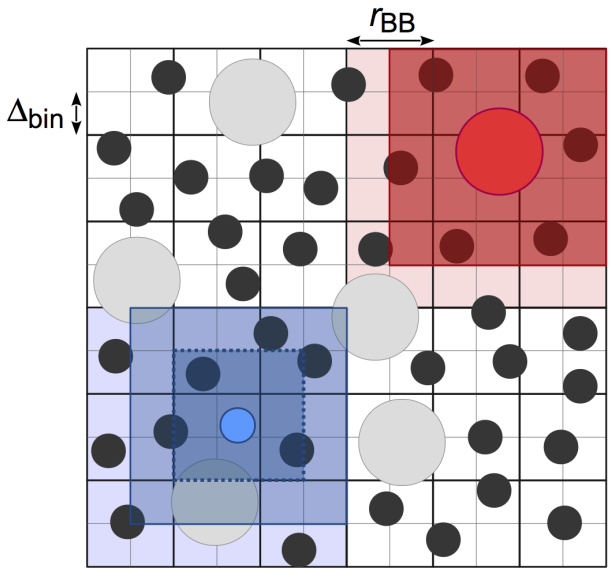


This method is very efficient for systems with nearly monodisperse cutoffs, but performance degrades for large cutoff radius asymmetries due to the significantly increased number of particles per cell and increased search volume. For example, the small A particles, who have a majority of neighbors who are also A particles within cutoff r_{AA} must now search through the full volume defined by r_{BB}. In practice, we have found that this neighbor list style is the best option for most users when the asymmetry between the largest and smallest cutoff radius is less than 2:1.

Note: Users may find that the cell-list neighbor list consumes a significant amount of memory, especially on CUDA devices. One cause of this can be non-uniform density distributions because the memory allocated for the cell list is proportional the maximum number of particles in any cell. Another common cause is large system volumes combined with small cutoffs, which results in a very large number of cells in the system. In these cases, consider using *hoomd.md.nlist.stencil* or *hoomd.md.nlist.tree*.

6.3 Stenciled cell list

Performance of the simple cell-list can be improved in the size asymmetric case by basing the bin size of the cell list on the **smallest** cutoff radius of all pair potentials attached to the neighbor list (P.J. in't Veld et al. 2008). From the previous example, the bin size is now based on r_{AA}. A *stencil* is then constructed on a per-pair basis that defines the bins to search. Some particles can now be excluded without distance check if they lie in bins outside the stencil. The small A particles only need to distance check other A particles in the dark blue cells (dashed outline). This reduces both the number of distances evaluations and the amount of particle data that is read.

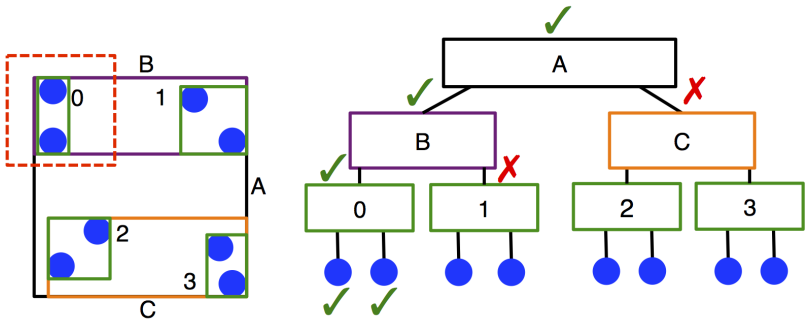


We have found that the stenciled cell list (`hoomd.md.nlist.stencil`) performs well for size asymmetric systems that have comparable concentrations of both small and large particles. Performance may degrade when the fraction of large particles is low (< 20%). The memory consumed by the stenciled cell list is typically much lower than that used for a comparable simple cell list because of the way the stencils constructed to query the cell list. However, this comes at the expense of higher register usage on CUDA devices, which may lead to reduced performance compared to the simple cell list in some cases depending on your CUDA device's architecture.

Note: Users may still find that the stenciled cell list consumes a significant amount of memory for systems with large volumes and small cutoffs. In this case, the bin size should be made larger (possibly at the expense of performance), or `hoomd.md.nlist.tree` should be used instead.

6.4 LBVH tree

Linear bounding volume hierarchies (LBVHs) are an entirely different approach to accelerating the neighbor search. LBVHs are binary tree structures that partition the system based on *objects* rather than space (see schematic below). This means that the memory they require scales with the number of particles in the system rather than the system volume, which may be particularly advantageous for large, sparse systems. Because of their lightweight memory footprint, LBVHs can also be constructed per-type, and this makes searching the trees very efficient in size asymmetric systems. The LBVH algorithm is $O(N \log N)$ to search the tree.



We have found that LBVHs (`hoomd.md.nlist.tree`) are very useful for systems with size asymmetry greater

than 2:1 between the largest and smallest cutoffs, and when the fraction of large particles is dilute (< 20%). These conditions are typical of many colloidal systems. Additionally, LBVHs can be used advantageously in sparse systems or systems with large volumes, where they have less overhead and memory demands than cell lists.

6.5 Multiple neighbor lists

Multiple neighbor lists can be created to accelerate simulations where there is significant disparity in the pairwise cutoffs between pair potentials. If one pair force has a maximum cutoff radius much smaller than another pair force, the pair force calculation for the short cutoff will be slowed down considerably because many particles in the neighbor list will have to be read and skipped because they lie outside the shorter cutoff. Attaching each potential to a different neighbor list may improve performance of the pair force calculation at the expense of duplicate computation of the neighbor list. When using multiple neighbor lists, it may be advantageous to adopt two different neighbor list styles. For example, in a colloidal suspension of a small number of large colloids dispersed in many solvent particles, a modest performance gain may be achieved by computing the solvent-solvent neighbors using *hoomd.md.nlist.cell*, but the solvent-colloid and colloid-colloid interactions using *hoomd.md.nlist.tree*. Particles can be excluded from neighbor lists by setting their cutoff radius to `False` or a negative value.

CHAPTER 7

MPI domain decomposition

7.1 Overview

HOOMD-blue supports multi-GPU (and multi-CPU) simulations using MPI. It uses a spatial domain decomposition approach similar to the one used by LAMMPS. Every GPU is assigned a sub-domain of the simulation box, the dimensions of which are calculated by dividing the lengths of the simulation box by the number of processors per dimension. These domain boundaries can also be adjusted to different fractional widths while still maintaining a 3d grid, which can be advantageous in systems with density gradients. The product of the number of processors along all dimensions must equal the number of processors in the MPI job. As in single-GPU simulations, there is a one-to-one mapping between host CPU cores (MPI ranks) and the GPUs.

Job scripts do not need to be modified to take advantage of multi-GPU execution. However, not all features are available in MPI mode. The list of single-GPU only features can be found below.

See J. Glaser et. al. 2015 for more implementation details.

7.2 Compilation

For detailed compilation instructions, see *Compile HOOMD-blue*.

Compilation flags pertinent to MPI simulations are:

- **ENABLE_MPI** (to enable multi-GPU simulations, must be set to b ON)
- **ENABLE_MPI_CUDA** (optional, enables CUDA-aware MPI library support, see below)

7.3 Usage

To execute a hoomd job script on multiple GPUs, run:

```
mpirun -n 8 python script.py --mode=gpu
```

This will execute HOOMD on 8 processors. HOOMD automatically detects which GPUs are available and assigns them to MPI ranks. The syntax and name of the `mpirun` command may be different between different MPI libraries and system architectures, check with your system documentation to find out what launcher to use. When running on multiple nodes, the job script must be available to all nodes via a network file system.

HOOMD chooses the best spatial sub-division according to a minimum-area rule. If needed, the dimensions of the decomposition be specified using the **linear**, **nx**, **ny** and **nz** *Command line options*. If your intention is to run HOOMD on a single GPU, you can invoke HOOMD with no MPI launcher:

```
python script.py
```

instead of giving the `-n 1` argument to `mpirun`.

Warning: Some cluster environments do not allow this and require the MPI launcher be used even for single rank jobs.

HOOMD-blue can also execute on many **CPU cores** in parallel:

```
mpirun -n 16 python script.py --mode=cpu
```

7.4 GPU selection in MPI runs

HOOMD-blue uses information from `mpirun` to determine the *local rank* on a node (0,1,2,...). Each rank will use the GPU id matching the local rank modulus the number of GPUs on the node. In this mode, do not run more ranks per node than there are GPUs or you will oversubscribe the GPUs. This selection mechanism selects GPUs from within the set of GPUs provided by the cluster scheduler.

In some MPI stacks, such as Intel MPI, this information is unavailable and HOOMD falls back on selecting `gpu_id = global_rank % num_gpus_on_node` and issues a notice message. This mode only works on clusters where scheduling is performed by node (not by core) and there are a uniform number of GPUs on each node.

In any case, a status message is printed on startup that lists which ranks are using which GPU ids. You can use this to verify proper GPU selection.

7.5 Best practices

HOOMD-blue's multi-GPU performance depends on many factors, such as the model of the actual GPU used, the type of interconnect between nodes, whether the MPI library supports CUDA, etc. Below we list some recommendations for obtaining optimal performance.

7.5.1 System size

Performance depends greatly on system size. Runs with fewer particles per GPU will execute slower due to communications overhead. HOOMD-blue has decent strong scaling down to small numbers of particles per GPU, but to obtain high efficiency (more than 60%) typical benchmarks need 100,000 or more particles per GPU. You should benchmark your own system of interest with short runs to determine a reasonable range of efficient scaling behavior. Different potentials and/or cutoff radii can greatly change scaling behavior.

7.5.2 CUDA-aware MPI libraries

The main benefit of using a CUDA-enabled MPI library is that it enables intra-node peer-to-peer (P2P) access between several GPUs on the same PCIe root complex, which increases bandwidth. Secondarily, it may offer some additional optimization for direct data transfer between the GPU and a network adapter. To use these features with an MPI library that supports it, set `ENABLE_MPI_CUDA` to **ON** for compilation.

Currently, we recommend building with `ENABLE_MPI_CUDA` **OFF**. On MPI libraries available at time of release, enabling `ENABLE_MPI_CUDA` cuts performance in half. Systems with *GPUDirect RDMA* enabled improve on this somewhat, but even on such systems typical benchmarks still run faster with `ENABLE_MPI_CUDA` **OFF**.

7.5.3 GPUDirect RDMA

HOOMD does support *GPUDirect RDMA* with network adapters that support it (i.e. Mellanox) and compatible GPUs (Kepler), through a CUDA-aware MPI library (i.e. OpenMPI 1.7.5 or MVAPICH 2.0b GDR). On HOOMD's side, nothing is required beyond setting `ENABLE_MPI_CUDA` to **ON** before compilation. On the side of the MPI library, special flags may need to be set to enable *GPUDirect RDMA*, consult the documentation of your MPI library for that.

7.5.4 Slab decomposition

For small numbers of GPUs per job (typically <= 8 for cubic boxes) that are non-prime, the performance may be increased by using a slab decomposition. A one-dimensional decomposition is enforced if the `--linear` command line option (*Command line options*) is given.

7.5.5 Neighbor list buffer length (r_buff)

The optimum value of the **r_buff** value of the neighbor list may be different between single- and multi-GPU runs. In multi-GPU runs, the buffering length also determines the width of the ghost layer runs and sets the size of messages exchanged between the processors. To determine the optimum value, use *hoomd.md.nlist.nlist.tune()*. command with the same number of MPI ranks that will be used for the production simulation.

7.5.6 Running with multiple partitions (–nrank command-line option)

HOOMD-blue supports simulation of multiple independent replicas, with the same number of GPUs per replica. To enable multi-replica mode, and to partition the total number of ranks **N** into **p = N/n** replicas, where **n** is the number of GPUs per replica, invoke HOOMD-blue with the **–nrank=n** command line option (see *Command line options*).

Inside the command script, the current partition can be queried using *hoomd.comm.get_partition()*.

7.6 Dynamic load balancing

HOOMD-blue supports non-uniform domain decomposition for systems with density gradients. A static domain decomposition on a regular 3d grid but non-uniform widths can be constructed using *hoomd.comm.decomposition*. Here, either the number of processors in a uniform decomposition or the fractional widths of $n-1$ domains can be set. Dynamic load balancing can be applied to any domain decomposition either one time or periodically throughout the simulation using *hoomd.update.balance*. The domain boundaries are adjusted to attempt to place an equal number of particles on each rank. The overhead from periodically updating the domain boundaries is reasonably small, so most simulations with non-uniform particle distributions will benefit from periodic dynamic load balancing.

7.6.1 Troubleshooting

- **My simulation does not run significantly faster on exactly two GPUs compared to one GPU.** This is expected. HOOMD uses special optimizations for single-GPU runs, which means that there is no overhead due to MPI calls. The communication overhead can be 20-25% of the total performance, and is only incurred when running on more than one GPU.
- **I get a message saying "Bond incomplete"** In multi-GPU simulations, there is an implicit restriction on the maximal length of a single bond. A bond cannot be longer than half the local domain size. If this happens, an error is thrown. The problem can be fixed by running HOOMD on fewer processors, or with a larger box size.
- **Simulations with large numbers of nodes are slow.** In simulations involving many nodes, collective MPI calls can take a significant portion of the run time. To find out if these are limiting you, run the simulation with the `profile=True` option to the *`hoomd.run()`* command. One reason for slow performance can be the distance check, which, by default, is applied every step to check if the neighbor list needs to be rebuild. It requires synchronization between all MPI ranks and is therefore slow. See *`hoomd.md.nlist.nlist.set_params()`* to increase the interval (**check_period**) between distance checks, to improve performance.
- **My simulation crashes on multiple GPUs when I set ENABLE_MPI_CUDA=ON** First, check that cuda-aware MPI support is enabled in your MPI library. Usually this is determined at compile time of the MPI library. For MVAPICH2, HOOMD automatically sets the required environment variable **MV2_USE_CUDA=1**. If you are using *GPUDirect RDMA* in a dual-rail configuration, special considerations need to be taken to ensure correct GPU-core affinity, not doing so may result in crashing or slow simulations.

CHAPTER 8

Autotuner

8.1 Overview

HOOMD-blue uses run-time autotuning to optimize GPU performance. Every time you run a hoomd script, hoomd starts autotuning values from a clean slate. Performance may vary during the first time steps of a simulation when the autotuner is scanning through possible values. Once the autotuner completes the first scan, performance will stabilize at optimized values. After approximately *period* steps, the autotuner will activate again and perform a quick scan to update timing data. With continual updates, tuned parameters will adapt to simulation conditions - so as you switch your simulation from NVT to NPT, compress the box, or change forces, the autotuner will keep everything running at optimal performance.

8.2 Benchmarking hoomd

Care must be taken in performance benchmarks. The initial warm up time of the tuner is significant, and performance measurements should only be taken after warm up. The total time needed for a scan may vary from system to system depending on parameters. For example, the `lj-liquid-bmark` script requires 10,000 steps for the initial tuning pass (2000 for subsequent updates). You can monitor the autotuner with the command line option `--notice-level=4`. Each tuner will print a status message when it completes the warm up period. The `nlist_binned` tuner will most likely take the longest time to complete.

When obtaining profile traces, disable the autotuner after the warm up period so that it does not decide to re-tune during the profile.

8.3 Controlling the autotuner

Default parameters should be sufficient for the autotuner to work well in almost any situation. Controllable parameters are:

- `period`: Approximate number of time steps before retuning occurs

- `enabled`: Boolean to control whether the autotuner is enabled. If disabled after the warm up period, no retuning will occur, but it will still use the found optimal values. If disabled during the warm up period, a warning is issued and the system will use non-optimal values.

The defaults are `period=100000`, and `enabled=True`. Other parameters can be set by calling *hoomd.option.set_autotuner_params()*. This period is short enough to pick up changes after just a few hundred thousand time steps, but long enough so that the performance loss of occasionally running at nonoptimal parameters is small (most per time step calls can complete tuning in less than 200 time steps).

CHAPTER 9

Restartable jobs

9.1 Overview

The ideal restartable job is a single job script that can be resubmitted over and over again to the job queue system. Each time the job starts, it picks up where it left off the last time and continues running until it is done. You can put all the logic necessary to do this in the hoomd python script itself, keeping the submission script simple:

```
# job.sh
mpirun hoomd run.py
```

With a properly configured python script, `qsub job.sh` is all that is necessary to submit the first run, continue a previous job that exited cleanly, and continue one that was prematurely killed.

9.2 Elements of a restartable script

A restartable needs to:

- Choose between an initial condition and the restart file when initializing.
- Write a restart file periodically.
- Set a phase for all analysis, dump, and update commands.
- Use *hoomd.run_upto()* to skip over time steps that were run in previous job submissions.
- Use only commands that are restart capable.
- Optionally ensure that jobs cleanly exit before the job walltime limit.

9.2.1 Choose the appropriate initialization file

Let's assume that the initial condition for the simulation is in `init.gsd`, and `restart.gsd` is saved periodically as the job runs. A single *hoomd.init.read_gsd()* command will load the restart file if it exists, otherwise it will

load the initial file. It is easiest to think about dump files, temperature ramps, etc... if `init.gsd` is at time step 0:

```
init.read_gsd(filename='init.gsd', restart='restart.gsd')
```

If you generate your initial configuration in python, you will need to add some logic to read `restart.gsd` if it exists or generate if not. This logic is left as an exercise to the reader.

9.2.2 Write restart files

You cannot predict when a hardware failure will cause your job to fail, so you need to save restart files at regular intervals as your run progresses. You will also need periodic restart files at a fast rate if you don't manage wall time to ensure clean job exits.

First, you need to select a restart period. The compute center you run on may offer a tool to help you determine an optimal restart period in minutes. A good starting point is to write a restart file every hour. Based on performance benchmarks, select a restart period in time steps:

```
dump.gsd(filename="restart.gsd", group=group.all(), truncate=True, period=10000,␣
↪phase=0)
```

9.2.3 Use the phase option

Set a a `phase >= 0` for all analysis routines, file dumps, and updaters you use with period > 1 (the default is 0). With `phase >= 0`, these routines will continue to run in a restarted job on the correct timesteps as if the job had not been restarted.

Do not use, `phase=-1`, as then these routines will start running immediately when a restart job begins:

```
dump.dcd(filename="trajectory.dcd", period=1e6, phase=0)
analyze.log(filename='temperature.log', quantities=['temperature'], period=5000,␣
↪phase=0)
zeroer= update.zero_momentum(period=1e6, phase=0)
```

9.2.4 Use run_upto

`hoomd.run_upto()` runs the simulation up to timestep `n`. Use this in restartable jobs to allow them to run a given number of steps, independent of the number of submissions needed to reach that:

```
run_upto(100e6)
```

9.2.5 Use restart capable commands

Most commands in hoomd that output to files are capable of appending to the end of a file so that restarted jobs continue adding data to the file as if the job had never been restarted.

However, not all features in hoomd are capable of restarting. Some are not even capable of appending to files. See the documentation for each individual command you use to tell whether it is compatible with restartable jobs. For those that are restart capable, do not set *overwrite=True*, or each time the job restarts it will erase the file and start writing a new one.

Some analysis routines in HOOMD-blue store internal state and may require a period that is commensurate with the restart period. See the documentation on the individual command you use to see if this is the case.

9.2.6 Cleanly exit before the walltime limit

Job queues will kill your job when it reaches the walltime limit. HOOMD can stop your run before that happens and give your job time to exit cleanly. Set the environment variable `HOOMD_WALLTIME_STOP` to enable this. Any *hoomd.run()* or *hoomd.run_upto()* command will exit before the specified time is reached. HOOMD monitors run performance and tries to ensure that it will end *before* `HOOMD_WALLTIME_STOP`. Set the variable to a unix epoch time. For example in a job script that should run 12 hours, set `HOOMD_WALLTIME_STOP` to 12 hours from now, minus 10 minutes to allow for job cleanup:

```
# job.sh
export HOOMD_WALLTIME_STOP=$((`date +%s` + 12 * 3600 - 10 * 60))
mpirun hoomd run.py
```

When using `HOOMD_WALLTIME_STOP`, *hoomd.run()* will throw the exception `WalltimeLimitReached` when it exits due to the walltime limit. Catch this exception so that your job can exit cleanly. Also, make sure to write out a final restart file at the end of your job so you have the final system state to continue from. Set the `limit_multiple` for the run to the restart period so that any analyzers that must run commensurate with the restart file have a chance to run. If you don't use any such commands, you can omit `limit_multiple` and the run will be free to end on any time step:

```
gsd_restart = dump.gsd(filename="restart.gsd", group=group.all(), truncate=True,␣
↪period=10000, phase=0)

try:
    run_upto(1e6, limit_multiple=10000)

    # Perform additional actions here that should only be done after the job has␣
↪completed all time steps.
except WalltimeLimitReached:
    # Perform actions here that need to be done each time you run into the wall clock␣
↪limit, or just pass
    pass

gsd_restart.write_restart()
# Perform additional job cleanup actions here. These will be executed each time the␣
↪job ends due to reaching the
# walltime limit AND when the job completes all of its time steps.
```

9.3 Examples

9.3.1 Simple example

Here is a simple example that puts all of these elements together:

```
# job.sh
export HOOMD_WALLTIME_STOP=$((`date +%s` + 12 * 3600 - 10 * 60))
mpirun hoomd run.py
```

```
# run.py
from hoomd import *
from hoomd import md
context.initialize()
```

(continues on next page)

(continued from previous page)

```
init.read_gsd(filename='init.gsd', restart='restart.gsd')

lj = md.pair.lj(r_cut=2.5)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)

md.integrate.mode_standard(dt=0.005)
md.integrate.nvt(group=group.all(), T=1.2, tau=0.5)

gsd_restart = dump.gsd(filename="restart.gsd", group=group.all(), truncate=True,␣
↪period=10000, phase=0)
dump.dcd(filename="trajectory.dcd", period=1e5, phase=0)
analyze.log(filename='temperature.log', quantities=['temperature'], period=5000,␣
↪phase=0)

try:
    run_upto(1e6, limit_multiple=10000)
except WalltimeLimitReached:
    pass

gsd_restart.write_restart()
```

9.3.2 Temperature ramp

Runs often have temperature ramps. These are trivial to make restartable using a variant. Just be sure to set the `zero=0` option so that the ramp starts at timestep 0 and does not begin at the top every time the job is submitted. The only change needed from the previous simple example is to use the variant in `integrate.nvt()`:

```
T_variant = variant.linear_interp(points = [(0, 2.0), (2e5, 0.5)], zero=0)
integrate.nvt(group=group.all(), T=T_variant, tau=0.5)
```

9.3.3 Multiple stage jobs

Not all ramps or staged job protocols can be expressed as variants. However, it is easy to implement multi-stage jobs using run_upto and `HOOMD_WALLTIME_STOP`. Here is an example of a more complex job that involves multiple stages:

```
# run.py
from hoomd import *
from hoomd import md
context.initialize()

init.read_gsd(filename='init.gsd', restart='restart.gsd')

lj = md.pair.lj(r_cut=2.5)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)

md.integrate.mode_standard(dt=0.005)

gsd_restart = dump.gsd(filename="restart.gsd", group=group.all(), truncate=True,␣
↪period=10000, phase=0)

try:
    # randomize at high temperature
```

(continues on next page)

(continued from previous page)

```
    nvt = md.integrate.nvt(group=group.all(), T=5.0, tau=0.5)
    run_upto(1e6, limit_multiple=10000)

    # equilibrate
    nvt.set_params(T=1.0)
    run_upto(2e6, limit_multiple=10000)

    # switch to nve and start saving data for the production run
    nvt.disable();
    md.integrate.nve(group=group.all())
    dump.dcd(filename="trajectory.dcd", period=1e5, phase=0)
    analyze.log(filename='temperature.log', quantities=['temperature'], period=5000,␣
↪phase=0)

    run_upto(12e6);

except WalltimeLimitReached:
    pass

gsd_restart.write_restart()
```

And here is another example that changes interaction parameters:

```
try:
    for i in range(1,11):
        lj.pair_coeff.set('A', 'A', epsilon=0.1*i)
        run_upto(1e6*i);
except WalltimeLimitReached:
    pass
```

9.3.4 Multiple hoomd invocations

`HOOMD_WALLTIME_STOP` is a global variable set at the start of a job script. So you can launch hoomd scripts multiple times from within a job script and any of those individual runs will exit cleanly when it reaches the walltime. You need to take care that you don't start any new scripts once the first exits due to a walltime limit. The BASH script logic necessary to implement this behavior is workflow dependent and left as an exercise to the reader.

CHAPTER 10

Variable period specification

Most updaters and analyzers in hoomd accept a variable period specification. Just specify a function taking a single argument to the period parameter.

For example, dump gsd files at time steps 1, 10, 100, 1000, . . . :

```
dump.gsd(filename="dump.gsd", period = lambda n: 10**n)
```

More examples:

```
dump.gsd(filename="dump.gsd", period = lambda n: n**2)
dump.gsd(filename="dump.gsd", period = lambda n: 2**n)
dump.gsd(filename="dump.gsd", period = lambda n: 1005 + 0.5 * 10**n)
```

The object passed into *period* must be callable, accept one argument, and return a floating point number or integer. The function should also be monotonically increasing.

- First, the current time step of the simulation is saved when the analyzer is created.
- n is also set to 1 when the analyzer is created
- Every time the analyzer performs it's output, it evaluates the given function at the current value of n and records that as the next time step to perform the analysis. n is then incremented by 1

CHAPTER 11

Developer Topics

11.1 Plugins and Components

HOOMD-Blue can be tuned for particular use cases in several ways. Many smaller workloads and higher-level use cases can be handled through python-level *hoomd.run()* callbacks. For heavier-duty work such as pair potential evaluation, HOOMD-Blue can be extended through `components`. Components provide sets of functionality with a common overarching goal; for example, the *hoomd.hpmc* component provides code for hard particle Monte Carlo methods within HOOMD-Blue.

Components can be compiled and installed as **builtin** components or as **external** components. Builtin components are built and installed alongside the rest of HOOMD-Blue, while external components are compiled after HOOMD-Blue has already been compiled and installed at its destination. They have the same capabilities, but builtin components are simpler to build while external components are more flexible for packaging purposes.

The HOOMD-Blue source provides an example component template in the *example_plugin* subdirectory which supports installation either as a builtin component or as an external component, depending on how it is configured.

To set up the example component as a builtin component, simply create a symbolic link to the **internal** *example_plugin* directory (*example_plugin/example_plugin*) inside the *hoomd* subdirectory:

```
$ cd /path/to/hoomd-blue/hoomd
$ ln -s ../example_plugin/example_plugin example_plugin
$ cd ../build && make install
```

Note that this has already been done for the case of the example component.

Alternatively, one can use the example component as an external component. This relies on the *FindHOOMD.cmake* cmake script to set up cmake in a way that closely mirrors the cmake environment that HOOMD Blue was originally compiled with. The process is very similar to the process of installing HOOMD Blue itself. For ease of configuration, it is best to make sure that the *hoomd* module that is automatically imported by python is the one you wish to configure the component against and install to:

```
$ cd /path/to/component
$ mkdir build && cd build
```

(continues on next page)

(continued from previous page)

```
$ # This python command should print the location you wish to install into
$ python -c 'import hoomd;print(hoomd.__file__)'
$ # Add any extra cmake arguments you need here (like -DPYTHON_EXECUTABLE)
$ cmake ..
$ make install
```

CHAPTER 12

hoomd

Overview

hoomd.get_step	Get the current simulation time step.
hoomd.run	Runs the simulation for a given number of time steps.
hoomd.run_upto	Runs the simulation up to a given time step number.

Details

HOOMD-blue python API

hoomd provides a high level user interface for executing simulations using HOOMD:

```
import hoomd
from hoomd import md
hoomd.context.initialize()

# create a 10x10x10 square lattice of particles with name A
hoomd.init.create_lattice(unitcell=hoomd.lattice.sc(a=2.0, type_name='A'), n=10)
# specify Lennard-Jones interactions between particle pairs
nl = md.nlist.cell()
lj = md.pair.lj(r_cut=3.0, nlist=nl)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
# integrate at constant temperature
all = hoomd.group.all();
md.integrate.mode_standard(dt=0.005)
hoomd.md.integrate.langevin(group=all, kT=1.2, seed=4)
# run 10,000 time steps
hoomd.run(10e3)
```

Stability

hoomd is **stable**. When upgrading from version 2.x to 2.y (y > x), existing job scripts that follow *documented* interfaces for functions and classes will not require any modifications. **Maintainer:** Joshua A. Anderson

Attention: This stability guarantee only applies to modules in the *hoomd* package. Subpackages (*hoomd.hpmc*, *hoomd.md*, etc. . .) may or may not have a stable API. The documentation for each subpackage specifies the level of API stability it provides.

hoomd.**get_step**()
Get the current simulation time step.

Returns The current simulation time step.

Example:

```
print(hoomd.get_step())
```

hoomd.**run**(*tsteps*, *profile=False*, *limit_hours=None*, *limit_multiple=1*, *callback_period=0*, *callback=None*, *quiet=False*)
Runs the simulation for a given number of time steps.

Parameters

- **tsteps** (*int*) – Number of time steps to advance the simulation.
- **profile** (*bool*) – Set to True to enable high level profiling output at the end of the run.
- **limit_hours** (*float*) – If not None, limit this run to a given number of hours.
- **limit_multiple** (*int*) – When stopping the run due to walltime limits, only stop when the time step is a multiple of limit_multiple.
- **callback** (*callable*) – Sets a Python function to be called regularly during a run.
- **callback_period** (*int*) – Sets the period, in time steps, between calls made to `callback`.
- **quiet** (*bool*) – Set to True to disable the status information printed to the screen by the run.

Example:

```
hoomd.run(10)
hoomd.run(10e6, limit_hours=1.0/3600.0, limit_multiple=10)
hoomd.run(10, profile=True)
hoomd.run(10, quiet=True)
hoomd.run(10, callback_period=2, callback=lambda step: print(step))
```

Execute the *run()* command to advance the simulation forward in time. During the run, all previously specified analyzers, updaters and the integrator are executed at the specified regular periods.

After *run()* completes, you may change parameters of the simulation and continue the simulation by executing *run()* again. Time steps are added cumulatively, so calling `run(1000)` and then `run(2000)` would run the simulation up to time step 3000.

run() cannot be executed before the system is initialized. In most cases, *run()* should only be called after after pair forces, bond forces, and an integrator are specified.

When *profile* is **True**, a detailed breakdown of how much time was spent in each portion of the calculation is printed at the end of the run. Collecting this timing information slows the simulation.

Wallclock limited runs:

There are a number of mechanisms to limit the time of a running hoomd script. Use these in a job queuing environment to allow your script to cleanly exit before reaching the system enforced walltime limit.

Force *run()* to end only on time steps that are a multiple of `limit_multiple`. Set this to the period at which you dump restart files so that you always end a *run()* cleanly at a point where you can restart from. Use `phase=0` on logs, file dumps, and other periodic tasks. With `phase=0`, these tasks will continue on the same sequence regardless of the restart period.

Set the environment variable `HOOMD_WALLTIME_STOP` prior to starting a hoomd script to stop the *run()* at a given wall clock time. *run()* monitors performance and tries to ensure that it will end *before* `HOOMD_WALLTIME_STOP`. This environment variable works even with multiple stages of runs in a script (use *run_upto()*. Set the variable to a unix epoch time. For example in a job script that should run 12 hours, set `HOOMD_WALLTIME_STOP` to 12 hours from now, minus 10 minutes to allow for job cleanup:

```
export HOOMD_WALLTIME_STOP=$((`date +%s` + 12 * 3600 - 10 * 60))
```

When using `HOOMD_WALLTIME_STOP`, *run()* will throw the exception `WalltimeLimitReached` if it exits due to the walltime limit.

`limit_hours` is another way to limit the length of a *run()*. Set it to a number of hours (use fractional values for minutes) to limit this particular *run()* to that length of time. This is less useful than `HOOMD_WALLTIME_STOP` in a job queuing environment.

Callbacks:

If `callback` is set to a Python function then this function will be called regularly at `callback_period` intervals. The callback function must receive one integer as argument and can return an integer. The argument passed to the callback is the current time step number. If the callback function returns a negative number, the run is immediately aborted.

If `callback_period` is set to 0 (the default) then the callback is only called once at the end of the run. Otherwise the callback is executed whenever the current time step number is a multiple of `callback_period`.

hoomd.**run_upto**(*step*, ***keywords*)

Runs the simulation up to a given time step number.

Parameters

- **step** (*int*) – Final time step of the simulation which to run
- **keywords** – Catch for all keyword arguments to pass on to *run()*

run_upto() runs the simulation, but only until it reaches the given time step. If the simulation has already reached the specified step, a message is printed and no simulation steps are run.

It accepts all keyword options that *run()* does.

Examples:

```
run_upto(1000)
run_upto(10000, profile=True)
run_upto(1e9, limit_hours=11)
```

Modules

12.1 hoomd.analyze

Overview

hoomd.analyze.callback	Callback analyzer.
hoomd.analyze.imd	Send simulation snapshots to VMD in real-time.
hoomd.analyze.log	Log a number of calculated quantities to a file.

Details

Commands that analyze the system and provide some output.

An analyzer examines the system state in some way every *period* time steps and generates some form of output based on the analysis. Check the documentation for individual analyzers to see what they do.

class hoomd.analyze.**callback**(*callback*, *period*, *phase=0*)

Callback analyzer.

Parameters

- **callback** (*callable*) – The python callback object
- **period** (*int*) – The callback is called every a period time steps
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where (step + phase) % period == 0.

Create an analyzer that runs a given python callback method at a defined period.

Examples:

```
def my_callback(timestep):
  print(timestep)

analyze.callback(callback = my_callback, period = 100)
```

disable()

Disable the analyzer.

Examples:

```
my_analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any *hoomd.run()* command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with *enable()*.

enable()

Enables the analyzer

Examples:

```
my_analyzer.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the period between analyzer executions

Parameters **period** (*int*) – New period to set (in time steps)

Examples:

```
analyzer.set_period(100)
analyzer.set_period(1)
```

While the simulation is running (*hoomd.run()*, the action of each analyzer is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.analyze.**imd**(*port*, *period=1*, *rate=1*, *pause=False*, *force=None*, *force_scale=0.1*, *phase=0*)
Send simulation snapshots to VMD in real-time.

Parameters

- **port** (*int*) – TCP/IP port to listen on.
- **period** (*int*) – Number of time steps to run before checking for new IMD messages.
- **rate** (*int*) – Number of periods between coordinate data transmissions.
- **pause** (*bool*) – Set to *True* to pause the simulation at the first time step until an imd connection is made.
- **force** (*hoomd.md.force.constant*) – A force that apply forces received from VMD.
- **force_scale** (*float*) – Factor by which to scale all forces received from VMD.
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.

hoomd.analyze.imd listens on a specified TCP/IP port for connections from VMD. Once that connection is established, it begins transmitting simulation snapshots to VMD every *rate* time steps.

To connect to a simulation running on the local host, issue the command:

```
imd connect localhost 54321
```

in the VMD command window (where 54321 is replaced with the port number you specify for *hoomd.analyze.imd*.

Note: If a period larger than 1 is set, the actual rate at which time steps are transmitted is `rate * period`.

Examples:

```
analyze.imd(port=54321, rate=100)
analyze.imd(port=54321, rate=100, pause=True)
imd = analyze.imd(port=12345, rate=1000)
```

disable()
Disable the analyzer.

Examples:

```
my_analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any *hoomd.run()* command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with *enable()*.

enable()
Enables the analyzer

Examples:

```
my_analyzer.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the period between analyzer executions

Parameters **period** (*int*) – New period to set (in time steps)

Examples:

```
analyzer.set_period(100)
analyzer.set_period(1)
```

While the simulation is running (*hoomd.run()*, the action of each analyzer is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.analyze.**log**(*filename*, *quantities*, *period*, *header_prefix=''*, *overwrite=False*, *phase=0*)
Log a number of calculated quantities to a file.

Parameters

- **filename** (*str*) – File to write the log to, or *None* for no file output.
- **quantities** (*list*) – List of quantities to log.
- **period** (*int*) – Quantities are logged every *period* time steps.
- **header_prefix** (*str*) – Specify a string to print before the header.
- **overwrite** (*bool*) – When False (the default) an existing log will be appended to. When True, an existing log file will be overwritten instead.
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.

hoomd.analyze.log reads a variety of calculated values, like energy and temperature, from specified forces, integrators, and updaters. It writes a single line to the specified output file every *period* time steps. The resulting file is suitable for direct import into a spreadsheet, MATLAB, or other software that can handle simple delimited files. See *Units* for information on the units in hoomd.

Quantities that can be logged at any time:

- **volume** - Volume of the simulation box (in volume units)
- **N** - Particle number (dimensionless)
- **lx** - Box length in x direction (in length units)
- **ly** - Box length in y direction (in length units)

- **lz** - Box length in z direction (in length units)
- **xy** - Box tilt factor in xy plane (dimensionless)
- **xz** - Box tilt factor in xz plane (dimensionless)
- **yz** - Box tilt factor in yz plane (dimensionless)
- **momentum** - Magnitude of the average momentum of all particles (in momentum units)
- **time** - Wall-clock running time from the start of the log (in seconds)

Thermodynamic properties: - The following quantities are always available and computed over all particles in the system (see *hoomd.compute.thermo* for detailed definitions):

- **num_particles**
- **ndof**
- **translational_ndof**
- **rotational_ndof**
- **potential_energy** (in energy units)
- **kinetic_energy** (in energy units)
- **translational_kinetic_energy** (in energy units)
- **rotational_kinetic_energy** (in energy units)
- **temperature** (in thermal energy units)
- **pressure** (in pressure units)
- **pressure_xx**, **pressure_xy**, **pressure_xz**, **pressure_yy**, **pressure_yz**, **pressure_zz** (in pressure units)
- The above quantities, tagged with a *_groupname* suffix are automatically available for any group passed to an integrate command
- Specify a compute.thermo directly to enable additional quantities for user-specified groups.

The following quantities are only available if the command is parentheses has been specified and is active for logging:

- Pair potentials
  - **pair_dpd_energy** (*hoomd.md.pair.dpd*) - Total DPD conservative potential energy (in energy units)
  - **pair_dpdlj_energy** (*hoomd.md.pair.dpdlj*) - Total DPDLJ conservative potential energy (in energy units)
  - **pair_eam_energy** (*hoomd.metal.pair.eam*) - Total EAM potential energy (in energy units)
  - **pair_ewald_energy** (*hoomd.md.pair.ewald*) - Short ranged part of the electrostatic energy (in energy units)
  - **pair_gauss_energy** (*hoomd.md.pair.gauss*) - Total Gaussian potential energy (in energy units)
  - **pair_lj_energy** (*hoomd.md.pair.lj*) - Total Lennard-Jones potential energy (in energy units)
  - **pair_morse_energy** (*hoomd.md.pair.yukawa*) - Total Morse potential energy (in energy units)
  - **pair_table_energy** (*hoomd.md.pair.table*) - Total potential energy from Tabulated potentials (in energy units)

  - **pair_slj_energy** (*hoomd.md.pair.slj*) - Total Shifted Lennard-Jones potential energy (in energy units)
  - **pair_yukawa_energy** (*hoomd.md.pair.yukawa*) - Total Yukawa potential energy (in energy units)
  - **pair_force_shifted_lj_energy** (*hoomd.md.pair.force_shifted_lj*) - Total Force-shifted Lennard-Jones potential energy (in energy units)
  - **pppm_energy** (*hoomd.md.charge.pppm*) - Long ranged part of the electrostatic energy (in energy units)
- Bond potentials
  - **bond_fene_energy** (*hoomd.md.bond.fene*) - Total fene bond potential energy (in energy units)
  - **bond_harmonic_energy** (*hoomd.md.bond.harmonic*) - Total harmonic bond potential energy (in energy units)
  - **bond_table_energy** (*hoomd.md.bond.table*) - Total table bond potential energy (in energy units)
- Angle potentials
  - **angle_harmonic_energy** (*hoomd.md.angle.harmonic*) - Total harmonic angle potential energy (in energy units)
- Dihedral potentials
  - **dihedral_harmonic_energy** (*hoomd.md.dihedral.harmonic*) - Total harmonic dihedral potential energy (in energy units)
- Special pair interactions - **special_pair_lj_energy** (*hoomd.md.special_pair.lj*) - Total energy of special pair interactions (in energy units)
- External potentials
  - **external_periodic_energy** (*hoomd.md.external.periodic*) - Total periodic potential energy (in energy units)
  - **external_e_field_energy** (*hoomd.md.external.e_field*) - Total e_field potential energy (in energy units)
- Wall potentials
  - **external_wall_lj_energy** (*hoomd.md.wall.lj*) - Total Lennard-Jones wall energy (in energy units)
  - **external_wall_gauss_energy** (*hoomd.md.wall.gauss*) - Total Gauss wall energy (in energy units)
  - **external_wall_slj_energy** (*hoomd.md.wall.slj*) - Total Shifted Lennard-Jones wall energy (in energy units)
  - **external_wall_yukawa_energy** (*hoomd.md.wall.yukawa*) - Total Yukawa wall energy (in energy units)
  - **external_wall_mie_energy** (*hoomd.md.wall.mie*) - Total Mie wall energy (in energy units)
- Integrators
  - **langevin_reservoir_energy_groupname** (*hoomd.md.integrate.langevin*) - Energy reservoir for the Langevin integrator (in energy units)
  - **nvt_reservoir_energy_groupname** (*hoomd.md.integrate.nvt*) - Energy reservoir for the NVT thermostat (in energy units)

- **nvt_mtk_reservoir_energy_groupname** (*hoomd.md.integrate.nvt*) - Energy reservoir for the NVT MTK thermostat (in energy units)
- **npt_thermostat_energy** (*hoomd.md.integrate.npt*) - Energy of the NPT thermostat
- **npt_barostat_energy** (*hoomd.md.integrate.npt* & *hoomd.md.integrate.nph*) - Energy of the NPT (or NPH) barostat

Additionally, all pair and bond potentials can be provided user-defined names that are appended as suffixes to the logged quantity (e.g. with `pair.lj(r_cut=2.5, name="alpha")`, the logged quantity would be pair_lj_energy_alpha):

By specifying a force, disabling it with the *log=True* option, and then logging it, different energy terms can be computed while only a subset of them actually drive the simulation. Common use-cases of this capability include separating out pair energy of given types (shown below) and free energy calculations. Be aware that the globally chosen *r_cut* value is the largest of all active pair potentials and those with *log=True*, so you will observe performance degradation if you *disable(log=True)* a potential with a large *r_cut*.

File output from analyze.log is optional. Specify *None* for the file name and no file will be output. Use this with the *query()* method to query the values of properties without the overhead of writing them to disk.

You can register custom python callback functions to provide logged quantities with *register_callback()*.

Examples:

```
lj1 = pair.lj(r_cut=3.0, name="lj1")
lj1.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
lj1.pair_coeff.set('A', 'B', epsilon=1.0, sigma=1.0)
lj1.pair_coeff.set('B', 'B', epsilon=1.0, sigma=1.0)

lj2 = pair.lj(r_cut=3.0, name="lj2")
lj2.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
lj2.pair_coeff.set('A', 'B', epsilon=0.0, sigma=1.0)
lj2.pair_coeff.set('B', 'B', epsilon=0.0, sigma=1.0)
lj2.disable(log=True)

analyze.log(filename='mylog.log', quantities=['pair_lj_energy_lj1', 'pair_lj_
↪energy_lj2'],
            period=100, header_prefix='#')


logger = analyze.log(filename='mylog.log', period=100,
                     quantities=['pair_lj_energy'])

analyze.log(quantities=['pair_lj_energy', 'bond_harmonic_energy',
            'kinetic_energy'], period=1000, filename='full.log')

analyze.log(filename='mylog.log', quantities=['pair_lj_energy'],
            period=100, header_prefix='#')

analyze.log(filename='mylog.log', quantities=['bond_harmonic_energy'],
            period=10, header_prefix='Log of harmonic energy, run 5\\n')
logger = analyze.log(filename='mylog.log', period=100,
                     quantities=['pair_lj_energy'], overwrite=True)

log = analyze.log(filename=None, quantities=['potential_energy'], period=1)
U = log.query('potential_energy')
```

By default, columns in the log file are separated by tabs, suitable for importing as a tab-delimited spreadsheet.

The delimiter can be changed to any string using *set_params()*

The *header_prefix* can be used in a number of ways. It specifies a simple string that will be printed before the header line of the output file. One handy way to use this is to specify header_prefix='#' so that `gnuplot` will ignore the header line automatically. Another use-case would be to specify a descriptive line containing details of the current run. Examples of each of these cases are given above.

Warning: When an existing log is appended to, the header is not printed. For the log to remain consistent with the header already in the file, you must specify the same quantities to log and in the same order for all runs of hoomd that append to the same log.

disable()
Disable the logger.

Examples:

```
logger.disable()
```

Executing the disable command will remove the logger from the system. Any *hoomd.run()* command executed after disabling the logger will not use that logger during the simulation. A disabled logger can be re-enabled with *enable()*.

enable()
Enables the logger

Examples:

```
logger.enable()
```

See *disable()*.

query(*quantity*)
Get the current value of a logged quantity.

Parameters **quantity** (*str*) – Name of the quantity to return.

query() works in two different ways depending on how the logger is configured. If the logger is writing to a file, *query()* returns the last value written to the file. If filename is *None*, then *query()* returns the value of the quantity computed at the current timestep.

Examples:

```
logdata = logger.query('pair_lj_energy')
log = analyze.log(filename=None, quantities=['potential_energy'], period=1)
U = log.query('potential_energy')
```

register_callback(*name*, *callback*)
Register a callback to produce a logged quantity.

Parameters

- **name** (*str*) – Name of the quantity
- **callback** (*callable*) – A python callable object (i.e. a lambda, function, or class that implements __call__)

The callback method must take a single argument, the current timestep, and return a single floating point value to be logged.

Note: One callback can query the value of another, but logged quantities are evaluated in order from left to right.

Examples:

```
logger = analyze.log(filename='log.dat', quantities=['my_quantity', 'cosm'],␣
↪period=100)
logger.register_callback('my_quantity', lambda timestep: timestep**2)
logger.register_callback('cosm', lambda timestep: math.cos(logger.query('my_
↪quantity')))
```

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*quantities=None*, *delimiter=None*)
Change the parameters of the log.

Parameters

- **quantities** (*list*) – New list of quantities to log (if specified)
- **delimiter** (*str*) – New delimiter between columns in the output file (if specified)

Examples:

```
logger.set_params(quantities=['bond_harmonic_energy'])
logger.set_params(delimiter=',');
logger.set_params(quantities=['bond_harmonic_energy'], delimiter=',');
```

set_period(*period*)
Changes the period between analyzer executions

Parameters **period** (*int*) – New period to set (in time steps)

Examples:

```
analyzer.set_period(100)
analyzer.set_period(1)
```

While the simulation is running (*hoomd.run()*, the action of each analyzer is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

12.2 hoomd.benchmark

Overview

hoomd.benchmark.series	Perform a series of benchmark runs.

Details

Benchmark utilities

Commands that help in benchmarking HOOMD-blue performance.

hoomd.benchmark.**series**(*warmup=100000*, *repeat=20*, *steps=10000*, *limit_hours=None*)

Perform a series of benchmark runs.

Parameters

- **warmup** (*int*) – Number of time steps to *hoomd.run()* to warm up the benchmark
- **repeat** (*int*) – Number of times to repeat the benchmark *steps*.
- **steps** (*int*) – Number of time steps to *hoomd.run()* at each benchmark point.
- **limit_hours** (*float*) – Limit each individual *hoomd.run()* length to this time.

series() executes *warmup* time steps. After that, it calls `run(steps)`, *repeat* times and returns a list containing the average TPS for each of those runs.

12.3 hoomd.cite

Overview

hoomd.cite.save	Saves the automatically generated bibliography to a BibTeX file

Details

Commands to support automatic citation generation.

Certain features of HOOMD-blue should be cited because they represent significant contributions from developers. In order to make these citations transparent and easy, HOOMD-blue will automatically print citation notices at run time when you use a feature that should be cited. Users should read and cite these publications in their work. Citations can be saved as a BibTeX file for easy incorporation into bibliographies.

`hoomd.cite.`**save**(*file='hoomd.bib'*)

Saves the automatically generated bibliography to a BibTeX file

Parameters **file** (*str*) – File name for the saved bibliography

After *save()* is called for the first time, the bibliography will (re-)generate each time that a new feature is added to ensure that all citations have been included. If `file` already exists, it will be overwritten.

Examples:

```
cite.save()
cite.save(file='cite.bib')
```

12.4 hoomd.comm

Overview

hoomd.comm.barrier	Perform a MPI barrier synchronization across all ranks in the partition.
hoomd.comm.barrier_all	Perform a MPI barrier synchronization across the whole MPI run.

Continued on next page

Table 5 – continued from previous page

hoomd.comm.decomposition	Set the domain decomposition.
hoomd.comm.get_num_ranks	Get the number of ranks in this partition.
hoomd.comm.get_partition	Get the current partition index.
hoomd.comm.get_rank	Get the current rank.

Details

MPI communication interface

Use methods in this module to query the number of MPI ranks, the current rank, etc...

hoomd.comm.**barrier**()
Perform a MPI barrier synchronization across all ranks in the partition.

Note: Does nothing in in non-MPI builds.

hoomd.comm.**barrier_all**()
Perform a MPI barrier synchronization across the whole MPI run.

Note: Does nothing in in non-MPI builds.

class hoomd.comm.**decomposition**(*x=None*, *y=None*, *z=None*, *nx=None*, *ny=None*, *nz=None*)
Set the domain decomposition.

Parameters

- **x** (*list*) – First nx-1 fractional domain widths (if *nx* is None)
- **y** (*list*) – First ny-1 fractional domain widths (if *ny* is None)
- **z** (*list*) – First nz-1 fractional domain widths (if *nz* is None)
- **nx** (*int*) – Number of processors to uniformly space in x dimension (if *x* is None)
- **ny** (*int*) – Number of processors to uniformly space in y dimension (if *y* is None)
- **nz** (*int*) – Number of processors to uniformly space in z dimension (if *z* is None)

A single domain decomposition is defined for the simulation. A standard domain decomposition divides the simulation box into equal volumes along the Cartesian axes while minimizing the surface area between domains. This works well for systems where particles are uniformly distributed and there is equal computational load for each domain, and is the default behavior in HOOMD-blue. If no decomposition is specified for an MPI run, a uniform decomposition is automatically constructed on initialization.

In simulations with density gradients, such as a vapor-liquid interface, there can be a considerable imbalance of particles between different ranks. The simulation time then becomes limited by the slowest processor. It may then be advantageous in certain systems to create domains of unequal volume, for example, by increasing the volume of less dense regions of the simulation box in order to balance the number of particles.

The decomposition command allows the user to control the geometry and positions of the decomposition. The fractional width of the first $n_i - 1$ domains is specified along each dimension, where n_i is the number of ranks desired along dimension i. If no cut planes are specified, then a uniform spacing is assumed. The number of domains with uniform spacing can also be specified. If the desired decomposition is not commensurate with the number of ranks available (for example, a 3x3x3 decomposition when only 8 ranks are available), then a default uniform spacing is chosen. For the best control, the user should specify the number of ranks in each dimension even if uniform spacing is desired.

decomposition can only be called *before* the system is initialized, at which point the particles are decomposed. An error is raised if the system is already initialized.

The decomposition can be adjusted dynamically if the best static decomposition is not known, or the system composition is changing dynamically. For this associated command, see update.balance().

Priority is always given to specified arguments over the command line arguments. If one of these is not set but a command line option is, then the command line option is used. Otherwise, a default decomposition is chosen.

Examples:

```
comm.decomposition(x=0.4, ny=2, nz=2)
comm.decomposition(nx=2, y=0.8, z=[0.2,0.3])
```

Warning: The decomposition command will override specified command line options.

Warning: This command must be invoked *before* the system is initialized because particles are decomposed at this time.

Note: The domain size cannot be chosen arbitrarily small. There are restrictions placed on the decomposition by the ghost layer width set by the pair potentials. An error will be raised at run time if the ghost layer width exceeds half the shortest domain size.

Warning: Both fractional widths and the number of processors cannot be set simultaneously, and an error will be raised if both are set.

set_params(*x=None*, *y=None*, *z=None*, *nx=None*, *ny=None*, *nz=None*)

Set parameters for the decomposition before initialization.

Parameters

- **x** (*list*) – First nx-1 fractional domain widths (if *nx* is None)
- **y** (*list*) – First ny-1 fractional domain widths (if *ny* is None)
- **z** (*list*) – First nz-1 fractional domain widths (if *nz* is None)
- **nx** (*int*) – Number of processors to uniformly space in x dimension (if *x* is None)
- **ny** (*int*) – Number of processors to uniformly space in y dimension (if *y* is None)
- **nz** (*int*) – Number of processors to uniformly space in z dimension (if *z* is None)

Examples:

```
decomposition.set_params(x=[0.2])
decomposition.set_params(nx=1, y=[0.3,0.4], nz=2)
```

hoomd.comm.**get_num_ranks**()

Get the number of ranks in this partition.

Returns The number of MPI ranks in this partition.

Note: Returns 1 in non-mpi builds.

hoomd.comm.**get_partition**()
Get the current partition index.

Returns Index of the current partition.

Note: Always returns 0 in non-mpi builds.

hoomd.comm.**get_rank**()
Get the current rank.

Returns Index of the current rank in this partition.

Note: Always returns 0 in non-mpi builds.

12.5 hoomd.compute

Overview

hoomd.compute.thermo	Compute thermodynamic properties of a group of particles.

Details

Compute system properties

A compute calculates properties of the system on demand. Most computes are automatically generated by the command that needs them (e.g. integrate.nvt creates a compute.thermo for temperature calculations). User-specified computes can be used when more flexibility is needed. Properties calculated by specified computes (automatically, or by the user) can be logged with analyze.log.

class hoomd.compute.**thermo**(*group*)
Compute thermodynamic properties of a group of particles.

Parameters **group** (*hoomd.group*) – Group to compute thermodynamic properties for.

hoomd.compute.thermo acts on a given group of particles and calculates thermodynamic properties of those particles when requested. A default *hoomd.compute.thermo* is created that operates on the group of all particles. Integration methods such as *hoomd.md.integrate.nvt* automatically create an internal *hoomd.compute.thermo* for the group that they operate on. If thermodynamic properties are needed on additional groups, a user can specify additional *hoomd.compute.thermo* commands.

Whether they are automatically created or created by a user, all specified thermos are available for logging via the *hoomd.analyze.log* command. Each one provides a set of quantities for logging, suffixed with *_groupname*, so that values for different groups are differentiated in the log file. The default *hoomd.compute.thermo* specified on the group of all particles has no suffix placed on its quantity names.

The quantities provided are (where **groupname** is replaced with the name of the group):

- **num_particles_groupname** - N number of particles in the group

- **ndof_groupname** - N_{dof} number of degrees of freedom given to the group by integrate commands
- **translational_ndof_groupname** - N_{dof} number of translational degrees of freedom given to the group by integrate commands
- **rotational_ndof_groupname** - N_{dof} number of rotational degrees of freedom given to the group by integrate commands
- **potential_energy_groupname** - U potential energy that the group contributes to the entire system (in energy units)
- **kinetic_energy_groupname** - K total kinetic energy of all particles in the group (in energy units)
- **translational_kinetic_energy_groupname** - K translational kinetic energy of all particles in the group (in energy units)
- **rotational_kinetic_energy_groupname** - K rotational kinetic energy of all particles in the group (in energy units)
- **temperature_groupname** - kT instantaneous thermal energy of the group (in energy units). Calculated as

$$kT = 2 \cdot \frac{K}{N_{\mathrm{dof}}}$$

- **pressure_groupname** - P instantaneous pressure of the group (in pressure units). Calculated as

$$W = \frac{1}{2} \sum_{i} \sum_{j \neq i} \vec{F}_{ij} \cdot \vec{r_{ij}} + \sum_{k} \vec{F}_{k} \cdot \vec{r_{k}}$$

  where $\vec{F}_{ij}$ are pairwise forces between particles and $\vec{F}_{k}$ are forces due to explicit constraints, implicit rigid body constraints, external walls, and fields. In 2D simulations, $P = (K + \frac{1}{2} \cdot W)/A$ where A is the area of the simulation box. of the simulation box.

- **pressure_xx_groupname**, **pressure_xy_groupname**, **pressure_xz_groupname**, **pressure_yy_groupname**, **pressure_yz_groupname**, **pressure_zz_groupname** - instantaneous pressure tensor of the group (in pressure units).

$$P_{ij} = \left[\sum_{k \in [0..N)} m_k v_{k,i} v_{k,j} + \sum_{k \in [0..N)} \sum_{l > k} \frac{1}{2} \left(\vec{r}_{kl,i} \vec{F}_{kl,j} + \vec{r}_{kl,j} \vec{F}_{kl,i} \right) \right] / V$$

See also:

hoomd.analyze.log.

Examples:

```
g = group.type(name='typeA', type='A')
compute.thermo(group=g)
```

disable()

Disables the thermo.

Examples:

```
my_thermo.disable()
```

Executing the disable command will remove the thermo compute from the system. Any *hoomd.run()* command executed after disabling a thermo compute will not be able to log computed values with *hoomd.analyze.log*.

A disabled thermo compute can be re-enabled with *enable()*.

enable()
Enables the thermo compute.

Examples:

```
my_thermo.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

12.6 hoomd.context

Overview

hoomd.context.SimulationContext	Simulation context
hoomd.context.initialize	Initialize the execution context

Details

Manage execution contexts.

Every hoomd simulation needs an execution context that describes what hardware it should execute on, the MPI configuration for the job, etc...

class hoomd.context.**SimulationContext**
Simulation context

Store all of the context related to a single simulation, including the system state, forces, updaters, integration methods, and all other commands specified on this simulation. All such commands in hoomd apply to the currently active simulation context. You swap between simulation contexts by using this class as a context manager:

```
sim1 = context.SimulationContext();
sim2 = context.SimulationContext();
with sim1:
  init.read_xml('init1.xml');
  lj = pair.lj(...)
  ...

with sim2:
  init.read_xml('init2.xml');
  gauss = pair.gauss(...)
  ...

# run simulation 1 for a bit
with sim1:
   run(100)

# run simulation 2 for a bit
with sim2:
   run(100)
```

(continues on next page)

(continued from previous page)

```
# set_current sets the current context without needing to use with
sim1.set_current()
run(100)
```

If you do not need to maintain multiple contexts, you can call *context.initialize()* to initialize a new context and erase the existing one:

```
context.initialize()
init.read_xml('init1.xml');
lj = pair.lj(...)
...
run(100);

context.initialize()
init.read_xml('init2.xml');
gauss = pair.gauss(...)
...
run(100)
```

sorter
hoomd.update.sort – Global particle sorter.

system_definition
hoomd.data.system_data – System definition.

The attributes are global to the context. User scripts may access documented attributes to control settings, access particle data, etc... See the linked documentation of each attribute for more details. For example, to disable the global sorter:

```
c = context.initialize();
c.sorter.disable();
```

on_gpu()
Test whether this job is running on a GPU.

Returns True if this invocation of HOOMD-blue is executing on a GPU. False if it is on the CPU.

set_current()
Force this to be the current context

hoomd.context.**initialize**(*args=None*, *memory_traceback=False*, *mpi_comm=None*)
Initialize the execution context

Parameters

- **args** (*str*) – Arguments to parse. When *None*, parse the arguments passed on the command line.
- **memory_traceback** (*bool*) – If true, enable memory allocation tracking (*only for debugging/profiling purposes*)
- **mpi_comm** – Accepts an mpi4py communicator. Use this argument to perform many independent hoomd simulations where you communicate between those simulations using your own mpi4py code.

hoomd.context.initialize() parses the command line arguments given, sets the options and initializes MPI and GPU execution (if any). By default, *hoomd.context.initialize()* reads arguments given on

the command line. Provide a string to *hoomd.context.initialize()* to set the launch configuration within the job script.

hoomd.context.initialize() can be called more than once in a script. However, the execution parameters are fixed on the first call and *args* is ignored. Subsequent calls to *hoomd.context.initialize()* create a new *SimulationContext* and set it current. This behavior is primarily to support use of hoomd in jupyter notebooks, so that a new clean simulation context is set when rerunning the notebook within an existing kernel.

Example:

```
from hoomd import *
context.initialize();
context.initialize("--mode=gpu --nrank=64");
context.initialize("--mode=cpu --nthreads=64");

world = MPI.COMM_WORLD
comm = world.Split(world.Get_rank(), 0)
hoomd.context.initialize(mpi_comm=comm)
```

12.7 hoomd.data

Overview

hoomd.data.SnapshotParticleData	Snapshot of particle data properties.
hoomd.data.angle_data_proxy	Access a single angle via a proxy.
hoomd.data.bond_data_proxy	Access a single bond via a proxy.
hoomd.data.boxdim	Define box dimensions.
hoomd.data.constraint_data_proxy	Access a single constraint via a proxy.
hoomd.data.dihedral_data_proxy	Access a single dihedral via a proxy.
hoomd.data.force_data_proxy	Access the force on a single particle via a proxy.
hoomd.data.gsd_snapshot	Read a snapshot from a GSD file.
hoomd.data.particle_data_proxy	Access a single particle via a proxy.
hoomd.data.make_snapshot	Make an empty snapshot.
hoomd.data.system_data	Access system data

Details

Access system configuration data.

Code in the data package provides high-level access to all of the particle, bond and other data that define the current state of the system. You can use python code to directly read and modify this data, allowing you to analyze simulation results while the simulation runs, or to create custom initial configurations with python code.

There are two ways to access the data.

1. Snapshots record the system configuration at one instant in time. You can store this state to analyze the data, restore it at a future point in time, or to modify it and reload it. Use snapshots for initializing simulations, or when you need to access or modify the entire simulation state.
2. Data proxies directly access the current simulation state. Use data proxies if you need to only touch a few particles or bonds at a a time.

Snapshots

Relevant methods:

- *hoomd.data.system_data.take_snapshot()* captures a snapshot of the current system state. A snapshot is a copy of the simulation state. As the simulation continues to progress, data in a captured snapshot will remain constant.
- *hoomd.data.system_data.restore_snapshot()* replaces the current system state with the state stored in a snapshot.
- *hoomd.data.make_snapshot()* creates an empty snapshot that you can populate with custom data.
- *hoomd.init.read_snapshot()* initializes a simulation from a snapshot.

Examples:

```
snapshot = system.take_snapshot()
system.restore_snapshot(snapshot)
snapshot = data.make_snapshot(N=100, particle_types=['A', 'B'], box=data.boxdim(L=10))
# ... populate snapshot with data ...
init.read_snapshot(snapshot)
```

Snapshot and MPI

In MPI simulations, the snapshot is only valid on rank 0 by default. make_snapshot, read_snapshot, and take_snapshot, restore_snapshot are collective calls, and need to be called on all ranks. But only rank 0 can access data in the snapshot:

```
snapshot = system.take_snapshot(all=True)
if comm.get_rank() == 0:
    s = init.create_random(N=100, phi_p=0.05);numpy.mean(snapshot.particles.velocity))
    snapshot.particles.position[0] = [1,2,3];

system.restore_snapshot(snapshot);
snapshot = data.make_snapshot(N=10, box=data.boxdim(L=10))
if comm.get_rank() == 0:
    snapshot.particles.position[:] = ....
init.read_snapshot(snapshot)
```

You can explicitly broadcast the information contained in the snapshot to all other ranks, using **broadcast**.

> snapshot = system.take_snapshot(all=True) snapshot.broadcast() # broadcast from rank 0 to all other ranks using MPI snapshot.broadcast_all() # broadcast from partition 0 to all other ranks and partitions using MPI

Simulation box

You can access the simulation box from a snapshot:

```
>>> print(snapshot.box)
Box: Lx=17.3646569289 Ly=17.3646569289 Lz=17.3646569289 xy=0.0 xz=0.0 yz=0.0␣
↪dimensions=3
```

and can change it:

```
>>> snapshot.box = data.boxdim(Lx=10, Ly=20, Lz=30, xy=1.0, xz=0.1, yz=2.0)
>>> print(snapshot.box)
Box: Lx=10 Ly=20 Lz=30 xy=1.0 xz=0.1 yz=2.0 dimensions=3
```

All particles must be inside the box before using the snapshot to initialize a simulation or restoring it. The dimensionality of the system (2D/3D) cannot change after initialization.

Particle properties

Particle properties are present in *snapshot.particles*. Each property is stored in a numpy array that directly accesses the memory of the snapshot. References to these arrays will become invalid when the snapshot itself is garbage collected.

- *N* is the number of particles in the particle data snapshot:

```
>>> print(snapshot.particles.N)
64000
```

- Change the number of particles in the snapshot with resize. Existing particle properties are preserved after the resize. Any newly created particles will have default values. After resizing, existing references to the numpy arrays will be invalid, access them again from *snapshot.particles.**:

```
>>> snapshot.particles.resize(1000);
```

- The list of all particle types in the simulation can be accessed and modified:

```
>>> print(snapshot.particles.types)
['A', 'B', 'C']
>>> snapshot.particles.types = ['1', '2', '3', '4'];
```

- Individual particles properties are stored in numpy arrays. Vector quantities are stored in Nx3 arrays of floats (or doubles) and scalar quantities are stored in N length 1D arrays:

```
>>> print(snapshot.particles.position[10])
[ 1.2398  -10.2687  100.6324]
```

- Various properties can be accessed of any particle, and the numpy arrays can be sliced or passed whole to other routines:

```
>>> print(snapshot.particles.typeid[10])
2
>>> print(snapshot.particles.velocity[10])
(-0.60267972946166992, 2.6205904483795166, -1.7868227958679199)
>>> print(snapshot.particles.mass[10])
1.0
>>> print(snapshot.particles.diameter[10])
1.0
```

- Particle properties can be set in the same way. This modifies the data in the snapshot, not the current simulation state:

```
>>> snapshot.particles.position[10] = [1,2,3]
>>> print(snapshot.particles.position[10])
[ 1.  2.  3.]
```

- Snapshots store particle types as integers that index into the type name array:

```
>>> print(snapshot.particles.typeid)
[ 0.  1.  2.  0.  1.  2.  0.  1.  2.  0.]
>>> snapshot.particles.types = ['A', 'B', 'C'];
>>> snapshot.particles.typeid[0] = 2;   # C
>>> snapshot.particles.typeid[1] = 0;   # A
>>> snapshot.particles.typeid[2] = 1;   # B
```

For a list of all particle properties in the snapshot see *hoomd.data.SnapshotParticleData*.

Bonds

Bonds are stored in *snapshot.bonds*. *hoomd.data.system_data.take_snapshot()* does not record the bonds by default, you need to request them with the argument *bonds=True*.

- *N* is the number of bonds in the bond data snapshot:

```
>>> print(snapshot.bonds.N)
100
```

- Change the number of bonds in the snapshot with resize. Existing bonds are preserved after the resize. Any newly created bonds will be initialized to 0. After resizing, existing references to the numpy arrays will be invalid, access them again from *snapshot.bonds.**:

```
>>> snapshot.bonds.resize(1000);
```

- Bonds are stored in an Nx2 numpy array *group*. The first axis accesses the bond *i*. The second axis *j* goes over the individual particles in the bond. The value of each element is the tag of the particle participating in the bond:

```
>>> print(snapshot.bonds.group)
[[0 1]
[1 2]
[3 4]
[4 5]]
>>> snapshot.bonds.group[0] = [10,11]
```

- Snapshots store bond types as integers that index into the type name array:

```
>>> print(snapshot.bonds.typeid)
[ 0.  1.  2.  0.  1.  2.  0.  1.  2.  0.]
>>> snapshot.bonds.types = ['A', 'B', 'C'];
>>> snapshot.bonds.typeid[0] = 2;   # C
>>> snapshot.bonds.typeid[1] = 0;   # A
>>> snapshot.bonds.typeid[2] = 1;   # B
```

Angles, dihedrals and impropers

Angles, dihedrals, and impropers are stored similar to bonds. The only difference is that the group array is sized appropriately to store the number needed for each type of bond.

- *snapshot.angles.group* is Nx3
- *snapshot.dihedrals.group* is Nx4
- *snapshot.impropers.group* is Nx4

Special pairs

Special pairs are exactly handled like bonds. The snapshot entry is called **pairs**.

Constraints

Pairwise distance constraints are added and removed like bonds. They are defined between two particles. The only difference is that instead of a type, constraints take a distance as parameter.

- *N* is the number of constraints in the constraint data snapshot:

```
>>> print(snapshot.constraints.N)
99
```

- Change the number of constraints in the snapshot with resize. Existing constraints are preserved after the resize. Any newly created constraints will be initialized to 0. After resizing, existing references to the numpy arrays will be invalid, access them again from *snapshot.constraints.**:

```
>>> snapshot.constraints.resize(1000);
```

- Bonds are stored in an Nx2 numpy array *group*. The first axis accesses the constraint *i*. The second axis *j* goes over the individual particles in the constraint. The value of each element is the tag of the particle participating in the constraint:

```
>>> print(snapshot.constraints.group)
[[4 5]
[6 7]
[6 8]
[7 8]]
>>> snapshot.constraints.group[0] = [10,11]
```

- Snapshots store constraint distances as floats:

```
>>> print(snapshot.constraints.value)
[ 1.5 2.3 1.0 0.1 ]
```

data_proxy Proxy access

For most of the cases below, it is assumed that the result of the initialization command was saved at the beginning of the script:

```
system = init.read_xml(filename="input.xml")
```

Warning: The performance of the proxy access is very slow. Use snapshots to access the whole system configuration efficiently.

Simulation box

You can access the simulation box:

```
>>> print(system.box)
Box: Lx=17.3646569289 Ly=17.3646569289 Lz=17.3646569289 xy=0.0 xz=0.0 yz=0.0
```

and can change it:

```
>>> system.box = data.boxdim(Lx=10, Ly=20, Lz=30, xy=1.0, xz=0.1, yz=2.0)
>>> print(system.box)
Box: Lx=10 Ly=20 Lz=30 xy=1.0 xz=0.1 yz=2.0
```

All particles must **always** remain inside the box. If a box is set in this way such that a particle ends up outside of the box, expect errors to be thrown or for hoomd to just crash. The dimensionality of the system cannot change after initialization.

Particle properties

For a list of all particle properties that can be read and/or set, see *hoomd.data.particle_data_proxy*. The examples here only demonstrate changing a few of them.

`system.particles` is a window into all of the particles in the system. It behaves like standard python list in many ways.

- Its length (the number of particles in the system) can be queried:

```
>>> len(system.particles)
64000
```

- A short summary can be printed of the list:

```
>>> print(system.particles)
Particle Data for 64000 particles of 1 type(s)
```

- The list of all particle types in the simulation can be accessed:

```
>>> print(system.particles.types)
['A']
>>> print system.particles.types
Particle types: ['A']
```

- Particle types can be added between *hoomd.run()* commands:

```
>>> system.particles.types.add('newType')
```

- Individual particles can be accessed at random:

```
>>> i = 4
>>> p = system.particles[i]
```

- Various properties can be accessed of any particle (note that p can be replaced with system.particles[i] and the results are the same):

```
>>> p.tag
4
>>> p.position
(27.296911239624023, -3.5986068248748779, 10.364067077636719)
>>> p.velocity
(-0.60267972946166992, 2.6205904483795166, -1.7868227958679199)
```

(continues on next page)

(continued from previous page)

```
>>> p.mass
1.0
>>> p.diameter
1.0
>>> p.type
'A'
>>> p.tag
4
```

- Particle properties can be set in the same way:

```
>>> p.position = (1,2,3)
>>> p.position
(1.0, 2.0, 3.0)
```

- Finally, all particles can be easily looped over:

```
for p in system.particles:
    p.velocity = (0,0,0)
```

Particles may be added at any time in the job script, and a unique tag is returned:

```
>>> system.particles.add('A')
>>> t = system.particles.add('B')
```

Particles may be deleted by index:

```
>>> del system.particles[0]
>>> print(system.particles[0])
tag         : 1
position    : (23.846603393554688, -27.558368682861328, -20.501256942749023)
image       : (0, 0, 0)
velocity    : (0.0, 0.0, 0.0)
acceleration: (0.0, 0.0, 0.0)
charge      : 0.0
mass        : 1.0
diameter    : 1.0
type        : A
typeid      : 0
body        : 4294967295
orientation : (1.0, 0.0, 0.0, 0.0)
net_force   : (0.0, 0.0, 0.0)
net_energy  : 0.0
net_torque  : (0.0, 0.0, 0.0)
```

Note: The particle with tag 1 is now at index 0. No guarantee is made about how the order of particles by index will or will not change, so do not write any job scripts which assume a given ordering.

To access particles in an index-independent manner, use their tags. For example, to remove all particles of type 'A', do:

```
tags = []
for p in system.particles:
    if p.type == 'A'
        tags.append(p.tag)
```

Then remove each of the particles by their unique tag:

```
for t in tags:
    system.particles.remove(t)
```

Particles can also be accessed through their unique tag:

```
t = system.particles.add('A')
p = system.particles.get(t)
```

Any defined group can be used in exactly the same way as `system.particles` above, only the particles accessed will be those just belonging to the group. For a specific example, the following will set the velocity of all particles of type A to 0:

```
groupA = group.type(name="a-particles", type='A')
for p in groupA:
    p.velocity = (0,0,0)
```

Bond Data

Bonds may be added at any time in the job script:

```
>>> system.bonds.add("bondA", 0, 1)
>>> system.bonds.add("bondA", 1, 2)
>>> system.bonds.add("bondA", 2, 3)
>>> system.bonds.add("bondA", 3, 4)
```

Individual bonds may be accessed by index:

```
>>> bnd = system.bonds[0]
>>> print(bnd)
tag           : 0
typeid        : 0
a             : 0
b             : 1
type          : bondA
>>> print(bnd.type)
bondA
>>> print(bnd.a)
0
>>> print(bnd.b)
1
```

Warning: The order in which bonds appear by index is not static and may change at any time!

Bonds may be deleted by index:

```
>>> del system.bonds[0]
>>> print(system.bonds[0])
tag           : 3
typeid        : 0
a             : 3
b             : 4
type          : bondA
```

To access bonds in an index-independent manner, use their tags. For example, to delete all bonds which connect to particle 2, first loop through the bonds and build a list of bond tags that match the criteria:

```
tags = []
for b in system.bonds:
    if b.a == 2 or b.b == 2:
        tags.append(b.tag)
```

Then remove each of the bonds by their unique tag:

```
for t in tags:
    system.bonds.remove(t)
```

Bonds can also be accessed through their unique tag:

```
t = system.bonds.add('polymer',0,1)
p = system.bonds.get(t)
```

Angle, Dihedral, and Improper Data

Angles, Dihedrals, and Impropers may be added at any time in the job script:

```
>>> system.angles.add("angleA", 0, 1, 2)
>>> system.dihedrals.add("dihedralA", 1, 2, 3, 4)
>>> system.impropers.add("dihedralA", 2, 3, 4, 5)
```

Individual angles, dihedrals, and impropers may be accessed, deleted by index or removed by tag with the same syntax as described for bonds, just replace *bonds* with *angles*, *dihedrals*, or, *impropers* and access the appropriate number of tag elements (a,b,c for angles) (a,b,c,d for dihedrals/impropers).

Constraints

Constraints may be added and removed from within the job script.

To add a constraint of length 1.5 between particles 0 and 1:

```
>>> t = system.constraints.add(0, 1, 1.5)
```

To remove it again:

```
>>> system.constraints.remove(t)
```

Forces

Forces can be accessed in a similar way:

```
>>> lj = pair.lj(r_cut=3.0)
>>> lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
>>> print(lj.forces[0])
tag         : 0
force       : (-0.077489577233791351, -0.029512746259570122, -0.13215918838977814)
virial      : -0.0931386947632
energy      : -0.0469368174672
```

(continues on next page)

(continued from previous page)

```
>>> f0 = lj.forces[0]
>>> print(f0.force)
(-0.077489577233791351, -0.029512746259570122, -0.13215918838977814)
>>> print(f0.virial)
-0.093138694763n
>>> print(f0.energy)
-0.0469368174672
```

In this manner, forces due to the lj pair force, bonds, and any other force commands in hoomd can be accessed independently from one another. See *hoomd.data.force_data_proxy* for a definition of each data field.

For advanced code using the particle data access from python, it is important to understand that the hoomd particles, forces, bonds, et cetera, are accessed as proxies. This means that after:

```
p = system.particles[i]
```

is executed, *p* **does not** store the position, velocity, ... of particle *i*. Instead, it stores *i* and provides an interface to get/set the properties on demand. This has some side effects you need to be aware of.

- First, it means that *p* (or any other proxy reference) always references the current state of the particle. As an example, note how the position of particle p moves after the run() command:

```
>>> p.position
(-21.317455291748047, -23.883811950683594, -22.159387588500977)
>>> run(1000)
** starting run **
** run complete **
>>> p.position
(-19.774742126464844, -23.564577102661133, -21.418502807617188)
```

- Second, it means that copies of the proxy reference cannot be changed independently:

```
p.position
>>> a = p
>>> a.position
(-19.774742126464844, -23.564577102661133, -21.418502807617188)
>>> p.position = (0,0,0)
>>> a.position
(0.0, 0.0, 0.0)
```

class hoomd.data.**SnapshotParticleData**

Snapshot of particle data properties.

Users should not create SnapshotParticleData directly. Use *hoomd.data.make_snapshot()* or *hoomd.data.system_data.take_snapshot()* to make snapshots.

N

int – Number of particles in the snapshot

types

list – List of string type names (assignable)

position

numpy.ndarray – (Nx3: numpy array containing the position of each particle (float or double)

orientation

numpy.ndarray – (Nx4) - numpy array containing the orientation quaternion of each particle (float or double)

velocity
numpy.ndarray – (Nx3) - numpy array containing the velocity of each particle (float or double)

acceleration
numpy.ndarray – (Nx3) - numpy array containing the acceleration of each particle (float or double)

typeid
numpy.ndarray – N-Length numpy array containing the type id of each particle (32-bit unsigned int)

mass
numpy.ndarray – N-Length numpy array containing the mass of each particle (float or double)

charge
numpy.ndarray – N L-ength numpy array containing the charge of each particle (float or double)

diameter
numpy.ndarray – N Length numpy array containing the diameter of each particle (float or double)

image
numpy.ndarray – (Nx3) - numpy array containing the image of each particle (32-bit int)

body
numpy.ndarray – N L-ength numpy array containing the body of each particle (32-bit unsigned int)

moment_inertia
numpy.ndarray – (Nx3) - numpy array containing the principal moments of inertia of each particle (float or double)

angmom
numpy.ndarray – (Nx4) - numpy array containing the angular momentum quaternion of each particle (float or double)

See also:

hoomd.data

resize(*N*)
Resize the snapshot to hold N particles.

Parameters **N** (*int*) – new size of the snapshot.

resize() changes the size of the arrays in the snapshot to hold *N* particles. Existing particle properties are preserved after the resize. Any newly created particles will have default values. After resizing, existing references to the numpy arrays will be invalid, access them again from *snapshot.particles.**

class hoomd.data.**angle_data_proxy**(*adata*, *tag*)
Access a single angle via a proxy.

angle_data_proxy provides access to all of the properties of a single angle in the system. See *hoomd.data* for examples.

tag
int – A unique integer attached to each angle (not in any particular range). A angle's tag remains fixed during its lifetime. (Tags previously used by removed angles may be recycled).

typeid
int – Type id of the angle.

a
int – The tag of the first particle in the angle.

b
int – The tag of the second particle in the angle.

c
int – The tag of the third particle in the angle.

type
str – angle type name.

In the current version of the API, only already defined type names can be used. A future improvement will allow dynamic creation of new type names from within the python API.

class hoomd.data.**bond_data_proxy**(*bdata*, *tag*)
Access a single bond via a proxy.

bond_data_proxy provides access to all of the properties of a single bond in the system. See *hoomd.data* for examples.

tag
int – A unique integer attached to each bond (not in any particular range). A bond's tag remains fixed during its lifetime. (Tags previously used by removed bonds may be recycled).

typeid
int – Type id of the bond.

a
int – The tag of the first particle in the bond.

b
int – The tag of the second particle in the bond.

type
str – Bond type name.

In the current version of the API, only already defined type names can be used. A future improvement will allow dynamic creation of new type names from within the python API.

class hoomd.data.**boxdim**(*Lx=1.0*, *Ly=1.0*, *Lz=1.0*, *xy=0.0*, *xz=0.0*, *yz=0.0*, *dimensions=3*, *L=None*, *volume=None*)
Define box dimensions.

Parameters

- **Lx** (*float*) – box extent in the x direction (distance units)
- **Ly** (*float*) – box extent in the y direction (distance units)
- **Lz** (*float*) – box extent in the z direction (distance units)
- **xy** (*float*) – tilt factor xy (dimensionless)
- **xz** (*float*) – tilt factor xz (dimensionless)
- **yz** (*float*) – tilt factor yz (dimensionless)
- **dimensions** (*int*) – Number of dimensions in the box (2 or 3).
- **L** (*float*) – shorthand for specifying Lx=Ly=Lz=L (distance units)
- **volume** (*float*) – Scale the given box dimensions up to the this volume (area if dimensions=2)

Simulation boxes in hoomd are specified by six parameters, *Lx*, *Ly*, *Lz*, *xy*, *xz* and *yz*. For full details, see TODO: ref page. A boxdim provides a way to specify all six parameters for a given box and perform some common operations with them. Modifying a boxdim does not modify the underlying simulation box in hoomd. A boxdim can be passed to an initialization method or to assigned to a saved sysdef variable (*system.box = new_box*) to set the simulation box.

Access attributes directly:

```
b = data.boxdim(L=20);
b.xy = 1.0;
b.yz = 0.5;
b.Lz = 40;
```

Two dimensional systems

2D simulations in hoomd are embedded in 3D boxes with short heights in the z direction. To create a 2D box, set dimensions=2 when creating the boxdim. This will force Lz=1 and xz=yz=0. init commands that support 2D boxes will pass the dimensionality along to the system. When you assign a new boxdim to an already initialized system, the dimensionality flag is ignored. Changing the number of dimensions during a simulation run is not supported.

In 2D boxes, *volume* is in units of area.

Shorthand notation

data.boxdim accepts the keyword argument *L=x* as shorthand notation for *Lx=x, Ly=x, Lz=x* in 3D and *Lx=x, Ly=z, Lz=1* in 2D. If you specify both *L=* and *Lx,Ly, or Lz*, then the value for *L* will override the others.

Examples:

- Cubic box with given volume: *data.boxdim(volume=V)*
- Triclinic box in 2D with given area: *data.boxdim(xy=1.0, dimensions=2, volume=A)*
- Rectangular box in 2D with given area and aspect ratio: *data.boxdim(Lx=1, Ly=aspect, dimensions=2, volume=A)*
- Cubic box with given length: *data.boxdim(L=10)*
- Fully define all box parameters: *data.boxdim(Lx=10, Ly=20, Lz=30, xy=1.0, xz=0.5, yz=0.1)*

get_lattice_vector(*i*)
Get a lattice vector.

Parameters **i** (*int*) – (=0,1,2) direction of lattice vector

Returns The lattice vector (3-tuple) along direction *i*.

get_volume()
Get the box volume.

Returns The box volume (area in 2D).

make_fraction(*v*)
Scale a vector to fractional coordinates.

Parameters **v** (*tuple*) – The vector to convert to fractional coordinates

make_fraction() takes a vector in a box and computes a vector where all components are between 0 and 1.

Returns The scaled vector.

min_image(*v*)
Apply the minimum image convention to a vector using periodic boundary conditions.

Parameters **v** (*tuple*) – The vector to apply minimum image to

Returns The minimum image as a tuple.

scale(*sx=1.0*, *sy=1.0*, *sz=1.0*, *s=None*)
Scale box dimensions.

Parameters

- **sx** (*float*) – scale factor in the x direction
- **sy** (*float*) – scale factor in the y direction
- **sz** (*float*) – scale factor in the z direction
- **s** (*float*) – Shorthand for sx=s, sy=x, sz=s

Scales the box by the given scale factors. Tilt factors are not modified.

Returns A reference to the modified box.

set_volume(*volume*)
Set the box volume.

Parameters **volume** (*float*) – new box volume (area if dimensions=2)

Scale the box to the given volume (or area).

Returns A reference to the modified box.

wrap(*v*, *img=(0, 0, 0)*)
Wrap a vector using the periodic boundary conditions.

Parameters

- **v** (*tuple*) – The vector to wrap
- **img** (*tuple*) – A vector of integer image flags that will be updated (optional)

Returns The wrapped vector and the image flags in a tuple.

class hoomd.data.**constraint_data_proxy**(*cdata*, *tag*)
Access a single constraint via a proxy.

constraint_data_proxy provides access to all of the properties of a single constraint in the system. See *hoomd.data* for examples.

tag
int – A unique integer attached to each constraint (not in any particular range). A constraint's tag remains fixed during its lifetime. (Tags previously used by removed constraints may be recycled).

d
float – The constraint distance.

a
int – The tag of the first particle in the constraint.

b
int – The tag of the second particle in the constraint.

class hoomd.data.**dihedral_data_proxy**(*ddata*, *tag*)
Access a single dihedral via a proxy.

dihedral_data_proxy provides access to all of the properties of a single dihedral in the system. See *hoomd.data* for examples.

tag
int – A unique integer attached to each dihedral (not in any particular range). A dihedral's tag remains fixed during its lifetime. (Tags previously used by removed dihedrals may be recycled).

typeid
int – Type id of the dihedral.

a
int – The tag of the first particle in the dihedral.

b
int – The tag of the second particle in the dihedral.

c
int – The tag of the third particle in the dihedral.

d
int – The tag of the fourth particle in the dihedral.

type
str – dihedral type name.

In the current version of the API, only already defined type names can be used. A future improvement will allow dynamic creation of new type names from within the python API.

class hoomd.data.**force_data_proxy**(*force*, *tag*)
Access the force on a single particle via a proxy.

force_data_proxy provides access to the current force, virial, and energy of a single particle due to a single force computation. See *hoomd.data* for examples.

force
tuple – (float, x, y, z) - the current force on the particle (force units)

virial
tuple – This particle's contribution to the total virial tensor.

energy
float – This particle's contribution to the total potential energy (energy units)

torque
float – (float x, y, z) - current torque on the particle (torque units)

hoomd.data.**gsd_snapshot**(*filename*, *frame=0*)
Read a snapshot from a GSD file.

Parameters

- **filename** (*str*) – GSD file to read the snapshot from.
- **frame** (*int*) – Frame to read from the GSD file. Negative values index from the end of the file.

hoomd.data.gsd_snapshot() opens the given GSD file and reads a snapshot from it.

hoomd.data.**make_snapshot**(*N*, *box*, *particle_types=['A']*, *bond_types=[]*, *angle_types=[]*, *dihedral_types=[]*, *improper_types=[]*, *pair_types=[]*, *dtype='float'*)
Make an empty snapshot.

Parameters

- **N** (*int*) – Number of particles to create.
- **box** (*hoomd.data.boxdim*) – Simulation box parameters.
- **particle_types** (*list*) – Particle type names (must not be zero length).
- **bond_types** (*list*) – Bond type names (may be zero length).
- **angle_types** (*list*) – Angle type names (may be zero length).

- **dihedral_types** (*list*) – Dihedral type names (may be zero length).
- **improper_types** (*list*) – Improper type names (may be zero length).
- **pair_types** (*list*) – Special pair type names (may be zero length). .. versionadded:: 2.1
- **dtype** (*str*) – Data type for the real valued numpy arrays in the snapshot. Must be either 'float' or 'double'.

Examples:

```
snapshot = data.make_snapshot(N=1000, box=data.boxdim(L=10))
snapshot = data.make_snapshot(N=64000, box=data.boxdim(L=1, dimensions=2,␣
↪volume=1000), particle_types=['A', 'B'])
snapshot = data.make_snapshot(N=64000, box=data.boxdim(L=20), bond_types=['polymer
↪'], dihedral_types=['dihedralA', 'dihedralB'], improper_types=['improperA',
↪'improperB', 'improperC'])
... set properties in snapshot ...
init.read_snapshot(snapshot);
```

hoomd.data.make_snapshot() creates all particles with **default properties**. You must set reasonable values for particle properties before initializing the system with *hoomd.init.read_snapshot()*.

The default properties are:

- position 0,0,0
- velocity 0,0,0
- image 0,0,0
- orientation 1,0,0,0
- typeid 0
- charge 0
- mass 1.0
- diameter 1.0

See also:

hoomd.init.read_snapshot()

class hoomd.data.**particle_data_proxy**(*pdata*, *tag*)

Access a single particle via a proxy.

particle_data_proxy provides access to all of the properties of a single particle in the system. See *hoomd.data* for examples.

tag

int – A unique name for the particle in the system. Tags run from 0 to N-1.

acceleration

tuple – A 3-tuple of floats (x, y, z). Acceleration is a calculated quantity and cannot be set. (in acceleration units)

typeid

int – The type id of the particle.

position

tuple – (x, y, z) (float, in distance units).

image
tuple – (x, y, z) (int).

velocity
tuple – (x, y, z) (float, in velocity units).

charge
float – Particle charge.

mass
float – (in mass units).

diameter
float – (in distance units).

type
str – Particle type name.

body
int – Rigid body id (-1 for free particles).

orientation
tuple – (w,x,y,z) (float, quaternion).

net_force
tuple – Net force on particle (x, y, z) (float, in force units).

net_energy
float – Net contribution of particle to the potential energy (in energy units).

net_torque
tuple – Net torque on the particle (x, y, z) (float, in torque units).

net_virial
tuple – Net virial for the particle (xx,yy,zz, xy, xz, yz)

class hoomd.data.**system_data**(*sysdef*)
Access system data

system_data provides access to the different data structures that define the current state of the simulation. See *hoomd.data* for a full explanation of how to use by example.

box
hoomd.data.boxdim

particles
hoomd.data.particle_data_proxy

bonds
hoomd.data.bond_data_proxy

angles
hoomd.data.angle_data_proxy

dihedrals
hoomd.data.dihedral_data_proxy

impropers
hoomd.data.dihedral_data_proxy

constraint
hoomd.data.constraint_data_proxy

pairs
hoomd.data.bond_data_proxy – .. versionadded:: 2.1

replicate(*nx=1*, *ny=1*, *nz=1*)
Replicates the system along the three spatial dimensions.

Parameters

- **nx** (*int*) – Number of times to replicate the system along the x-direction
- **ny** (*int*) – Number of times to replicate the system along the y-direction
- **nz** (*int*) – Number of times to replicate the system along the z-direction

This method replicates particles along all three spatial directions, as opposed to replication implied by periodic boundary conditions. The box is resized and the number of particles is updated so that the new box holds the specified number of replicas of the old box along all directions. Particle coordinates are updated accordingly to fit into the new box. All velocities and other particle properties are replicated as well. Also bonded groups between particles are replicated.

Examples:

```
system = init.read_xml("some_file.xml")
system.replicate(nx=2,ny=2,nz=2)
```

Note: The dimensions of the processor grid are not updated upon replication. For example, if an initially cubic box is replicated along only one spatial direction, this could lead to decreased performance if the processor grid was optimal for the original box dimensions, but not for the new ones.

restore_snapshot(*snapshot*)
Re-initializes the system from a snapshot.

Parameters **snapshot** – . The snapshot to initialize the system from.

Snapshots temporarily store system data. Snapshots contain the complete simulation state in a single object. They can be used to restart a simulation.

Example use cases in which a simulation may be restarted from a snapshot include python-script-level Monte-Carlo schemes, where the system state is stored after a move has been accepted (according to some criterion), and where the system is re-initialized from that same state in the case when a move is not accepted.

Example:

```
system = init.read_xml("some_file.xml")

... run a simulation ...

snapshot = system.take_snapshot(all=True)
...
system.restore_snapshot(snapshot)
```

Warning: restore_snapshot() may invalidate force coefficients, neighborlist r_cut values, and other per type quantities if called within a callback during a run(). You can restore a snapshot during a run only if the snapshot is of a previous state of the currently running system. Otherwise, you need to use restore_snapshot() between run() commands to ensure that all per type coefficients are updated properly.

take_snapshot(*particles=True*, *bonds=False*, *pairs=False*, *integrators=False*, *all=False*, *dtype='float'*)
Take a snapshot of the current system data.

Parameters

- **particles** (*bool*) – When True, particle data is included in the snapshot.
- **bonds** (*bool*) – When true, bond, angle, dihedral, improper and constraint data is included.
- **pairs** (*bool*) – When true, special pair data is included .. versionadded:: 2.1
- **integrators** (*bool*) – When true, integrator data is included the snapshot.
- **all** (*bool*) – When true, the entire system state is saved in the snapshot.
- **dtype** (*str*) – Datatype for the snapshot numpy arrays. Must be either 'float' or 'double'.

Returns The snapshot object.

This functions returns a snapshot object. It contains the current. partial or complete simulation state. With appropriate options it is possible to select which data properties should be included in the snapshot

Examples:

```
snapshot = system.take_snapshot()
snapshot = system.take_snapshot()
snapshot = system.take_snapshot(bonds=true)
```

12.8 hoomd.dump

Overview

hoomd.dump.dcd	Writes simulation snapshots in the DCD format
hoomd.dump.getar	Analyzer for dumping system properties to a getar file at intervals.
hoomd.dump.gsd	Writes simulation snapshots in the GSD format

Details

Write system configurations to files.

Commands in the dump package write the system state out to a file every *period* time steps. Check the documentation for details on which file format each command writes.

class hoomd.dump.**dcd**(*filename*, *period*, *group=None*, *overwrite=False*, *unwrap_full=False*, *unwrap_rigid=False*, *angle_z=False*, *phase=0*)
Writes simulation snapshots in the DCD format

Parameters

- **filename** (*str*) – File name to write.
- **period** (*int*) – Number of time steps between file dumps.
- **group** (*hoomd.group*) – Particle group to output to the dcd file. If left as None, all particles will be written.

- **overwrite** (*bool*) – When False, (the default) an existing DCD file will be appended to. When True, an existing DCD file *filename* will be overwritten.
- **unwrap_full** (*bool*) – When False, (the default) particle coordinates are always written inside the simulation box. When True, particles will be unwrapped into their current box image before writing to the dcd file.
- **unwrap_rigid** (*bool*) – When False, (the default) individual particles are written inside the simulation box which breaks up rigid bodies near box boundaries. When True, particles belonging to the same rigid body will be unwrapped so that the body is continuous. The center of mass of the body remains in the simulation box, but some particles may be written just outside it. *unwrap_rigid* is ignored when *unwrap_full* is True.
- **angle_z** (*bool*) – When True, the particle orientation angle is written to the z component (only useful for 2D simulations)
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.

Every *period* time steps a new simulation snapshot is written to the specified file in the DCD file format. DCD only stores particle positions, in distance units - see *Units*.

Due to constraints of the DCD file format, once you stop writing to a file via *disable()*, you cannot continue writing to the same file, nor can you change the period of the dump at any time. Either of these tasks can be performed by creating a new dump file with the needed settings.

Examples:

```
dump.dcd(filename="trajectory.dcd", period=1000)
dcd = dump.dcd(filename"data/dump.dcd", period=1000)
```

Warning: When you use dump.dcd to append to an existing dcd file:

- The period must be the same or the time data in the file will not be consistent.
- dump.dcd will not write out data at time steps that already are present in the dcd file to maintain a consistent timeline

disable()
Disable the analyzer.

Examples:

```
my_analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any *hoomd.run()* command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with *enable()*.

enable()
The DCD dump writer cannot be re-enabled

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the period between analyzer executions

Parameters **period** (*int*) – New period to set (in time steps)

Examples:

```
analyzer.set_period(100)
analyzer.set_period(1)
```

While the simulation is running (*hoomd.run()*, the action of each analyzer is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.dump.**getar**(*filename*, *mode='w'*, *static=[]*, *dynamic={}*, *_register=True*)

Analyzer for dumping system properties to a getar file at intervals.

Getar files are a simple interface on top of archive formats (such as zip and tar) for storing trajectory data efficiently. A more thorough description of the format and a description of a python API to read and write these files is available at the libgetar documentation.

Properties to dump can be given either as a *getar.DumpProp* object or a name. Supported property names are specified in the Supported Property Table in *hoomd.init.read_getar*.

Files can be opened in write, append, or one-shot mode. Write mode overwrites files with the same name, while append mode adds to them. One-shot mode is intended for restorable system backups and is described below.

One-shot mode

In one-shot mode, activated by passing mode='1' to the getar constructor, properties are written to a temporary file, which then overwrites the file with the given filename. In this way, the file with the given filename should always have the most recent frame of successfully written data. This mode is designed for being able to dump restoration data often without wasting large amounts of space saving earlier data. Note that this create-and-overwrite process can be stressful on filesystems, particularly lustre filesystems, and can get your account blocked on some supercomputer resources if overused.

For convenience, you can also specify **composite properties**, which are expanded according to the table below.

twidwidNaResult
twidwidglobox_all
di-
men-
sions
twidwidangle_all_type_names,
an-
gle_tag,
an-
gle_type
twidwidbond_all_type_names,
bond_tag,
bond_type
twidwidhedral_all_type_names,
di-
he-
dral_tag,
di-
he-
dral_type
twidwidproper_all_type_names,
im-
proper_tag,
im-
proper_type
twidwidangular_all_momentum,
body,
charge,
di-
am-
e-
ter,
im-
age,
mass,
mo-
ment_inertia,
ori-
en-
ta-
tion,
po-
si-
tion,
type,
type_names,
ve-
loc-
ity
twidwidparticle_all,
an-
gle_all,
bond_all,
di-
he-
dral_all,
im-
proper_all,
global_all

Particle-related metadata

Metadata about particle shape (for later visualization or use in restartable scripts) can be stored in a simple form through *hoomd.dump.getar.writeJSON()*, which encodes JSON records as strings and stores them inside the dump file. Currently, classes inside *hoomd.dem* and *hoomd.hpmc* are equipped with *get_type_shapes()* methods which can provide per-particle-type shape information as a list.

Example:

```
dump = hoomd.dump.getar.simple('dump.sqlite', 1e3,
    static=['viz_static'],
    dynamic=['viz_aniso_dynamic'])

dem_wca = hoomd.dem.WCA(nlist, radius=0.5)
dem_wca.setParams('A', vertices=vertices, faces=faces)
dump.writeJSON('type_shapes.json', dem_wca.get_type_shapes())

mc = hpmc.integrate.convex_polygon(seed=415236)
mc.shape_param.set('A', vertices=[(-0.5, -0.5), (0.5, -0.5), (0.5, 0.5), (-0.5, 0.
↪5)])
dump.writeJSON('type_shapes.json', mc.get_type_shapes(), dynamic=True)
```

class DumpProp(*name*, *highPrecision=False*, *compression=hoomd.dump.getar.Compression.FastCompress*)

Create a dump property specification.

Parameters

- **name** – Name of the property to dump
- **highPrecision** – True if the property should be dumped in high precision, if possible
- **compression** – Compression level to save the property with, if possible

__init__(*filename*, *mode='w'*, *static=[]*, *dynamic={}*, *_register=True*)

Initialize a getar dumper. Creates or appends an archive at the given file location according to the mode and prepares to dump the given sets of properties.

Parameters

- **filename** (*str*) – Name of the file to open
- **mode** (*str*) – Run mode; see mode list below.
- **static** (*list*) – List of static properties to dump immediately
- **dynamic** (*dict*) – Dictionary of {prop: period} periodic dumps
- **_register** (*bool*) – If True, register as a hoomd analyzer (internal)

Note that zip32-format archives can not be appended to at the moment; for details and solutions, see the libgetar documentation, section "Zip vs. Zip64." The gtar.fix module was explicitly made for this purpose, but be careful not to call it from within a running GPU HOOMD simulation due to strangeness in the CUDA driver.

Valid mode arguments:

- 'w': Write, and overwrite if file exists
- 'a': Write, and append if file exists
- '1': One-shot mode: keep only one frame of data. For details on one-shot mode, see the "One-shot mode" section of *getar*.

Property specifications can be either a property name (as a string) or *DumpProp* objects if you desire greater control over how the property will be dumped.

Example:

```
# detailed API; see `dump.getar.simple` for simpler wrappers
zip = dump.getar('dump.zip', static=['types'],
          dynamic={'orientation': 10000,
                   'velocity': 5000,
                   dump.getar.DumpProp('position', highPrecision=True): 10000}
↪)
```

close()
Closes the trajectory if it is open. Finalizes any IO beforehand.

disable()
Disable the analyzer.

Examples:

```
my_analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any *hoomd.run()* command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with *enable()*.

enable()
Enables the analyzer

Examples:

```
my_analyzer.enable()
```

See *disable()*.

classmethod immediate(*filename*, *static*, *dynamic*)
Immediately dump the given static and dynamic properties to the given filename.

For detailed explanation of arguments, see *getar*.

Example:

```
hoomd.dump.getar.immediate(
    'snapshot.tar', static=['viz_static'], dynamic=['viz_dynamic'])
```

restore_state()
Restore the state information from the file used to initialize the simulations

classmethod simple(*filename*, *period*, *mode='w'*, *static=[]*, *dynamic=[]*, *high_precision=False*)
Create a *getar* dump object with a simpler interface.

Static properties will be dumped once immediately, and dynamic properties will be dumped every *period* steps. For detailed explanation of arguments, see *getar*.

Parameters

- **filename** (*str*) – Name of the file to open
- **period** (*int*) – Period to dump the given dynamic properties with
- **mode** (*str*) – Run mode; see mode list in *getar*.
- **static** (*list*) – List of static properties to dump immediately

- **dynamic** (*list*) – List of properties to dump every *period* steps
- **high_precision** (*bool*) – If True, dump precision properties

Example:

```
# [optionally] dump metadata beforehand with libgetar
with gtar.GTAR('dump.sqlite', 'w') as trajectory:
    metadata = json.dumps(hoomd.meta.dump_metadata())
    trajectory.writeStr('hoomd_metadata.json', metadata)
# for later visualization of anisotropic systems
zip2 = hoomd.dump.getar.simple(
     'dump.sqlite', 100000, 'a', static=['viz_static'], dynamic=['viz_aniso_
↪dynamic'])
# as backup to restore from later
backup = hoomd.dump.getar.simple(
    'backup.tar', 10000, '1', static=['viz_static'], dynamic=['viz_aniso_
↪dynamic'])
```

writeJSON(*name*, *contents*, *dynamic=True*)

Encodes the given JSON-encodable object as a string and writes it (immediately) as a quantity with the given name. If dynamic is True, writes the record as a dynamic record with the current timestep.

Parameters

- **name** (*str*) – Name of the record to save
- **contents** (*str*) – Any datatype encodable by the `json` module
- **dynamic** (*bool*) – If True, dump a dynamic quantity with the current timestep; otherwise, dump a static quantity

Example:

```
dump = hoomd.dump.getar.simple('dump.sqlite', 1e3,
    static=['viz_static'], dynamic=['viz_dynamic'])
dump.writeJSON('params.json', dict(temperature=temperature,␣
↪pressure=pressure))
dump.writeJSON('metadata.json', hoomd.meta.dump_metadata())
```

class hoomd.dump.**gsd**(*filename*, *period*, *group*, *overwrite=False*, *truncate=False*, *phase=0*, *time_step=None*, *static=None*, *dynamic=None*)

Writes simulation snapshots in the GSD format

Parameters

- **filename** (*str*) – File name to write
- **period** (*int*) – Number of time steps between file dumps, or None to write a single file immediately.
- **group** (*hoomd.group*) – Particle group to output to the gsd file.
- **overwrite** (*bool*) – When False (the default), any existing GSD file will be appended to. When True, an existing GSD file *filename* will be overwritten.
- **truncate** (*bool*) – When False (the default), frames are appended to the GSD file. When True, truncate the file and write a new frame 0 every time.
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.
- **time_step** (*int*) – Time step to write to the file (only used when period is None)

- **dynamic** (*list*) – A list of quantity categories to save every frame. (added in version 2.2)
- **static** (*list*) – A list of quantity categories save only in frame 0 (may not be set in conjunction with *dynamic*, deprecated in version 2.2).

Write a simulation snapshot to the specified GSD file at regular intervals. GSD is capable of storing all particle and bond data fields in hoomd, in every frame of the trajectory. This allows GSD to store simulations where the number of particles, number of particle types, particle types, diameter, mass, charge, or anything is changing over time.

To save on space, GSD does not write values that are all set at defaults. So if all masses are left set at the default of 1.0, mass will not take up any space in the file. Additionally, only **dynamic** quantities are written to all frames, non-dynamic quantities are only written to frame 0. The GSD schema defines that data not present in frame *i* is to be read from frame 0. This makes every single frame of a GSD file fully specified and simulations initialized with *hoomd.init.read_gsd()* can select any frame of the file.

You can control what quantities are dynamic by category. `property` is always dynamic. The categories listed in the **dynamic** will also be written out to every frame.

- `attribute`
  - particles/N
  - particles/types
  - particles/typeid
  - particles/mass
  - particles/charge
  - particles/diameter
  - particles/body
  - particles/moment_inertia
- `property`
  - particles/position
  - particles/orientation
- `momentum`
  - particles/velocity
  - particles/angmom
  - particles/image
- `topology`
  - bonds/
  - angles/
  - dihedrals/
  - impropers/
  - constraints/
  - pairs/

See https://bitbucket.org/glotzer/gsd and http://gsd.readthedocs.io/ for more information on GSD files.

If you only need to store a subset of the system, you can save file size and time spent analyzing data by specifying a group to write out. *gsd* will write out all of the particles in the group in ascending tag order. When the group is not *hoomd.group.all()*, *gsd* will not write the topology fields.

To write restart files with gsd, set *truncate=True*. This will cause *gsd* to write a new frame 0 to the file every period steps.

State data

gsd can save internal state data for the following hoomd objects:

- *HPMC integrators*

Call *dump_state()* with the object as an argument to enable saving its state. State saved in this way can be restored after initializing the system with *hoomd.init.read_gsd()*.

Examples:

```
dump.gsd(filename="trajectory.gsd", period=1000, group=group.all(), phase=0)
dump.gsd(filename="restart.gsd", truncate=True, period=10000, group=group.all(),␣
↪phase=0)
dump.gsd(filename="configuration.gsd", overwrite=True, period=None, group=group.
↪all(), time_step=0)
dump.gsd(filename="momentum_too.gsd", period=1000, group=group.all(), phase=0,␣
↪dynamic=['momentum'])
dump.gsd(filename="saveall.gsd", overwrite=True, period=1000, group=group.all(),␣
↪dynamic=['attribute', 'momentum', 'topology'])
```

disable()
Disable the analyzer.

Examples:

```
my_analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any *hoomd.run()* command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with *enable()*.

dump_state(*obj*)
Write state information for a hoomd object.

Call *dump_state()* if you want to write the state of a hoomd object to the gsd file.

New in version 2.2.

enable()
Enables the analyzer

Examples:

```
my_analyzer.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the period between analyzer executions

Parameters **period** (*int*) – New period to set (in time steps)

Examples:

```
analyzer.set_period(100)
analyzer.set_period(1)
```

While the simulation is running (*hoomd.run()*, the action of each analyzer is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

write_restart()
Write a restart file at the current time step.

Call *write_restart()* at the end of a simulation where are writing a gsd restart file with `truncate=True` to ensure that you have the final frame of the simulation written before exiting. See *Restartable jobs* for examples.

12.9 hoomd.group

Overview

hoomd.group.all	Groups all particles.
hoomd.group.charged	Groups particles that are charged.
hoomd.group.cuboid	Groups particles in a cuboid.
hoomd.group.difference	Create a new group from the set difference or complement of two existing groups.
hoomd.group.group	Defines a group of particles
hoomd.group.intersection	Create a new group from the set intersection of two existing groups.
hoomd.group.nonrigid	Groups particles that do not belong to rigid bodies.
hoomd.group.rigid	Groups particles that belong to rigid bodies.
hoomd.group.rigid_center	Groups particles that are center particles of rigid bodies.
hoomd.group.tag_list	Groups particles by tag list.
hoomd.group.tags	Groups particles by tag.
hoomd.group.type	Groups particles by type.
hoomd.group.union	Create a new group from the set union of two existing groups.

Details

Commands for grouping particles

This package contains various commands for making groups of particles

hoomd.group.**all**()
Groups all particles.

Creates a particle group from all particles in the simulation.

Examples:

```
all = group.all()
```

hoomd.group.**charged**(*name='charged'*)

Groups particles that are charged.

Parameters **name** (*str*) – User-assigned name for this group.

Creates a particle group containing all particles that have a non-zero charge.

Warning: This group currently does not support being updated when the number of particles changes.

Examples:

```
a = group.charged()
b = group.charged(name="cp")
```

hoomd.group.**cuboid**(*name*, *xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, *zmin=None*, *zmax=None*)

Groups particles in a cuboid.

Parameters

- **name** (*str*) – User-assigned name for this group
- **xmin** (*float*) – (if set) Lower left x-coordinate of the cuboid (in distance units)
- **xmax** (*float*) – (if set) Upper right x-coordinate of the cuboid (in distance units)
- **ymin** (*float*) – (if set) Lower left y-coordinate of the cuboid (in distance units)
- **ymax** (*float*) – (if set) Upper right y-coordinate of the cuboid (in distance units)
- **zmin** (*float*) – (if set) Lower left z-coordinate of the cuboid (in distance units)
- **zmax** (*float*) – (if set) Upper right z-coordinate of the cuboid (in distance units)

If any of the above parameters is not set, it will automatically be placed slightly outside of the simulation box dimension, allowing easy specification of slabs.

Creates a particle group from particles that fall in the defined cuboid. Membership tests are performed via `xmin <= x < xmax` (and so forth for y and z) so that directly adjacent cuboids do not have overlapping group members.

Note: Membership in *cuboid* is defined at time of group creation. Once created, any particles added to the system will not be added to the group. Any particles that move into the cuboid region will not be added automatically, and any that move out will not be removed automatically.

Between runs, you can force a group to update its membership with the particles currently in the originally defined region using *hoomd.group.group.force_update()*.

Examples:

```
slab = group.cuboid(name="slab", ymin=-3, ymax=3)
cube = group.cuboid(name="cube", xmin=0, xmax=5, ymin=0, ymax=5, zmin=0, zmax=5)
run(100)
# Remove particles that left the region and add particles that entered the region.
cube.force_update()
```

hoomd.group.**difference**(*name*, *a*, *b*)

Create a new group from the set difference or complement of two existing groups.

Parameters

- **name** (*str*) – User-assigned name for this group.
- **a** (*group*) – First group.
- **b** (*group*) – Second group.

The set difference of *a* and *b* is defined to be the set of particles that are in *a* and not in *b*. This can be useful for inverting the sense of a group (see below).

A new group called *name* is created.

Warning: The group is static and will not update if particles are added to or removed from the system.

Examples:

```
groupA = group.type(name='groupA', type='A')
all = group.all()
nottypeA = group.difference(name="particles-not-typeA", a=all, b=groupA)
```

class hoomd.group.**group**(*name*, *cpp_group*)

Defines a group of particles

group should not be created directly in user code. The following methods can be used to create particle groups.

- *hoomd.group.all()*
- *hoomd.group.cuboid()*
- *hoomd.group.nonrigid()*
- *hoomd.group.rigid()*
- *hoomd.group.tags()*
- *hoomd.group.tag_list()*
- *hoomd.group.type()*
- *hoomd.group.charged()*

The above functions assign a descriptive name based on the criteria chosen. That name can be easily changed if desired:

```
groupA = group.type('A')
groupA.name = "my new group name"
```

Once a group has been created, it can be combined with others via set operations to form more complicated groups. Available operations are:

- *hoomd.group.difference()*
- *hoomd.group.intersection()*
- *hoomd.group.union()*

Note: Groups need to be consistent with the particle data. If a particle member is removed from the simulation, it will be temporarily removed from the group as well, that is, even though the group reports that tag as a member,

it will act as if the particle was not existent. If a particle with the same tag is later added to the simulation, it will become member of the group again.

Examples:

```
# create a group containing all particles in group A and those with
# tags 100-199
groupA = group.type('A')
group100_199 = group.tags(100, 199);
group_combined = group.union(name="combined", a=groupA, b=group100_199);

# create a group containing all particles in group A that also have
# tags 100-199
group_combined2 = group.intersection(name="combined2", a=groupA, b=group100_199);

# create a group containing all particles that are not in group A
all = group.all()
group_notA = group.difference(name="notA", a=all, b=groupA)
```

A group can also be queried with python sequence semantics.

Examples:

```
groupA = group.type('A')
# print the number of particles in group A
print len(groupA)
# print the position of the first particle in the group
print groupA[0].position
# set the velocity of all particles in groupA to 0
for p in groupA:
    p.velocity = (0,0,0)
```

For more information and examples on accessing the data in this way, see *hoomd.data*.

force_update()

Force an update of the group.

Re-evaluate all particles against the original group selection criterion and build a new member list based on the current state of the system. For example, call *hoomd.group.group.force_update()* set set a cuboid group's membership to particles that are currently in the defined region.

Groups made by a combination (union, intersection, difference) of other groups will not update their membership, they are always static.

hoomd.group.**intersection**(*name*, *a*, *b*)

Create a new group from the set intersection of two existing groups.

Parameters

- **name** (*str*) – User-assigned name for this group.
- **a** (*group*) – First group.
- **b** (*group*) – Second group.

A new group is created that contains all particles of *a* that are also in *b*, and is given the name *name*.

Warning: The group is static and will not update if particles are added to or removed from the system.

Examples:

```
groupA = group.type(name='groupA', type='A')
group100_199 = group.tags(name='100_199', tag_min=100, tag_max=199);
groupC = group.intersection(name="groupC", a=groupA, b=group100_199)
```

hoomd.group.**nonrigid**()

Groups particles that do not belong to rigid bodies.

Creates a particle group from particles. All particles that **do not** belong to a rigid body will be added to the group. The group is always named 'nonrigid'.

Examples:

```
nonrigid = group.nonrigid()
```

hoomd.group.**rigid**()

Groups particles that belong to rigid bodies.

Creates a particle group from particles. All particles that belong to a rigid body will be added to the group. The group is always named 'rigid'.

Examples:

```
rigid = group.rigid()
```

hoomd.group.**rigid_center**()

Groups particles that are center particles of rigid bodies.

Creates a particle group from particles. All particles that are central particles of rigid bodies be added to the group. The group is always named 'rigid_center'.

Examples:

```
rigid = group.rigid_center()
```

hoomd.group.**tag_list**(*name*, *tags*)

Groups particles by tag list.

Parameters

- **tags** (*list*) – List of particle tags to include in the group
- **name** (*str*) – User-assigned name for this group.

Creates a particle group from particles with the given tags. Can be used to implement advanced grouping not available with existing group commands.

Examples:

```
a = group.tag_list(name="a", tags = [0, 12, 18, 205])
b = group.tag_list(name="b", tags = range(20,400))
```

hoomd.group.**tags**(*tag_min*, *tag_max=None*, *name=None*, *update=False*)

Groups particles by tag.

Parameters

- **tag_min** (*int*) – First tag in the range to include (inclusive)
- **tag_max** (*int*) – Last tag in the range to include (inclusive)

- **name** (*str*) – User-assigned name for this group. If a name is not specified, a default one will be generated.
- **update** (*bool*) – When True, update list of group members when particles are added to or removed from the simulation.

Creates a particle group from particles that match the given tag range.

The *tag_max* is optional. If it is not specified, then a single particle with `tag=tag_min` will be added to the group.

Examples:

```
half1 = group.tags(name="first-half", tag_min=0, tag_max=999)
half2 = group.tags(name="second-half", tag_min=1000, tag_max=1999)
```

hoomd.group.**type**(*type*, *name=None*, *update=False*)

Groups particles by type.

Parameters

- **type** (*str*) – Name of the particle type to add to the group.
- **name** (*str*) – User-assigned name for this group. If a name is not specified, a default one will be generated.
- **update** (*bool*) – When true, update list of group members when particles are added to or removed from the simulation.

Creates a particle group from particles that match the given type. The group can then be used by other hoomd commands (such as analyze.msd) to specify which particles should be operated on.

Note: Membership in *hoomd.group.type()* is defined at time of group creation. Once created, any particles added to the system will be added to the group if *update* is set to *True*. However, if you change a particle type it will not be added to or removed from this group.

Between runs, you can force a group to update its membership with the particles currently in the originally specified type using *hoomd.group.group.force_update()*.

Examples:

```
groupA = group.type(name='a-particles', type='A')
groupB = group.type(name='b-particles', type='B')
groupB = group.type(name='b-particles', type='B',update=True)
```

hoomd.group.**union**(*name*, *a*, *b*)

Create a new group from the set union of two existing groups.

Parameters

- **name** (*str*) – User-assigned name for this group.
- **a** (*group*) – First group.
- **b** (*group*) – Second group.

A new group is created that contains all particles present in either group *a* or *b*, and is given the name *name*.

Warning: The group is static and will not update if particles are added to or removed from the system.

Examples:

```
groupA = group.type(name='groupA', type='A')
groupB = group.type(name='groupB', type='B')
groupAB = group.union(name="ab-particles", a=groupA, b=groupB)
```

12.10 hoomd.init

Overview

hoomd.init.create_lattice	Create a lattice.
hoomd.init.read_getar	Initialize a system from a trajectory archive (.tar, .getar, .sqlite) file.
hoomd.init.read_gsd	Read initial system state from an GSD file.
hoomd.init.read_snapshot	Initializes the system from a snapshot.

Details

Data initialization commands

Commands in the *hoomd.init* package initialize the particle system.

hoomd.init.**create_lattice**(*unitcell*, *n*)

Create a lattice.

Parameters

- **unitcell** (*hoomd.lattice.unitcell*) – The unit cell of the lattice.
- **n** (*list*) – Number of replicates in each direction.

create_lattice() take a unit cell and replicates it the requested number of times in each direction. The resulting simulation box is commensurate with the given unit cell. A generic *hoomd.lattice.unitcell* may have arbitrary vectors $\vec{a}_1$, $\vec{a}_2$, and $\vec{a}_3$. *create_lattice()* will rotate the unit cell so that $\vec{a}_1$ points in the x direction and $\vec{a}_2$ is in the xy plane so that the lattice may be represented as a HOOMD simulation box.

When *n* is a single value, the lattice is replicated *n* times in each direction. When *n* is a list, the lattice is replicated *n[0]* times in the $\vec{a}_1$ direction, *n[1]* times in the $\vec{a}_2$ direction and *n[2]* times in the $\vec{a}_3$ direction.

Examples:

```
hoomd.init.create_lattice(unitcell=hoomd.lattice.sc(a=1.0),
                          n=[2,4,2]);

hoomd.init.create_lattice(unitcell=hoomd.lattice.bcc(a=1.0),
                          n=10);

hoomd.init.create_lattice(unitcell=hoomd.lattice.sq(a=1.2),
                          n=[100,10]);

hoomd.init.create_lattice(unitcell=hoomd.lattice.hex(a=1.0),
                          n=[100,58]);
```

hoomd.init.**read_getar**(*filename*, *modes={'any': 'any'}*)

Initialize a system from a trajectory archive (.tar, .getar, .sqlite) file. Returns a HOOMD *system_data* object.

Parameters

- **filename** (*str*) – Name of the file to read from
- **modes** (*dict*) – dictionary of {property: frame} values; see below

Getar files are a simple interface on top of archive formats (such as zip and tar) for storing trajectory data efficiently. A more thorough description of the format and a description of a python API to read and write these files is available at the libgetar documentation.

The **modes** argument is a dictionary. The keys of this dictionary should be either property names (see the Supported Property Table below) or tuples of property names.

If the key is a tuple of property names, data for those names will be restored from the same frame. Other acceptable keys are "any" to restore any properties which are present from the file, "angle_any" to restore any angle-related properties present, "bond_any", and so forth. The values associated with each key in the dictionary should be "any" (in which case any frame present for the data will be restored, even if the frames are different for two property names in a tuple), "latest" (grab the most recent frame data), "earliest", or a specific timestep value.

Example:

```
# creating file to initialize beforehand using libgetar
with gtar.GTAR('init.zip', 'w') as traj:
    traj.writePath('position.f32.ind', positions)
    traj.writePath('velocity.f32.ind', velocities)
    traj.writePath('metadata.json', json.dumps(metadata))
system = hoomd.init.read_getar('init.zip')
# using the backup created in the `hoomd.dump.getar.simple` example
system = hoomd.init.read_getar('backup.tar')
```

Supported Property Table

Name	Type	Size	Shape	Notes
	[String]		(N_angle_types,)	list containing the name of each angle type in JSON format
	int		(N_angle, 3)	list of particle tags for each angle interaction
	int		(N_angle,)	list of angle interaction types
			(N, 4)	particle angular momentum quaternion
			(N,)	particle rigid body index
	[String]		(N_bond_types,)	containing the

hoomd.init.**read_gsd**(*filename*, *restart=None*, *frame=0*, *time_step=None*)
Read initial system state from an GSD file.

Parameters

- **filename** (*str*) – File to read.
- **restart** (*str*) – If it exists, read the file *restart* instead of *filename*.
- **frame** (*int*) – Index of the frame to read from the GSD file. Negative values index from the end of the file.
- **time_step** (*int*) – (if specified) Time step number to initialize instead of the one stored in the GSD file.

All particles, bonds, angles, dihedrals, impropers, constraints, and box information are read from the given GSD file at the given frame index. To read and write GSD files outside of hoomd, see http://gsd.readthedocs.io/. *hoomd.dump.gsd* writes GSD files.

For restartable jobs, specify the initial condition in *filename* and the restart file in *restart*. *hoomd.init.read_gsd()* will read the restart file if it exists, otherwise it will read *filename*.

If *time_step* is specified, its value will be used as the initial time step of the simulation instead of the one read from the GSD file *filename*. *time_step* is not applied when the file *restart* is read.

The result of *hoomd.init.read_gsd()* can be saved in a variable and later used to read and/or change particle properties later in the script. See *hoomd.data* for more information.

See also:

hoomd.dump.gsd

hoomd.init.**read_snapshot**(*snapshot*)
Initializes the system from a snapshot.

Parameters **snapshot** (*hoomd.data* snapshot) – The snapshot to initialize the system.

Snapshots temporarily store system data. Snapshots contain the complete simulation state in a single object. Snapshots are set to time_step 0, and should not be used to restart a simulation.

Example use cases in which a simulation may be started from a snapshot include user code that generates initial particle positions.

Example:

```
snapshot = my_system_create_routine(.. parameters ..)
system = init.read_snapshot(snapshot)
```

See also:

hoomd.data

hoomd.init.**restore_getar**(*filename*, *modes={'any': 'any'}*)
Restore a subset of the current system's parameters from a trajectory archive (.tar, .zip, .sqlite) file. For a detailed discussion of arguments, see *read_getar()*.

Parameters

- **filename** (*str*) – Name of the file to read from
- **modes** (*dict*) – dictionary of {property: frame} values, as described in *read_getar()*

12.11 hoomd.lattice

Overview

hoomd.lattice.bcc	Create a body centered cubic lattice (3D).
hoomd.lattice.fcc	Create a face centered cubic lattice (3D).
hoomd.lattice.hex	Create a hexagonal lattice (2D).
hoomd.lattice.sc	Create a simple cubic lattice (3D).
hoomd.lattice.sq	Create a square lattice (2D).
hoomd.lattice.unitcell	Define a unit cell.

Details

Define lattices.

hoomd.lattice provides a general interface to define lattices to initialize systems.

See also:

hoomd.init.create_lattice().

hoomd.lattice.**bcc**(*a*, *type_name='A'*)

Create a body centered cubic lattice (3D).

Parameters

- **a** (*float*) – Lattice constant.
- **type_name** (*str*) – Particle type name.

The body centered cubic unit cell has 2 particles:

$$\vec{r} = \begin{pmatrix} 0 & 0 & 0 \\ \frac{a}{2} & \frac{a}{2} & \frac{a}{2} \end{pmatrix}$$

And the box matrix:

$$\mathbf{h} = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{pmatrix}$$

hoomd.lattice.**fcc**(*a*, *type_name='A'*)

Create a face centered cubic lattice (3D).

Parameters

- **a** (*float*) – Lattice constant.
- **type_name** (*str*) – Particle type name.

The face centered cubic unit cell has 4 particles:

$$\vec{r} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & \frac{a}{2} & \frac{a}{2} \\ \frac{a}{2} & 0 & \frac{a}{2} \\ \frac{a}{2} & \frac{a}{2} & 0 \end{pmatrix}$$

And the box matrix:

$$\mathbf{h} = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{pmatrix}$$

hoomd.lattice.**hex**(*a*, *type_name='A'*)

Create a hexagonal lattice (2D).

Parameters

- **a** (*float*) – Lattice constant.
- **type_name** (*str*) – Particle type name.

hex creates a hexagonal lattice in a rectangular box. It has 2 particles, one at the corner and one at the center of the rectangle. This is not the primitive unit cell, but is more convenient to work with because of its shape.

$$\vec{r} = \begin{pmatrix} 0 & 0 \\ \frac{a}{2} & \sqrt{3}\frac{a}{2} \end{pmatrix}$$

And the box matrix:

$$\mathbf{h} = \begin{pmatrix} a & 0 \\ 0 & \sqrt{3}a \\ 0 & 0 \end{pmatrix}$$

hoomd.lattice.**sc**(*a*, *type_name='A'*)

Create a simple cubic lattice (3D).

Parameters

- **a** (*float*) – Lattice constant.
- **type_name** (*str*) – Particle type name.

The simple cubic unit cell has 1 particle:

$$\vec{r} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$$

And the box matrix:

$$\mathbf{h} = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{pmatrix}$$

hoomd.lattice.**sq**(*a*, *type_name='A'*)

Create a square lattice (2D).

Parameters

- **a** (*float*) – Lattice constant.
- **type_name** (*str*) – Particle type name.

The simple square unit cell has 1 particle:

$$\vec{r} = \begin{pmatrix} 0 & 0 \end{pmatrix}$$

And the box matrix:

$$\mathbf{h} = \begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix}$$

class hoomd.lattice.**unitcell**(*N*, *a1*, *a2*, *a3*, *dimensions=3*, *position=None*, *type_name=None*, *mass=None*, *charge=None*, *diameter=None*, *moment_inertia=None*, *orientation=None*)

Define a unit cell.

Parameters

- **N** (*int*) – Number of particles in the unit cell.
- **a1** (*list*) – Lattice vector (3-vector).
- **a2** (*list*) – Lattice vector (3-vector).
- **a3** (*list*) – Lattice vector (3-vector). Set to [0,0,1] in 2D lattices.
- **dimensions** (*int*) – Dimensionality of the lattice (2 or 3).
- **position** (*list*) – List of particle positions.
- **type_name** (*list*) – List of particle type names.
- **mass** (*list*) – List of particle masses.
- **charge** (*list*) – List of particle charges.
- **diameter** (*list*) – List of particle diameters.
- **moment_inertia** (*list*) – List of particle moments of inertia.
- **orientation** (*list*) – List of particle orientations.

A unit cell is a box definition (*a1*, *a2*, *a3*, *dimensions*), and particle properties for *N* particles. You do not need to specify all particle properties. Any property omitted will be initialized to defaults (see *hoomd.data.make_snapshot()*). *hoomd.init.create_lattice* initializes the system with many copies of a unit cell.

unitcell is a completely generic unit cell representation. See other classes in the *hoomd.lattice* module for convenience wrappers for common lattices.

Example:

```
uc = hoomd.lattice.unitcell(N = 2,
                            a1 = [1,0,0],
                            a2 = [0.2,1.2,0],
                            a3 = [-0.2,0, 1.0],
                            dimensions = 3,
                            position = [[0,0,0], [0.5, 0.5, 0.5]],
                            type_name = ['A', 'B'],
                            mass = [1.0, 2.0],
                            charge = [0.0, 0.0],
                            diameter = [1.0, 1.3],
                            moment_inertia = [[1.0, 1.0, 1.0], [0.0, 0.0, 0.0]],
                            orientation = [[0.707, 0, 0, 0.707], [1.0, 0, 0, 0]]);
```

Note: *a1*, *a2*, *a3* must define a right handed coordinate system.

get_snapshot()

Get a snapshot.

Returns A snapshot representing the lattice.

> **Attention:** HOOMD-blue requires upper-triangular box matrices. The general box matrix *(a1, a2, a3)* set for this *unitcell* and the particle positions and orientations will be rotated from provided values into upper triangular form.

get_type_list()
Get a list of the unique type names in the unit cell.

Returns A list of the unique type names present in the unit cell.

get_typeid_mapping()
Get a type name to typeid mapping.

Returns A dict that maps type names to integer type ids.

12.12 hoomd.meta

Overview

hoomd.meta.dump_metadata	Writes simulation metadata into a file.

Details

Write out simulation and environment context metadata.

Metadata is stored in form of key-value pairs in a JSON file and used to summarize the per-run simulation parameters so that they can be easily taken up by other scripts and stored in a database.

Example:

```
metadata = meta.dump_metadata()
meta.dump_metadata(filename = "metadata.json", user = {'debug': True}, indent=2)
```

hoomd.meta.**dump_metadata**(*filename=None*, *user=None*, *indent=4*)
Writes simulation metadata into a file.

Parameters

- **filename** (*str*) – The name of the file to write JSON metadata to (optional)
- **user** (*dict*) – Additional metadata.
- **indent** (*int*) – The json indentation size

Returns metadata as a dictionary

When called, this function will query all registered forces, updaters etc. and ask them to provide metadata. E.g. a pair potential will return information about parameters, the Logger will output the filename it is logging to, etc.

Custom metadata can be provided as a dictionary to *user*.

The output is aggregated into a dictionary and written to a JSON file, together with a timestamp. The file is overwritten if it exists.

12.13 hoomd.option

Overview

hoomd.option.get_user	Get user options.
hoomd.option.set_autotuner_params	Set autotuner parameters.
hoomd.option.set_msg_file	Set the message file.
hoomd.option.set_notice_level	Set the notice level.

Details

Set global options.

Options may be set on the command line or from a job script using *hoomd.context.initialize()*. The option.set_* commands override any settings made previously.

hoomd.option.**get_user**()
Get user options.

Returns List of user options passed in via –user="arg1 arg2 ..."

hoomd.option.**set_autotuner_params**(*enable=True*, *period=100000*)
Set autotuner parameters.

Parameters

- **enable** (*bool*) –
- **period** (*int*) – Approximate period in time steps between retuning.

TODO: reference autotuner page here.

hoomd.option.**set_msg_file**(*fname*)
Set the message file.

Parameters **fname** (*str*) – Specifies the name of the file to write. The file will be overwritten. Set to None to direct messages back to stdout/stderr.

The message file may be changed before or after initialization, and may be changed many times during a job script. Changing the message file will only affect messages sent after the change.

Note: Overrides `--msg-file` on the command line.

---

hoomd.option.**set_notice_level**(*notice_level*)
Set the notice level.

Parameters **notice_level** (*int*) –

The notice level may be changed before or after initialization, and may be changed many times during a job script.

Note: Overrides `--notice-level` on the command line.

---

hoomd.option.**set_num_threads**(*num_threads*)
Set the number of CPU (TBB) threads HOOMD uses

Parameters **num_threads** (*int*) – The number of threads

Note: Overrides `--nthreads` on the command line.

12.14 hoomd.update

Overview

hoomd.update.balance	Adjusts the boundaries of a domain decomposition on a regular 3D grid.
hoomd.update.box_resize	Rescale the system box size.
hoomd.update.sort	Sorts particles in memory to improve cache coherency.

Details

Modify the system state periodically.

When an updater is specified, it acts on the particle system every *period* steps to change it in some way. See the documentation of specific updaters to find out what they do.

class hoomd.update.**balance**(*x=True*, *y=True*, *z=True*, *tolerance=1.02*, *maxiter=1*, *period=1000*, *phase=0*)

Adjusts the boundaries of a domain decomposition on a regular 3D grid.

Parameters

- **x** (*bool*) – If True, balance in x dimension.
- **y** (*bool*) – If True, balance in y dimension.
- **z** (*bool*) – If True, balance in z dimension.
- **tolerance** (*float*) – Load imbalance tolerance (if <= 1.0, balance every step).
- **maxiter** (*int*) – Maximum number of iterations to attempt in a single step.
- **period** (*int*) – Balancing will be attempted every a period time steps
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.

Every *period* steps, the boundaries of the processor domains are adjusted to distribute the particle load close to evenly between them. The load imbalance is defined as the number of particles owned by a rank divided by the average number of particles per rank if the particles had a uniform distribution:

$$I = \frac{N(i)}{N/P}$$

where :math:‘ N(i) ‘ is the number of particles on processor i, N is the total number of particles, and P is the number of ranks.

In order to adjust the load imbalance, the sizes are rescaled by the inverse of the imbalance factor. To reduce oscillations and communication overhead, a domain cannot move more than 5% of its current size in a single rebalancing step, and the edge of a domain cannot move more than half the distance to its neighbors.

Simulations with interfaces (so that there is a particle density gradient) or clustering should benefit from load balancing. The potential speedup is roughly $I - 1.0$, so that if the largest imbalance is 1.4, then the user can expect a roughly 40% speedup in the simulation. This is of course an estimate that assumes that all algorithms are roughly linear in N, all GPUs are fully occupied, and the simulation is limited by the speed of the slowest processor. It also assumes that all particles roughly equal. If you have a simulation where, for example, some particles have significantly more pair force neighbors than others, this estimate of the load imbalance may not produce the optimal results.

A load balancing adjustment is only performed when the maximum load imbalance exceeds a *tolerance*. The ideal load balance is 1.0, so setting *tolerance* less than 1.0 will force an adjustment every *period*. The load balancer can attempt multiple iterations of balancing every *period*, and up to *maxiter* attempts can be made. The optimal values of *period* and *maxiter* will depend on your simulation.

Load balancing can be performed independently and sequentially for each dimension of the simulation box. A small performance increase may be obtained by disabling load balancing along dimensions that are known to be homogeneous. For example, if there is a planar vapor-liquid interface normal to the z axis, then it may be advantageous to disable balancing along x and y.

In systems that are well-behaved, there is minimal overhead of balancing with a small *period*. However, if the system is not capable of being balanced (for example, due to the density distribution or minimum domain size), having a small *period* and high *maxiter* may lead to a large performance loss. In such systems, it is currently best to either balance infrequently or to balance once in a short test run and then set the decomposition statically in a separate initialization.

Balancing is ignored if there is no domain decomposition available (MPI is not built or is running on a single rank).

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*x=None*, *y=None*, *z=None*, *tolerance=None*, *maxiter=None*)
Change load balancing parameters.

Parameters

- **x** (*bool*) – If True, balance in x dimension.
- **y** (*bool*) – If True, balance in y dimension.
- **z** (*bool*) – If True, balance in z dimension.

- **tolerance** (*float*) – Load imbalance tolerance (if <= 1.0, balance every step).
- **maxiter** (*int*) – Maximum number of iterations to attempt in a single step.

Examples:

```
balance.set_params(x=True, y=False)
balance.set_params(tolerance=0.02, maxiter=5)
```

set_period(*period*)

Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.update.**box_resize**(*Lx=None*, *Ly=None*, *Lz=None*, *xy=None*, *xz=None*, *yz=None*, *period=1*, *L=None*, *phase=0*, *scale_particles=True*)

Rescale the system box size.

Parameters

- **L** (*hoomd.variant*) – (if set) box length in the x,y, and z directions as a function of time (in distance units)
- **Lx** (*hoomd.variant*) – (if set) box length in the x direction as a function of time (in distance units)
- **Ly** (*hoomd.variant*) – (if set) box length in the y direction as a function of time (in distance units)
- **Lz** (*hoomd.variant*) – (if set) box length in the z direction as a function of time (in distance units)
- **xy** (*hoomd.variant*) – (if set) X-Y tilt factor as a function of time (dimensionless)
- **xz** (*hoomd.variant*) – (if set) X-Z tilt factor as a function of time (dimensionless)
- **yz** (*hoomd.variant*) – (if set) Y-Z tilt factor as a function of time (dimensionless)
- **period** (*int*) – The box size will be updated every *period* time steps.
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.
- **scale_particles** (*bool*) – When True (the default), scale particles into the new box. When False, do not change particle positions when changing the box.

Every *period* time steps, the system box dimensions is updated to values given by the user (in a variant). As an option, the particles can either be left in place as the box is changed or their positions can be scaled with the box.

Note: If *period* is set to None, then the given box lengths are applied immediately and periodic updates are not performed.

L, Lx, Ly, Lz, xy, xz, yz can either be set to a constant number or a *hoomd.variant*. if any of the box parameters are not specified, they are set to maintain the same value in the current box.

Use L as a shorthand to specify Lx, Ly, and Lz to the same value.

By default, particle positions are rescaled with the box. Set *scale_particles=False* to leave particles in place when changing the box.

If, under rescaling, tilt factors get too large, the simulation may slow down due to too many ghost atoms being communicated. *hoomd.update.box_resize* does NOT reset the box to orthorhombic shape if this occurs (and does not move the next periodic image into the primary cell).

Examples:

```
update.box_resize(L = hoomd.variant.linear_interp([(0, 20), (1e6, 50)]))
box_resize = update.box_resize(L = hoomd.variant.linear_interp([(0, 20), (1e6,␣
↪50)]), period = 10)
update.box_resize(Lx = hoomd.variant.linear_interp([(0, 20), (1e6, 50)]),
                  Ly = hoomd.variant.linear_interp([(0, 20), (1e6, 60)]),
                  Lz = hoomd.variant.linear_interp([(0, 10), (1e6, 80)]))
update.box_resize(Lx = hoomd.variant.linear_interp([(0, 20), (1e6, 50)]), Ly = 10,
↪ Lz = 10)

# Shear the box in the xy plane using Lees-Edwards boundary conditions
update.box_resize(xy = hoomd.variant.linear_interp([(0,0), (1e6, 1)]))
```

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.update.**sort**

Sorts particles in memory to improve cache coherency.

Warning: Do not specify *hoomd.update.sort* explicitly in your script. HOOMD creates a sorter by default.

Every *period* time steps, particles are reordered in memory based on a Hilbert curve. This operation is very efficient, and the reordered particles significantly improve performance of all other algorithmic steps in HOOMD.

The reordering is accomplished by placing particles in spatial bins. A Hilbert curve is generated that traverses these bins and particles are reordered in memory in the same order in which they fall on the curve. The grid dimension used over the course of the simulation is held constant, and the default is chosen to be as fine as possible without utilizing too much memory. The grid size can be changed with *set_params()*.

Warning: Memory usage by the sorter grows quickly with the grid size:

- grid=128 uses 8 MB
- grid=256 uses 64 MB
- grid=512 uses 512 MB
- grid=1024 uses 4096 MB

Note: 2D simulations do not use any additional memory and default to grid=4096.

A sorter is created by default. To disable it or modify parameters, save the context and access the sorter through it:

```
c = context.initialize();
hoomd.init.create_random(N=1000, phi_p=0.2)
# the sorter is only available after initialization
c.sorter.disable()
```

disable()

Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()

Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*grid=None*)
Change sorter parameters.

Parameters **grid** (*int*) – New grid dimension (if set)

Examples:: sorter.set_params(grid=128)

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

12.15 hoomd.util

Overview

hoomd.util.cuda_profile_start	Start CUDA profiling.
hoomd.util.cuda_profile_stop	Stop CUDA profiling.
hoomd.util.quiet_status	Quiet the status line output.
hoomd.util.unquiet_status	Resume the status line output.

Details

Utilities.

hoomd.util.**cuda_profile_start**()
Start CUDA profiling.

When using nvvp to profile CUDA kernels in hoomd jobs, you usually don't care about all the initialization and startup. cuda_profile_start() allows you to not even record that. To use, uncheck the box "start profiling on application start" in your nvvp session configuration. Then, call cuda_profile_start() in your hoomd script when you want nvvp to start collecting information.

Example:

```
from hoomd import *
init.read_xml("init.xml");
# setup....
run(30000);  # warm up and auto-tune kernel block sizes
option.set_autotuner_params(enable=False);  # prevent block sizes from further␣
↪autotuning
```

(continues on next page)

(continued from previous page)

```
cuda_profile_start();
run(100);
```

hoomd.util.**cuda_profile_stop**()
Stop CUDA profiling.

See also:

cuda_profile_start().

hoomd.util.**quiet_status**()
Quiet the status line output.

After calling *hoomd.util.quiet_status()*, hoomd will no longer print out the line of code that executes each hoomd script command. Call *hoomd.util.unquiet_status()* to enable the status messages again. Messages are only enabled after a number of *hoomd.util.unquiet_status()* calls equal to the number of prior *hoomd.util.quiet_status()* calls.

hoomd.util.**unquiet_status**()
Resume the status line output.

See also:

hoomd.util.quiet_status()

12.16 hoomd.variant

Overview

hoomd.variant.linear_interp	Linearly interpolated variant.

Details

Specify values that vary over time.

This package contains various commands for creating quantities that can vary smoothly over the course of a simulation. For example, set the temperature in a NVT simulation to slowly heat or cool the system over a long simulation.

class hoomd.variant.**linear_interp**(*points*, *zero='now'*)
Linearly interpolated variant.

Parameters

- **points** (*list*) – Set points in the linear interpolation (see below)
- **zero** (*int*) – Specify absolute time step number location for 0 in *points*. Use 'now' to indicate the current step.

hoomd.variant.linear_interp creates a time-varying quantity where the value at each time step is determined by linear interpolation between a given set of points.

At time steps before the initial point, the value is identical to the value at the first given point. At time steps after the final point, the value is identical to the value at the last given point. All points between are determined by linear interpolation.

Time steps given to *hoomd.variant.linear_interp* are relative to the current step of the simulation, and starts counting from 0 at the time of creation. Set *zero* to control the relative starting point.

points is a list of `(time step, set value)` tuples. For example, to specify a series of points that goes from 10 at time step 0 to 20 at time step 100 and then back down to 5 at time step 200:

```
points = [(0, 10), (100, 20), (200, 5)]
```

Any number of points can be specified in any order. However, listing them monotonically increasing in time will result in a much more human readable set of values.

Examples:

```
L = variant.linear_interp(points = [(0, 10), (100, 20), (200, 5)])
V = variant.linear_interp(points = [(0, 10), (1e6, 20)], zero=80000)
integrate.nvt(group=all, tau = 0.5,
    T = variant.linear_interp(points = [(0, 1.0), (1e5, 2.0)])
```

12.17 hoomd.hdf5

Overview

hoomd.hdf5.File
hoomd.hdf5.log

Details

Commands that require the h5py package at runtime.

All commands that are part of this module require the h5py package a python API for hdf5. In addition, this module is an opt-in. As a consequence you'll need to import it via *import hoomd.hdf5* before you can use any command.

class hoomd.hdf5.**File**(**args*, ***kwargs*)

Thin wrapper of the h5py.File class.

This class ensures, that opening and close operations within a context manager are only executed on the root MPI rank.

Note: This class can be used like the h5py.File class, but the user has to make sure that all operations are only executed on the root rank.

---

class hoomd.hdf5.**log**(*h5file*, *period*, *quantities=[]*, *matrix_quantities=[]*, *phase=0*)

Log a number of calculated quantities or matrices to a hdf5 file.

Parameters

- **h5file** (*hoomd.hdf5.File*) – Instance describing the opened h5file.
- **period** (*int*) – Quantities are logged every *period* time steps
- **quantities** (*list*) – Quantities to log.
- **matrix_quantities** (*list*) – Matrix quantities to log.
- **overwrite** (*bool*) – When False (the default) the existing log will be append. When True the file will be overwritten.

- **phase** (*int*) – When -1, start on the current time step. When >= 0 execute on steps where *(step +phase) % period == 0*.

For details on the loggable quantities refer *hoomd.analyze.log* for details.

The non-matrix quantities are combined in an array 'quantities' in the hdf5 file. The attributes list all the names of the logged quantities and their position in the file.

Matrix quantities are logged as a separate data set each in the file. The name of the data set corresponds to the name of the quantity. The first dimension of the data set is counting the logged time step. The other dimension correspond to the dimensions of the logged matrix.

Note: The number and order of non-matrix quantities cannot change compared to data which is already stored in the hdf5 file. As a result, if you append to a file make sure you are logging the same values as before. In addition, also during a run with multiple *hoomd.run()* commands the logged values can not change.

Note: The dimensions of logged matrix quantities cannot change compared to a matrix with same name stored in the file. This applies for appending files as well as during a single simulation run.

Examples:

```
with hoomd.hdf5.File("log.h5", "w") as h5file:
   #general setup

   log = hoomd.hdf5.log(filename='log.h5', quantities=['my_quantity', 'cosm'],␣
↪matrix_quantities = ['random_matrix'], period=100)
   log.register_callback('my_quantity', lambda timestep: timestep**2)
   log.register_callback('cosm', lambda timestep: math.cos(logger.query('my_
↪quantity')))
   def random_matrix(timestep):
      return numpy.random.rand(23, 56)
   log.register_callback('random_matrix', random_matrix, True)
   #more setup
   run(200)
```

disable()

Disable the logger.

Examples:

```
logger.disable()
```

Executing the disable command will remove the logger from the system. Any *hoomd.run()* command executed after disabling the logger will not use that logger during the simulation. A disabled logger can be re-enabled with *enable()*.

enable()

Enables the logger

Examples:

```
logger.enable()
```

See *disable()*.

query(*quantity*, *force_matrix=False*)

Get the last logged value of a quantity which has been written to the file. If quantity is registered as a non-matrix quantity, its value is returned. If it is not registered as a non-matrix quantity, it is assumed to be a matrix quantity. If a quantity exists as non-matrix and matrix quantity with the same name, force_matrix can be used to obtain the matrix value.

Parameters

- **quantity** (*str*) – name of the quantity to query
- **force_matrix** (*bool*) – the name of the quantity is

Note: Matrix quantities are not efficiently cached by the class, so calling this function multiple time, may not be efficient.

---

register_callback(*name*, *callback*, *matrix=False*)

Register a callback to produce a logged quantity.

Parameters

- **name** (*str*) – Name of the quantity
- **callback** (*callable*) – A python callable object (i.e. a lambda, function, or class that implements __call__)
- **matrix** (*bool*) – Is the callback a computing a matrix and thus returning a numpy array instead of a single float?

The callback method must take a single argument, the current timestep, and return a single floating point value to be logged. If the callback returns a matrix quantity the return value must be a numpy array constant dimensions of each call.

Note: One callback can query the value of another, but logged quantities are evaluated in order from left to right.

Examples:

```
log = hoomd.hdf5.log(filename='log.h5', quantities=['my_quantity', 'cosm'],␣
↪matrix_quantities = ['random_matrix'], period=100)
log.register_callback('my_quantity', lambda timestep: timestep**2)
log.register_callback('cosm', lambda timestep: math.cos(logger.query('my_
↪quantity')))
def random_matrix(timestep):
    return numpy.random.rand(23, 56)
log.register_callback('random_matrix', random_matrix, True)
```

set_params(*quantities=None*, *matrix_quantities=None*)

Change the parameters of the log.

Warning: Do not change the number or order of logged non-matrix quantities compared to values stored in the file.

CHAPTER 13

hpmc

Details

Hard particle Monte Carlo

HPMC performs hard particle Monte Carlo simulations of a variety of classes of shapes.

Overview

HPMC implements hard particle Monte Carlo in HOOMD-blue.

Logging

The following quantities are provided by the integrator for use in HOOMD-blue's *hoomd.analyze.log*.

- `hpmc_sweep` - Number of sweeps completed since the start of the MC integrator
- `hpmc_translate_acceptance` - Fraction of translation moves accepted (averaged only over the last time step)
- `hpmc_rotate_acceptance` - Fraction of rotation moves accepted (averaged only over the last time step)
- `hpmc_d` - Maximum move displacement
- `hpmc_a` - Maximum rotation move
- `hpmc_move_ratio` - Probability of making a translation move (1- P(rotate move))
- `hpmc_overlap_count` - Count of the number of particle-particle overlaps in the current system configuration

With non-interacting depletant (**implicit=True**), the following log quantities are available:

- `hpmc_fugacity` - The current value of the depletant fugacity (in units of density, volume^-1)
- `hpmc_ntrial` - The current number of configurational bias attempts per overlapping depletant

- `hpmc_insert_count` - Number of depletants inserted per colloid
- `hpmc_reinsert_count` - Number of overlapping depletants reinserted per colloid by configurational bias MC
- `hpmc_free_volume_fraction` - Fraction of free volume to total sphere volume after a trial move has been proposed (sampled inside a sphere around the new particle position)
- `hpmc_overlap_fraction` - Fraction of depletants in excluded volume after trial move to depletants in free volume before move
- `hpmc_configurational_bias_ratio` - Ratio of configurational bias attempts to depletant insertions

With patch energies defined, the following quantities are available: - `hpmc_patch_energy` - The potential energy of the system resulting from the patch interaction. - `hpmc_patch_rcut` - The cutoff radius in the patch energy interaction.

`compute.free_volume` provides the following loggable quantities: - `hpmc_free_volume` - The free volume estimate in the simulation box obtained by MC sampling (in volume units)

`update.boxmc` provides the following loggable quantities:

- `hpmc_boxmc_trial_count` - Number of box changes attempted since the start of the boxmc updater
- `hpmc_boxmc_volume_acceptance` - Fraction of volume/length change trials accepted (averaged from the start of the last run)
- `hpmc_boxmc_ln_volume_acceptance` - Fraction of log(volume) change trials accepted (averaged from the start of the last run)
- `hpmc_boxmc_shear_acceptance` - Fraction of shear trials accepted (averaged from the start of the last run)
- `hpmc_boxmc_aspect_acceptance` - Fraction of aspect trials accepted (averaged from the start of the last run)
- `hpmc_boxmc_betaP` Current value of the βp value of the boxmc updater

`update.muvt` provides the following loggable quantities.

- `hpmc_muvt_insert_acceptance` - Fraction of particle insertions accepted (averaged from start of run)
- `hpmc_muvt_remove_acceptance` - Fraction of particle removals accepted (averaged from start of run)
- `hpmc_muvt_volume_acceptance` - Fraction of particle removals accepted (averaged from start of run)

`update.clusters()` provides the following loggable quantities.

- `hpmc_clusters_moves` - Fraction of cluster moves divided by the number of particles
- `hpmc_clusters_pivot_acceptance` - Fraction of pivot moves accepted
- `hpmc_clusters_reflection_acceptance` - Fraction of reflection moves accepted
- `hpmc_clusters_swap_acceptance` - Fraction of swap moves accepted
- `hpmc_clusters_avg_size` - Average cluster size

Timestep definition

HOOMD-blue started as an MD code where **timestep** has a clear meaning. MC simulations are run for timesteps. In exact terms, this means different things on the CPU and GPU and something slightly different when using MPI. The behavior is approximately normalized so that user scripts do not need to drastically change run() lengths when switching from one execution resource to another.

In the GPU implementation, one trial move is applied to a number of randomly chosen particles in each cell during one timestep. The number of selected particles is `nselect*ceil(avg particles per cell)` where *nselect* is a user-chosen parameter. The default value of *nselect* is 4, which achieves optimal performance for a wide variety of benchmarks. Detailed balance is obeyed at the level of a timestep. In short: One timestep **is NOT equal** to one sweep, but is approximately *nselect* sweeps, which is an overestimation.

In the single-threaded CPU implementation, one trial move is applied *nselect* times to each of the *N* particles during one timestep. In parallel MPI runs, one trial moves is applied *nselect* times to each particle in the active region. There is a small strip of inactive region near the boundaries between MPI ranks in the domain decomposition. The trial moves are performed in a shuffled order so detailed balance is obeyed at the level of a timestep. In short: One timestep **is approximately** *nselect* sweeps (*N* trial moves). In single-threaded runs, the approximation is exact, but it is slightly underestimated in MPI parallel runs.

To approximate a fair comparison of dynamics between CPU and GPU timesteps, log the `hpmc_sweep` quantity to get the number sweeps completed so far at each logged timestep.

See J. A. Anderson et. al. 2016 for design and implementation details.

Stability

hoomd.hpmc is **stable**. When upgrading from version 2.x to 2.y (y > x), existing job scripts that follow *documented* interfaces for functions and classes will not require any modifications. **Maintainer:** Joshua A. Anderson

Modules

13.1 hpmc.analyze

Overview

hpmc.analyze.sdf

Details

Compute properties of hard particle configurations.

class hoomd.hpmc.analyze.**sdf**(*mc*, *filename*, *xmax*, *dx*, *navg*, *period*, *overwrite=False*, *phase=0*)

Compute the scale distribution function.

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator.
- **filename** (*str*) – Output file name.
- **xmax** (*float*) – Maximum *x* value at the right hand side of the rightmost bin (distance units).
- **dx** (*float*) – Bin width (distance units).
- **navg** (*int*) – Number of times to average before writing the histogram to the file.
- **period** (*int*) – Number of timesteps between histogram evaluations.
- **overwrite** (*bool*) – Set to True to overwrite *filename* instead of appending to it.

- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.

sdf computes a distribution function of scale parameters x. For each particle, it finds the smallest scale factor $1 + x$ that would cause the particle to touch one of its neighbors and records that in the histogram $s(x)$. The histogram is discrete and $s(x_i) = s[i]$ where $x_i = i \cdot dx + dx/2$.

In an NVT simulation, the extrapolation of $s(x)$ to $x = 0$, $s(0+)$ is related to the pressure.

$$\frac{P}{kT} = \rho \left(1 + \frac{s(0+)}{2d} \right)$$

where d is the dimensionality of the system and ρ is the number density.

Extrapolating $s(0+)$ is not trivial. Here are some suggested parameters, but they may not work in all cases.

- *xmax* = 0.02
- *dx* = 1e-4
- Polynomial curve fit of degree 5.

In systems near densest packings, `dx=1e-5` may be needed along with either a smaller xmax or a smaller region to fit. A good rule of thumb might be to fit a region where `numpy.sum(s[0:n]*dx)` ~ 0.5 - but this needs further testing to confirm.

sdf averages *navg* histograms together before writing them out to a text file in a plain format: "timestep bin_0 bin_1 bin_2 .... bin_n".

sdf works well with restartable jobs. Ensure that `navg*period` is an integer fraction $1/k$ of the restart period. Then *sdf* will have written the final output to its file just before the restart gets written. The new data needed for the next line of values is entirely collected after the restart.

Warning: *sdf* does not compute correct pressures for simulations with concave particles.

Numpy extrapolation code:

```
def extrapolate(s, dx, xmax, degree=5):
  # determine the number of values to fit
  n_fit = int(math.ceil(xmax/dx));
  s_fit = s[0:n_fit];
  # construct the x coordinates
  x_fit = numpy.arange(0,xmax,dx)
  x_fit += dx/2;
  # perform the fit and extrapolation
  p = numpy.polyfit(x_fit, s_fit, degree);
  return numpy.polyval(p, 0.0);
```

Examples:

```
mc = hpmc.integrate.sphere(seed=415236)
analyze.sdf(mc=mc, filename='sdf.dat', xmax=0.02, dx=1e-4, navg=100, period=100)
analyze.sdf(mc=mc, filename='sdf.dat', xmax=0.002, dx=1e-5, navg=100, period=100)
```

disable()

Disable the analyzer.

Examples:

```
my_analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any *hoomd.run()* command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with *enable()*.

enable()
Enables the analyzer

Examples:

```
my_analyzer.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the period between analyzer executions

Parameters **period** (*int*) – New period to set (in time steps)

Examples:

```
analyzer.set_period(100)
analyzer.set_period(1)
```

While the simulation is running (*hoomd.run()*, the action of each analyzer is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

13.2 hpmc.compute

Overview

hpmc.compute.free_volume	Compute the free volume available to a test particle by stochastic integration.

Details

Compute properties of hard particle configurations.

class hoomd.hpmc.compute.**free_volume**(*mc*, *seed*, *suffix=''*, *test_type=None*, *nsample=None*)
Compute the free volume available to a test particle by stochastic integration.

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator.
- **seed** (*int*) – Random seed for MC integration.
- **type** (*str*) – Type of particle to use for integration
- **nsample** (*int*) – Number of samples to use in MC integration
- **suffix** (*str*) – Suffix to use for log quantity

:py:class‘free_volume‘ computes the free volume of a particle assembly using stochastic integration with a test particle type. It works together with an HPMC integrator, which defines the particle types used in the simulation. As parameters it requires the number of MC integration samples (*nsample*), and the type of particle (*test_type*) to use for the integration.

Once initialized, the compute provides a log quantity called **hpmc_free_volume**, that can be logged via *hoomd.analyze.log*. If a suffix is specified, the log quantities name will be **hpmc_free_volume_suffix**.

Examples:

```
mc = hpmc.integrate.sphere(seed=415236)
compute.free_volume(mc=mc, seed=123, test_type='B', nsample=1000)
log = analyze.log(quantities=['hpmc_free_volume'], period=100, filename='log.dat',
↪ overwrite=True)
```

disable()
Disables the compute.

Examples:

```
c.disable()
```

Executing the disable command will remove the compute from the system. Any *hoomd.run()* command executed after disabling a compute will not be able to log computed values with *hoomd.analyze.log*.

A disabled compute can be re-enabled with *enable()*.

enable()
Enables the compute.

Examples:

```
c.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

13.3 hpmc.data

Overview

hpmc.data.param_dict	Manage shape parameters.

Details

Shape data structures.

class hoomd.hpmc.data.**param_dict**(*mc*)
Manage shape parameters.

The parameters for all hpmc integrator shapes (*hoomd.hpmc.integrate*) are specified using this class. Parameters are specified per particle type. Every HPMC integrator has a member shape_param that can read and set parameters of the shapes.

param_dict can be used as a dictionary to access parameters by type. You can read individual parameters or set parameters with *set()*.

Example:

```
mc = hpmc.integrate.sphere();
mc.shape_param['A'].set(diameter=2.0)
mc.shape_param['B'].set(diameter=0.1)
dA = mc.shape_param['A'].diameter
dB = mc.shape_param['B'].diameter
```

set (*types*, ***params*)

Sets parameters for particle type(s).

Parameters

- **type** (*str*) – Particle type (string) or list of types
- **params** – Named parameters (see specific integrator for required parameters - *hoomd.hpmc.integrate*)

Calling set() results in one or more parameters being set for a shape. Types are identified by name, and parameters are also added by name. Which parameters you need to specify depends on the hpmc integrator you are setting these coefficients for, see the corresponding documentation.

All possible particle types types defined in the simulation box must be specified before executing *hoomd.run()*. You will receive an error if you fail to do so. It is an error to specify coefficients for particle types that do not exist in the simulation.

To set the same parameters for many particle types, provide a list of type names instead of a single one. All types in the list will be set to the same parameters. A convenient wildcard that lists all types of particles in the simulation can be gotten from a saved *sysdef* from the init command.

Examples:

```
mc.shape_param.set('A', diameter=1.0)
mc.shape_param.set('B', diameter=2.0)
mc.shape_param.set(['A', 'B'], diameter=2.0)
```

Note: Single parameters can not be updated. If both *diameter* and *length* are required for a particle type, then executing coeff.set('A', diameter=1.5) will fail one must call coeff.set('A', diameter=1.5, length=2.0)

13.4 hpmc.field

Overview

hpmc.field.callback	Use a python-defined energy function in MC integration
hpmc.field.external_field_composite	Manage multiple external fields.
hpmc.field.frenkel_ladd_energy	Compute the Frenkel-Ladd Energy of a crystal.
hpmc.field.lattice_field	Restrain particles on a lattice
hpmc.field.wall	Manage walls (an external field type).

Details

Apply external fields to HPMC simulations.

class hoomd.hpmc.field.**callback**(*mc*, *energy_function*, *composite=False*)
Use a python-defined energy function in MC integration

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator.
- **callback** (*callable*) – A python function to evaluate the energy of a configuration
- **composite** (*bool*) – True if this evaluator is part of a composite external field

Example:

```
def energy(snapshot):
    # evaluate the energy in a linear potential gradient along the x-axis
    gradient = (5,0,0)
    e = 0
    for p in snap.particles.position:
        e -= numpy.dot(gradient,p)
    return e

mc = hpmc.integrate.sphere(seed=415236);
mc.shape_param.set('A',diameter=1.0)
hpmc.field.callback(mc=mc, energy_function=energy);
run(100)
```

disable()
Disables the compute.

Examples:

```
c.disable()
```

Executing the disable command will remove the compute from the system. Any *hoomd.run()* command executed after disabling a compute will not be able to log computed values with *hoomd.analyze.log*.

A disabled compute can be re-enabled with *enable()*.

enable()
Enables the compute.

Examples:

```
c.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

class hoomd.hpmc.field.**external_field_composite**(*mc*, *fields=None*)
Manage multiple external fields.

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator (don't specify a new integrator later, external_field_composite will continue to use the old one)
- **fields** (*list*) – List of external fields to combine together.

external_field_composite allows the user to create and compute multiple external fields. Once created use *add_field()* to add a new field.

Once initialized, the compute provides a log quantities that other external fields create. See those external fields to find the quantities.

Examples:

```
mc = hpmc.integrate.shape(...);
walls = hpmc.field.walls(...)
lattice = hpmc.field.lattice(...)
composite_field = hpmc.field.external_field_composite(mc, fields=[walls, lattice])
```

add_field(*fields*)
Add an external field to the ensemble.

Parameters **fields** (*list*) – list of fields to add

Example:

```
mc = hpmc.integrate.shape(...);
composite_field = hpmc.compute.external_field_composite(mc)
walls = hpmc.compute.walls(..., setup=False)
lattice = hpmc.compute.lattice(..., setup=False)
composite_field.add_field(fields=[walls, lattice])
```

disable()
Disables the compute.

Examples:

```
c.disable()
```

Executing the disable command will remove the compute from the system. Any *hoomd.run()* command executed after disabling a compute will not be able to log computed values with *hoomd.analyze.log*.

A disabled compute can be re-enabled with *enable()*.

enable()
Enables the compute.

Examples:

```
c.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

class hoomd.hpmc.field.**frenkel_ladd_energy**(*mc*, *ln_gamma*, *q_factor*, *r0*, *q0*, *drift_period*, *symmetry=[]*)
Compute the Frenkel-Ladd Energy of a crystal.

Parameters

- **ln_gamma** (*float*) – log of the translational spring constant
- **q_factor** (*float*) – scale factor between the translational spring constant and rotational spring constant
- **r0** (*list*) – reference lattice positions

- **q0** (*list*) – reference lattice orientations
- **drift_period** (*int*) – period call the remove drift updater

frenkel_ladd_energy interacts with *lattice_field* and *hoomd.hpmc.update.remove_drift*.

Once initialized, the compute provides the log quantities from the *lattice_field*.

Example:

```
mc = hpmc.integrate.convex_polyhedron(seed=seed);
mc.shape_param.set("A", vertices=verts)
mc.set_params(d=0.005, a=0.005)
#set the FL parameters
fl = hpmc.compute.frenkel_ladd_energy(mc=mc, ln_gamma=0.0, q_factor=10.0, r0=rs,␣
↪q0=qs, drift_period=1000)
```

disable()
Disables the compute.

Examples:

```
c.disable()
```

Executing the disable command will remove the compute from the system. Any *hoomd.run()* command executed after disabling a compute will not be able to log computed values with *hoomd.analyze.log*.

A disabled compute can be re-enabled with *enable()*.

enable()
Enables the compute.

Examples:

```
c.enable()
```

See *disable()*.

reset_statistics()
Reset the statistics counters.

Example:

```
mc = hpmc.integrate.sphere(seed=415236);
fl = hpmc.compute.frenkel_ladd_energy(mc=mc, ln_gamma=0.0, q_factor=10.0,␣
↪r0=rs, q0=qs, drift_period=1000)
ks = np.linspace(1000, 0.01, 100);
for k in ks:
  fl.set_params(ln_gamma=math.log(k), q_factor=10.0);
  fl.reset_statistics();
  run(1000)
```

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*ln_gamma=None*, *q_factor=None*)
Set the Frenkel-Ladd parameters.

Parameters

- **ln_gamma** (*float*) – log of the translational spring constant

- **q_factor** (*float*) – scale factor between the translational spring constant and rotational spring constant

Example:

```
mc = hpmc.integrate.sphere(seed=415236);
fl = hpmc.compute.frenkel_ladd_energy(mc=mc, ln_gamma=0.0, q_factor=10.0,␣
↪r0=rs, q0=qs, drift_period=1000)
ks = np.linspace(1000, 0.01, 100);
for k in ks:
  fl.set_params(ln_gamma=math.log(k), q_factor=10.0);
  fl.reset_statistics();
  run(1000)
```

class hoomd.hpmc.field.**lattice_field**(*mc*, *position=[]*, *orientation=[]*, *k=0.0*, *q=0.0*, *symmetry=[]*, *composite=False*)

Restrain particles on a lattice

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator.
- **position** (*list*) – list of positions to restrain each particle (distance units).
- **orientation** (*list*) – list of orientations to restrain each particle (quaternions).
- **k** (*float*) – translational spring constant.
- **q** (*float*) – rotational spring constant.
- **symmetry** (*list*) – list of equivalent quaternions for the shape.
- **composite** (*bool*) – Set this to True when this field is part of a *external_field_composite*.

lattice_field specifies that a harmonic spring is added to every particle:

$$V_i(r) = k_r * (r_i - r_{oi})^2$$
$$V_i(q) = k_q * (q_i - q_{oi})^2$$

Note: 1/2 is not included in the formulas, specify your spring constants accordingly.

- k_r - translational spring constant.
- r_o - lattice positions (in distance units).
- k_q - rotational spring constant.
- q_o - lattice orientations (quaternion)

Once initialized, the compute provides the following log quantities that can be logged via analyze.log:

- **lattice_energy** – total lattice energy
- **lattice_energy_pp_avg** – average lattice energy per particle
- **lattice_energy_pp_sigma** – standard deviation of the lattice energy per particle
- **lattice_translational_spring_constant** – translational spring constant
- **lattice_rotational_spring_constant** – rotational spring constant
- **lattice_num_samples** – number of samples used to compute the average and standard deviation

Example:

```
mc = hpmc.integrate.sphere(seed=415236);
hpmc.field.lattice_field(mc=mc, position=fcc_lattice, k=1000.0);
log = analyze.log(quantities=['lattice_energy'], period=100, filename='log.dat',␣
↪overwrite=True);
```

disable()
Disables the compute.

Examples:

```
c.disable()
```

Executing the disable command will remove the compute from the system. Any *hoomd.run()* command executed after disabling a compute will not be able to log computed values with *hoomd.analyze.log*.

A disabled compute can be re-enabled with *enable()*.

enable()
Enables the compute.

Examples:

```
c.enable()
```

See *disable()*.

get_average_energy()

Get the average energy per particle of the lattice field. This is a collective call and must be called on all ranks.

Example:: mc = hpmc.integrate.sphere(seed=415236); lattice = hpmc.field.lattice_field(mc=mc, position=fcc_lattice, k=exp(15)); run(20000) avg_eng = lattice.get_average_energy() // should be about 1.5kT

get_energy()

Get the current energy of the lattice field. This is a collective call and must be called on all ranks.

Example:: mc = hpmc.integrate.sphere(seed=415236); lattice = hpmc.field.lattice_field(mc=mc, position=fcc_lattice, k=1000.0); run(20000) eng = lattice.get_energy()

get_sigma_energy()

Gives the standard deviation of the average energy per particle of the lattice field. This is a collective call and must be called on all ranks.

Example:: mc = hpmc.integrate.sphere(seed=415236); lattice = hpmc.field.lattice_field(mc=mc, position=fcc_lattice, k=exp(15)); run(20000) sig_eng = lattice.get_sigma_energy()

reset(*timestep=None*)
Reset the statistics counters.

Parameters **timestep** (*int*) – the timestep to pass into the reset function.

Example:

```
mc = hpmc.integrate.sphere(seed=415236);
lattice = hpmc.field.lattice_field(mc=mc, position=fcc_lattice, k=1000.0);
ks = np.linspace(1000, 0.01, 100);
for k in ks:
```

(continues on next page)

(continued from previous page)

```
  lattice.set_params(k=k, q=0.0);
  lattice.reset();
  run(1000)
```

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*k*, *q*)
Set the translational and rotational spring constants.

Parameters

- **k** (*float*) – translational spring constant.
- **q** (*float*) – rotational spring constant.

Example:

```
mc = hpmc.integrate.sphere(seed=415236);
lattice = hpmc.field.lattice_field(mc=mc, position=fcc_lattice, k=1000.0);
ks = np.linspace(1000, 0.01, 100);
for k in ks:
  lattice.set_params(k=k, q=0.0);
  run(1000)
```

set_references(*position=[]*, *orientation=[]*)
Reset the reference positions or reference orientations.

Parameters

- **position** (*list*) – list of positions to restrain each particle.
- **orientation** (*list*) – list of orientations to restrain each particle.

Example:

```
mc = hpmc.integrate.sphere(seed=415236);
lattice = hpmc.field.lattice_field(mc=mc, position=fcc_lattice, k=1000.0);
lattice.set_references(position=bcc_lattice)
```

class hoomd.hpmc.field.**wall**(*mc*, *composite=False*)
Manage walls (an external field type).

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator.
- **composite** (*bool*) – Set this to True when this field is part of a *external_field_composite*.

wall allows the user to implement one or more walls. If multiple walls are added, then particles are confined by the INTERSECTION of all of these walls. In other words, particles are confined by all walls if they independently satisfy the confinement condition associated with each separate wall. Once you've created an instance of this class, use *add_sphere_wall()* to add a new spherical wall, *add_cylinder_wall()* to add a new cylindrical wall, or *add_plane_wall()* to add a new plane wall.

Specialized overlap checks have been written for supported combinations of wall types and particle shapes. These combinations are: * Sphere particles: sphere walls, cylinder walls, plane walls * Convex polyhedron particles: sphere walls, cylinder walls, plane walls * Convex spheropolyhedron particles: sphere walls

Once initialized, the compute provides the following log quantities that can be logged via *hoomd.analyze.log*:

- **hpmc_wall_volume** : the volume associated with the intersection of implemented walls. This number is only meaningful if the user has initially provided it through *set_volume()*. It will subsequently change when the box is resized and walls are scaled appropriately.
- **hpmc_wall_sph_rsq-i** : the squared radius of the spherical wall indexed by i, beginning at 0 in the order the sphere walls were added to the system.
- **hpmc_wall_cyl_rsq-i** : the squared radius of the cylindrical wall indexed by i, beginning at 0 in the order the cylinder walls were added to the system.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
ext_wall.set_volume(4./3.*np.pi);
log = analyze.log(quantities=['hpmc_wall_volume','hpmc_wall_sph_rsq-0'],␣
↪period=100, filename='log.dat', overwrite=True);
```

add_cylinder_wall(*radius*, *origin*, *orientation*, *inside=True*)

Add a cylindrical wall to the simulation.

Parameters

- **radius** (*float*) – radius of cylindrical wall
- **origin** (*tuple*) – origin (center) of cylindrical wall
- **orientation** (*tuple*) – vector that defines the direction of the long axis of the cylinder. will be normalized automatically by hpmc.
- **inside** (*bool*) – When True, then particles are CONFINED by the wall if they exist entirely inside the cylinder (in the portion of connected space that contains the origin). When False, then particles are CONFINED by the wall if they exist entirely outside the cylinder (in the portion of connected space that does not contain the origin). DEFAULTS to True.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_cylinder_wall(radius = 1.0, origin = [0, 0, 0], orientation = [0,
↪ 0, 1], inside = True);
```

add_plane_wall(*normal*, *origin*)

Add a plane wall to the simulation.

Parameters

- **normal** (*tuple*) – vector normal to the plane. this, in combination with a point on the plane, defines the plane entirely. It will be normalized automatically by hpmc. The direction of the normal vector defines the confinement condition associated with the plane wall. If every part of a particle exists in the halfspace into which the normal points, then that particle is CONFINED by the plane wall.
- **origin** (*tuple*) – a point on the plane wall. this, in combination with the normal vector, defines the plane entirely.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_plane_wall(normal = [0, 0, 1], origin = [0, 0, 0]);
```

add_sphere_wall(*radius*, *origin*, *inside=True*)
Add a spherical wall to the simulation.

Parameters

- **radius** (*float*) – radius of spherical wall
- **origin** (*tuple*) – origin (center) of spherical wall.
- **inside** (*bool*) – When True, particles are CONFINED by the wall if they exist entirely inside the sphere (in the portion of connected space that contains the origin). When False, then particles are CONFINED by the wall if they exist entirely outside the sphere (in the portion of connected space that does not contain the origin).

Quick Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
```

count_overlaps(*exit_early=False*)
Count the overlaps associated with the walls.

Parameters **exit_early** (*bool*) – When True, stop counting overlaps after the first one is found.

Returns The number of overlaps associated with the walls

A particle "overlaps" with a wall if it fails to meet the confinement condition associated with the wall.

Example

mc = hpmc.integrate.sphere(seed = 415236); ext_wall = hpmc.compute.wall(mc); ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True); run(100) num_overlaps = ext_wall.count_overlaps();

disable()
Disables the compute.

Examples:

```
c.disable()
```

Executing the disable command will remove the compute from the system. Any *hoomd.run()* command executed after disabling a compute will not be able to log computed values with *hoomd.analyze.log*.

A disabled compute can be re-enabled with *enable()*.

enable()
Enables the compute.

Examples:

```
c.enable()
```

See *disable()*.

get_curr_box()
Get the simulation box that the wall class is currently storing.

Returns The boxdim object that the wall class is currently storing.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
ext_wall.set_volume(4./3.*np.pi);
run(100)
curr_box = ext_wall.get_curr_box();
```

get_cylinder_wall_param(*index*, *param*)
Access a parameter associated with a particular cylinder wall.

Parameters

- **index** (*int*) – index of the cylinder wall to be accessed. indices begin at 0 in the order the cylinder walls were added to the system.
- **param** (*str*) – name of parameter to be accessed. options are "rsq" (squared radius of cylinder wall), "origin" (origin of cylinder wall), "orientation" (orientation of cylinder wall), and "inside" (confinement condition associated with cylinder wall).

Returns Value of queried parameter.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_cylinder_wall(radius = 1.0, origin = [0, 0, 0], orientation = [0,
↪ 0, 1], inside = True);
rsq = ext_wall.get_cylinder_wall_param(index = 0, param = "rsq");
```

get_num_cylinder_walls()
Get the current number of cylinder walls in the simulation.

Returns The current number of cylinder walls in the simulation.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_cylinder_wall(radius = 1.0, origin = [0, 0, 0], orientation = [0,
↪ 0, 1], inside = True);
num_cyl_walls = ext_wall.get_num_cylinder_walls();
```

get_num_plane_walls()
Get the current number of plane walls in the simulation.

Returns The current number of plane walls in the simulation.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_plane_wall(normal = [0, 0, 1], origin = [0, 0, 0]);
num_plane_walls = ext_wall.get_num_plane_walls();
```

get_num_sphere_walls()
Get the current number of sphere walls in the simulation.

Returns: the current number of sphere walls in the simulation

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
num_sph_walls = ext_wall.get_num_sphere_walls();
```

get_plane_wall_param(*index*, *param*)
Access a parameter associated with a particular plane wall.

Parameters

- **index** (*int*) – index of the plane wall to be accessed. indices begin at 0 in the order the plane walls were added to the system.
- **param** (*str*) – name of parameter to be accessed. options are "normal" (vector normal to the plane wall), and "origin" (point on the plane wall)

Returns Value of queried parameter.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_plane_wall(normal = [0, 0, 1], origin = [0, 0, 0]);
n = ext_wall.get_plane_wall_param(index = 0, param = "normal");
```

get_sphere_wall_param(*index*, *param*)
Access a parameter associated with a particular sphere wall.

Parameters

- **index** (*int*) – index of the sphere wall to be accessed. indices begin at 0 in the order the sphere walls were added to the system.
- **param** (*str*) – name of parameter to be accessed. options are "rsq" (squared radius of sphere wall), "origin" (origin of sphere wall), and "inside" (confinement condition associated with sphere wall)

Returns Value of queried parameter.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
rsq = ext_wall.get_sphere_wall_param(index = 0, param = "rsq");
```

get_volume()
Get the current volume associated with the intersection of all walls in the system.

If this quantity has not previously been set by the user, this returns a meaningless value.

Returns The current volume associated with the intersection of all walls in the system.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
ext_wall.set_volume(4./3.*np.pi);
run(100)
curr_vol = ext_wall.get_volume();
```

remove_cylinder_wall(*index*)
Remove a particular cylinder wall from the simulation.

Parameters **index** (*int*) – index of the cylinder wall to be removed. indices begin at 0 in the order the cylinder walls were added to the system.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_cylinder_wall(radius = 1.0, origin = [0, 0, 0], orientation = [0,
↪ 0, 1], inside = True);
ext_wall.remove_cylinder_wall(index = 0);
```

remove_plane_wall(*index*)
Remove a particular plane wall from the simulation.

Parameters **index** (*int*) – index of the plane wall to be removed. indices begin at 0 in the order the plane walls were added to the system.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_plane_wall(normal = [0, 0, 1], origin = [0, 0, 0]);
ext_wall.remove_plane_wall(index = 0);
```

remove_sphere_wall(*index*)
Remove a particular sphere wall from the simulation.

Parameters **index** (*int*) – index of the sphere wall to be removed. indices begin at 0 in the order the sphere walls were added to the system.

Quick Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
ext_wall.remove_sphere_wall(index = 0);
```

restore_state()
Restore the state information from the file used to initialize the simulations

set_curr_box(*Lx=None*, *Ly=None*, *Lz=None*, *xy=None*, *xz=None*, *yz=None*)
Set the simulation box that the wall class is currently storing.

You may want to set this independently so that you can cleverly control whether or not the walls actually scale in case you manually resize your simulation box. The walls scale automatically when they get the signal that the global box, associated with the system definition, has scaled. They do so, however, with a scale factor associated with the ratio of the volume of the global box to the volume of the box that the walls class is currently storing. (After the scaling the box that the walls class is currently storing is updated appropriately.) If you want to change the simulation box WITHOUT scaling the walls, then, you must first

update the simulation box that the walls class is storing, THEN update the global box associated with the system definition.

Example:

```
init_box = hoomd.data.boxdim(L=10, dimensions=3);
snap = hoomd.data.make_snapshot(N=1, box=init_box, particle_types=['A']);
system = hoomd.init.read_snapshot(snap);
system.particles[0].position = [0,0,0];
system.particles[0].type = 'A';
mc = hpmc.integrate.sphere(seed = 415236);
mc.shape_param.set('A', diameter = 2.0);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 3.0, origin = [0, 0, 0], inside = True);
ext_wall.set_curr_box(Lx=2.0*init_box.Lx, Ly=2.0*init_box.Ly, Lz=2.0*init_box.
↪Lz, xy=init_box.xy, xz=init_box.xz, yz=init_box.yz);
system.sysdef.getParticleData().setGlobalBox(ext_wall.get_curr_box()._
↪getBoxDim())
```

set_cylinder_wall(*index*, *radius*, *origin*, *orientation*, *inside=True*)

Change the parameters associated with a particular cylinder wall.

Parameters

- **index** (*int*) – index of the cylinder wall to be modified. indices begin at 0 in the order the cylinder walls were added to the system.
- **radius** (*float*) – New radius of cylindrical wall
- **origin** (*tuple*) – New origin (center) of cylindrical wall
- **orientation** (*tuple*) – New vector that defines the direction of the long axis of the cylinder. will be normalized automatically by hpmc.
- **inside** (*bool*) – New confinement condition. When True, then particles are CONFINED by the wall if they exist entirely inside the cylinder (in the portion of connected space that contains the origin). When False, then particles are CONFINED by the wall if they exist entirely outside the cylinder (in the portion of connected space that does not contain the origin). DEFAULTS to True.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_cylinder_wall(radius = 1.0, origin = [0, 0, 0], orientation = [0,
↪ 0, 1], inside = True);
ext_wall.set_cylinder_wall(index = 0, radius = 3.0, origin = [0, 0, 0],␣
↪orientation = [0, 0, 1], inside = True);
```

set_plane_wall(*index*, *normal*, *origin*)

Change the parameters associated with a particular plane wall.

Parameters

- **index** (*int*) – index of the plane wall to be modified. indices begin at 0 in the order the plane walls were added to the system.
- **normal** (*tuple*) – new vector normal to the plane. this, in combination with a point on the plane, defines the plane entirely. It will be normalized automatically by hpmc. The direction of the normal vector defines the confinement condition associated with the plane

wall. If every part of a particle exists in the halfspace into which the normal points, then that particle is CONFINED by the plane wall.

- **origin** (*tuple*) – new point on the plane wall. this, in combination with the normal vector, defines the plane entirely.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_plane_wall(normal = [0, 0, 1], origin = [0, 0, 0]);
ext_wall.set_plane_wall(index = 0, normal = [0, 0, 1], origin = [0, 0, 1]);
```

set_sphere_wall(*index*, *radius*, *origin*, *inside=True*)

Change the parameters associated with a particular sphere wall.

Parameters

- **index** (*int*) – index of the sphere wall to be modified. indices begin at 0 in the order the sphere walls were added to the system.
- **radius** (*float*) – New radius of spherical wall
- **origin** (*tuple*) – New origin (center) of spherical wall.
- **inside** (*bool*) – New confinement condition. When True, particles are CONFINED by the wall if they exist entirely inside the sphere (in the portion of connected space that contains the origin). When False, then particles are CONFINED by the wall if they exist entirely outside the sphere (in the portion of connected space that does not contain the origin).

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
ext_wall.set_sphere_wall(index = 0, radius = 3.0, origin = [0, 0, 0], inside␣
↪= True);
```

set_volume(*volume*)

Set the volume associated with the intersection of all walls in the system.

This number will subsequently change when the box is resized and walls are scaled appropriately.

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
ext_wall.set_volume(4./3.*np.pi);
```

13.5 hpmc.integrate

Overview

hpmc.integrate.convex_polygon	HPMC integration for convex polygons (2D).

Continued on next page

Table 5 – continued from previous page

hpmc.integrate.convex_polyhedron	HPMC integration for convex polyhedra (3D).
hpmc.integrate.convex_polyhedron_union	HPMC integration for unions of convex polyhedra (3D).
hpmc.integrate.convex_spheropolygon	HPMC integration for convex spheropolygons (2D).
hpmc.integrate.convex_spheropolyhedron	HPMC integration for spheropolyhedra (3D).
hpmc.integrate.convex_spheropolyhedron_union	HPMC integration for unions of convex polyhedra (3D).
hpmc.integrate.ellipsoid	HPMC integration for ellipsoids (2D/3D).
hpmc.integrate.faceted_sphere	HPMC integration for faceted spheres (3D).
hpmc.integrate.interaction_matrix	Define pairwise interaction matrix
hpmc.integrate.mode_hpmc	Base class HPMC integrator.
hpmc.integrate.polyhedron	HPMC integration for general polyhedra (3D).
hpmc.integrate.simple_polygon	HPMC integration for simple polygons (2D).
hpmc.integrate.sphere	HPMC integration for spheres (2D/3D).
hpmc.integrate.sphere_union	HPMC integration for unions of spheres (3D).
hpmc.integrate.sphinx	HPMC integration for sphinx particles (3D).

Details

class hoomd.hpmc.integrate.**convex_polygon**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *restore_state=False*)

HPMC integration for convex polygons (2D).

Parameters

- **seed** (*int*) – Random number seed
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

Note: For concave polygons, use *simple_polygon*.

Convex polygon parameters:

- *vertices* (**required**) - vertices of the polygon as is a list of (x,y) tuples of numbers (distance units)
  - Vertices **MUST** be specified in a *counter-clockwise* order.
  - The origin **MUST** be contained within the vertices.
  - Points inside the polygon **MUST NOT** be included.
  - The origin centered circle that encloses all vertices should be of minimal size for optimal performance (e.g. don't put the origin right next to an edge).
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Warning: HPMC does not check that all requirements are met. Undefined behavior will result if they are violated.

Examples:

```
mc = hpmc.integrate.convex_polygon(seed=415236)
mc = hpmc.integrate.convex_polygon(seed=415236, d=0.3, a=0.4)
mc.shape_param.set('A', vertices=[(-0.5, -0.5), (0.5, -0.5), (0.5, 0.5), (-0.5, 0.
↪5)]);
print('vertices = ', mc.shape_param['A'].vertices)
```

get_type_shapes()

Get all the types of shapes in the current simulation :returns: A list of dictionaries, one for each particle type in the system. Currently assumes that all 3D shapes are convex.

class hoomd.hpmc.integrate.**convex_polyhedron**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*, *max_verts=None*, *restore_state=False*)

HPMC integration for convex polyhedra (3D).

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – (Override the automatic choice for the number of trial moves to perform in each cell.
- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **max_verts** (*int*) – Set the maximum number of vertices in a polyhedron. (deprecated in version 2.2)
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

Convex polyhedron parameters:

- *vertices* (**required**) - vertices of the polyhedron as is a list of (x,y,z) tuples of numbers (distance units)
  - The origin **MUST** be contained within the vertices.
  - The origin centered circle that encloses all vertices should be of minimal size for optimal performance (e.g. don't put the origin right next to a face).
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Warning: HPMC does not check that all requirements are met. Undefined behavior will result if they are violated.

Example:

```
mc = hpmc.integrate.convex_polyhedron(seed=415236)
mc = hpmc.integrate.convex_polyhedron(seed=415236, d=0.3, a=0.4)
mc.shape_param.set('A', vertices=[(0.5, 0.5, 0.5), (0.5, -0.5, -0.5), (-0.5, 0.5,␣
↪-0.5), (-0.5, -0.5, 0.5)]);
print('vertices = ', mc.shape_param['A'].vertices)
```

Depletants Example:

```
mc = hpmc.integrate.convex_polyhedron(seed=415236, d=0.3, a=0.4, implicit=True,␣
↪depletant_mode='circumsphere')
mc.set_param(nselect=1,nR=3,depletant_type='B')
mc.shape_param.set('A', vertices=[(0.5, 0.5, 0.5), (0.5, -0.5, -0.5), (-0.5, 0.5,␣
↪-0.5), (-0.5, -0.5, 0.5)]);
mc.shape_param.set('B', vertices=[(0.05, 0.05, 0.05), (0.05, -0.05, -0.05), (-0.
↪05, 0.05, -0.05), (-0.05, -0.05, 0.05)]);
```

get_type_shapes()

Get all the types of shapes in the current simulation :returns: A list of dictionaries, one for each particle type in the system. Currently assumes that all 3D shapes are convex.

class hoomd.hpmc.integrate.**convex_polyhedron_union**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*)

HPMC integration for unions of convex polyhedra (3D).

Deprecated since version 2.4: Replaced by *convex_spheropolyhedron_union*. This class stays in place for compatibility with older scripts.

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **max_members** (*int*) – Set the maximum number of members in the convex polyhedron union
- **capacity** (*int*) – Set to the number of constituent convex polyhedra per leaf node

New in version 2.2.

Convex polyhedron union parameters:

- *vertices* (**required**) - list of vertex lists of the polyhedra in particle coordinates.
- *centers* (**required**) - list of centers of constituent polyhedra in particle coordinates.
- *orientations* (**required**) - list of orientations of constituent polyhedra.
- *overlap* (**default: 1 for all particles**) - only check overlap between constituent particles for which *overlap[i] & overlap[j]* is !=0, where '&' is the bitwise AND operator.
- *sweep_radii* (**default: 0 for all particle**) - radii of spheres sweeping out each constituent polyhedron
  - New in version 2.4.
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking.
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Example:

```
mc = hpmc.integrate.convex_polyhedron_union(seed=27)
mc = hpmc.integrate.convex_polyhedron_union(seed=27, d=0.3, a=0.4)
cube_verts = [[-1,-1,-1],[-1,-1,1],[-1,1,1],[-1,1,-1],
              [1,-1,-1],[1,-1,1],[1,1,1],[1,1,-1]]
mc.shape_param.set('A', vertices=[cube_verts, cube_verts],
                        centers=[[-1,0,0],[1,0,0]],orientations=[[1,0,0,0],[1,0,0,
↪0]]);
print('vertices of the first cube = ', mc.shape_param['A'].members[0].vertices)
print('center of the first cube = ', mc.shape_param['A'].centers[0])
print('orientation of the first cube = ', mc.shape_param['A'].orientations[0])
```

class hoomd.hpmc.integrate.**convex_spheropolygon**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *restore_state=False*)

HPMC integration for convex spheropolygons (2D).

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

Spheropolygon parameters:

- *vertices* (**required**) - vertices of the polygon as is a list of (x,y) tuples of numbers (distance units)
  - The origin **MUST** be contained within the shape.
  - The origin centered circle that encloses all vertices should be of minimal size for optimal performance (e.g. don't put the origin right next to an edge).

- *sweep_radius* (**default: 0.0**) - the radius of the sphere swept around the edges of the polygon (distance units) - **optional**
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Useful cases:

- A 1-vertex spheropolygon is a disk.
- A 2-vertex spheropolygon is a spherocylinder.

Warning: HPMC does not check that all requirements are met. Undefined behavior will result if they are violated.

Examples:

```
mc = hpmc.integrate.convex_spheropolygon(seed=415236)
mc = hpmc.integrate.convex_spheropolygon(seed=415236, d=0.3, a=0.4)
mc.shape_param.set('A', vertices=[(-0.5, -0.5), (0.5, -0.5), (0.5, 0.5), (-0.5, 0.
↪5)], sweep_radius=0.1, ignore_statistics=False);
mc.shape_param.set('A', vertices=[(0,0)], sweep_radius=0.5, ignore_
↪statistics=True);
print('vertices = ', mc.shape_param['A'].vertices)
```

get_type_shapes()
Get all the types of shapes in the current simulation :returns: A list of dictionaries, one for each particle type in the system. Currently assumes that all 3D shapes are convex.

class hoomd.hpmc.integrate.**convex_spheropolyhedron**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*, *max_verts=None*, *restore_state=False*)

HPMC integration for spheropolyhedra (3D).

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **max_verts** (*int*) – Set the maximum number of vertices in a polyhedron. (deprecated in version 2.2)

- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

A spheropolyhedron can also represent spheres (0 or 1 vertices), and spherocylinders (2 vertices).

Spheropolyhedron parameters:

- *vertices* (**required**) - vertices of the polyhedron as is a list of (x,y,z) tuples of numbers (distance units)
  - The origin **MUST** be contained within the vertices.
  - The origin centered sphere that encloses all vertices should be of minimal size for optimal performance (e.g. don't put the origin right next to a face).
  - A sphere can be represented by specifying zero vertices (i.e. vertices=[]) and a non-zero radius R
  - Two vertices and a non-zero radius R define a prolate spherocylinder.
- *sweep_radius* (**default: 0.0**) - the radius of the sphere swept around the edges of the polygon (distance units) - **optional**
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Warning: HPMC does not check that all requirements are met. Undefined behavior will result if they are violated.

Example:

```
mc = hpmc.integrate.convex_spheropolyhedron(seed=415236)
mc = hpmc.integrate.convex_spheropolyhedron(seed=415236, d=0.3, a=0.4)
mc.shape_param['tetrahedron'].set(vertices=[(0.5, 0.5, 0.5), (0.5, -0.5, -0.5), (-
↪0.5, 0.5, -0.5), (-0.5, -0.5, 0.5)]);
print('vertices = ', mc.shape_param['A'].vertices)
mc.shape_param['SphericalDepletant'].set(vertices=[], sweep_radius=0.1, ignore_
↪statistics=True);
```

Depletants example:

```
mc = hpmc.integrate.convex_spheropolyhedron(seed=415236, d=0.3, a=0.4,␣
↪implicit=True, depletant_mode='circumsphere')
mc.set_param(nR=3,depletant_type='SphericalDepletant')
mc.shape_param['tetrahedron'].set(vertices=[(0.5, 0.5, 0.5), (0.5, -0.5, -0.5), (-
↪0.5, 0.5, -0.5), (-0.5, -0.5, 0.5)]);
mc.shape_param['SphericalDepletant'].set(vertices=[], sweep_radius=0.1);
```

get_type_shapes()

Get all the types of shapes in the current simulation :returns: A list of dictionaries, one for each particle type in the system. Currently assumes that all 3D shapes are convex.

class hoomd.hpmc.integrate.**convex_spheropolyhedron_union**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*)

HPMC integration for unions of convex polyhedra (3D).

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **max_members** (*int*) – Set the maximum number of members in the convex polyhedron union
- **capacity** (*int*) – Set to the number of constituent convex polyhedra per leaf node

New in version 2.2.

Convex polyhedron union parameters:

- *vertices* (**required**) - list of vertex lists of the polyhedra in particle coordinates.
- *centers* (**required**) - list of centers of constituent polyhedra in particle coordinates.
- *orientations* (**required**) - list of orientations of constituent polyhedra.
- *overlap* (**default: 1 for all particles**) - only check overlap between constituent particles for which *overlap[i] & overlap[j]* is !=0, where '&' is the bitwise AND operator.
- *sweep_radii* (**default: 0 for all particle**) - radii of spheres sweeping out each constituent polyhedron
  - New in version 2.4.
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking.
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Example:

```
mc = hpmc.integrate.convex_spheropolyhedron_union(seed=27)
mc = hpmc.integrate.convex_spheropolyhedron_union(seed=27, d=0.3, a=0.4)
cube_verts = [[-1,-1,-1],[-1,-1,1],[-1,1,1],[-1,1,-1],
              [1,-1,-1],[1,-1,1],[1,1,1],[1,1,-1]]
mc.shape_param.set('A', vertices=[cube_verts, cube_verts],
                        centers=[[-1,0,0],[1,0,0]],orientations=[[1,0,0,0],[1,0,0,
↪0]]);
print('vertices of the first cube = ', mc.shape_param['A'].members[0].vertices)
print('center of the first cube = ', mc.shape_param['A'].centers[0])
print('orientation of the first cube = ', mc.shape_param['A'].orientations[0])
```

class hoomd.hpmc.integrate.**ellipsoid**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*, *restore_state=False*)

HPMC integration for ellipsoids (2D/3D).

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

Ellipsoid parameters:

- *a* (**required**) - principle axis a of the ellipsoid (radius in the x direction) (distance units)
- *b* (**required**) - principle axis b of the ellipsoid (radius in the y direction) (distance units)
- *c* (**required**) - principle axis c of the ellipsoid (radius in the z direction) (distance units)
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Example:

```
mc = hpmc.integrate.ellipsoid(seed=415236)
mc = hpmc.integrate.ellipsoid(seed=415236, d=0.3, a=0.4)
mc.shape_param.set('A', a=0.5, b=0.25, c=0.125);
print('ellipsoids parameters (a,b,c) = ', mc.shape_param['A'].a, mc.shape_param['A
↪'].b, mc.shape_param['A'].c)
```

Depletants Example:

```
mc = hpmc.integrate.ellipsoid(seed=415236, d=0.3, a=0.4, implicit=True, depletant_
↪mode='circumsphere')
mc.set_param(nselect=1,nR=50,depletant_type='B')
mc.shape_param.set('A', a=0.5, b=0.25, c=0.125);
mc.shape_param.set('B', a=0.05, b=0.05, c=0.05);
```

class hoomd.hpmc.integrate.**faceted_sphere**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*, *restore_state=False*)

HPMC integration for faceted spheres (3D).

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.

- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

A faceted sphere is a sphere intersected with halfspaces. The equation defining each halfspace is given by:

$$n_i \cdot r + b_i \leq 0$$

where n_i is the face normal, and b_i is the offset.

Warning: The origin must be chosen so as to lie **inside the shape**, or the overlap check will not work. This condition is not checked.

Faceted sphere parameters:

- *normals* (**required**) - list of (x,y,z) tuples defining the facet normals (distance units)
- *offsets* (**required**) - list of offsets (distance unit^2)
- *diameter* (**required**) - diameter of sphere
- *vertices* (**required**) - list of vertices for intersection polyhedron
- *origin* (**required**) - origin vector
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Warning: Planes must not be coplanar.

Example:

```
mc = hpmc.integrate.faceted_sphere(seed=415236)
mc = hpmc.integrate.faceted_sphere(seed=415236, d=0.3, a=0.4)
mc.shape_param.set('A', normals=[(-1,0,0),(1,0,0),(0,-1,0),(0,1,0),(0,0,-1),(0,0,
↪1)],diameter=1.0);
print('diameter = ', mc.shape_param['A'].diameter)
```

Depletants Example:

```
mc = hpmc.integrate.pathcy_sphere(seed=415236, d=0.3, a=0.4, implicit=True,␣
↪depletant_mode='circumsphere')
mc.set_param(nselect=1,nR=3,depletant_type='B')
```

(continues on next page)

(continued from previous page)

```
mc.shape_param.set('A', normals=[(-1,0,0),(1,0,0),(0,-1,0),(0,1,0),(0,0,-1),(0,0,
↪1)],diameter=1.0);
mc.shape_param.set('B', normals=[],diameter=0.1);
```

class hoomd.hpmc.integrate.**interaction_matrix**

Define pairwise interaction matrix

All shapes use *interaction_matrix* to define the interaction matrix between different pairs of particles indexed by type. The set of pair coefficients is a symmetric matrix defined over all possible pairs of particle types.

By default, all elements of the interaction matrix are 1, that means that overlaps are checked between all pairs of types. To disable overlap checking for a specific type pair, set the coefficient for that pair to 0.

Access the interaction matrix with a saved integrator object like so:

```
from hoomd import hpmc

mc = hpmc.integrate.some_shape(arguments...)
mv.overlap_checks.set('A', 'A', enable=False)
mc.overlap_checks.set('A', 'B', enable=True)
mc.overlap_checks.set('B', 'B', enable=False)
```

New in version 2.1.

set(*a*, *b*, *enable*)

Sets parameters for one type pair.

Parameters

- **a** (*str*) – First particle type in the pair (or a list of type names)
- **b** (*str*) – Second particle type in the pair (or a list of type names)
- **enable** – Set to True to enable overlap checks for this pair, False otherwise

By default, all interaction matrix elements are set to 'True'.

It is not an error, to specify matrix elements for particle types that do not exist in the simulation.

There is no need to specify matrix elements for both pairs 'A', 'B' and 'B', 'A'. Specifying only one is sufficient.

To set the same elements between many particle types, provide a list of type names instead of a single one. All pairs between the two lists will be set to the same parameters.

Examples:

```
mc.overlap_checks.set('A', 'A', False);
mc.overlap_checks.set('B', 'B', False);
mc.overlap_checks.set('A', 'B', True);
mc.overlap_checks.set(['A', 'B', 'C', 'D'], 'F', True);
mc.overlap_checks.set(['A', 'B', 'C', 'D'], ['A', 'B', 'C', 'D'], False);
```

class hoomd.hpmc.integrate.**mode_hpmc**(*implicit*, *depletant_mode=None*)

Base class HPMC integrator.

mode_hpmc is the base class for all HPMC integrators. It provides common interface elements. Users should not instantiate this class directly. Methods documented here are available to all hpmc integrators.

State data

HPMC integrators can save and restore the following state information to gsd files:

- Maximum trial move displacement *d*
- Maximum trial rotation move *a*
- Shape parameters for all types.

State data are *not* written by default. You must explicitly request that state data for an mc integrator is written to a gsd file (see *hoomd.dump.gsd.dump_state()*).

```
mc = hoomd.hpmc.shape(...)
gsd = hoomd.dump.gsd(...)
gsd.dump_state(mc)
```

State data are *not* restored by default. You must explicitly request that state data be restored when initializing the integrator.

```
init.read_gsd(...)
mc = hoomd.hpmc.shape(..., restore_state=True)
```

See the *State data* section of the HOOMD GSD schema for details on GSD data chunk names and how the data are stored.

Depletants

HPMC supports integration with depletants. An ideal gas of depletants is generated 'on-the-fly' and and used in the Metropolis acceptance criterion for the 'colloid' particles. Depletants are of arbitrary shape, however they are assumed to be 'hard' only with respect to the colloids, and mutually interpenetrable. The main idea is described in See J. Glaser et. al. 2015 .

As of version 2.2, hoomd.hpmc supports a new acceptance rule for depletants which is enabled with the **depletant_mode='overlap_regions'** argument. The new mode results in free diffusion of colloids that do not share any overlap volume with other colloids. This speeds up equilibration of dilute systems of colloids in a dense depletant bath. Both modes yield the same equilibrium statistics, but different dynamics (Glaser, to be published).

count_overlaps()

Count the number of overlaps.

Returns The number of overlaps in the current system configuration

Example:

```
mc = hpmc.integrate.shape(..);
mc.shape_param.set(....);
run(100)
num_overlaps = mc.count_overlaps();
```

get_a(*type=None*)

Get the maximum trial rotation.

Parameters **type** (*str*) – Type name to query.

Returns The current value of the 'a' parameter of the integrator.

get_configurational_bias_ratio()
Get the average ratio of configurational bias attempts to depletant insertion moves.

Only supported with **depletant_mode=='circumsphere'**.

Returns The average configurational bias ratio during the last *hoomd.run()*.

Example:

```
mc = hpmc.integrate.shape(..,implicit=True);
mc.shape_param.set(....);
run(100)
cb_ratio = mc.get_configurational_bias_ratio();
```

get_counters()
Get all trial move counters.

Returns A dictionary containing all trial moves counted during the last *hoomd.run()*.

The dictionary contains the entries:

- *translate_accept_count* - count of the number of accepted translate moves
- *translate_reject_count* - count of the number of rejected translate moves
- *rotate_accept_count* - count of the number of accepted rotate moves
- *rotate_reject_count* - count of the number of rejected rotate moves
- *overlap_checks* - estimate of the number of overlap checks performed
- *translate_acceptance* - Average translate acceptance ratio over the run
- *rotate_acceptance* - Average rotate acceptance ratio over the run
- *move_count* - Count of the number of trial moves during the run

get_d(*type=None*)
Get the maximum trial displacement.

Parameters **type** (*str*) – Type name to query.

Returns The current value of the 'd' parameter of the integrator.

get_depletant_type()
Get the depletant type

Returns The type of particle used as depletant (the 'depletant_type' argument of the integrator).

get_move_ratio()
Get the current probability of attempting translation moves.

Returns: The current value of the 'move_ratio' parameter of the integrator.

get_mps()
Get the number of trial moves per second.

Returns The number of trial moves per second performed during the last *hoomd.run()*.

get_nR()
Get depletant density

Returns The current value of the 'nR' parameter of the integrator.

get_nselect()
Get nselect parameter.

Returns The current value of the 'nselect' parameter of the integrator.

get_ntrial()
Get ntrial parameter.

Returns The current value of the 'ntrial' parameter of the integrator.

get_rotate_acceptance()
Get the average acceptance ratio for rotate moves.

Returns The average rotate accept ratio during the last *hoomd.run()*.

Example:

```
mc = hpmc.integrate.shape(..);
mc.shape_param.set(....);
run(100)
t_accept = mc.get_rotate_acceptance();
```

get_translate_acceptance()
Get the average acceptance ratio for translate moves.

Returns The average translate accept ratio during the last *hoomd.run()*.

Example:

```
mc = hpmc.integrate.shape(..);
mc.shape_param.set(....);
run(100)
t_accept = mc.get_translate_acceptance();
```

get_type_shapes()
Get all the types of shapes in the current simulation

Since this behaves differently for different types of shapes, the default behavior just raises an exception. Subclasses can override this to properly return

map_overlaps()
Build an overlap map of the system

Returns List of tuples. True/false value of the i,j entry indicates overlap/non-overlap of the ith and jth particles (by tag)

Note: *map_overlaps()* does not support MPI parallel simulations.

Example

mc = hpmc.integrate.shape(...) mc.shape_param.set(...) overlap_map = np.asarray(mc.map_overlaps())

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*d=None*, *a=None*, *move_ratio=None*, *nselect=None*, *nR=None*, *depletant_type=None*, *ntrial=None*, *deterministic=None*)
Changes parameters of an existing integration mode.

Parameters

- **d** (*float*) – (if set) Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.

- **a** (*float*) – (if set) Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – (if set) New value for the move ratio.
- **nselect** (*int*) – (if set) New value for the number of particles to select for trial moves in one cell.
- **nR** (*int*) – (if set) **Implicit depletants only**: Number density of implicit depletants in free volume.
- **depletant_type** (*str*) – (if set) **Implicit depletants only**: Particle type to use as implicit depletant.
- **ntrial** (*int*) – (if set) **Implicit depletants only**: Number of re-insertion attempts per overlapping depletant. (Only supported with **depletant_mode='circumsphere'**)
- **deterministic** (*bool*) – (if set) Make HPMC integration deterministic on the GPU by sorting the cell list.

Note: Simulations are only deterministic with respect to the same execution configuration (CPU or GPU) and number of MPI ranks. Simulation output will not be identical if either of these is changed.

---

setup_pos_writer(*pos*, *colors={}*)

Set pos_writer definitions for specified shape parameters.

Parameters

- **pos** (*hoomd.deprecated.dump.pos*) – pos writer to setup
- **colors** (*dict*) – dictionary of type name to color mappings

setup_pos_writer() uses the shape_param settings to specify the shape definitions (via set_def) to the provided pos file writer. This overrides any previous values specified to *hoomd.deprecated.dump.pos.set_def()*.

colors allows you to set per-type colors for particles. Specify colors as strings in the injavis format. When colors is not specified for a type, all colors default to `005984FF`.

Examples:

```
mc = hpmc.integrate.shape(...);
mc.shape_param.set(....);
pos = pos_writer.dumpy.pos("dump.pos", period=100);
mc.setup_pos_writer(pos, colors=dict(A='005984FF'));
```

test_overlap(*type_i*, *type_j*, *rij*, *qi*, *qj*, *use_images=True*, *exclude_self=False*)

Test overlap between two particles.

Parameters

- **type_i** (*str*) – Type of first particle
- **type_j** (*str*) – Type of second particle
- **rij** (*tuple*) – Separation vector **rj**-**ri** between the particle centers
- **qi** (*tuple*) – Orientation quaternion of first particle
- **qj** (*tuple*) – Orientation quaternion of second particle
- **use_images** (*bool*) – If True, check for overlap between the periodic images of the particles by adding the image vector to the separation vector

- **exclude_self** (*bool*) – If both **use_images** and **exclude_self** are true, exclude the primary image

For two-dimensional shapes, pass the third dimension of **rij** as zero.

Returns True if the particles overlap.

class hoomd.hpmc.integrate.**polyhedron**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*, *restore_state=False*)

HPMC integration for general polyhedra (3D).

This shape uses an internal OBB tree for fast collision queries. Depending on the number of constituent spheres in the tree, different values of the number of spheres per leaf node may yield different optimal performance. The capacity of leaf nodes is configurable.

Only triangle meshes and spheres are supported. The mesh must be free of self-intersections.

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

Polyhedron parameters:

- *vertices* (**required**) - vertices of the polyhedron as is a list of (x,y,z) tuples of numbers (distance units)
  - The origin **MUST** strictly be contained in the generally nonconvex volume defined by the vertices and faces
  - The (0,0,0) centered sphere that encloses all vertices should be of minimal size for optimal performance (e.g. don't translate the shape such that (0,0,0) right next to a face).
- *faces* (**required**) - a list of vertex indices for every face
- *sweep_radius* (**default: 0.0**) - rounding radius applied to polyhedron
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.
- *capacity* (**default: 4**) - set to the maximum number of particles per leaf node for better performance
  - New in version 2.2.
- *origin* (**default: (0,0,0)**) - a point strictly inside the shape, needed for correctness of overlap checks
  - New in version 2.2.

- *hull_only* (**default: True**) - if True, only consider intersections between hull polygons
  - New in version 2.2.

Warning: HPMC does not check that all requirements are met. Undefined behavior will result if they are violated.

Example:

```
mc = hpmc.integrate.polyhedron(seed=415236)
mc = hpmc.integrate.polyhedron(seed=415236, d=0.3, a=0.4)
mc.shape_param.set('A', vertices=[(-0.5, -0.5, -0.5), (-0.5, -0.5, 0.5), (-0.5, 0.
↪5, -0.5), (-0.5, 0.5, 0.5), \
    (0.5, -0.5, -0.5), (0.5, -0.5, 0.5), (0.5, 0.5, -0.5), (0.5, 0.5, 0.5)],\
    faces = [(7, 3, 1, 5), (7, 5, 4, 6), (7, 6, 2, 3), (3, 2, 0, 1), (0, 2, 6, 4),
↪ (1, 0, 4, 5)]);
print('vertices = ', mc.shape_param['A'].vertices)
print('faces = ', mc.shape_param['A'].faces)
```

Depletants Example:

```
mc = hpmc.integrate.polyhedron(seed=415236, d=0.3, a=0.4, implicit=True,␣
↪depletant_mode='circumsphere')
mc.set_param(nselect=1,nR=3,depletant_type='B')
faces = [(7, 3, 1, 5), (7, 5, 4, 6), (7, 6, 2, 3), (3, 2, 0, 1), (0, 2, 6, 4), (1,
↪ 0, 4, 5)];
mc.shape_param.set('A', vertices=[(-0.5, -0.5, -0.5), (-0.5, -0.5, 0.5), (-0.5, 0.
↪5, -0.5), (-0.5, 0.5, 0.5), \
    (0.5, -0.5, -0.5), (0.5, -0.5, 0.5), (0.5, 0.5, -0.5), (0.5, 0.5, 0.5)],␣
↪faces = faces);
mc.shape_param.set('B', vertices=[(-0.05, -0.05, -0.05), (-0.05, -0.05, 0.05), (-
↪0.05, 0.05, -0.05), (-0.05, 0.05, 0.05), \
    (0.05, -0.05, -0.05), (0.05, -0.05, 0.05), (0.05, 0.05, -0.05), (0.05, 0.05,␣
↪0.05)], faces = faces, origin = (0,0,0));
```

class hoomd.hpmc.integrate.**simple_polygon**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *restore_state=False*)

HPMC integration for simple polygons (2D).

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

Note: For simple polygons that are not concave, use *convex_polygon*, it will execute much faster than *simple_polygon*.

Simple polygon parameters:

- *vertices* (**required**) - vertices of the polygon as is a list of (x,y) tuples of numbers (distance units)
  - Vertices **MUST** be specified in a *counter-clockwise* order.
  - The polygon may be concave, but edges must not cross.
  - The origin doesn't necessarily need to be inside the shape.
  - The origin centered circle that encloses all vertices should be of minimal size for optimal performance.
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Warning: HPMC does not check that all requirements are met. Undefined behavior will result if they are violated.

Examples:

```
mc = hpmc.integrate.simple_polygon(seed=415236)
mc = hpmc.integrate.simple_polygon(seed=415236, d=0.3, a=0.4)
mc.shape_param.set('A', vertices=[(0, 0.5), (-0.5, -0.5), (0, 0), (0.5, -0.5)]);
print('vertices = ', mc.shape_param['A'].vertices)
```

get_type_shapes()
Get all the types of shapes in the current simulation :returns: A list of dictionaries, one for each particle type in the system. Currently assumes that all 3D shapes are convex.

class hoomd.hpmc.integrate.**sphere**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*, *restore_state=False*)

HPMC integration for spheres (2D/3D).

Parameters

- **seed** (*int*) – Random number seed
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (float, only with **orientable=True**) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type. (added in version 2.3)
- **move_ratio** (float, only used with **orientable=True**) – Ratio of translation moves to rotation moves. (added in version 2.3)
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

Hard particle Monte Carlo integration method for spheres.

Sphere parameters:

- *diameter* (**required**) - diameter of the sphere (distance units)
- *orientable* (**default: False**) - set to True for spheres with orientation (added in version 2.3)
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Examples:

```
mc = hpmc.integrate.sphere(seed=415236)
mc = hpmc.integrate.sphere(seed=415236, d=0.3)
mc.shape_param.set('A', diameter=1.0)
mc.shape_param.set('B', diameter=2.0)
mc.shape_param.set('C', diameter=1.0, orientable=True)
print('diameter = ', mc.shape_param['A'].diameter)
```

Depletants Example:

```
mc = hpmc.integrate.sphere(seed=415236, d=0.3, a=0.4, implicit=True, depletant_
↪mode='circumsphere')
mc.set_param(nselect=8,nR=3,depletant_type='B')
mc.shape_param.set('A', diameter=1.0)
mc.shape_param.set('B', diameter=.1)
```

get_type_shapes()
Get all the types of shapes in the current simulation :returns: A list of dictionaries, one for each particle type in the system. Currently assumes that all 3D shapes are convex.

class hoomd.hpmc.integrate.**sphere_union**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*, *max_members=None*, *restore_state=False*)

HPMC integration for unions of spheres (3D).

This shape uses an internal OBB tree for fast collision queries. Depending on the number of constituent spheres in the tree, different values of the number of spheres per leaf node may yield different optimal performance. The capacity of leaf nodes is configurable.

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.
- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **max_members** (*int*) – Set the maximum number of members in the sphere union * .. deprecated:: 2.2

- **capacity** (*int*) – Set to the number of constituent spheres per leaf node. (added in version 2.2)
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

Sphere union parameters:

- *diameters* (**required**) - list of diameters of the spheres (distance units).
- *centers* (**required**) - list of centers of constituent spheres in particle coordinates.
- *overlap* (**default: 1 for all spheres**) - only check overlap between constituent particles for which *overlap[i] & overlap[j]* is !=0, where '&' is the bitwise AND operator.
  - New in version 2.1.
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking.
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.
- ***capacity* (default: 4) - set to the maximum number of particles per leaf node for better performance**
  - New in version 2.2.

Example:

```
mc = hpmc.integrate.sphere_union(seed=415236)
mc = hpmc.integrate.sphere_union(seed=415236, d=0.3, a=0.4)
mc.shape_param.set('A', diameters=[1.0, 1.0], centers=[(-0.25, 0.0, 0.0), (0.25,␣
↪0.0, 0.0)]);
print('diameter of the first sphere = ', mc.shape_param['A'].members[0].diameter)
print('center of the first sphere = ', mc.shape_param['A'].centers[0])
```

Depletants Example:

```
mc = hpmc.integrate.sphere_union(seed=415236, d=0.3, a=0.4, implicit=True,␣
↪depletant_mode='circumsphere')
mc.set_param(nselect=1,nR=50,depletant_type='B')
mc.shape_param.set('A', diameters=[1.0, 1.0], centers=[(-0.25, 0.0, 0.0), (0.25,␣
↪0.0, 0.0)]);
mc.shape_param.set('B', diameters=[0.05], centers=[(0.0, 0.0, 0.0)]);
```

class hoomd.hpmc.integrate.**sphinx**(*seed*, *d=0.1*, *a=0.1*, *move_ratio=0.5*, *nselect=4*, *implicit=False*, *depletant_mode='circumsphere'*, *restore_state=False*)

HPMC integration for sphinx particles (3D).

Parameters

- **seed** (*int*) – Random number seed.
- **d** (*float*) – Maximum move displacement, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **a** (*float*) – Maximum rotation move, Scalar to set for all types, or a dict containing {type:size} to set by type.
- **move_ratio** (*float*) – Ratio of translation moves to rotation moves.
- **nselect** (*int*) – The number of trial moves to perform in each cell.

- **implicit** (*bool*) – Flag to enable implicit depletants.
- **depletant_mode** (string, only with **implicit=True**) – Where to place random depletants, either 'circumsphere' or 'overlap_regions' (added in version 2.2)
- **restore_state** (*bool*) – Restore internal state from initialization file when True. See *mode_hpmc* for a description of what state data restored. (added in version 2.2)

Sphinx particles are dimpled spheres (spheres with 'positive' and 'negative' volumes).

Sphinx parameters:

- *diameters* - diameters of spheres (positive OR negative real numbers)
- *centers* - centers of spheres in local coordinate frame
- *ignore_statistics* (**default: False**) - set to True to disable ignore for statistics tracking
- *ignore_overlaps* (**default: False**) - set to True to disable overlap checks between this and other types with *ignore_overlaps=True*
  - Deprecated since version 2.1: Replaced by *interaction_matrix*.

Quick Example:

```
mc = hpmc.integrate.sphinx(seed=415236)
mc = hpmc.integrate.sphinx(seed=415236, d=0.3, a=0.4)
mc.shape_param.set('A', centers=[(0,0,0),(1,0,0)], diameters=[1,.25])
print('diameters = ', mc.shape_param['A'].diameters)
```

Depletants Example:

```
mc = hpmc.integrate.sphinx(seed=415236, d=0.3, a=0.4, implicit=True, depletant_
↪mode='circumsphere')
mc.set_param(nselect=1,nR=3,depletant_type='B')
mc.shape_param.set('A', centers=[(0,0,0),(1,0,0)], diameters=[1,-.25])
mc.shape_param.set('B', centers=[(0,0,0)], diameters=[.15])
```

13.6 hpmc.update

Overview

hpmc.update.boxmc	Apply box updates to sample isobaric and related ensembles.
hpmc.update.muvt	Insert and remove particles in the muVT ensemble.
hpmc.update.remove_drift	Remove the center of mass drift from a system restrained on a lattice.
hpmc.update.wall	Apply wall updates with a user-provided python callback.

Details

HPMC updaters.

class hoomd.hpmc.update.**boxmc**(*mc*, *betaP*, *seed*)

Apply box updates to sample isobaric and related ensembles.

Parameters

- **mc** (*hoomd.hpmc.integrate*) – HPMC integrator object for system on which to apply box updates
- **betaP** (*float* or *hoomd.variant*) – $\frac{p}{k_B T}$. (units of inverse area in 2D or inverse volume in 3D) Apply your chosen reduced pressure convention externally.
- **seed** (*int*) – random number seed for MC box changes

One or more Monte Carlo move types are applied to evolve the simulation box. By default, no moves are applied. Activate desired move types using the following methods with a non-zero weight:

- *aspect()* - box aspect ratio moves
- *length()* - change box lengths independently
- *shear()* - shear the box
- *volume()* - scale the box lengths uniformly
- *ln_volume()* - scale the box lengths uniformly with logarithmic increments

Pressure inputs to update.boxmc are defined as βP. Conversions from a specific definition of reduced pressure P^* are left for the user to perform.

Note: All *delta* and *weight* values for all move types default to 0.

Example:

```
mc = hpmc.integrate.sphere(seed=415236, d=0.3)
boxMC = hpmc.update.boxmc(mc, betaP=1.0, seed=9876)
boxMC.set_betap(2.0)
boxMC.ln_volume(delta=0.01, weight=2.0)
boxMC.length(delta=(0.1,0.1,0.1), weight=4.0)
run(30) # perform approximately 10 volume moves and 20 length moves
```

aspect(*delta=None*, *weight=None*)

Enable/disable aspect ratio move and set parameters.

Parameters

- **delta** (*float*) – maximum relative change of aspect ratio
- **weight** (*float*) – relative weight of this box move type relative to other box move types. 0 disables this move type.

Rescale aspect ratio along a randomly chosen dimension.

Note: When an argument is None, the value is left unchanged from its current state.

Example:

```
box_update.aspect(delta=0.01)
box_update.aspect(delta=0.01, weight=2)
box_update.aspect(delta=0.01, weight=0.15)
```

Returns A dict with the current values of *delta*, and *weight*.

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Example:

```
box_updater.set_params(isotropic=True)
run(1e5)
box_updater.disable()
update.box_resize(dLy = 10)
box_updater.enable()
run(1e5)
```

See updater base class documentation for more information

get_aspect_acceptance()
Get the average acceptance ratio for aspect changing moves.

Returns The average aspect change acceptance for the last run

Example:

```
mc = hpmc.integrate.shape(..);
mc_shape_param[name].set(....);
box_update = hpmc.update.boxmc(mc, betaP=10, seed=1)
run(100)
a_accept = box_update.get_aspect_acceptance()
```

get_ln_volume_acceptance()
Get the average acceptance ratio for log(V) changing moves.

Returns The average volume change acceptance for the last run

Example:

```
mc = hpmc.integrate.shape(..);
mc.shape_param[name].set(....);
box_update = hpmc.update.boxmc(mc, betaP=10, seed=1)
run(100)
v_accept = box_update.get_ln_volume_acceptance()
```

get_shear_acceptance()
Get the average acceptance ratio for shear changing moves.

Returns The average shear change acceptance for the last run

Example:

```
mc = hpmc.integrate.shape(..);
mc.shape_param[name].set(....);
box_update = hpmc.update.boxmc(mc, betaP=10, seed=1)
```

(continues on next page)

(continued from previous page)

```
run(100)
s_accept = box_update.get_shear_acceptance()
```

get_volume_acceptance()
Get the average acceptance ratio for volume changing moves.

Returns The average volume change acceptance for the last run

Example:

```
mc = hpmc.integrate.shape(..);
mc.shape_param[name].set(....);
box_update = hpmc.update.boxmc(mc, betaP=10, seed=1)
run(100)
v_accept = box_update.get_volume_acceptance()
```

length(*delta=None*, *weight=None*)
Enable/disable isobaric box dimension move and set parameters.

Parameters

- **delta** (`float` or `tuple`) – maximum change of the box thickness for each pair of parallel planes connected by the corresponding box edges. I.e. maximum change of HOOMD-blue box parameters Lx, Ly, Lz. A single float *x* is equivalent to (*x*, *x*, *x*).
- **weight** (*float*) – relative weight of this box move type relative to other box move types. 0 disables this move type.

Sample the isobaric distribution of box dimensions by rescaling the plane-to-plane distance of box faces, Lx, Ly, Lz (see *Periodic boundary conditions*).

Note: When an argument is None, the value is left unchanged from its current state.

Example:

```
box_update.length(delta=(0.01, 0.01, 0.0)) # 2D box changes
box_update.length(delta=(0.01, 0.01, 0.01), weight=2)
box_update.length(delta=0.01, weight=2)
box_update.length(delta=(0.10, 0.01, 0.01), weight=0.15) # sample Lx more␣
↪aggressively
```

Returns A `dict` with the current values of *delta* and *weight*.

ln_volume(*delta=None*, *weight=None*)
Enable/disable isobaric volume move and set parameters.

Parameters

- **delta** (*float*) – maximum change of **ln(V)** (where V is box area (2D) or volume (3D)).
- **weight** (*float*) – relative weight of this box move type relative to other box move types. 0 disables this move type.

Sample the isobaric distribution of box volumes by rescaling the box.

Note: When an argument is None, the value is left unchanged from its current state.

Example:

```
box_update.ln_volume(delta=0.001)
box_update.ln_volume(delta=0.001, weight=2)
box_update.ln_volume(delta=0.001, weight=0.15)
```

Returns A `dict` with the current values of *delta* and *weight*.

restore_state()
Restore the state information from the file used to initialize the simulations

set_betap(*betaP*)
Update the pressure set point for Metropolis Monte Carlo volume updates.

Parameters **betaP** (float) or (*hoomd.variant*) – $\frac{p}{k_B T}$. (units of inverse area in 2D or inverse volume in 3D) Apply your chosen reduced pressure convention externally.

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

shear(*delta=None*, *weight=None*, *reduce=None*)
Enable/disable box shear moves and set parameters.

Parameters

- **delta** (*tuple*) – maximum change of the box tilt factor xy, xz, yz.
- **reduce** (*float*) – Maximum number of lattice vectors of shear to allow before applying lattice reduction. Shear of +/- 0.5 cannot be lattice reduced, so set to a value < 0.5 to disable (default 0) Note that due to precision errors, lattice reduction may introduce small overlaps which can be resolved, but which temporarily break detailed balance.
- **weight** (*float*) – relative weight of this box move type relative to other box move types. 0 disables this move type.

Sample the distribution of box shear by adjusting the HOOMD-blue tilt factor parameters xy, xz, and yz. (see *Periodic boundary conditions*).

Note: When an argument is None, the value is left unchanged from its current state.

Example:

```
box_update.shear(delta=(0.01, 0.00, 0.0)) # 2D box changes
box_update.shear(delta=(0.01, 0.01, 0.01), weight=2)
box_update.shear(delta=(0.10, 0.01, 0.01), weight=0.15) # sample xy more␣
↪aggressively
```

Returns A `dict` with the current values of *delta*, *weight*, and *reduce*.

volume(*delta=None*, *weight=None*)

Enable/disable isobaric volume move and set parameters.

Parameters

- **delta** (*float*) – maximum change of the box area (2D) or volume (3D).
- **weight** (*float*) – relative weight of this box move type relative to other box move types. 0 disables this move type.

Sample the isobaric distribution of box volumes by rescaling the box.

Note: When an argument is None, the value is left unchanged from its current state.

Example:

```
box_update.volume(delta=0.01)
box_update.volume(delta=0.01, weight=2)
box_update.volume(delta=0.01, weight=0.15)
```

Returns A dict with the current values of *delta* and *weight*.

class hoomd.hpmc.update.**clusters**(*mc*, *seed*, *period=1*)

Equilibrate the system according to the geometric cluster algorithm (GCA).

The GCA as described in Liu and Lujten (2004), http://doi.org/10.1103/PhysRevLett.92.035504 is used for hard shape, patch interactions and depletants.

With depletants, Clusters are defined by a simple distance cut-off criterion. Two particles belong to the same cluster if the circumspheres of the depletant-excluded volumes overlap.

Supported moves include pivot moves (point reflection), line reflections (pi rotation around an axis), and type swaps. Only the pivot move is rejection free. With anisotropic particles, the pivot move cannot be used because it would create a chiral mirror image of the particle, and only line reflections are employed. Line reflections are not rejection free because of periodic boundary conditions, as discussed in Sinkovits et al. (2012), http://doi.org/10.1063/1.3694271 .

The type swap move works between two types of spherical particles and exchanges their identities.

The *clusters* updater support TBB execution on multiple CPU cores. See *Compiling HOOMD-blue* for more information on how to compile HOOMD with TBB support.

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator.
- **seed** (*int*) – The seed of the pseudo-random number generator (Needs to be the same across partitions of the same Gibbs ensemble)
- **period** (*int*) – Number of timesteps between histogram evaluations.

Example:

```
mc = hpmc.integrate.sphere(seed=415236)
hpmc.update.clusters(mc=mc, seed=123)
```

disable()

Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

get_pivot_acceptance()
Get the average acceptance ratio for pivot moves

Returns The average acceptance rate for pivot moves during the last run

get_reflection_acceptance()
Get the average acceptance ratio for reflection moves

Returns The average acceptance rate for reflection moves during the last run

get_swap_acceptance()
Get the average acceptance ratio for swap moves

Returns The average acceptance rate for type swap moves during the last run

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*move_ratio=None*, *flip_probability=None*, *swap_move_ratio=None*, *delta_mu=None*, *swap_types=None*)
Set options for the clusters moves.

Parameters

- **move_ratio** (*float*) – Set the ratio between pivot and reflection moves (default 0.5)
- **flip_probability** (*float*) – Set the probability for transforming an individual cluster (default 0.5)
- **swap_move_ratio** (*float*) – Set the ratio between type swap moves and geometric moves (default 0.5)
- **delta_mu** (*float*) – The chemical potential difference between types to be swapped
- **swap_types** (*list*) – A pair of two types whose identities are swapped

Note: When an argument is None, the value is left unchanged from its current state.

Example:

```
clusters = hpmc.update.clusters(mc, seed=123)
clusters.set_params(move_ratio = 1.0)
clusters.set_params(swap_types=['A','B'], delta_mu = -0.001)
```

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.hpmc.update.**muvt**(*mc*, *seed*, *period=1*, *transfer_types=None*, *ngibbs=1*)
Insert and remove particles in the muVT ensemble.

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator.
- **seed** (*int*) – The seed of the pseudo-random number generator (Needs to be the same across partitions of the same Gibbs ensemble)
- **period** (*int*) – Number of timesteps between histogram evaluations.
- **transfer_types** (*list*) – List of type names that are being transferred from/to the reservoir or between boxes (if *None*, all types)
- **ngibbs** (*int*) – The number of partitions to use in Gibbs ensemble simulations (if == 1, perform grand canonical muVT)

The muVT (or grand-canonical) ensemble simulates a system at constant fugacity.

Gibbs ensemble simulations are also supported, where particles and volume are swapped between two or more boxes. Every box correspond to one MPI partition, and can therefore run on multiple ranks. See *hoomd.comm* and the –nrank command line option for how to split a MPI task into partitions.

Note: Multiple Gibbs ensembles are also supported in a single parallel job, with the ngibbs option to update.muvt(), where the number of partitions can be a multiple of ngibbs.

Example:

```
mc = hpmc.integrate.sphere(seed=415236)
update.muvt(mc=mc, period)
```

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()
Restore the state information from the file used to initialize the simulations

set_fugacity(*type*, *fugacity*)
Change muVT fugacities.

Parameters

- **type** (*str*) – Particle type to set parameters for
- **fugacity** (*float*) – Fugacity of this particle type (dimension of volume^-1)

Example

muvt = hpmc.update.muvt(mc, period = 10) muvt.set_fugacity(type='A',fugacity=1.23) variant = hoomd.variant.linear_interp(points= [(0,1e1), (1e5, 4.56)]) muvt.set_fugacity(type='A', fugacity=variant)

set_params(*dV=None*, *move_ratio=None*, *transfer_ratio=None*)
Set muVT parameters.

Parameters

- **dV** (*float*) – (if set) Set volume rescaling factor (dimensionless)
- **move_ratio** (*float*) – (if set) Set the ratio between volume and exchange/transfer moves (applies to Gibbs ensemble)
- **transfer_ratio** (*float*) – (if set) Set the ratio between transfer and exchange moves

Example:

```
muvt = hpmc.update.muvt(mc, period = 10)
muvt.set_params(dV=0.1)
muvt.set_params(n_trial=2)
muvt.set_params(move_ratio=0.05)
```

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.hpmc.update.**remove_drift**(*mc*, *external_lattice*, *period=1*)
Remove the center of mass drift from a system restrained on a lattice.

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator.

- **external_lattice** (*hoomd.hpmc.field.lattice_field*) – lattice field where the lattice is defined.
- **period** (*int*) – the period to call the updater

The command hpmc.update.remove_drift sets up an updater that removes the center of mass drift of a system every period timesteps,

Example:

```
mc = hpmc.integrate.convex_polyhedron(seed=seed);
mc.shape_param.set("A", vertices=verts)
mc.set_params(d=0.005, a=0.005)
lattice = hpmc.compute.lattice_field(mc=mc, position=fcc_lattice, k=1000.0);
remove_drift = update.remove_drift(mc=mc, external_lattice=lattice, period=1000);
```

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.hpmc.update.**wall**(*mc*, *walls*, *py_updater*, *move_ratio*, *seed*, *period=1*)
Apply wall updates with a user-provided python callback.

Parameters

- **mc** (*hoomd.hpmc.integrate*) – MC integrator.
- **walls** (*hoomd.hpmc.field.wall*) – the wall class instance to be updated

- **py_updater** (*callable*) – the python callback that performs the update moves. This must be a python method that is a function of the timestep of the simulation. It must actually update the *hoomd.hpmc.field.wall*) managed object.
- **move_ratio** (*float*) – the probability with which an update move is attempted
- **seed** (*int*) – the seed of the pseudo-random number generator that determines whether or not an update move is attempted
- **period** (*int*) – the number of timesteps between update move attempt attempts Every *period* steps, a walls update move is tried with probability *move_ratio*. This update move is provided by the *py_updater* callback. Then, update.wall only accepts an update move provided by the python callback if it maintains confinement conditions associated with all walls. Otherwise, it reverts back to a non-updated copy of the walls.

Once initialized, the update provides the following log quantities that can be logged via *hoomd.analyze.log*:

- **hpmc_wall_acceptance_ratio** - the acceptance ratio for wall update moves

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
def perturb(timestep):
  r = np.sqrt(ext_wall.get_sphere_wall_param(index = 0, param = "rsq"));
  ext_wall.set_sphere_wall(index = 0, radius = 1.5*r, origin = [0, 0, 0], inside␣
↪= True);
wall_updater = hpmc.update.wall(mc, ext_wall, perturb, move_ratio = 0.5, seed =␣
↪27, period = 50);
log = analyze.log(quantities=['hpmc_wall_acceptance_ratio'], period=100, filename=
↪'log.dat', overwrite=True);
```

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
def perturb(timestep):
  r = np.sqrt(ext_wall.get_sphere_wall_param(index = 0, param = "rsq"));
  ext_wall.set_sphere_wall(index = 0, radius = 1.5*r, origin = [0, 0, 0], inside␣
↪= True);
wall_updater = hpmc.update.wall(mc, ext_wall, perturb, move_ratio = 0.5, seed =␣
↪27, period = 50);
```

disable()

Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()

Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

get_accepted_count(*mode=0*)
Get the number of accepted wall update moves.

Parameters **mode** (*int*) – specify the type of count to return. If mode!=0, return absolute quantities. If mode=0, return quantities relative to the start of the run. DEFAULTS to 0.

Returns the number of accepted wall update moves

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
def perturb(timestep):
  r = np.sqrt(ext_wall.get_sphere_wall_param(index = 0, param = "rsq"));
  ext_wall.set_sphere_wall(index = 0, radius = 1.5*r, origin = [0, 0, 0],␣
↪inside = True);
wall_updater = hpmc.update.wall(mc, ext_wall, perturb, move_ratio = 0.5, seed␣
↪= 27, period = 50);
run(100);
acc_count = wall_updater.get_accepted_count(mode = 0);
```

get_total_count(*mode=0*)
Get the number of attempted wall update moves.

Parameters **mode** (*int*) – specify the type of count to return. If mode!=0, return absolute quantities. If mode=0, return quantities relative to the start of the run. DEFAULTS to 0.

Returns the number of attempted wall update moves

Example:

```
mc = hpmc.integrate.sphere(seed = 415236);
ext_wall = hpmc.compute.wall(mc);
ext_wall.add_sphere_wall(radius = 1.0, origin = [0, 0, 0], inside = True);
def perturb(timestep):
  r = np.sqrt(ext_wall.get_sphere_wall_param(index = 0, param = "rsq"));
  ext_wall.set_sphere_wall(index = 0, radius = 1.5*r, origin = [0, 0, 0],␣
↪inside = True);
wall_updater = hpmc.update.wall(mc, ext_wall, perturb, move_ratio = 0.5, seed␣
↪= 27, period = 50);
run(100);
tot_count = wall_updater.get_total_count(mode = 0);
```

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

13.7 hpmc.util

Overview

hpmc.util.tune	Tune mc parameters.
hpmc.util.tune_npt	Tune the HPMC *hoomd.hpmc.update.boxmc* using *tune*.

Details

HPMC utilities

class hoomd.hpmc.util.**tune**(*obj=None*, *tunables=[]*, *max_val=[]*, *target=0.2*, *max_scale=2.0*, *gamma=2.0*, *type=None*, *tunable_map=None*, **args*, ***kwargs*)

Tune mc parameters.

hoomd.hpmc.util.tune provides a general tool to observe Monte Carlo move acceptance rates and adjust the move sizes when called by a user script. By default, it understands how to read and adjust the trial move domain for translation moves and rotation moves for an hpmc.integrate instance. Other move types for integrators or updaters can be handled with a customized tunable map passed when creating the tuner or in a subclass definition. E.g. see use an implementation of *tune_npt*

Parameters

- **obj** – HPMC Integrator or Updater instance
- **tunables** (*list*) – list of strings naming parameters to tune. By default, allowed element values are 'd' and 'a'.
- **max_val** (*list*) – maximum allowed values for corresponding tunables
- **target** (*float*) – desired acceptance rate
- **max_scale** (*float*) – maximum amount to scale a parameter in a single update
- **gamma** (*float*) – damping factor (>= 0.0) to keep from scaling parameter values too fast
- **type** (*str*) – Name of a single hoomd particle type for which to tune move sizes. If None (default), all types are tuned with the same statistics.
- **tunable_map** (*dict*) – For each tunable, provide a dictionary of values and methods to be used by the tuner (see below)
- **args** – Additional positional arguments
- **kwargs** – Additional keyword arguments

Example:

```
mc = hpmc.integrate.convex_polyhedron()
mc.set_params(d=0.01, a=0.01, move_ratio=0.5)
tuner = hpmc.util.tune(mc, tunables=['d', 'a'], target=0.2, gamma=0.5)
for i in range(10):
    run(1e4)
    tuner.update()
```

Note: You should run enough steps to get good statistics for the acceptance ratios. 10,000 trial moves seems like a good number. E.g. for 10,000 or more particles, tuning after a single timestep should be fine. For npt moves made once per timestep, tuning as frequently as 1,000 timesteps could get a rough convergence of acceptance ratios, which is probably good enough since we don't really know the optimal acceptance ratio, anyway.

Warning: There are some sanity checks that are not performed. For example, you shouldn't try to scale 'd' in a single particle simulation.

Details:

If `gamma == 0`, each call to *update()* rescales the current value of the tunable(s) by the ratio of the observed acceptance rate to the target value. For `gamma > 0`, the scale factor is the reciprocal of a weighted mean of the above ratio with 1, according to

scale = (1.0 + gamma) / (target/acceptance + gamma)

The names in `tunables` must match one of the keys in `tunable_map`, which in turn correspond to the keyword parameters of the MC object being updated.

`tunable_map` is a `dict` of `dict`. The keys of the outer `dict` are strings that can be specified in the `tunables` parameter. The value of this outer `dict` is another `dict` with the following four keys: 'get', 'acceptance', 'set', and 'maximum'.

A default `tunable_map` is provided but can be modified or extended by setting the following dictionary key/value pairs in the entry for tunable.

- get (`callable`): function called by tuner (no arguments) to retrieve current tunable value
- acceptance (`callable`): function called by tuner (no arguments) to get relevant acceptance rate
- set (`callable`): function to call to set new value (optional). Must take one argument (the new value). If not provided, `obj.set_params(tunable=x)` will be called to set the new value.
- maximum (`float`): maximum value the tuner may set for the tunable parameter

The default `tunable_map` defines the `callable` for 'set' to call *hoomd.hpmc.integrate.mode_hpmc.set_params()* with `tunable={type: newval}` instead of `tunable=newval` if the `type` argument is given when creating the `tune` object.

update()
Calculate and set tunable parameters using statistics from the run just completed.

class hoomd.hpmc.util.**tune_npt**(*obj=None*, *tunables=[]*, *max_val=[]*, *target=0.2*, *max_scale=2.0*, *gamma=2.0*, *type=None*, *tunable_map=None*, **args*, ***kwargs*)
Tune the HPMC *hoomd.hpmc.update.boxmc* using *tune*.

This is a thin wrapper to `tune` that simply defines an alternative `tunable_map` dictionary. In this case, the `obj` argument must be an instance of *hoomd.hpmc.update.boxmc*. Several tunables are defined.

'dLx', 'dLy', and 'dLz' use the acceptance rate of volume moves to set `delta[0]`, `delta[1]`, and `delta[2]`, respectively in a call to *hoomd.hpmc.update.boxmc.length()*.

'dV' uses the volume acceptance to call *hoomd.hpmc.update.boxmc.volume()*.

'dlnV' uses the ln_volume acceptance to call *hoomd.hpmc.update.boxmc.ln_volume()*.

'dxy', 'dxz', and 'dyz' tunables use the shear acceptance to set `delta[0]`, `delta[1]`, and `delta[2]`, respectively in a call to *hoomd.hpmc.update.boxmc.shear()*.

Refer to the documentation for *hoomd.hpmc.update.boxmc* for information on how these parameters are used, since they are not all applicable for a given use of `boxmc`.

Note: A bigger damping factor gamma may be appropriate for tuning box volume changes because there may be multiple parameters affecting each acceptance rate.

Example:

```
mc = hpmc.integrate.convex_polyhedron()
mc.set_params(d=0.01, a=0.01, move_ratio=0.5)
updater = hpmc.update.boxmc(mc, betaP=10)
updater.length(0.1, weight=1)
tuner = hpmc.util.tune_npt(updater, tunables=['dLx', 'dLy', 'dLz'], target=0.3,␣
↪gamma=1.0)
for i in range(10):
    run(1e4)
    tuner.update()
```

update()
Calculate and set tunable parameters using statistics from the run just completed.

CHAPTER 14

md

Details

Molecular Dynamics

Perform Molecular Dynamics simulations with HOOMD-blue.

Stability

hoomd.md is **stable**. When upgrading from version 2.x to 2.y (y > x), existing job scripts that follow *documented* interfaces for functions and classes will not require any modifications. **Maintainer:** Joshua A. Anderson

Modules

14.1 md.angle

Overview

md.angle.harmonic	Harmonic angle potential.
md.angle.cosinesq	Cosine squared angle potential.
md.angle.table	Tabulated angle potential.

Details

Angle potentials.

Angles add forces between specified triplets of particles and are typically used to model chemical angles between two bonds.

By themselves, angles that have been specified in an initial configuration do nothing. Only when you specify an angle force (i.e. angle.harmonic), are forces actually calculated between the listed particles.

class hoomd.md.angle.**coeff**

Define angle coefficients.

The coefficients for all angle force are specified using this class. Coefficients are specified per angle type.

There are two ways to set the coefficients for a particular angle potential. The first way is to save the angle potential in a variable and call *set()* directly. See below for an example of this.

The second method is to build the coeff class first and then assign it to the angle potential. There are some advantages to this method in that you could specify a complicated set of angle potential coefficients in a separate python file and import it into your job script.

Example:

```
my_coeffs = hoomd.md.angle.coeff();
my_angle_force.angle_coeff.set('polymer', k=330.0, r=0.84)
my_angle_force.angle_coeff.set('backbone', k=330.0, r=0.84)
```

set(*type*, ***coeffs*)

Sets parameters for angle types.

Parameters

- **type** (*str*) – Type of angle (or a list of type names)
- **coeffs** – Named coefficients (see below for examples)

Calling *set()* results in one or more parameters being set for a angle type. Types are identified by name, and parameters are also added by name. Which parameters you need to specify depends on the angle potential you are setting these coefficients for, see the corresponding documentation.

All possible angle types as defined in the simulation box must be specified before executing run(). You will receive an error if you fail to do so. It is not an error, however, to specify coefficients for angle types that do not exist in the simulation. This can be useful in defining a potential field for many different types of angles even when some simulations only include a subset.

Examples:

```
my_angle_force.angle_coeff.set('polymer', k=330.0, r0=0.84)
my_angle_force.angle_coeff.set('backbone', k=1000.0, r0=1.0)
my_angle_force.angle_coeff.set(['angleA','angleB'], k=100, r0=0.0)
```

Note: Single parameters can be updated. If both `k` and `r0` have already been set for a particle type, then executing `coeff.set('polymer', r0=1.0)` will update the value of `r0` and leave the other parameters as they were previously set.

class hoomd.md.angle.**cosinesq**

Cosine squared angle potential.

The command angle.cosinesq specifies a cosine squared potential energy between every triplet of particles with an angle specified between them.

$$V(\theta) = \frac{1}{2}k\left(\cos\theta - \cos\theta_0\right)^2$$

where θ is the angle between the triplet of particles. This angle style is also known as g96, since they were used in the gromos96 force field. These are also the types of angles used with the coarse-grained MARTINI force field.

Coefficients:

- θ_0 - rest angle `t0` (in radians)
- k - potential constant `k` (in units of energy)

Coefficients k and θ_0 must be set for each type of angle in the simulation using the method *angle_coeff.set()*. Note that the value of k for this angle potential is not comparable to the value of k for harmonic angles, as they have different units.

Examples:

```
cosinesq = angle.cosinesq()
cosinesq.angle_coeff.set('polymer', k=3.0, t0=0.7851)
cosinesq.angle_coeff.set('backbone', k=100.0, t0=1.0)
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.angle.**harmonic**

Harmonic angle potential.

The command angle.harmonic specifies a harmonic potential energy between every triplet of particles with an angle specified between them.

$$V(\theta) = \frac{1}{2}k\left(\theta - \theta_0\right)^2$$

where θ is the angle between the triplet of particles.

Coefficients:

- θ_0 - rest angle `t0` (in radians)
- k - potential constant `k` (in units of energy/radians^2)

Coefficients k and θ_0 must be set for each type of angle in the simulation using the method *angle_coeff.set()*.

Examples:

```
harmonic = angle.harmonic()
harmonic.angle_coeff.set('polymer', k=3.0, t0=0.7851)
harmonic.angle_coeff.set('backbone', k=100.0, t0=1.0)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.angle.**table**(*width*, *name=None*)

Tabulated angle potential.

Parameters

- **width** (*int*) – Number of points to use to interpolate V and F (see documentation above)
- **name** (*str*) – Name of the force instance

table specifies that a tabulated angle potential should be added to every bonded triple of particles in the simulation.

The torque T is (in units of force * distance) and the potential $V(\theta)$ is (in energy units):

$$T(\theta) = T_{\mathrm{user}}(\theta)$$
$$V(\theta) = V_{\mathrm{user}}(\theta)$$

where θ is the angle from A-B to B-C in the triple.

$T_{\mathrm{user}}(\theta)$ and $V_{\mathrm{user}}(\theta)$ are evaluated on *width* grid points between 0 and π. Values are interpolated linearly between grid points. For correctness, you must specify: $T = -\frac{\partial V}{\partial \theta}$

Parameters:

- $T_{\mathrm{user}}(\theta)$ and $V_{\mathrm{user}}(\theta)$ - evaluated by *func* (see example)
- coefficients passed to *func* - *coeff* (see example)

The table *width* is set once when *table* is specified. There are two ways to specify the other parameters.

Set table from a given function

When you have a functional form for T and F, you can enter that directly into python. *table* will evaluate the given function over *width* points between 0 and π and use the resulting values in the table:

```
def harmonic(theta, kappa, theta_0):
    V = 0.5 * kappa * (theta-theta_0)**2;
    T = -kappa*(theta-theta_0);
    return (V, T)

btable = angle.table(width=1000)
btable.angle_coeff.set('angle1', func=harmonic, coeff=dict(kappa=330, theta_0=0))
btable.angle_coeff.set('angle2', func=harmonic,coeff=dict(kappa=30, theta_0=0.1))
```

Set a table from a file

When you have no function for for T or F, or you otherwise have the data listed in a file, *table* can use the given values directly. You must first specify the number of rows in your tables when initializing *table*. Then use *set_from_file()* to read the file:

```
btable = angle.table(width=1000)
btable.set_from_file('polymer', 'angle.dat')
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_from_file(*anglename*, *filename*)
Set a angle pair interaction from a file.

Parameters

- **anglename** (*str*) – Name of angle
- **filename** (*str*) – Name of the file to read

The provided file specifies V and F at equally spaced theta values:

```
#t  V    T
0.0 2.0 -3.0
1.5707 3.0 -4.0
3.1414 2.0 -3.0
```

Warning: The theta values are not used by the code. It is assumed that a table that has N rows will start at 0, end at π and that $\delta\theta = \pi/(N-1)$. The table is read directly into the grid points used to evaluate $T_{\mathrm{user}}(\theta)$ and $V_{\mathrm{user}}(\theta)$.

14.2 md.bond

Overview

md.bond.fene	FENE bond potential.
md.bond.harmonic	Harmonic bond potential.
md.bond.table	Tabulated bond potential.

Details

Bond potentials.

Bonds add forces between specified pairs of particles and are typically used to model chemical bonds between two particles.

By themselves, bonds that have been specified in an initial configuration do nothing. Only when you specify an bond force (i.e. bond.harmonic), are forces actually calculated between the listed particles.

class hoomd.md.bond.**coeff**
Define bond coefficients.

The coefficients for all bond potentials are specified using this class. Coefficients are specified per bond type.

There are two ways to set the coefficients for a particular bond potential. The first way is to save the bond potential in a variable and call *set()* directly. See below for an example of this.

The second method is to build the coeff class first and then assign it to the bond potential. There are some advantages to this method in that you could specify a complicated set of bond potential coefficients in a separate python file and import it into your job script.

Example:

```
my_coeffs = hoomd.md.bond.coeff();
my_bond_force.bond_coeff.set('polymer', k=330.0, r=0.84)
my_bond_force.bond_coeff.set('backbone', k=330.0, r=0.84)
```

set (*type*, ***coeffs*)

Sets parameters for bond types.

Parameters

- **type** (*str*) – Type of bond (or a list of type names)
- **coeffs** – Named coefficients (see below for examples)

Calling *set()* results in one or more parameters being set for a bond type. Types are identified by name, and parameters are also added by name. Which parameters you need to specify depends on the bond potential you are setting these coefficients for, see the corresponding documentation.

All possible bond types as defined in the simulation box must be specified before executing run(). You will receive an error if you fail to do so. It is not an error, however, to specify coefficients for bond types that do not exist in the simulation. This can be useful in defining a potential field for many different types of bonds even when some simulations only include a subset.

Examples:

```
my_bond_force.bond_coeff.set('polymer', k=330.0, r0=0.84)
my_bond_force.bond_coeff.set('backbone', k=1000.0, r0=1.0)
my_bond_force.bond_coeff.set(['bondA','bondB'], k=100, r0=0.0)
```

Note: Single parameters can be updated. If both `k` and `r0` have already been set for a particle type, then executing `coeff.set('polymer', r0=1.0)` will update the value of `r0` and leave the other parameters as they were previously set.

class hoomd.md.bond.**fene** (*name=None*)

FENE bond potential.

Parameters **name** (*str*) – Name of the bond instance.

fene specifies a FENE potential energy between the two particles in each defined bond.

$$V(r) = -\frac{1}{2}kr_0^2 \ln\left(1 - \left(\frac{r-\Delta}{r_0}\right)^2\right) + V_{\mathrm{WCA}}(r)$$

where $\vec{r}$ is the vector pointing from one particle to the other in the bond, $\Delta = (d_i + d_j)/2 - 1$, d_i is the diameter of particle i, and

$$\begin{aligned} V_{\mathrm{WCA}}(r) = &\quad 4\varepsilon\left[\left(\tfrac{\sigma}{r-\Delta}\right)^{12} - \left(\tfrac{\sigma}{r-\Delta}\right)^{6}\right] + \varepsilon \quad r - \Delta < 2^{\frac{1}{6}}\sigma \\ = &\quad 0 \qquad\qquad\qquad\qquad\quad r - \Delta \geq 2^{\frac{1}{6}}\sigma \end{aligned}$$

Coefficients:

- k - attractive force strength `k` (in units of energy/distance^2)
- r_0 - size parameter `r0` (in distance units)
- ε - repulsive force strength `epsilon` (in energy units)
- σ - repulsive force interaction distance `sigma` (in distance units)

Examples:

```
fene = bond.fene()
fene.bond_coeff.set('polymer', k=30.0, r0=1.5, sigma=1.0, epsilon= 2.0)
fene.bond_coeff.set('backbone', k=100.0, r0=1.0, sigma=1.0, epsilon= 2.0)
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.bond.**harmonic**(*name=None*)
Harmonic bond potential.

Parameters **name** (*str*) – Name of the bond instance.

harmonic specifies a harmonic potential energy between the two particles in each defined bond.

$$V(r) = \frac{1}{2}k\left(r - r_0\right)^2$$

where $\vec{r}$ is the vector pointing from one particle to the other in the bond.

Coefficients:

- k - force constant `k` (in units of energy/distance^2)
- r_0 - bond rest length `r0` (in distance units)

Example:

```
harmonic = bond.harmonic(name="mybond")
harmonic.bond_coeff.set('polymer', k=330.0, r0=0.84)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.bond.**table**(*width*, *name=None*)

Tabulated bond potential.

Parameters

- **width** (*int*) – Number of points to use to interpolate V and F
- **name** (*str*) – Name of the potential instance

table specifies that a tabulated bond potential should be applied between the two particles in each defined bond.

The force $\vec{F}$ is (in force units) and the potential $V(r)$ is (in energy units):

$$\begin{aligned} \vec{F}(\vec{r}) = & \quad 0 & r < r_{\mathrm{min}} \\ = & \quad F_{\mathrm{user}}(r)\hat{r} & r < r_{\mathrm{max}} \\ = & \quad 0 & r \geq r_{\mathrm{max}} \end{aligned}$$

$$\begin{aligned} V(r) = & \quad 0 & r < r_{\mathrm{min}} \\ = & \quad V_{\mathrm{user}}(r) & r < r_{\mathrm{max}} \\ = & \quad 0 & r \geq r_{\mathrm{max}} \end{aligned}$$

where $\vec{r}$ is the vector pointing from one particle to the other in the bond. Care should be taken to define the range of the bond so that it is not possible for the distance between two bonded particles to be outside the specified range. On the CPU, this will throw an error. On the GPU, this will throw an error if GPU error checking is enabled.

$F_{\mathrm{user}}(r)$ and $V_{\mathrm{user}}(r)$ are evaluated on *width* grid points between r_{min} and r_{max}. Values are interpolated linearly between grid points. For correctness, you must specify the force defined by: $F = -\frac{\partial V}{\partial r}$

The following coefficients must be set for each bond type:

- $F_{\mathrm{user}}(r)$ and $V_{\mathrm{user}}(r)$ - evaluated by `func` (see example)
- coefficients passed to *func* - `coeff` (see example)
- r_{min} - `rmin` (in distance units)
- r_{max} - `rmax` (in distance units)

The table *width* is set once when bond.table is specified. There are two ways to specify the other parameters.

Set table from a given function

When you have a functional form for V and F, you can enter that directly into python. *table* will evaluate the given function over *width* points between *rmin* and *rmax* and use the resulting values in the table:

```
def harmonic(r, rmin, rmax, kappa, r0):
    V = 0.5 * kappa * (r-r0)**2;
    F = -kappa*(r-r0);
    return (V, F)

btable = bond.table(width=1000)
btable.bond_coeff.set('bond1', func=harmonic, rmin=0.2, rmax=5.0,␣
↪coeff=dict(kappa=330, r0=0.84))
btable.bond_coeff.set('bond2', func=harmonic, rmin=0.2, rmax=5.0,␣
↪coeff=dict(kappa=30, r0=1.0))
```

Set a table from a file

When you have no function for for V or F, or you otherwise have the data listed in a file, *table* can use the given values directly. You must first specify the number of rows in your tables when initializing bond.table. Then use *set_from_file()* to read the file:

```
btable = bond.table(width=1000)
btable.set_from_file('polymer', 'btable.file')
```

Note: For potentials that diverge near r=0, make sure to set `rmin` to a reasonable value. If a potential does not diverge near r=0, then a setting of `rmin=0` is valid.

Note: Ensure that `rmin` and `rmax` cover the range of possible bond lengths. When gpu error checking is on, a error will be thrown if a bond distance is outside than this range.

---

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_from_file(*bondname*, *filename*)
Set a bond pair interaction from a file.

Parameters

- **bondname** (*str*) – Name of bond
- **filename** (*str*) – Name of the file to read

The provided file specifies V and F at equally spaced r values. Example:

```
#r  V    F
1.0 2.0 -3.0
1.1 3.0 -4.0
1.2 2.0 -3.0
1.3 1.0 -2.0
1.4 0.0 -1.0
1.5 -1.0 0.0
```

The first r value sets `rmin`, the last sets `rmax`. Any line with # as the first non-whitespace character is is treated as a comment. The `r` values must monotonically increase and be equally spaced. The table is read directly into the grid points used to evaluate $F_{\mathrm{user}}(r)$ and $V_{\mathrm{user}}(r)$.

14.3 md.charge

Overview

md.charge.pppm	Long-range electrostatics computed with the PPPM method.

Details

Electrostatic potentials.

Charged interactions are usually long ranged, and for computational efficiency this is split into two parts, one part computed in real space and on in Fourier space. You don't need to worry about this implementation detail, however, as charge commands in hoomd automatically initialize and configure both the long and short range parts.

Only one method of computing charged interactions should be used at a time. Otherwise, they would add together and produce incorrect results.

class hoomd.md.charge.**pppm**(*group*, *nlist*)

Long-range electrostatics computed with the PPPM method.

Parameters

- **group** (*hoomd.group*) – Group on which to apply long range PPPM forces. The short range part is always applied between all particles.
- **nlist** (*hoomd.md.nlist*) – Neighbor list

D. LeBard et. al. 2012 describes the PPPM implementation details in HOOMD-blue. Please cite it if you utilize the PPPM functionality in your work.

pppm specifies **both** the long-ranged **and** short range parts of the electrostatic force should be computed between all charged particles in the simulation. In other words, *pppm* initializes and sets all parameters for its own *hoomd.md.pair.ewald*, so do not specify an additional one.

The command supports additional screening of interactions, according to the Ewald summation for Yukawa potentials. This is useful if one wants to compute a screened interaction (i.e. a solution to the linearized Poisson-Boltzmann equation), yet the cut-off radius is so large that the computation with a purely short-ranged potential would become inefficient. In that case, the inverse Debye screening length can be supplied using *set_params()*. Also see *Salin, G and Caillol, J. 2000, <http://dx.doi.org/10.1063/1.1326477>*.

Parameters:

- Nx - Number of grid points in x direction
- Ny - Number of grid points in y direction
- Nz - Number of grid points in z direction
- order - Number of grid points in each direction to assign charges to
- r_{cut} - Cutoff for the short-ranged part of the electrostatics calculation

Parameters Nx, Ny, Nz, order, r_{cut} must be set using *set_params()* before any *hoomd.run()* can take place.

See *Units* for information on the units assigned to charges in hoomd.

Note: *pppm* takes a particle group as an option. This should be the group of all charged particles (*hoomd.group.charged()*). However, note that this group is static and determined at the time *pppm* is specified. If you are going to add charged particles at a later point in the simulation with the data access API, ensure that this group includes those particles as well.

Important: In MPI simulations, the number of grid point along every dimensions must be a power of two.

Example:

```
charged = group.charged();
pppm = charge.pppm(group=charged)
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*Nx*, *Ny*, *Nz*, *order*, *rcut*, *alpha=0.0*)
Sets PPPM parameters.

Parameters

- **Nx** (*int*) – Number of grid points in x direction

- **Ny** (*int*) – Number of grid points in y direction
- **Nz** (*int*) – Number of grid points in z direction
- **order** (*int*) – Number of grid points in each direction to assign charges to
- **rcut** (*float*) – Cutoff for the short-ranged part of the electrostatics calculation
- **alpha** (float, **optional**) – Debye screening parameter (in units 1/distance) .. version-added:: 2.1

Examples:

```
pppm.set_params(Nx=64, Ny=64, Nz=64, order=6, rcut=2.0)
```

Note that the Fourier transforms are much faster for number of grid points of the form 2^N.

14.4 md.constrain

Overview

md.constrain.distance	Constrain pairwise particle distances.
md.constrain.rigid	Constrain particles in rigid bodies.
md.constrain.sphere	Constrain particles to the surface of a sphere.
md.constrain.oneD	Constrain particles to move along a specific direction only

Details

Constraints.

Constraint forces constrain a given set of particle to a given surface, to have some relative orientation, or impose some other type of constraint. For example, a group of particles can be constrained to the surface of a sphere with *sphere*.

As with other force commands in hoomd, multiple constrain commands can be issued to specify multiple constraints, which are additively applied.

Warning: Constraints will be invalidated if two separate constraint commands apply to the same particle.

The degrees of freedom removed from the system by constraints are correctly taken into account when computing the temperature for thermostatting and logging.

class hoomd.md.constrain.**distance**

Constrain pairwise particle distances.

distance specifies that forces will be applied to all particles pairs for which constraints have been defined.

The constraint algorithm implemented is described in:

- [1] M. Yoneya, H. J. C. Berendsen, and K. Hirasawa, "A Non-Iterative Matrix Method for Constraint Molecular Dynamics Simulations," Mol. Simul., vol. 13, no. 6, pp. 395–405, 1994.
- [2] M. Yoneya, "A Generalized Non-iterative Matrix Method for Constraint Molecular Dynamics Simulations," J. Comput. Phys., vol. 172, no. 1, pp. 188–197, Sep. 2001.

In brief, the second derivative of the Lagrange multipliers with respect to time is set to zero, such that both the distance constraints and their time derivatives are conserved within the accuracy of the Velocity Verlet scheme, i.e. within Δt^2. The corresponding linear system of equations is solved. Because constraints are satisfied at $t + 2\Delta t$, the scheme is self-correcting and drifts are avoided.

Warning: In MPI simulations, all particles connected through constraints will be communicated between processors as ghost particles. Therefore, it is an error when molecules defined by constraints extend over more than half the local domain size.

Caution: constrain.distance() does not currently interoperate with integrate.brownian() or integrate.langevin()

Example:

```
constrain.distance()
```

disable()
Disable the force.

Example:

```
force.disable()
```

Executing the disable command removes the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*

enable()
Enable the force.

Example:

```
force.enable()
```

See *disable()*.

set_params(*rel_tol=None*)
Set parameters for constraint computation.

Parameters **rel_tol** (*float*) – The relative tolerance with which constraint violations are detected (**optional**).

Example:

```
dist = constrain.distance()
dist.set_params(rel_tol=0.0001)
```

class hoomd.md.constrain.**oneD**(*group, constraint_vector=[0, 0, 1]*)
Constrain particles to move along a specific direction only

Parameters

- **group** (*hoomd.group*) – Group on which to apply the constraint.
- **constraint_vector** (*list*) – [x,y,z] list indicating the direction that the particles are restricted to

oneD specifies that forces will be applied to all particles in the given group to constrain them to only move along a given vector.

Example:

```
constrain.oneD(group=groupA, constraint_vector=[1,0,0])
```

New in version 2.1.

disable()
Disable the force.

Example:

```
force.disable()
```

Executing the disable command removes the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*

enable()
Enable the force.

Example:

```
force.enable()
```

See *disable()*.

class hoomd.md.constrain.**rigid**
Constrain particles in rigid bodies.

Overview

Rigid bodies are defined by a single central particle and a number of constituent particles. All of these are particles in the HOOMD system configuration and can interact with other particles via force computes. The mass and moment of inertia of the central particle set the full mass and moment of inertia of the rigid body (constituent particle mass is ignored).

The central particle is at the center of mass of the rigid body and the orientation quaternion defines the rotation from the body space into the simulation box. In body space, the center of mass of the body is at 0,0,0 and the moment of inertia is diagonal. You specify the constituent particles to *rigid* for each type of body in body coordinates. Then, *rigid* takes control of those particles, and sets their position and orientation in the simulation box relative to the position and orientation of the central particle. *rigid* also transfers forces and torques from constituent particles to the central particle. Then, MD integrators can use these forces and torques to integrate the equations of motion of the central particles (representing the whole rigid body) forward in time.

Defining bodies

rigid accepts one local body environment per body type. The type of a body is the particle type of the central particle in that body. In this way, each particle of type *R* in the system configuration defines a body of type *R*.

As a convenience, you do not need to create placeholder entries for all of the constituent particles in your initial configuration. You only need to specify the positions and orientations of all the central particles. When you call *create_bodies()*, it will create all constituent particles that do not exist. (those that already exist e.g. in a restart file are left unchanged).

Example that creates rigid rods:

```
# Place the type R central particles
uc = hoomd.lattice.unitcell(N = 1,
                            a1 = [10.8, 0,   0],
                            a2 = [0,    1.2, 0],
                            a3 = [0,    0,   1.2],
                            dimensions = 3,
                            position = [[0,0,0]],
                            type_name = ['R'],
                            mass = [1.0],
                            moment_inertia = [[0,
                                               1/12*1.0*8**2,
                                               1/12*1.0*8**2]],
                            orientation = [[1, 0, 0, 0]]);
system = hoomd.init.create_lattice(unitcell=uc, n=[2,18,18]);

# Add constituent particles of type A and create the rods
system.particles.types.add('A');
rigid = hoomd.md.constrain.rigid();
rigid.set_param('R',
                types=['A']*8,
                positions=[(-4,0,0),(-3,0,0),(-2,0,0),(-1,0,0),
                           (1,0,0),(2,0,0),(3,0,0),(4,0,0)]);

rigid.create_bodies()
```

Danger: Automatic creation of constituent particles can change particle tags. If bonds have been defined between particles in the initial configuration, or bonds connect to constituent particles, rigid bodies should be created manually.

When you create the constituent particles manually (i.e. in an input file or with snapshots), the central particle of a rigid body must have a lower tag than all of its constituent particles. Constituent particles follow in monotonically increasing tag order, corresponding to the order they were defined in the argument to *set_param()*. The order of central and contiguous particles need **not** to be contiguous. Additionally, you must set the `body` field for each of the particles in the rigid body to the tag of the central particle (for both the central and constituent particles). Set `body` to -1 for particles that do not belong to a rigid body. After setting an initial configuration that contains properly defined bodies and all their constituent particles, call *validate_bodies()* to verify that the bodies are defined and prepare the constraint.

You must call either *create_bodies()* or *validate_bodies()* prior to starting a simulation *hoomd.run()*.

Integrating bodies

Most integrators in HOOMD support the integration of rotational degrees of freedom. When there are rigid bodies present in the system, do not apply integrators to the constituent particles, only the central and non-rigid particles.

Example:

```
rigid = hoomd.group.rigid_center();
hoomd.md.integrate.langevin(group=rigid, kT=1.0, seed=42);
```

Thermodynamic quantities of bodies

HOOMD computes thermodynamic quantities (temperature, kinetic energy, etc. . .) appropriately when there are rigid bodies present in the system. When it does so, it ignores all constituent particles and computes the translational and rotational energies of the central particles, which represent the whole body. *hoomd.analyze.log* can log the translational and rotational energy terms separately.

Restarting simulations with rigid bodies.

To restart, use *hoomd.dump.gsd* to write restart files. GSD stores all of the particle data fields needed to reconstruct the state of the system, including the body tag, rotational momentum, and orientation of the body. Restarting from a gsd file is equivalent to manual constituent particle creation. You still need to specify the same local body space environment to *rigid* as you did in the earlier simulation.

create_bodies(*create=True*)
Create copies of rigid bodies.

Parameters **create** (*bool*) – When True, create rigid bodies, otherwise validate existing ones.

disable()
Disable the force.

Example:

```
force.disable()
```

Executing the disable command removes the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*

enable()
Enable the force.

Example:

```
force.enable()
```

See *disable()*.

set_param(*type_name*, *types*, *positions*, *orientations=None*, *charges=None*, *diameters=None*)
Set constituent particle types and coordinates for a rigid body.

Parameters

- **type_name** (*str*) – The type of the central particle
- **types** (*list*) – List of types of constituent particles
- **positions** (*list*) – List of relative positions of constituent particles
- **orientations** (*list*) – List of orientations of constituent particles (**optional**)
- **charge** (*list*) – List of charges of constituent particles (**optional**)
- **diameters** (*list*) – List of diameters of constituent particles (**optional**)

Caution: The constituent particle type must be exist. If it does not exist, it can be created on the fly using `system.particles.types.add('A_const')` (see *hoomd.data*).

Example:

```
rigid = constrain.rigid()
rigid.set_param('A', types = ['A_const', 'A_const'], positions = [(0,0,1),(0,
↪0,-1)])
rigid.set_param('B', types = ['B_const', 'B_const'], positions = [(0,0,.5),(0,
↪0,-.5)])
```

validate_bodies()
Validate that bodies are well defined and prepare for the simulation run.

class hoomd.md.constrain.**sphere**(*group*, *P*, *r*)
Constrain particles to the surface of a sphere.

Parameters

- **group** (*hoomd.group*) – Group on which to apply the constraint.
- **P** (*tuple*) – (x,y,z) tuple indicating the position of the center of the sphere (in distance units).
- **r** (*float*) – Radius of the sphere (in distance units).

sphere specifies that forces will be applied to all particles in the given group to constrain them to a sphere. Currently does not work with Brownian or Langevin dynamics (*hoomd.md.integrate.brownian* and *hoomd.md.integrate.langevin*).

Example:

```
constrain.sphere(group=groupA, P=(0,10,2), r=10)
```

disable()
Disable the force.

Example:

```
force.disable()
```

Executing the disable command removes the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*

enable()
Enable the force.

Example:

```
force.enable()
```

See *disable()*.

14.5 md.dihedral

Overview

md.dihedral.harmonic	Harmonic dihedral potential.

Continued on next page

Table 5 – continued from previous page

md.dihedral.opls	OPLS dihedral force
md.dihedral.table	Tabulated dihedral potential.

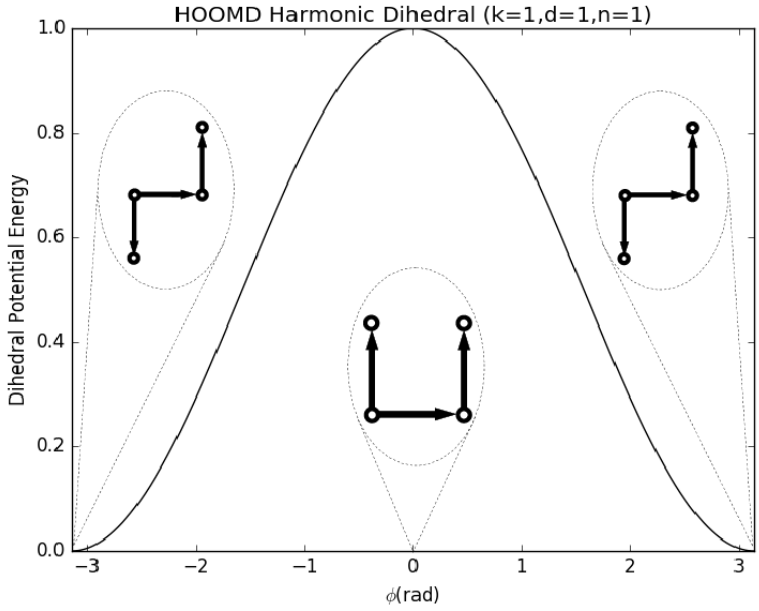
Details

Dihedral potentials.

Dihedrals add forces between specified quadruplets of particles and are typically used to model rotation about chemical bonds.

By themselves, dihedrals that have been specified in an input file do nothing. Only when you specify an dihedral force (i.e. dihedral.harmonic), are forces actually calculated between the listed particles.

Important: There are multiple conventions pertaining to the dihedral angle (phi) in the literature. HOOMD utilizes the convention shown in the following figure, where vectors are defined from the central particles to the outer particles. These vectors correspond to a stretched state (phi=180 deg) when they are anti-parallel and a compact state (phi=0 deg) when they are parallel.



class hoomd.md.dihedral.**coeff**

Defines dihedral coefficients.

The coefficients for all dihedral force are specified using this class. Coefficients are specified per dihedral type.

There are two ways to set the coefficients for a particular dihedral force. The first way is to save the dihedral force in a variable and call *set()* directly. See below for an example of this.

The second method is to build the *coeff* class first and then assign it to the dihedral force. There are some advantages to this method in that you could specify a complicated set of dihedral force coefficients in a separate python file and import it into your job script.

Examples:

```
my_coeffs = dihedral.coeff();
my_dihedral_force.dihedral_coeff.set('polymer', k=330.0, r=0.84)
my_dihedral_force.dihedral_coeff.set('backbone', k=330.0, r=0.84)
```

set (*type*, ***coeffs*)
Sets parameters for dihedral types.

Parameters

- **type** (*str*) – Type of dihedral, or list of types
- **coeffs** – Named coefficients (see below for examples)

Calling *set()* results in one or more parameters being set for a dihedral type. Types are identified by name, and parameters are also added by name. Which parameters you need to specify depends on the dihedral force you are setting these coefficients for, see the corresponding documentation.

All possible dihedral types as defined in the simulation box must be specified before executing run(). You will receive an error if you fail to do so. It is not an error, however, to specify coefficients for dihedral types that do not exist in the simulation. This can be useful in defining a force field for many different types of dihedrals even when some simulations only include a subset.

To set the same coefficients between many particle types, provide a list of type names instead of a single one. All types in the list will be set to the same parameters.

Examples:

```
my_dihedral_force.dihedral_coeff.set('polymer', k=330.0, r0=0.84)
my_dihedral_force.dihedral_coeff.set('backbone', k=1000.0, r0=1.0)
my_dihedral_force.dihedral_coeff.set(['dihedralA','dihedralB'], k=100, r0=0.0)
```

Note: Single parameters can be updated. If both `k` and `r0` have already been set for a particle type, then executing `coeff.set('polymer', r0=1.0)` will update the value of `r0` and leave the other parameters as they were previously set.

class hoomd.md.dihedral.**harmonic**
Harmonic dihedral potential.

harmonic specifies a harmonic dihedral potential energy between every defined dihedral quadruplet of particles in the simulation:

$$V(r) = \frac{1}{2}k\left(1 + d\cos\left(n * \phi(r)\right)\right)$$

where ϕ is angle between two sides of the dihedral.

Coefficients:

- k - strength of force (in energy units)
- d - sign factor (unitless)
- n - angle scaling factor (unitless)

Coefficients k, d, n must be set for each type of dihedral in the simulation using *dihedral_coeff.set()*.

Examples:

```
harmonic.dihedral_coeff.set('phi-ang', k=30.0, d=-1, n=3)
harmonic.dihedral_coeff.set('psi-ang', k=100.0, d=1, n=4)
```

disable (*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log*=*True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.dihedral.**opls**
OPLS dihedral force

opls specifies an OPLS-style dihedral potential energy between every defined dihedral.

$$V(r) = \frac{1}{2}k_1\left(1 + \cos\left(\phi\right)\right) + \frac{1}{2}k_2\left(1 - \cos\left(2\phi\right)\right) + \frac{1}{2}k_3\left(1 + \cos\left(3\phi\right)\right) + \frac{1}{2}k_4\left(1 - \cos\left(4\phi\right)\right)$$

where ϕ is the angle between two sides of the dihedral and k_n are the force coefficients in the Fourier series (in energy units).

k_1, k_2, k_3, and k_4 must be set for each type of dihedral in the simulation using *dihedral_coeff.set()*.

Example:

```
opls_di.dihedral_coeff.set('dihedral1', k1=30.0, k2=15.5, k3=2.2, k4=23.8)
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.dihedral.**table**(*width*, *name=None*)
Tabulated dihedral potential.

Parameters

- **width** (*int*) – Number of points to use to interpolate V and T (see documentation above)

- **name** (*str*) – Name of the force instance

table specifies that a tabulated dihedral force should be applied to every define dihedral.

$T_{\text{user}}(\theta)$ and $V_{\text{user}}(\theta)$ are evaluated on *width* grid points between $-\pi$ and π. Values are interpolated linearly between grid points. For correctness, you must specify the derivative of the potential with respect to the dihedral angle, defined by: $T = -\frac{\partial V}{\partial \theta}$.

Parameters:

- $T_{\text{user}}(\theta)$ and $V_{\text{user}}(\theta)$ - evaluated by `func` (see example)
- coefficients passed to *func* - *coeff* (see example)

Set table from a given function

When you have a functional form for V and T, you can enter that directly into python. *table* will evaluate the given function over *width* points between $-\pi$ and π and use the resulting values in the table:

```
def harmonic(theta, kappa, theta0):
   V = 0.5 * kappa * (theta-theta0)**2;
   F = -kappa*(theta-theta0);
   return (V, F)

dtable = dihedral.table(width=1000)
dtable.dihedral_coeff.set('dihedral1', func=harmonic, coeff=dict(kappa=330, theta_
↪0=0.0))
dtable.dihedral_coeff.set('dihedral2', func=harmonic,coeff=dict(kappa=30, theta_
↪0=1.0))
```

Set a table from a file

When you have no function for for *V* or *T*, or you otherwise have the data listed in a file, dihedral.table can use the given values directly. You must first specify the number of rows in your tables when initializing *table*. Then use *set_from_file()* to read the file.

dtable = dihedral.table(width=1000) dtable.set_from_file('polymer', 'dihedral.dat')

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_from_file(*dihedralname*, *filename*)
Set a dihedral pair interaction from a file.

Parameters

- **dihedralname** (*str*) – Name of dihedral
- **filename** (*str*) – Name of the file to read

The provided file specifies V and F at equally spaced theta values.

Example:

```
#t  V    T
-3.141592653589793 2.0 -3.0
-1.5707963267948966 3.0 -4.0
0.0 2.0 -3.0
1.5707963267948966 3.0 -4.0
3.141592653589793 2.0 -3.0
```

Note: The theta values are not used by the code. It is assumed that a table that has N rows will start at $-\pi$, end at π and that $\delta\theta = 2\pi/(N-1)$. The table is read directly into the grid points used to evaluate $T_{\mathrm{user}}(\theta)$ and $V_{\mathrm{user}}(\theta)$.

14.6 md.external

Overview

md.external.periodic	One-dimension periodic potential.
md.external.e_field	Electric field.

Details

External forces.

Apply an external force to all particles in the simulation. This module organizes all external forces. As an example, a force derived from a *periodic* potential can be used to induce a concentration modulation in the system.

class hoomd.md.external.**coeff**

Defines external potential coefficients.

The coefficients for all external forces are specified using this class. Coefficients are specified per particle type.

Example:

```
my_external_force.force_coeff.set('A', A=1.0, i=1, w=0.02, p=3)
my_external_force.force_coeff.set('B', A=-1.0, i=1, w=0.02, p=3)
```

set (*type*, ***coeffs*)

Sets parameters for particle types.

Parameters

- **type** (*str*) – Type of particle (or list of types)
- **coeff** – Named coefficients (see below for examples)

Calling *set()* results in one or more parameters being set for a particle type. Types are identified by name, and parameters are also added by name. Which parameters you need to specify depends on the external force you are setting these coefficients for, see the corresponding documentation.

All possible particle types as defined in the simulation box must be specified before executing run(). You will receive an error if you fail to do so. It is not an error, however, to specify coefficients for particle types that do not exist in the simulation. This can be useful in defining a force field for many different types of particles even when some simulations only include a subset.

To set the same coefficients between many particle types, provide a list of type names instead of a single one. All types in the list will be set to the same parameters. A convenient wildcard that lists all types of particles in the simulation can be gotten from a saved system from the init command.

Examples:

```
coeff.set('A', A=1.0, i=1, w=0.02, p=3)
coeff.set('B', A=-1.0, i=1, w=0.02, p=3)
coeff.set(['A','B'], i=1, w=0.02, p=3)
```

Note: Single parameters can be updated. For example, executing `coeff.set('A', A=1.0)` will update the value of `A` and leave the other parameters as they were previously set.

class hoomd.md.external.**e_field**(*field*, *name=''*)

Electric field.

e_field specifies that an external force should be added to every particle in the simulation that results from an electric field.

The external potential $V(\vec{r})$ is implemented using the following formula:

$$V(\vec{r}) = -q_i \vec{E} \cdot \vec{r}$$

where q_i is the particle charge and $\vec{E}$ is the field vector

Example:

```
# Apply an electric field in the x-direction
e_field = external.e_field((1,0,0))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.external.**periodic**(*name=''*)

One-dimension periodic potential.

periodic specifies that an external force should be added to every particle in the simulation to induce a periodic modulation in the particle concentration. The force parameters can be set on a per particle type basis. The potential can e.g. be used to induce an ordered phase in a block-copolymer melt.

The external potential $V(\vec{r})$ is implemented using the following formula:

$$V(\vec{r}) = A * \tanh\left[\frac{1}{2\pi p w}\cos\left(p\vec{b_i}\cdot\vec{r}\right)\right]$$

where A is the ordering parameter, $\vec{b_i}$ is the reciprocal lattice vector direction $i = 0..2$, p the periodicity and w the interface width (relative to the distance $2\pi/|\mathbf{b_i}|$ between planes in the i-direction). The modulation is one-dimensional. It extends along the lattice vector $\mathbf{a}_i$ of the simulation cell.

Examples:

```
# Apply a periodic composition modulation along the first lattice vector
periodic = external.periodic()
periodic.force_coeff.set('A', A=1.0, i=0, w=0.02, p=3)
periodic.force_coeff.set('B', A=-1.0, i=0, w=0.02, p=3)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

14.7 md.force

Overview

md.force.active	Active force.
md.force.constant	Constant force.
md.force.dipole	Treat particles as dipoles in an electric field.

Details

Apply forces to particles.

class hoomd.md.force.**active**(*seed*, *group*, *f_lst=None*, *t_lst=None*, *orientation_link=True*, *orientation_reverse_link=False*, *rotation_diff=0*, *constraint=None*)

Active force.

Parameters

- **seed** (*int*) – required user-specified seed number for random number generator.
- **f_list** (*list*) – An array of (x,y,z) tuples for the active force vector for each individual particle.
- **t_list** (*list*) – An array of (x,y,z) tuples that indicate active torque vectors for each particle
- **group** (*hoomd.group*) – Group for which the force will be set
- **orientation_link** (*bool*) – if True then forces and torques are applied in the particle's reference frame. If false, then the box reference frame is used. Only relevant for non-point-like anisotropic particles.
- **orientation_reverse_link** (*bool*) – When True, the particle's orientation is set to match the active force vector. Useful for for using a particle's orientation to log the active force vector. Not recommended for anisotropic particles. Quaternion rotation assumes base vector of (0,0,1).

- **rotation_diff** (*float*) – rotational diffusion constant, D_r, for all particles in the group.
- **constraint** (*hoomd.md.update.constraint_ellipsoid*) – such as update.constraint_ellipsoid.

active specifies that an active force should be added to all particles. Obeys $\delta \mathbf{r}_i = \delta t v_0 \hat{p}_i$, where v_0 is the active velocity. In 2D $\hat{p}_i = (\cos\theta_i, \sin\theta_i)$ is the active force vector for particle i; and the diffusion of the active force vector follows $\delta\theta/\delta t = \sqrt{2D_r/\delta t}\Gamma$, where D_r is the rotational diffusion constant, and the gamma function is a unit-variance random variable, whose components are uncorrelated in time, space, and between particles. In 3D, $\hat{p}_i$ is a unit vector in 3D space, and diffusion follows $\delta\hat{p}_i/\delta t = \sqrt{2D_r/\delta t}\Gamma(\hat{p}_i(\cos\theta - 1) + \hat{p}_r \sin\theta)$, where $\hat{p}_r$ is an uncorrelated random unit vector. The persistence length of an active particle's path is v_0/D_r.

Attention: *active()* does not support MPI execution.

Examples:

```
force.active( seed=13, f_list=[tuple(3,0,0) for i in range(N)])

ellipsoid = update.constraint_ellipsoid(group=groupA, P=(0,0,0), rx=3, ry=4, rz=5)
force.active( seed=7, f_list=[tuple(1,2,3) for i in range(N)], orientation_
↪link=False, rotation_diff=100, constraint=ellipsoid)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.force.**constant**(*fx=None*, *fy=None*, *fz=None*, *fvec=None*, *tvec=None*, *group=None*, *callback=None*)

Constant force.

Parameters

- **fvec** (*tuple*) – force vector (in force units)
- **tvec** (*tuple*) – torque vector (in torque units)
- **fx** (*float*) – x component of force, retained for backwards compatibility
- **fy** (*float*) – y component of force, retained for backwards compatibility
- **fz** (*float*) – z component of force, retained for backwards compatibility
- **group** (*hoomd.group*) – Group for which the force will be set.
- **callback** (*callable*) – A python callback invoked every time the forces are computed

constant specifies that a constant force should be added to every particle in the simulation or optionally to all particles in a group.

Note: Forces are kept constant during the simulation. If a callback should re-compute particle forces every time step, it needs to overwrite the old forces of **all** particles with new values.

Note: Per-particle forces take precedence over a particle group, which takes precedence over constant forces for all particles.

Examples:

```
force.constant(fx=1.0, fy=0.5, fz=0.25)
const = force.constant(fvec=(0.4,1.0,0.5))
const = force.constant(fvec=(0.4,1.0,0.5),group=fluid)
const = force.constant(fvec=(0.4,1.0,0.5), tvec=(0,0,1) ,group=fluid)

def update_forces(timestep):
    global const
    const.set_force(tag=1, fvec=(1.0*timestep,2.0*timestep,3.0*timestep))
const = force.constant(callback=update_forces)
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.force.**dipole**(*field_x*, *field_y*, *field_z*, *p*)
Treat particles as dipoles in an electric field.

Parameters

- **field_x** (*float*) – x-component of the field (units?)
- **field_y** (*float*) – y-component of the field (units?)
- **field_z** (*float*) – z-component of the field (units?)

- **p** (*float*) – magnitude of the particles' dipole moment in the local z direction

Examples:

```
force.external_field_dipole(field_x=0.0, field_y=1.0 ,field_z=0.5, p=1.0)
const_ext_f_dipole = force.external_field_dipole(field_x=0.0, field_y=1.0 ,field_
↪z=0.5, p=1.0)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params (*field_y*, *field_z*, *p*)
Change the constant field and dipole moment.

Parameters

- **field_x** (*float*) – x-component of the field (units?)
- **field_y** (*float*) – y-component of the field (units?)
- **field_z** (*float*) – z-component of the field (units?)
- **p** (*float*) – magnitude of the particles' dipole moment in the local z direction

Examples:

```
const_ext_f_dipole = force.external_field_dipole(field_x=0.0, field_y=1.0 ,
↪field_z=0.5, p=1.0)
const_ext_f_dipole.setParams(field_x=0.1, field_y=0.1, field_z=0.0, p=1.0))
```

14.8 md.improper

Overview

md.improper.harmonic	Harmonic improper potential.

Details

Improper potentials.

Impropers add forces between specified quadruplets of particles and are typically used to model rotation about chemical bonds without having bonds to connect the atoms. Their most common use is to keep structural elements flat, i.e. model the effect of conjugated double bonds, like in benzene rings and its derivatives.

By themselves, impropers that have been specified in an input file do nothing. Only when you specify an improper force (i.e. improper.harmonic), are forces actually calculated between the listed particles.

class hoomd.md.improper.**coeff**
Define improper coefficients.

The coefficients for all improper force are specified using this class. Coefficients are specified per improper type.

Examples:

```
my_coeffs = improper.coeff();
my_improper_force.improper_coeff.set('polymer', k=330.0, r=0.84)
my_improper_force.improper_coeff.set('backbone', k=330.0, r=0.84)
```

set (*type*, ***coeffs*)
Sets parameters for one improper type.

Parameters

- **type** (*str*) – Type of improper (or list of types).
- **coeffs** – Named coefficients (see below for examples)

Calling *set()* results in one or more parameters being set for a improper type. Types are identified by name, and parameters are also added by name. Which parameters you need to specify depends on the improper force you are setting these coefficients for, see the corresponding documentation.

All possible improper types as defined in the simulation box must be specified before executing *hoomd.run()*. You will receive an error if you fail to do so. It is not an error, however, to specify coefficients for improper types that do not exist in the simulation. This can be useful in defining a force field for many different types of impropers even when some simulations only include a subset.

To set the same coefficients between many particle types, provide a list of type names instead of a single one. All types in the list will be set to the same parameters.

Examples:

```
my_improper_force.improper_coeff.set('polymer', k=330.0, r0=0.84)
my_improper_force.improper_coeff.set('backbone', k=1000.0, r0=1.0)
my_improper_force.improper_coeff.set(['improperA','improperB'], k=100, r0=0.0)
```

Note: Single parameters can be updated. If both `k` and `r0` have already been set for a particle type, then executing `coeff.set('polymer', r0=1.0)` will update the value of `r0` and leave the other parameters as they were previously set.

class hoomd.md.improper.**harmonic**

Harmonic improper potential.

The command improper.harmonic specifies a harmonic improper potential energy between every quadruplet of particles in the simulation.

$$V(r) = \frac{1}{2}k\left(\chi - \chi_0\right)^2$$

where χ is angle between two sides of the improper.

Coefficients:

- k - strength of force, `k` (in energy units)
- χ_0 - equilibrium angle, `chi` (in radians)

Coefficients k and χ_0 must be set for each type of improper in the simulation using *improper_coeff.set()*.

Examples:

```
harmonic.improper_coeff.set('heme-ang', k=30.0, chi=1.57)
harmonic.improper_coeff.set('hydro-bond', k=20.0, chi=1.57)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log*=*True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

14.9 md.integrate

Overview

md.integrate.berendsen	Applies the Berendsen thermostat.
md.integrate.brownian	Brownian dynamics.
md.integrate.langevin	Langevin dynamics.
md.integrate.mode_standard	Enables a variety of standard integration methods.
md.integrate.mode_minimize_fire	Energy Minimizer (FIRE).
md.integrate.npt	NPT Integration via MTK barostat-thermostat.
md.integrate.nph	NPH Integration via MTK barostat-thermostat..
md.integrate.nve	NVE Integration via Velocity-Verlet
md.integrate.nvt	NVT Integration via the Nosé-Hoover thermostat.

Details

Integration methods.

To integrate the system forward in time, an integration mode must be set. Only one integration mode can be active at a time, and the last `integrate.mode_*` command before the *hoomd.run()* command is the one that will take effect. It is possible to set one mode, run for a certain number of steps and then switch to another mode before the next run command.

The most commonly used mode is :py:class‘mode_standard‘. It specifies a standard mode where, at each time step, all of the specified forces are evaluated and used in moving the system forward to the next step. :py:class‘mode_standard‘ doesn't integrate any particles by itself, one or more compatible integration methods must be specified before the staring a *hoomd.run()*. Like commands that specify forces, integration methods are **persistent** and remain set until they are disabled.

To clarify, the following series of commands will run for 1000 time steps in the NVT ensemble and then switch to NVE for another 1000 steps:

```
all = group.all()
integrate.mode_standard(dt=0.005)
nvt = integrate.nvt(group=all, kT=1.2, tau=0.5)
run(1000)
nvt.disable()
integrate.nve(group=all)
run(1000)
```

You can change integrator parameters between runs:

```
integrator = integrate.nvt(group=all, kT=1.2, tau=0.5)
run(100)
integrator.set_params(kT=1.0)
run(100)
```

This code snippet runs the first 100 time steps with kT=1.2 and the next 100 with kT=1.0.

class hoomd.md.integrate.**berendsen**(*group*, *kT*, *tau*)

Applies the Berendsen thermostat.

Parameters

- **group** (*hoomd.group*) – Group to which the Berendsen thermostat will be applied.
- **kT** (*hoomd.variant* or `float`) – Temperature of thermostat. (in energy units).
- **tau** (*float*) – Time constant of thermostat. (in time units)

berendsen rescales the velocities of all particles on each time step. The rescaling is performed so that the difference in the current temperature from the set point decays exponentially: Berendsen et. al. 1984.

$$\frac{dT_{\mathrm{cur}}}{dt} = \frac{T - T_{\mathrm{cur}}}{\tau}$$

Attention: *berendsen* does not function with MPI parallel simulations.

Attention: *berendsen* does not integrate rotational degrees of freedom.

disable()

Disables the integration method.

Examples:

```
method.disable()
```

Executing the disable command will remove the integration method from the simulation. Any *hoomd.run()* command executed after disabling an integration method will not apply the integration method to the particles during the simulation. A disabled integration method can be re-enabled with *enable()*.

enable()
Enables the integration method.

Examples:

```
method.enable()
```

See also:

disable().

randomize_velocities(*seed*)
Assign random velocities and angular momenta to particles in the group, sampling from the Maxwell-Boltzmann distribution. This method considers the dimensionality of the system and particle anisotropy, and removes drift (the center of mass velocity).

New in version 2.3.

Parameters **seed** (*int*) – Random number seed

Note: Randomization is applied at the start of the next call to *hoomd.run()*.

Example:

```
integrator = md.integrate.berendsen(group=group.all(), kT=1.0, tau=0.5)
integrator.randomize_velocities(seed=42)
run(100)
```

class hoomd.md.integrate.**brownian**(*group*, *kT*, *seed*, *dscale=False*, *noiseless_t=False*, *noiseless_r=False*)
Brownian dynamics.

Parameters

- **group** (*hoomd.group*) – Group of particles to apply this method to.
- **kT** (*hoomd.variant* or *float*) – Temperature of the simulation (in energy units).
- **seed** (*int*) – Random seed to use for generating $\vec{F}_{\mathrm{R}}$.
- **dscale** (*bool*) – Control λ options. If 0 or False, use γ values set per type. If non-zero, $\gamma = \lambda d_i$.
- **noiseless_t** (*bool*) – If set true, there will be no translational noise (random force)
- **noiseless_r** (*bool*) – If set true, there will be no rotational noise (random torque)

brownian integrates particles forward in time according to the overdamped Langevin equations of motion,

sometimes called Brownian dynamics, or the diffusive limit.

$$\frac{d\vec{x}}{dt} = \frac{\vec{F}_{\mathrm{C}} + \vec{F}_{\mathrm{R}}}{\gamma}$$
$$\langle \vec{F}_{\mathrm{R}} \rangle = 0$$
$$\langle |\vec{F}_{\mathrm{R}}|^2 \rangle = 2dkT\gamma/\delta t$$
$$\langle \vec{v}(t) \rangle = 0$$
$$\langle |\vec{v}(t)|^2 \rangle = dkT/m$$

where $\vec{F}_{\mathrm{C}}$ is the force on the particle from all potentials and constraint forces, γ is the drag coefficient, $\vec{F}_{\mathrm{R}}$ is a uniform random force, $\vec{v}$ is the particle's velocity, and d is the dimensionality of the system. The magnitude of the random force is chosen via the fluctuation-dissipation theorem to be consistent with the specified drag and temperature, T. When $kT = 0$, the random force $\vec{F}_{\mathrm{R}} = 0$.

brownian generates random numbers by hashing together the particle tag, user seed, and current time step index. See C. L. Phillips et. al. 2011 for more information.

Attention: Change the seed if you reset the simulation time step to 0. If you keep the same seed, the simulation will continue with the same sequence of random numbers used previously and may cause unphysical correlations.

For MPI runs: all ranks other than 0 ignore the seed input and use the value of rank 0.

brownian uses the integrator from I. Snook, The Langevin and Generalised Langevin Approach to the Dynamics of Atomic, Polymeric and Colloidal Systems, 2007, section 6.2.5, with the exception that $\vec{F}_{\mathrm{R}}$ is drawn from a uniform random number distribution.

In Brownian dynamics, particle velocities are completely decoupled from positions. At each time step, *brownian* draws a new velocity distribution consistent with the current set temperature so that *hoomd.compute.thermo* will report appropriate temperatures and pressures if logged or needed by other commands.

Brownian dynamics neglects the acceleration term in the Langevin equation. This assumption is valid when overdamped: $\frac{m}{\gamma} \ll \delta t$. Use *langevin* if your system is not overdamped.

You can specify γ in two ways:

1. Use *set_gamma()* to specify it directly, with independent values for each particle type in the system.
2. Specify λ which scales the particle diameter to $\gamma = \lambda d_i$. The units of λ are mass / distance / time.

brownian must be used with integrate.mode_standard.

kT can be a variant type, allowing for temperature ramps in simulation runs.

A *hoomd.compute.thermo* is automatically created and associated with *group*.

Examples:

```
all = group.all();
integrator = integrate.brownian(group=all, kT=1.0, seed=5)
integrator = integrate.brownian(group=all, kT=1.0, dscale=1.5)
typeA = group.type('A');
integrator = integrate.brownian(group=typeA, kT=hoomd.variant.linear_interp([(0,␣
↪4.0), (1e6, 1.0)]), seed=10)
```

disable()

Disables the integration method.

Examples:

```
method.disable()
```

Executing the disable command will remove the integration method from the simulation. Any *hoomd.run()* command executed after disabling an integration method will not apply the integration method to the particles during the simulation. A disabled integration method can be re-enabled with *enable()*.

enable()
Enables the integration method.

Examples:

```
method.enable()
```

See also:

disable().

set_gamma(*a*, *gamma*)
Set gamma for a particle type.

Parameters

- **a** (*str*) – Particle type name
- **gamma** (*float*) – γ for particle type a (in units of force/velocity)

set_gamma() sets the coefficient γ for a single particle type, identified by name. The default is 1.0 if not specified for a type.

It is not an error to specify gammas for particle types that do not exist in the simulation. This can be useful in defining a single simulation script for many different types of particles even when some simulations only include a subset.

Examples:

```
bd.set_gamma('A', gamma=2.0)
```

set_gamma_r(*a*, *gamma_r*)
Set gamma_r for a particle type.

Parameters

- **a** (*str*) – Particle type name
- **gamma_r** (*float or tuple*) – γ_r for particle type a (in units of force/velocity), optionally for all body frame directions

set_gamma_r() sets the coefficient γ_r for a single particle type, identified by name. The default is 1.0 if not specified for a type. It must be positive or zero, if set zero, it will have no rotational damping or random torque, but still with updates from normal net torque.

Examples:

```
bd.set_gamma_r('A', gamma_r=2.0)
bd.set_gamma_r('A', gamma_r=(1,2,3))
```

set_params(*kT=None*)
Change langevin integrator parameters.

Parameters **kT** (*hoomd.variant* or *float*) – New temperature (if set) (in energy units).

Examples:

```
integrator.set_params(kT=2.0)
```

class hoomd.md.integrate.**langevin**(*group*, *kT*, *seed*, *dscale=False*, *tally=False*, *noiseless_t=False*, *noiseless_r=False*)

Langevin dynamics.

Parameters

- **group** (*hoomd.group*) – Group of particles to apply this method to.
- **kT** (*hoomd.variant* or *float*) – Temperature of the simulation (in energy units).
- **seed** (*int*) – Random seed to use for generating $\vec{F}_{\mathrm{R}}$.
- **dscale** (*bool*) – Control λ options. If 0 or False, use γ values set per type. If non-zero, $\gamma = \lambda d_i$.
- **tally** (*bool*) – (optional) If true, the energy exchange between the thermal reservoir and the particles is tracked. Total energy conservation can then be monitored by adding `langevin_reservoir_energy_groupname` to the logged quantities.
- **noiseless_t** (*bool*) – If set true, there will be no translational noise (random force)
- **noiseless_r** (*bool*) – If set true, there will be no rotational noise (random torque)

Translational degrees of freedom

langevin integrates particles forward in time according to the Langevin equations of motion:

$$m\frac{d\vec{v}}{dt} = \vec{F}_{\mathrm{C}} - \gamma \cdot \vec{v} + \vec{F}_{\mathrm{R}}$$
$$\langle \vec{F}_{\mathrm{R}} \rangle = 0$$
$$\langle |\vec{F}_{\mathrm{R}}|^2 \rangle = 2dkT\gamma/\delta t$$

where $\vec{F}_{\mathrm{C}}$ is the force on the particle from all potentials and constraint forces, γ is the drag coefficient, $\vec{v}$ is the particle's velocity, $\vec{F}_{\mathrm{R}}$ is a uniform random force, and d is the dimensionality of the system (2 or 3). The magnitude of the random force is chosen via the fluctuation-dissipation theorem to be consistent with the specified drag and temperature, T. When $kT = 0$, the random force $\vec{F}_{\mathrm{R}} = 0$.

langevin generates random numbers by hashing together the particle tag, user seed, and current time step index. See C. L. Phillips et. al. 2011 for more information.

Attention: Change the seed if you reset the simulation time step to 0. If you keep the same seed, the simulation will continue with the same sequence of random numbers used previously and may cause unphysical correlations.

For MPI runs: all ranks other than 0 ignore the seed input and use the value of rank 0.

Langevin dynamics includes the acceleration term in the Langevin equation and is useful for gently thermalizing systems using a small gamma. This assumption is valid when underdamped: $\frac{m}{\gamma} \gg \delta t$. Use *brownian* if your system is not underdamped.

langevin uses the same integrator as *nve* with the additional force term $-\gamma \cdot \vec{v} + \vec{F}_{\mathrm{R}}$. The random force $\vec{F}_{\mathrm{R}}$ is drawn from a uniform random number distribution.

You can specify γ in two ways:

1. Use *set_gamma()* to specify it directly, with independent values for each particle type in the system.

2. Specify λ which scales the particle diameter to $\gamma = \lambda d_i$. The units of λ are mass / distance / time.

langevin must be used with *mode_standard*.

kT can be a variant type, allowing for temperature ramps in simulation runs.

A *hoomd.compute.thermo* is automatically created and associated with *group*.

Warning: When restarting a simulation, the energy of the reservoir will be reset to zero.

Examples:

```
all = group.all();
integrator = integrate.langevin(group=all, kT=1.0, seed=5)
integrator = integrate.langevin(group=all, kT=1.0, dscale=1.5, tally=True)
typeA = group.type('A');
integrator = integrate.langevin(group=typeA, kT=hoomd.variant.linear_interp([(0,␣
↪4.0), (1e6, 1.0)]), seed=10)
```

disable()

Disables the integration method.

Examples:

```
method.disable()
```

Executing the disable command will remove the integration method from the simulation. Any *hoomd.run()* command executed after disabling an integration method will not apply the integration method to the particles during the simulation. A disabled integration method can be re-enabled with *enable()*.

enable()

Enables the integration method.

Examples:

```
method.enable()
```

See also:

disable().

set_gamma(*a*, *gamma*)

Set gamma for a particle type.

Parameters

- **a** (*str*) – Particle type name
- **gamma** (*float*) – γ for particle type a (in units of force/velocity)

set_gamma() sets the coefficient γ for a single particle type, identified by name. The default is 1.0 if not specified for a type.

It is not an error to specify gammas for particle types that do not exist in the simulation. This can be useful in defining a single simulation script for many different types of particles even when some simulations only include a subset.

Examples:

```
bd.set_gamma('A', gamma=2.0)
```

set_gamma_r(*a*, *gamma_r*)
Set gamma_r for a particle type.

Parameters

- **a** (*str*) – Particle type name
- **gamma_r** (*float or tuple*) – γ_r for particle type a (in units of force/velocity), optionally for all body frame directions

set_gamma_r() sets the coefficient γ_r for a single particle type, identified by name. The default is 1.0 if not specified for a type. It must be positive or zero, if set zero, it will have no rotational damping or random torque, but still with updates from normal net torque.

Examples:

```
langevin.set_gamma_r('A', gamma_r=2.0)
langevin.set_gamma_r('A', gamma_r=(1.0,2.0,3.0))
```

set_params(*kT=None*, *tally=None*)
Change langevin integrator parameters.

Parameters

- **kT** (*hoomd.variant* or float) – New temperature (if set) (in energy units).
- **tally** (*bool*) – (optional) If true, the energy exchange between the thermal reservoir and the particles is tracked. Total energy conservation can then be monitored by adding `langevin_reservoir_energy_groupname` to the logged quantities.

Examples:

```
integrator.set_params(kT=2.0)
integrator.set_params(tally=False)
```

class hoomd.md.integrate.**mode_minimize_fire**(*dt*, *Nmin=5*, *finc=1.1*, *fdec=0.5*, *alpha_start=0.1*, *falpha=0.99*, *ftol=0.1*, *wtol=0.1*, *Etol=1e-05*, *min_steps=10*, *group=None*, *aniso=None*)

Energy Minimizer (FIRE).

Parameters

- **group** (*hoomd.group*) – Particle group to apply minimization to. Deprecated in version 2.2: *hoomd.md.integrate.mode_minimize_fire()* now accepts integration methods, such as *hoomd.md.integrate.nve()* and *hoomd.md.integrate.nph()*. The functions operate on user-defined groups. If **group** is defined here, automatically *hoomd.md.integrate.nve()* will be used for integration
- **dt** (*float*) – This is the maximum step size the minimizer is permitted to use. Consider the stability of the system when setting. (in time units)
- **Nmin** (*int*) – Number of steps energy change is negative before allowing α and δt to adapt.
- **finc** (*float*) – Factor to increase δt by
- **fdec** (*float*) – Factor to decrease δt by
- **alpha_start** (*float*) – Initial (and maximum) α
- **falpha** (*float*) – Factor to decrease αt by
- **ftol** (*float*) – force convergence criteria (in units of force over mass)

- **wtol** (*float*) – angular momentum convergence criteria (in units of angular momentum)
- **Etol** (*float*) – energy convergence criteria (in energy units)
- **min_steps** (*int*) – A minimum number of attempts before convergence criteria are considered
- **aniso** (*bool*) – Whether to integrate rotational degrees of freedom (bool), default None (autodetect). Added in version 2.2

New in version 2.1.

Changed in version 2.2.

mode_minimize_fire uses the Fast Inertial Relaxation Engine (FIRE) algorithm to minimize the energy for a group of particles while keeping all other particles fixed. This method is published in Bitzek, et. al., PRL, 2006.

At each time step, δt, the algorithm uses the NVE Integrator to generate a x, v, and F, and then adjusts v according to

$$\vec{v} = (1-\alpha)\vec{v} + \alpha\hat{F}|\vec{v}|$$

where α and δt are dynamically adaptive quantities. While a current search has been lowering the energy of system for more than N_{min} steps, α is decreased by $\alpha \rightarrow \alpha f_{alpha}$ and δt is increased by $\delta t \rightarrow max(\delta t \cdot f_{inc}, \delta t_{max})$. If the energy of the system increases (or stays the same), the velocity of the particles is set to 0, $\alpha \rightarrow \alpha_{start}$ and $\delta t \rightarrow \delta t \cdot f_{dec}$. Convergence is determined by both the force per particle and the change in energy per particle dropping below *ftol* and *Etol*, respectively or

$$\frac{\sum |F|}{N * \sqrt{N_{dof}}} < ftol \;\; and \;\; \Delta \frac{\sum |E|}{N} < Etol$$

where N is the number of particles the minimization is acting over (i.e. the group size) Either of the two criterion can be effectively turned off by setting the tolerance to a large number.

If the minimization is acted over a subset of all the particles in the system, the "other" particles will be kept frozen but will still interact with the particles being moved.

Examples:

```
fire=integrate.mode_minimize_fire(dt=0.05, ftol=1e-2, Etol=1e-7)
nve=integrate.nve(group=group.all())
while not(fire.has_converged()):
   run(100)
```

Examples:

```
fire=integrate.mode_minimize_fire(dt=0.05, ftol=1e-2, Etol=1e-7)
nph=integrate.nph(group=group.all(),P=0.0,gamma=.5)
while not(fire.has_converged()):
   run(100)
```

Note: The algorithm requires a base integrator to update the particle position and velocities. Usually this will be either NVE (to minimize energy) or NPH (to minimize energy and relax the box). The quantity minimized is in any case the energy (not the enthalpy or any other quantity).

Note: As a default setting, the algorithm will start with a $\delta t = \frac{1}{10}\delta t_{max}$ and attempts at least 10 search steps. In practice, it was found that this prevents the simulation from making too aggressive a first step, but also from

quitting before having found a good search direction. The minimum number of attempts can be set by the user.

Attention: *mode_minimize_fire* does not function with MPI parallel simulations.

get_energy()
Returns the energy after the last iteration of the minimizer

has_converged()
Test if the energy minimizer has converged.

Returns True when the minimizer has converged. Otherwise, return False.

reset()
Reset the minimizer to its initial state.

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*aniso=None*)
Changes parameters of an existing integration mode.

Parameters **aniso** (*bool*) – Anisotropic integration mode (bool), default None (autodetect).

Examples:

```
integrator_mode.set_params(aniso=False)
```

class hoomd.md.integrate.**mode_standard**(*dt*, *aniso=None*)
Enables a variety of standard integration methods.

Parameters

- **dt** (*float*) – Each time step of the simulation *hoomd.run()* will advance the real time of the system forward by *dt* (in time units).
- **aniso** (*bool*) – Whether to integrate rotational degrees of freedom (bool), default None (autodetect).

mode_standard performs a standard time step integration technique to move the system forward. At each time step, all of the specified forces are evaluated and used in moving the system forward to the next step.

By itself, *mode_standard* does nothing. You must specify one or more integration methods to apply to the system. Each integration method can be applied to only a specific group of particles enabling advanced simulation techniques.

The following commands can be used to specify the integration methods used by integrate.mode_standard.

- *brownian*
- *langevin*
- *nve*
- *nvt*
- *npt*
- *nph*

There can only be one integration mode active at a time. If there are more than one `integrate.mode_*` commands in a hoomd script, only the most recent before a given *hoomd.run()* will take effect.

Examples:

```
integrate.mode_standard(dt=0.005)
integrator_mode = integrate.mode_standard(dt=0.001)
```

Some integration methods (notable *nvt*, *npt* and *nph* maintain state between different *hoomd.run()* commands, to allow for restartable simulations. After adding or removing particles, however, a new *hoomd.run()* will continue from the old state and the integrator variables will re-equilibrate. To ensure equilibration from a unique reference state (such as all integrator variables set to zero), the method :py:method:reset_methods() can be use to re-initialize the variables.

reset_methods()
(Re-)initialize the integrator variables in all integration methods

New in version 2.2.

Examples:

```
run(100)
# .. modify the system state, e.g. add particles ..
integrator_mode.reset_methods()
run(100)
```

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*dt=None*, *aniso=None*)
Changes parameters of an existing integration mode.

Parameters

- **dt** (*float*) – New time step delta (if set) (in time units).
- **aniso** (*bool*) – Anisotropic integration mode (bool), default None (autodetect).

Examples:

```
integrator_mode.set_params(dt=0.007)
integrator_mode.set_params(dt=0.005, aniso=False)
```

class hoomd.md.integrate.**nph**(***params*)
NPH Integration via MTK barostat-thermostat..

Parameters

- **params** – keyword arguments passed to *npt*.
- **gamma** – (float, units of energy): Damping factor for the box degrees of freedom

nph performs constant pressure (NPH) simulations using a Martyna-Tobias-Klein barostat, an explicitly reversible and measure-preserving integration scheme. It allows for fully deformable simulation cells and uses the same underlying integrator as *npt* (with *nph=True*).

The available options are identical to those of *npt*, except that *kT* cannot be specified. For further information, refer to the documentation of *npt*.

Note: A time scale *tauP* for the relaxation of the barostat is required. This is defined as the relaxation time the barostat would have at an average temperature *T_0 = 1*, and it is related to the internally used (Andersen) Barostat mass W via $W = dNT_0^2$, where d is the dimensionality and N the number of particles.

---

nph is an integration method and must be used with *mode_standard*.

Examples:

```
# Triclinic unit cell
nph=integrate.nph(group=all, P=2.0, tauP=1.0, couple="none", all=True)
# Cubic unit cell
nph = integrate.nph(group=all, P=2.0, tauP=1.0)
# Relax the box
nph = integrate.nph(group=all, P=0, tauP=1.0, gamma=0.1)
```

disable()
Disables the integration method.

Examples:

```
method.disable()
```

Executing the disable command will remove the integration method from the simulation. Any *hoomd.run()* command executed after disabling an integration method will not apply the integration method to the particles during the simulation. A disabled integration method can be re-enabled with *enable()*.

enable()
Enables the integration method.

Examples:

```
method.enable()
```

See also:

disable().

randomize_velocities(*kT*, *seed*)
Assign random velocities and angular momenta to particles in the group, sampling from the Maxwell-Boltzmann distribution. This method considers the dimensionality of the system and particle anisotropy, and removes drift (the center of mass velocity).

New in version 2.3.

Starting in version 2.5, *randomize_velocities* also chooses random values for the internal integrator variables.

Parameters

- **kT** (*float*) – Temperature (in energy units)
- **seed** (*int*) – Random number seed

Note: Randomization is applied at the start of the next call to *hoomd.run()*.

Example:

```
integrator = md.integrate.nph(group=group.all(), P=2.0, tauP=1.0)
integrator.randomize_velocities(kT=1.0, seed=42)
run(100)
```

set_params(*kT=None*, *tau=None*, *S=None*, *P=None*, *tauP=None*, *rescale_all=None*, *gamma=None*)
Changes parameters of an existing integrator.

Parameters

- **kT** (*hoomd.variant* or *float*) – New temperature (if set) (in energy units)

- **tau** (*float*) – New coupling constant (if set) (in time units)
- **S** (list of *hoomd.variant* or float) – New stress components set point (if set) for the barostat (in pressure units). In Voigt notation: [Sxx, Syy, Szz, Syz, Sxz, Sxy]
- **P** (*hoomd.variant* or float) – New isotropic pressure set point (if set) for the barostat (in pressure units). Overrides *S* if set.
- **tauP** (*float*) – New barostat coupling constant (if set) (in time units)
- **rescale_all** (*bool*) – When True, rescale all particles, not only those in the group

Examples:

```
integrator.set_params(tau=0.6)
integrator.set_params(dt=3e-3, kT=2.0, P=1.0)
```

class hoomd.md.integrate.**npt**(*group*, *kT=None*, *tau=None*, *S=None*, *P=None*, *tauP=None*, *couple='xyz'*, *x=True*, *y=True*, *z=True*, *xy=False*, *xz=False*, *yz=False*, *all=False*, *nph=False*, *rescale_all=None*, *gamma=None*)

NPT Integration via MTK barostat-thermostat.

Parameters

- **group** (*hoomd.group*) – Group of particles on which to apply this method.
- **kT** (*hoomd.variant* or float) – Temperature set point for the thermostat, not needed if *nph=True* (in energy units).
- **tau** (*float*) – Coupling constant for the thermostat, not needed if *nph=True* (in time units).
- **S** (list of *hoomd.variant* or float) – Stress components set point for the barostat (in pressure units). In Voigt notation: [Sxx, Syy, Szz, Syz, Sxz, Sxy]
- **P** (*hoomd.variant* or float) – Isotropic pressure set point for the barostat (in pressure units). Overrides *S* if set.
- **tauP** (*float*) – Coupling constant for the barostat (in time units).
- **couple** (*str*) – Couplings of diagonal elements of the stress tensor, can be "none", "xy", "xz","yz", or "xyz" (default).
- **x** (*bool*) – if True, rescale *Lx* and x component of particle coordinates and velocities
- **y** (*bool*) – if True, rescale *Ly* and y component of particle coordinates and velocities
- **z** (*bool*) – if True, rescale *Lz* and z component of particle coordinates and velocities
- **xy** (*bool*) – if True, rescale xy tilt factor and x and y components of particle coordinates and velocities
- **xz** (*bool*) – if True, rescale xz tilt factor and x and z components of particle coordinates and velocities
- **yz** (*bool*) – if True, rescale yz tilt factor and y and z components of particle coordinates and velocities
- **all** (*bool*) – if True, rescale all lengths and tilt factors and components of particle coordinates and velocities
- **nph** (*bool*) – if True, integrate without a thermostat, i.e. in the NPH ensemble
- **rescale_all** (*bool*) – if True, rescale all particles, not only those in the group

- **gamma** – (float): Dimensionless damping factor for the box degrees of freedom (default: 0)

npt performs constant pressure, constant temperature simulations, allowing for a fully deformable simulation box.

The integration method is based on the rigorous Martyna-Tobias-Klein equations of motion for NPT. For optimal stability, the update equations leave the phase-space measure invariant and are manifestly time-reversible.

By default, *npt* performs integration in a cubic box under hydrostatic pressure by simultaneously rescaling the lengths *Lx*, *Ly* and *Lz* of the simulation box.

npt can also perform more advanced integration modes. The integration mode is specified by a set of couplings and by specifying the box degrees of freedom that are put under barostat control.

Couplings define which diagonal elements of the pressure tensor $P_{\alpha,\beta}$ should be averaged over, so that the corresponding box lengths are rescaled by the same amount.

Valid couplings are:

- none (all box lengths are updated independently)
- xy (*Lx* and *Ly* are coupled)
- xz (*Lx* and *Lz* are coupled)
- yz (*Ly* and *Lz* are coupled)
- xyz (*Lx* and *Ly* and *Lz* are coupled)

The default coupling is **xyz**, i.e. the ratios between all box lengths stay constant.

Degrees of freedom of the box specify which lengths and tilt factors of the box should be updated, and how particle coordinates and velocities should be rescaled.

Valid keywords for degrees of freedom are:

- x (the box length Lx is updated)
- y (the box length Ly is updated)
- z (the box length Lz is updated)
- xy (the tilt factor xy is updated)
- xz (the tilt factor xz is updated)
- yz (the tilt factor yz is updated)
- all (all elements are updated, equivalent to x, y, z, xy, xz, and yz together)

Any of the six keywords can be combined together. By default, the x, y, and z degrees of freedom are updated.

Note: If any of the diagonal x, y, z degrees of freedom is not being integrated, pressure tensor components along that direction are not considered for the remaining degrees of freedom.

For example:

- Specifying xyz couplings and x, y, and z degrees of freedom amounts to cubic symmetry (default)
- Specifying xy couplings and x, y, and z degrees of freedom amounts to tetragonal symmetry.
- Specifying no couplings and all degrees of freedom amounts to a fully deformable triclinic unit cell

npt Can also apply a constant stress to the simulation box. To do so, specify the symmetric stress tensor S instead of an isotropic pressure P.

Note: *npt* assumes that isotropic pressures are positive. Conventions for the stress tensor sometimes assume negative values on the diagonal. You need to set these values negative manually in HOOMD.

---

npt is an integration method. It must be used with *mode_standard*.

npt uses the proper number of degrees of freedom to compute the temperature and pressure of the system in both 2 and 3 dimensional systems, as long as the number of dimensions is set before the *npt* command is specified.

For the MTK equations of motion, see:

- G. J. Martyna, D. J. Tobias, M. L. Klein 1994
- M. E. Tuckerman et. al. 2006
- T. Yu et. al. 2010
- Glaser et. al (2013), to be published

Both kT and P can be variant types, allowing for temperature/pressure ramps in simulation runs.

τ is related to the Nosé mass Q by

$$\tau = \sqrt{\frac{Q}{gk_BT_0}}$$

where g is the number of degrees of freedom, and k_BT_0 is the set point (kT above).

A *hoomd.compute.thermo* is automatically specified and associated with *group*.

Examples:

```
integrate.npt(group=all, kT=1.0, tau=0.5, tauP=1.0, P=2.0)
integrator = integrate.npt(group=all, tau=1.0, kT=0.65, tauP = 1.2, P=2.0)
# orthorhombic symmetry
integrator = integrate.npt(group=all, tau=1.0, kT=0.65, tauP = 1.2, P=2.0, couple=
↪"none")
# tetragonal symmetry
integrator = integrate.npt(group=all, tau=1.0, kT=0.65, tauP = 1.2, P=2.0, couple=
↪"xy")
# triclinic symmetry
integrator = integrate.npt(group=all, tau=1.0, kT=0.65, tauP = 1.2, P=2.0, couple=
↪"none", rescale_all=True)
```

disable()
Disables the integration method.

Examples:

```
method.disable()
```

Executing the disable command will remove the integration method from the simulation. Any *hoomd.run()* command executed after disabling an integration method will not apply the integration method to the particles during the simulation. A disabled integration method can be re-enabled with *enable()*.

enable()
Enables the integration method.

Examples:

```
method.enable()
```

See also:

disable().

randomize_velocities(*seed*)

Assign random velocities and angular momenta to particles in the group, sampling from the Maxwell-Boltzmann distribution. This method considers the dimensionality of the system and particle anisotropy, and removes drift (the center of mass velocity).

New in version 2.3.

Starting in version 2.5, *randomize_velocities* also chooses random values for the internal integrator variables.

Parameters **seed** (*int*) – Random number seed

Note: Randomization is applied at the start of the next call to *hoomd.run()*.

Example:

```
integrator = md.integrate.npt(group=group.all(), kT=1.0, tau=0.5, tauP=1.0,␣
↪P=2.0)
integrator.randomize_velocities(seed=42)
run(100)
```

set_params(*kT=None*, *tau=None*, *S=None*, *P=None*, *tauP=None*, *rescale_all=None*, *gamma=None*)

Changes parameters of an existing integrator.

Parameters

- **kT** (*hoomd.variant* or *float*) – New temperature (if set) (in energy units)
- **tau** (*float*) – New coupling constant (if set) (in time units)
- **S** (*list* of *hoomd.variant* or *float*) – New stress components set point (if set) for the barostat (in pressure units). In Voigt notation: [Sxx, Syy, Szz, Syz, Sxz, Sxy]
- **P** (*hoomd.variant* or *float*) – New isotropic pressure set point (if set) for the barostat (in pressure units). Overrides *S* if set.
- **tauP** (*float*) – New barostat coupling constant (if set) (in time units)
- **rescale_all** (*bool*) – When True, rescale all particles, not only those in the group

Examples:

```
integrator.set_params(tau=0.6)
integrator.set_params(dt=3e-3, kT=2.0, P=1.0)
```

class hoomd.md.integrate.**nve**(*group*, *limit=None*, *zero_force=False*)

NVE Integration via Velocity-Verlet

Parameters

- **group** (*hoomd.group*) – Group of particles on which to apply this method.
- **limit** (*bool*) – (optional) Enforce that no particle moves more than a distance of a limit in a single time step

- **zero_force** (*bool*) – When set to true, particles in the a group are integrated forward in time with constant velocity and any net force on them is ignored.

nve performs constant volume, constant energy simulations using the standard Velocity-Verlet method. For poor initial conditions that include overlapping atoms, a limit can be specified to the movement a particle is allowed to make in one time step. After a few thousand time steps with the limit set, the system should be in a safe state to continue with unconstrained integration.

Another use-case for *nve* is to fix the velocity of a certain group of particles. This can be achieved by setting the velocity of those particles in the initial condition and setting the *zero_force* option to True for that group. A True value for *zero_force* causes integrate.nve to ignore any net force on each particle and integrate them forward in time with a constant velocity.

Note: With an active limit, Newton's third law is effectively **not** obeyed and the system can gain linear momentum. Activate the *hoomd.md.update.zero_momentum* updater during the limited nve run to prevent this.

---

nve is an integration method. It must be used with *mode_standard*.

A *hoomd.compute.thermo* is automatically specified and associated with *group*.

Examples:

```
all = group.all()
integrate.nve(group=all)
integrator = integrate.nve(group=all)
typeA = group.type('A')
integrate.nve(group=typeA, limit=0.01)
integrate.nve(group=typeA, zero_force=True)
```

disable()

Disables the integration method.

Examples:

```
method.disable()
```

Executing the disable command will remove the integration method from the simulation. Any *hoomd.run()* command executed after disabling an integration method will not apply the integration method to the particles during the simulation. A disabled integration method can be re-enabled with *enable()*.

enable()

Enables the integration method.

Examples:

```
method.enable()
```

See also:

disable().

randomize_velocities(*kT*, *seed*)

Assign random velocities and angular momenta to particles in the group, sampling from the Maxwell-Boltzmann distribution. This method considers the dimensionality of the system and particle anisotropy, and removes drift (the center of mass velocity).

New in version 2.3.

Parameters

- **kT** (*float*) – Temperature (in energy units)
- **seed** (*int*) – Random number seed

Note: Randomization is applied at the start of the next call to *hoomd.run()*.

Example:

```
integrator = md.integrate.nve(group=group.all())
integrator.randomize_velocities(kT=1.0, seed=42)
run(100)
```

set_params(*limit=None*, *zero_force=None*)
Changes parameters of an existing integrator.

Parameters

- **limit** (*bool*) – (if set) New limit value to set. Removes the limit if limit is False
- **zero_force** (*bool*) – (if set) New value for the zero force option

Examples:

```
integrator.set_params(limit=0.01)
integrator.set_params(limit=False)
```

class hoomd.md.integrate.**nvt**(*group*, *kT*, *tau*)
NVT Integration via the Nosé-Hoover thermostat.

Parameters

- **group** (*hoomd.group*) – Group of particles on which to apply this method.
- **kT** (*hoomd.variant* or *float*) – Temperature set point for the Nosé-Hoover thermostat. (in energy units).
- **tau** (*float*) – Coupling constant for the Nosé-Hoover thermostat. (in time units).

nvt performs constant volume, constant temperature simulations using the Nosé-Hoover thermostat, using the MTK equations described in Refs. G. J. Martyna, D. J. Tobias, M. L. Klein 1994 and J. Cao, G. J. Martyna 1996.

nvt is an integration method. It must be used in connection with *mode_standard*.

nvt uses the proper number of degrees of freedom to compute the temperature of the system in both 2 and 3 dimensional systems, as long as the number of dimensions is set before the integrate.nvt command is specified.

τ is related to the Nosé mass Q by

$$\tau = \sqrt{\frac{Q}{g k_B T_0}}$$

where g is the number of degrees of freedom, and $k_B T_0$ is the set point (kT above).

kT can be a variant type, allowing for temperature ramps in simulation runs.

A *hoomd.compute.thermo* is automatically specified and associated with *group*.

Examples:

```
all = group.all()
integrate.nvt(group=all, kT=1.0, tau=0.5)
integrator = integrate.nvt(group=all, tau=1.0, kT=0.65)
typeA = group.type('A')
integrator = integrate.nvt(group=typeA, tau=1.0, kT=hoomd.variant.linear_
↪interp([(0, 4.0), (1e6, 1.0)]))
```

disable()
Disables the integration method.

Examples:

```
method.disable()
```

Executing the disable command will remove the integration method from the simulation. Any *hoomd.run()* command executed after disabling an integration method will not apply the integration method to the particles during the simulation. A disabled integration method can be re-enabled with *enable()*.

enable()
Enables the integration method.

Examples:

```
method.enable()
```

See also:

disable().

randomize_velocities(*seed*)
Assign random velocities and angular momenta to particles in the group, sampling from the Maxwell-Boltzmann distribution. This method considers the dimensionality of the system and particle anisotropy, and removes drift (the center of mass velocity).

New in version 2.3.

Starting in version 2.5, *randomize_velocities* also chooses random values for the internal integrator variables.

Parameters **seed** (*int*) – Random number seed

Note: Randomization is applied at the start of the next call to *hoomd.run()*.

Example:

```
integrator = md.integrate.nvt(group=group.all(), kT=1.0, tau=0.5)
integrator.randomize_velocities(seed=42)
run(100)
```

set_params(*kT=None*, *tau=None*)
Changes parameters of an existing integrator.

Parameters

- **kT** (*float*) – New temperature (if set) (in energy units)
- **tau** (*float*) – New coupling constant (if set) (in time units)

Examples:

```
integrator.set_params(tau=0.6)
integrator.set_params(tau=0.7, kT=2.0)
```

14.10 md.nlist

Overview

md.nlist.cell	Cell list based neighbor list
md.nlist.stencil	Cell list based neighbor list using stencils
md.nlist.tree	Fast neighbor list for size asymmetric particles.

Details

Neighbor list acceleration structures.

Neighbor lists accelerate pair force calculation by maintaining a list of particles within a cutoff radius. Multiple pair forces can utilize the same neighbor list. Neighbors are included using a pairwise cutoff $r_{\mathrm{cut}}(i,j)$ that is the maximum of all $r_{\mathrm{cut}}(i,j)$ set for the pair forces attached to the list.

Multiple neighbor lists can be created to accelerate simulations where there is significant disparity in $r_{\mathrm{cut}}(i,j)$ between pair potentials. If one pair force has a cutoff radius much smaller than another pair force, the pair force calculation for the short cutoff will be slowed down considerably because many particles in the neighbor list will have to be read and skipped because they lie outside the shorter cutoff.

The simplest way to build a neighbor list is $O(N^2)$: each particle loops over all other particles and only includes those within the neighbor list cutoff. This algorithm is no longer implemented in HOOMD-blue because it is slow and inefficient. Instead, three accelerated algorithms based on cell lists and bounding volume hierarchy trees are implemented. The cell list implementation is fastest when the cutoff radius is similar between all pair forces (smaller than 2:1 ratio). The stencil implementation is a different variant of the cell list, and is usually fastest when there is large disparity in the pair cutoff radius and a high number fraction of particles with the bigger cutoff (at least 30%). The tree implementation is faster when there is large size disparity and the number fraction of big objects is low. Because the performance of these algorithms depends sensitively on your system and hardware, you should carefully test which option is fastest for your simulation.

Particles can be excluded from the neighbor list based on certain criteria. Setting $r_{\mathrm{cut}}(i,j) \leq 0$ will exclude this cross interaction from the neighbor list on build time. Particles can also be excluded by topology or for belonging to the same rigid body (see *nlist.reset_exclusions()*).

Examples:

```
nl_c = nlist.cell(check_period=1)
nl_t = nlist.tree(r_buff = 0.8)
lj1 = pair.lj(r_cut = 3.0, nlist=nl_c)
lj2 = pair.lj(r_cut = 10.0, nlist=nl_t)
```

class hoomd.md.nlist.**cell**(*r_buff=0.4*, *check_period=1*, *d_max=None*, *dist_check=True*, *name=None*, *deterministic=False*)

Cell list based neighbor list

Parameters

- **r_buff** (*float*) – Buffer width.
- **check_period** (*int*) – How often to attempt to rebuild the neighbor list.

- **d_max** (*float*) – The maximum diameter a particle will achieve, only used in conjunction with slj diameter shifting.
- **dist_check** (*bool*) – Flag to enable / disable distance checking.
- **name** (*str*) – Optional name for this neighbor list instance.
- **deterministic** (*bool*) – When True, enable deterministic runs on the GPU by sorting the cell list.

cell creates a cell list based neighbor list object to which pair potentials can be attached for computing non-bonded pairwise interactions. Cell listing allows for *O(N)* construction of the neighbor list. Particles are first spatially sorted into cells based on the largest pairwise cutoff radius attached to this instance of the neighbor list. Particles then query their adjacent cells, and neighbors are included based on pairwise cutoffs. This method is very efficient for systems with nearly monodisperse cutoffs, but performance degrades for large cutoff radius asymmetries due to the significantly increased number of particles per cell. Users can create multiple neighbor lists, and may see significant performance increases by doing so for systems with size asymmetry, especially when used in conjunction with *tree*.

Use base class methods to change parameters (*set_params*), reset the exclusion list (*reset_exclusions*) or tune *r_buff* (*tune*).

Examples:

```
nl_c = nlist.cell(check_period = 1)
nl_c.set_params(r_buff=0.5)
nl_c.reset_exclusions([]);
nl_c.tune()
```

Note: *d_max* should only be set when slj diameter shifting is required by a pair potential. Currently, slj is the only pair potential requiring this shifting, and setting *d_max* for other potentials may lead to significantly degraded performance or incorrect results.

class hoomd.md.nlist.**nlist**

Base class neighbor list.

Methods provided by this base class are available to all subclasses.

query_update_period()

Query the maximum possible check_period.

query_update_period() examines the counts of nlist rebuilds during the previous *hoomd.run()*. It returns `s-1`, where *s* is the smallest update period experienced during that time. Use it after a medium-length warm up run with *check_period=1* to determine what check_period to set for production runs.

Warning: If the previous *hoomd.run()* was short, insufficient sampling may cause the queried update period to be large enough to result in dangerous builds during longer runs. Unless you use a really long warm up run, subtract an additional 1 from this when you set check_period for additional safety.

reset_exclusions(*exclusions=None*)

Resets all exclusions in the neighborlist.

Parameters **exclusions** (*list*) – Select which interactions should be excluded from the pair interaction calculation.

By default, the following are excluded from short range pair interactions"

- Directly bonded particles.
- Directly constrained particles.
- Particles that are in the same rigid body.

reset_exclusions allows the defaults to be overridden to add other exclusions or to remove the exclusion for bonded or constrained particles.

Specify a list of desired types in the *exclusions* argument (or an empty list to clear all exclusions). All desired exclusions have to be explicitly listed, i.e. '1-3' does **not** imply '1-2'.

Valid types are:

- **bond** - Exclude particles that are directly bonded together.
- **constraint** - Exclude particles that are directly constrained.
- **angle** - Exclude the two outside particles in all defined angles.
- **dihedral** - Exclude the two outside particles in all defined dihedrals.
- **pair** - Exclude particles in all defined special pairs.
- **body** - Exclude particles that belong to the same body.

The following types are determined solely by the bond topology. Every chain of particles in the simulation connected by bonds (1-2-3-4) will be subject to the following exclusions, if enabled, whether or not explicit angles or dihedrals are defined:

- **1-2** - Same as bond
- **1-3** - Exclude particles connected with a sequence of two bonds.
- **1-4** - Exclude particles connected with a sequence of three bonds.

Examples:

```
nl.reset_exclusions(exclusions = ['1-2'])
nl.reset_exclusions(exclusions = ['1-2', '1-3', '1-4'])
nl.reset_exclusions(exclusions = ['bond', 'angle'])
nl.reset_exclusions(exclusions = ['bond', 'angle','constraint'])
nl.reset_exclusions(exclusions = [])
```

set_params(*r_buff=None*, *check_period=None*, *d_max=None*, *dist_check=True*)

Change neighbor list parameters.

Parameters

- **r_buff** (*float*) – (if set) changes the buffer radius around the cutoff (in distance units)
- **check_period** (*int*) – (if set) changes the period (in time steps) between checks to see if the neighbor list needs updating
- **d_max** (*float*) – (if set) notifies the neighbor list of the maximum diameter that a particle attain over the following run() commands. (in distance units)
- **dist_check** (*bool*) – When set to False, disable the distance checking logic and always regenerate the nlist every *check_period* steps

set_params() changes one or more parameters of the neighbor list. *r_buff* and *check_period* can have a significant effect on performance. As *r_buff* is made larger, the neighbor list needs to be updated less often, but more particles are included leading to slower force computations. Smaller values of *r_buff* lead to faster force computation, but more often neighbor list updates, slowing overall performance again. The sweet spot for the best performance needs to be found by experimentation. The default of *r_buff = 0.4* works well in practice for Lennard-Jones liquid simulations.

As *r_buff* is changed, *check_period* must be changed correspondingly. The neighbor list is updated no sooner than *check_period* time steps after the last update. If *check_period* is set too high, the neighbor list may not be updated when it needs to be.

For safety, the default check_period is 1 to ensure that the neighbor list is always updated when it needs to be. Increasing this to an appropriate value for your simulation can lead to performance gains of approximately 2 percent.

check_period should be set so that no particle moves a distance more than *r_buff/2.0* during a the *check_period*. If this occurs, a *dangerous build* is counted and printed in the neighbor list statistics at the end of a `hoomd.run()`.

When using `hoomd.md.pair.slj`, *d_max* **MUST** be set to the maximum diameter that a particle will attain at any point during the following `hoomd.run()` commands (see `hoomd.md.pair.slj` for more information). When using in conjunction, `hoomd.md.pair.slj` will automatically set *d_max* for the nlist. This can be overridden (e.g. if multiple potentials using diameters are used) by using `set_params()` after the `hoomd.md.pair.slj` class has been initialized.

Caution: When **not** using `hoomd.md.pair.slj`, *d_max* **MUST** be left at the default value of 1.0 or the simulation will be incorrect if d_max is less than 1.0 and slower than necessary if d_max is greater than 1.0.

Examples:

```
nl.set_params(r_buff = 0.9)
nl.set_params(check_period = 11)
nl.set_params(r_buff = 0.7, check_period = 4)
nl.set_params(d_max = 3.0)
```

tune(*warmup=200000*, *r_min=0.05*, *r_max=1.0*, *jumps=20*, *steps=5000*, *set_max_check_period=False*, *quiet=False*)

Make a series of short runs to determine the fastest performing r_buff setting.

Parameters

- **warmup** (*int*) – Number of time steps to run() to warm up the benchmark
- **r_min** (*float*) – Smallest value of r_buff to test
- **r_max** (*float*) – Largest value of r_buff to test
- **jumps** (*int*) – Number of different r_buff values to test
- **steps** (*int*) – Number of time steps to run() at each point
- **set_max_check_period** (*bool*) – Set to True to enable automatic setting of the maximum nlist check_period
- **quiet** (*bool*) – Quiet the individual run() calls.

`tune()` executes *warmup* time steps. Then it sets the nlist *r_buff* value to *r_min* and runs for *steps* time steps. The TPS value is recorded, and the benchmark moves on to the next *r_buff* value completing at *r_max* in *jumps* jumps. Status information is printed out to the screen, and the optimal *r_buff* value is left set for further `hoomd.run()` calls to continue at optimal settings.

Each benchmark is repeated 3 times and the median value chosen. Then, *warmup* time steps are run again at the optimal *r_buff* in order to determine the maximum value of check_period. In total, `(2*warmup + 3*jump*steps)` time steps are run.

Note: By default, the maximum check_period is **not** set for safety. If you wish to have it set when the call completes, call with the parameter *set_max_check_period=True*.

Returns (optimal_r_buff, maximum check_period)

class hoomd.md.nlist.**stencil**(*r_buff=0.4*, *check_period=1*, *d_max=None*, *dist_check=True*, *cell_width=None*, *name=None*, *deterministic=False*)

Cell list based neighbor list using stencils

Parameters

- **r_buff** (*float*) – Buffer width.
- **check_period** (*int*) – How often to attempt to rebuild the neighbor list.
- **d_max** (*float*) – The maximum diameter a particle will achieve, only used in conjunction with slj diameter shifting.
- **dist_check** (*bool*) – Flag to enable / disable distance checking.
- **cell_width** (*float*) – The underlying stencil bin width for the cell list
- **name** (*str*) – Optional name for this neighbor list instance.
- **deterministic** (*bool*) – When True, enable deterministic runs on the GPU by sorting the cell list.

stencil creates a cell list based neighbor list object to which pair potentials can be attached for computing non-bonded pairwise interactions. Cell listing allows for O(N) construction of the neighbor list. Particles are first spatially sorted into cells based on the largest pairwise cutoff radius attached to this instance of the neighbor list.

M.P. Howard et al. 2016 describes this neighbor list implementation in HOOMD-blue. Cite it if you utilize this neighbor list style in your work.

This neighbor-list style differs from *cell* based on how the adjacent cells are searched for particles. The cell list *cell_width* is set by default using the shortest active cutoff radius in the system. One *stencil* is computed per particle type based on the largest cutoff radius that type participates in, which defines the bins that the particle must search in. Distances to the bins in the stencil are precomputed so that certain particles can be quickly excluded from the neighbor list, leading to improved performance compared to *cell* when there is size disparity in the cutoff radius.

The performance of the stencil depends strongly on the choice of *cell_width*. The best performance is obtained when the cutoff radii are multiples of the *cell_width*, and when the *cell_width* covers the simulation box with a roughly integer number of cells. The *cell_width* can be set manually, or be automatically scanning through a range of possible bin widths using *tune_cell_width()*.

Examples:

```
nl_s = nlist.stencil(check_period = 1)
nl_s.set_params(r_buff=0.5)
nl_s.reset_exclusions([]);
nl_s.tune()
nl_s.tune_cell_width(min_width=1.5, max_width=3.0)
```

Note: *d_max* should only be set when slj diameter shifting is required by a pair potential. Currently, slj is the only pair potential requiring this shifting, and setting *d_max* for other potentials may lead to significantly

degraded performance or incorrect results.

---

set_cell_width(*cell_width*)
Set the cell width

Parameters **cell_width** (*float*) – New cell width.

tune_cell_width(*warmup=200000*, *min_width=None*, *max_width=None*, *jumps=20*, *steps=5000*)
Make a series of short runs to determine the fastest performing bin width.

Parameters

- **warmup** (*int*) – Number of time steps to run() to warm up the benchmark
- **min_width** (*float*) – Minimum cell bin width to try
- **max_width** (*float*) – Maximum cell bin width to try
- **jumps** (*int*) – Number of different bin width to test
- **steps** (*int*) – Number of time steps to run() at each point

tune_cell_width() executes *warmup* time steps. Then it sets the nlist *cell_width* value to *min_width* and runs for *steps* time steps. The TPS value is recorded, and the benchmark moves on to the next *cell_width* value completing at *max_width* in *jumps* jumps. Status information is printed out to the screen, and the optimal *cell_width* value is left set for further runs() to continue at optimal settings.

Each benchmark is repeated 3 times and the median value chosen. In total, `(warmup + 3*jump*steps)` time steps are run.

Returns The optimal cell width.

class hoomd.md.nlist.**tree**(*r_buff=0.4*, *check_period=1*, *d_max=None*, *dist_check=True*, *name=None*)
Fast neighbor list for size asymmetric particles.

Parameters

- **r_buff** (*float*) – Buffer width.
- **check_period** (*int*) – How often to attempt to rebuild the neighbor list.
- **d_max** (*float*) – The maximum diameter a particle will achieve, only used in conjunction with slj diameter shifting.
- **dist_check** (*bool*) – Flag to enable / disable distance checking.
- **name** (*str*) – Optional name for this neighbor list instance.

tree creates a neighbor list using bounding volume hierarchy (BVH) tree traversal. Pair potentials are attached for computing non-bonded pairwise interactions. A BVH tree of axis-aligned bounding boxes is constructed per particle type, and each particle queries each tree to determine its neighbors. This method of searching leads to significantly improved performance compared to cell lists in systems with moderate size asymmetry, but has poorer performance for monodisperse systems. The user should carefully benchmark neighbor list build times to select the appropriate neighbor list construction type.

M.P. Howard et al. 2016 describes this neighbor list implementation in HOOMD-blue. Cite it if you utilize this neighbor list style in your work.

Users can create multiple neighbor lists, and may see significant performance increases by doing so for systems with size asymmetry, especially when used in conjunction with nlist.cell.

Examples:

```
nl_t = nlist.tree(check_period = 1)
nl_t.set_params(r_buff=0.5)
nl_t.reset_exclusions([]);
nl_t.tune()
```

Note: *d_max* should only be set when slj diameter shifting is required by a pair potential. Currently, slj is the only pair potential requiring this shifting, and setting *d_max* for other potentials may lead to significantly degraded performance or incorrect results.

Attention: BVH tree neighbor lists are currently only supported on Kepler (sm_30) architecture devices and newer.

14.11 md.pair

Overview

md.pair.buckingham	Buckingham pair potential.
md.pair.dipole	Screened dipole-dipole interactions.
md.pair.dpd	Dissipative Particle Dynamics.
md.pair.dpdlj	Dissipative Particle Dynamics with a LJ conservative force
md.pair.dpd_conservative	DPD Conservative pair force.
md.pair.ewald	Ewald pair potential.
md.pair.force_shifted_lj	Force-shifted Lennard-Jones pair potential.
md.pair.gauss	Gaussian pair potential.
md.pair.gb	Gay-Berne anisotropic pair potential.
md.pair.lj	Lennard-Jones pair potential.
md.pair.lj1208	Lennard-Jones 12-8 pair potential.
md.pair.mie	Mie pair potential.
md.pair.morse	Morse pair potential.
md.pair.moliere	Moliere pair potential.
md.pair.pair	Common pair potential documentation.
md.pair.reaction_field	Onsager reaction field pair potential.
md.pair.slj	Shifted Lennard-Jones pair potential.
md.pair.square_density	Soft potential for simulating a van-der-Waals liquid
md.pair.table	Tabulated pair potential.
md.pair.tersoff	Tersoff Potential.
md.pair.yukawa	Yukawa pair potential.
md.pair.zbl	ZBL pair potential.

Details

Pair potentials.

Generally, pair forces are short range and are summed over all non-bonded particles within a certain cutoff radius of each particle. Any number of pair forces can be defined in a single simulation. The net force on each particle due to all types of pair forces is summed.

Pair forces require that parameters be set for each unique type pair. Coefficients are set through the aid of the *coeff* class. To set these coefficients, specify a pair force and save it in a variable:

```
my_force = pair.some_pair_force(arguments...)
```

Then the coefficients can be set using the saved variable:

```
my_force.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
my_force.pair_coeff.set('A', 'B', epsilon=1.0, sigma=2.0)
my_force.pair_coeff.set('B', 'B', epsilon=2.0, sigma=1.0)
```

This example set the parameters *epsilon* and *sigma* (which are used in *lj*). Different pair forces require that different coefficients are set. Check the documentation of each to see the definition of the coefficients.

class hoomd.md.pair.**DLVO**(*r_cut*, *nlist*, *d_max=None*, *name=None*)

DLVO colloidal interaction

DLVO specifies that a DLVO dispersion and electrostatic interaction should be applied between every non-excluded particle pair in the simulation.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.
- **d_max** (*float*) – Maximum diameter particles in the simulation will have (in distance units)

DLVO evaluates the forces for the pair potential .. math:

```
   V_{\mathrm{DLVO}}(r)  = & - \frac{A}{6} \left[
       \frac{2a_1a_2}{r^2 - (a_1+a_2)^2} + \frac{2a_1a_2}{r^2 - (a_1-a_2)^2}
       + \log \left( \frac{r^2 - (a_1+a_2)^2}{r^2 - (a_1+a_2)^2} \right) \right]
       + \frac{a_1 a_2}{a_1+a_2} Z e^{-\kappa(r - (a_1+a_2))} & r < (r_{\mathrm
↪{cut}} + \Delta)
       = & 0 & r \ge (r_{\mathrm{cut}} + \Delta)

where math:`a_i` is the radius of particle :math:`i`, :math:`\Delta = (d_i + d_j)/
↪2` and
:math:`d_i` is the diameter of particle :math:`i`.
```

The first term corresponds to the attractive van der Waals interaction with A being the Hamaker constant, the second term to the repulsive double-layer interaction between two spherical surfaces with Z proportional to the surface electric potential.

See Israelachvili 2011, pp. 317.

The DLVO potential does not need charge, but does need diameter. See *slj* for an explanation on how diameters are handled in the neighbor lists.

Due to the way that DLVO modifies the cutoff condition, it will not function properly with the xplor shifting mode. See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes.

Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in units of energy*distance)

- κ - *kappa* (in units of 1/distance)
- r_{cut} - *r_cut* (in units of distance) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on} - *r_on* (in units of distance) - *optional*: defaults to the global r_cut specified in the pair command

New in version 2.2.

Example:

```
nl = nlist.cell()
DLVO.pair_coeff.set('A', 'A', epsilon=1.0, kappa=1.0)
DLVO.pair_coeff.set('A', 'B', epsilon=2.0, kappa=0.5, r_cut=3.0, r_on=2.0);
DLVO.pair_coeff.set(['A', 'B'], ['C', 'D'], epsilon=0.5, kappa=3.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**ai_pair**(*r_cut*, *nlist*, *name=None*)
Generic anisotropic pair potential.

Users should not instantiate *ai_pair* directly. It is a base class that provides common features to all anisotropic pair forces. Rather than repeating all of that documentation in a dozen different places, it is collected here.

All anisotropic pair potential commands specify that a given potential energy, force and torque be computed on all non-excluded particle pairs in the system within a short range cutoff distance r_{cut}. The interaction energy, forces and torque depend on the inter-particle separation $\vec{r}$ and on the orientations $\vec{q}_i$, q_j, of the particles.

compute_energy(*tags1*, *tags2*)
Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \text{tags1}, j \in \text{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff

valid values for mode are: "none" (the default) and "shift":

- *none* - No shifting is performed and potentials are abruptly cut off
- *shift* - A constant shift is applied to the entire potential so that it is 0 at the cutoff

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
```

class hoomd.md.pair.**buckingham**(*r_cut*, *nlist*, *name=None*)
Buckingham pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

buckingham specifies that a Buckingham pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{Buckingham}}(r) = & \quad A \exp\left(-\frac{r}{\rho}\right) - \frac{C}{r} & r < r_{\mathrm{cut}} \\ = & \quad 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- A - *A* (in energy units)
- ρ - *rho* (in distance units)
- - *C* (in energy/distance units)

- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

New in version 2.2.

Changed in version 2.2.

Example:

```
nl = nlist.cell()
buck = pair.buckingham(r_cut=3.0, nlist=nl)
buck.pair_coeff.set('A', 'A', A=1.0, rho=1.0, C=1.0)
buck.pair_coeff.set('A', 'B', A=2.0, rho=1.0, C=1.0, r_cut=3.0, r_on=2.0);
buck.pair_coeff.set('B', 'B', A=1.0, rho=1.0, C=1.0, r_cut=2**(1.0/6.0), r_on=2.
↪0);
buck.pair_coeff.set(['A', 'B'], ['C', 'D'], A=1.5, rho=2.0, C=1.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log*=*True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**coeff**
Define pair coefficients

All pair forces use *coeff* to specify the coefficients between different pairs of particles indexed by type. The set of pair coefficients is a symmetric matrix defined over all possible pairs of particle types.

There are two ways to set the coefficients for a particular pair force. The first way is to save the pair force in a variable and call *set()* directly.

The second method is to build the *coeff* class first and then assign it to the pair force. There are some advantages to this method in that you could specify a complicated set of pair coefficients in a separate python file and import it into your job script.

Example (**force_field.py**):

```
from hoomd import md
my_coeffs = md.pair.coeff();
my_force.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
my_force.pair_coeff.set('A', 'B', epsilon=1.0, sigma=2.0)
my_force.pair_coeff.set('B', 'B', epsilon=2.0, sigma=1.0)
```

Example job script:

```
from hoomd import md
import force_field

.....
my_force = md.pair.some_pair_force(arguments...)
my_force.pair_coeff = force_field.my_coeffs
```

set (*a*, *b*, ***coeffs*)

Sets parameters for one type pair.

Parameters

- **a** (*str*) – First particle type in the pair (or a list of type names)
- **b** (*str*) – Second particle type in the pair (or a list of type names)
- **coeffs** – Named coefficients (see below for examples)

Calling *set()* results in one or more parameters being set for a single type pair or set of type pairs. Particle types are identified by name, and parameters are also added by name. Which parameters you need to specify depends on the pair force you are setting these coefficients for, see the corresponding documentation.

All possible type pairs as defined in the simulation box must be specified before executing *hoomd.run()*. You will receive an error if you fail to do so. It is not an error, however, to specify coefficients for particle types that do not exist in the simulation. This can be useful in defining a force field for many different types of particles even when some simulations only include a subset.

There is no need to specify coefficients for both pairs 'A', 'B' and 'B', 'A'. Specifying only one is sufficient.

To set the same coefficients between many particle types, provide a list of type names instead of a single one. All pairs between the two lists will be set to the same parameters.

Examples:

```
coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
coeff.set('B', 'B', epsilon=2.0, sigma=1.0)
coeff.set('A', 'B', epsilon=1.5, sigma=1.0)
coeff.set(['A', 'B', 'C', 'D'], 'F', epsilon=2.0)
coeff.set(['A', 'B', 'C', 'D'], ['A', 'B', 'C', 'D'], epsilon=1.0)
```

(continues on next page)

(continued from previous page)

```
system = init.read_xml('init.xml')
coeff.set(system.particles.types, system.particles.types, epsilon=2.0)
coeff.set('A', system.particles.types, epsilon=1.2)
```

Note: Single parameters can be updated. If both epsilon and sigma have already been set for a type pair, then executing `coeff.set('A', 'B', epsilon=1.1)` will update the value of epsilon and leave sigma as it was previously set.

Some pair potentials assign default values to certain parameters. If the default setting for a given coefficient (as documented in the respective pair command) is not set explicitly, the default will be used.

class hoomd.md.pair.**dipole**(*r_cut*, *nlist*, *name=None*)

Screened dipole-dipole interactions.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

dipole computes the (screened) interaction between pairs of particles with dipoles and electrostatic charges. The total energy computed is:

$$U_{dipole} = U_{dd} + U_{de} + U_{ee}$$

$$U_{dd} = Ae^{-\kappa r}\left(\frac{\vec{\mu_i}\cdot\vec{\mu_j}}{r^3} - 3\frac{(\vec{\mu_i}\cdot\vec{r_{ji}})(\vec{\mu_j}\cdot\vec{r_{ji}})}{r^5}\right)$$

$$U_{de} = Ae^{-\kappa r}\left(\frac{(\vec{\mu_j}\cdot\vec{r_{ji}})q_i}{r^3} - \frac{(\vec{\mu_i}\cdot\vec{r_{ji}})q_j}{r^3}\right)$$

$$U_{ee} = Ae^{-\kappa r}\frac{q_i q_j}{r}$$

Use *pair_coeff.set* to set potential coefficients. *dipole* does not implement and energy shift / smoothing modes due to the function of the force.

The following coefficients must be set per unique pair of particle types:

- mu - magnitude of $\vec{\mu} = \mu(1, 0, 0)$ in the particle local reference frame
- A - electrostatic energy scale A (default value 1.0)
- kappa - inverse screening length κ

Example:

```
# A/A interact only with screened electrostatics
dipole.pair_coeff.set('A', 'A', mu=0.0, A=1.0, kappa=1.0)
dipole.pair_coeff.set('A', 'B', mu=0.5, kappa=1.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \text{tags1}, j \in \text{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(**args*, ***kwargs*)
dipole has no energy shift modes

class hoomd.md.pair.**dpd**(*r_cut*, *nlist*, *kT*, *seed*, *name=None*)
Dissipative Particle Dynamics.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **kT** (*hoomd.variant* or float) – Temperature of thermostat (in energy units).
- **seed** (*int*) – seed for the PRNG in the DPD thermostat.
- **name** (*str*) – Name of the force instance.

dpd specifies that a DPD pair force should be applied between every non-excluded particle pair in the simulation, including an interaction potential, pairwise drag force, and pairwise random force. See Groot and Warren 1997.

$$F = F_{\mathrm{C}}(r) + F_{\mathrm{R,ij}}(r_{ij}) + F_{\mathrm{D,ij}}(v_{ij})$$

$$\begin{aligned} F_{\mathrm{C}}(r) = & \quad A \cdot w(r_{ij}) \\ F_{\mathrm{R,ij}}(r_{ij}) = & \quad -\theta_{ij}\sqrt{3}\sqrt{\frac{2k_b\gamma T}{\Delta t}} \cdot w(r_{ij}) \\ F_{\mathrm{D,ij}}(r_{ij}) = & \quad -\gamma w^2(r_{ij})\left(\hat{r}_{ij} \circ v_{ij}\right) \end{aligned}$$

$$\begin{aligned} w(r_{ij}) = & \quad (1 - r/r_{\mathrm{cut}}) \quad r < r_{\mathrm{cut}} \\ = & \quad 0 \quad r \geq r_{\mathrm{cut}} \end{aligned}$$

where $\hat{r}_{ij}$ is a normalized vector from particle i to particle j, $v_{ij} = v_i - v_j$, and θ_{ij} is a uniformly distributed random number in the range [-1, 1].

dpd generates random numbers by hashing together the particle tags in the pair, the user seed, and the current time step index.

Attention: Change the seed if you reset the simulation time step to 0. If you keep the same seed, the simulation will continue with the same sequence of random numbers used previously and may cause unphysical correlations.

For MPI runs: all ranks other than 0 ignore the seed input and use the value of rank 0.

C. L. Phillips et. al. 2011 describes the DPD implementation details in HOOMD-blue. Cite it if you utilize the DPD functionality in your work.

dpd does not implement and energy shift / smoothing modes due to the function of the force. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- A - *A* (in force units)
- γ - *gamma* (in units of force/velocity)
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

To use the DPD thermostat, an *hoomd.md.integrate.nve* integrator must be applied to the system and the user must specify a temperature. Use of the dpd thermostat pair force with other integrators will result in unphysical behavior. To use pair.dpd with a different conservative potential than F_C, set A to zero and define the conservative pair potential separately. Note that DPD thermostats are often defined in terms of σ where $\sigma = \sqrt{2k_b\gamma T}$.

Example:

```
nl = nlist.cell()
dpd = pair.dpd(r_cut=1.0, nlist=nl, kT=1.0, seed=0)
dpd.pair_coeff.set('A', 'A', A=25.0, gamma = 4.5)
dpd.pair_coeff.set('A', 'B', A=40.0, gamma = 4.5)
dpd.pair_coeff.set('B', 'B', A=25.0, gamma = 4.5)
dpd.pair_coeff.set(['A', 'B'], ['C', 'D'], A=12.0, gamma = 1.2)
dpd.set_params(kT = 1.0)
integrate.mode_standard(dt=0.02)
integrate.nve(group=group.all())
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log*=*True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*kT=None*)
Changes parameters.

Parameters **kT** (*hoomd.variant* or float) – Temperature of thermostat (in energy units).

Example:

```
dpd.set_params(kT=2.0)
```

class hoomd.md.pair.**dpd_conservative**(*r_cut*, *nlist*, *name=None*)
DPD Conservative pair force.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

dpd_conservative specifies the conservative part of the DPD pair potential should be applied between every non-excluded particle pair in the simulation. No thermostat (e.g. Drag Force and Random Force) is applied, as is in *dpd*.

$$\begin{aligned} V_{\mathrm{DPD-C}}(r) = & A \cdot (r_{\mathrm{cut}} - r) - \frac{1}{2} \cdot \frac{A}{r_{\mathrm{cut}}} \cdot \left(r_{\mathrm{cut}}^2 - r^2\right) & r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

dpd_conservative does not implement and energy shift / smoothing modes due to the function of the force. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- A - *A* (in force units)
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
dpdc = pair.dpd_conservative(r_cut=3.0, nlist=nl)
dpdc.pair_coeff.set('A', 'A', A=1.0)
dpdc.pair_coeff.set('A', 'B', A=2.0, r_cut = 1.0)
dpdc.pair_coeff.set('B', 'B', A=1.0)
dpdc.pair_coeff.set(['A', 'B'], ['C', 'D'], A=5.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*coeff*)
dpd_conservative has no energy shift modes

class hoomd.md.pair.**dpdlj**(*r_cut*, *nlist*, *kT*, *seed*, *name=None*)
Dissipative Particle Dynamics with a LJ conservative force

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).

- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **kT** (*hoomd.variant* or *float*) – Temperature of thermostat (in energy units).
- **seed** (*int*) – seed for the PRNG in the DPD thermostat.
- **name** (*str*) – Name of the force instance.

dpdlj specifies that a DPD thermostat and a Lennard-Jones pair potential should be applied between every non-excluded particle pair in the simulation.

C. L. Phillips et. al. 2011 describes the DPD implementation details in HOOMD-blue. Cite it if you utilize the DPD functionality in your work.

$$F = F_{\mathrm{C}}(r) + F_{\mathrm{R,ij}}(r_{ij}) + F_{\mathrm{D,ij}}(v_{ij})$$

$$\begin{aligned} F_{\mathrm{C}}(r) = & \quad \partial V_{\mathrm{LJ}}/\partial r \\ F_{\mathrm{R,ij}}(r_{ij}) = & \quad -\theta_{ij}\sqrt{3}\sqrt{\frac{2k_b\gamma T}{\Delta t}} \cdot w(r_{ij}) \\ F_{\mathrm{D,ij}}(r_{ij}) = & \quad -\gamma w^2(r_{ij})\left(\hat{r}_{ij} \circ v_{ij}\right) \end{aligned}$$

$$\begin{aligned} V_{\mathrm{LJ}}(r) = & \quad 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \alpha\left(\frac{\sigma}{r}\right)^{6}\right] & r < r_{\mathrm{cut}} \\ = & \quad 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

$$\begin{aligned} w(r_{ij}) = & \quad (1 - r/r_{\mathrm{cut}}) & r < r_{\mathrm{cut}} \\ = & \quad 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

where $\hat{r}_{ij}$ is a normalized vector from particle i to particle j, $v_{ij} = v_i - v_j$, and θ_{ij} is a uniformly distributed random number in the range [-1, 1].

Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- σ - *sigma* (in distance units)
- α - *alpha* (unitless) - *optional*: defaults to 1.0
- γ - *gamma* (in units of force/velocity)
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

To use the DPD thermostat, an *hoomd.md.integrate.nve* integrator must be applied to the system and the user must specify a temperature. Use of the dpd thermostat pair force with other integrators will result in unphysical behavior.

Example:

```
nl = nlist.cell()
dpdlj = pair.dpdlj(r_cut=2.5, nlist=nl, kT=1.0, seed=0)
dpdlj.pair_coeff.set('A', 'A', epsilon=1.0, sigma = 1.0, gamma = 4.5)
dpdlj.pair_coeff.set('A', 'B', epsilon=0.0, sigma = 1.0 gamma = 4.5)
```

(continues on next page)

(continued from previous page)

```
dpdlj.pair_coeff.set('B', 'B', epsilon=1.0, sigma = 1.0 gamma = 4.5, r_cut = 2.
↪0**(1.0/6.0))
dpdlj.pair_coeff.set(['A', 'B'], ['C', 'D'], epsilon = 3.0,sigma=1.0, gamma = 1.2)
dpdlj.set_params(T = 1.0)
integrate.mode_standard(dt=0.005)
integrate.nve(group=group.all())
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \text{tags1}, j \in \text{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*kT=None*, *mode=None*)
Changes parameters.

Parameters

- **T** (*hoomd.variant* or *float*) – Temperature (if set) (in energy units)
- **mode** (*str*) – energy shift/smoothing mode (default noshift).

Examples:

```
dpdlj.set_params(kT=variant.linear_interp(points = [(0, 1.0), (1e5, 2.0)]))
dpdlj.set_params(kT=2.0, mode="shift")
```

class hoomd.md.pair.**ewald**(*r_cut*, *nlist*, *name=None*)
Ewald pair potential.

ewald specifies that a Ewald pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{ewald}}(r) = & \quad q_i q_j \left[\mathrm{erfc}\left(\kappa r + \frac{\alpha}{2\kappa}\right)\exp(\alpha r) + \mathrm{erfc}\left(\kappa r - \frac{\alpha}{2\kappa}\right)\exp(-\alpha r)\right] & r < r_{\mathrm{cut}} \\ = & \quad 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

The Ewald potential is designed to be used in conjunction with *hoomd.md.charge.pppm*.

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- κ - *kappa* (Splitting parameter, in 1/distance units)
- α - ***alpha*** **(Debye screening length, in 1/distance units)** New in version 2.1.

- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
ewald = pair.ewald(r_cut=3.0, nlist=nl)
ewald.pair_coeff.set('A', 'A', kappa=1.0)
ewald.pair_coeff.set('A', 'A', kappa=1.0, alpha=1.5)
ewald.pair_coeff.set('A', 'B', kappa=1.0, r_cut=3.0, r_on=2.0);
```

Warning: **DO NOT** use in conjunction with *hoomd.md.charge.pppm*. It automatically creates and configures *ewald* for you.

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log*=*True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*coeff*)
ewald has no energy shift modes

class hoomd.md.pair.**force_shifted_lj**(*r_cut*, *nlist*, *name=None*)
Force-shifted Lennard-Jones pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

force_shifted_lj specifies that a modified Lennard-Jones pair force should be applied between non-excluded particle pair in the simulation. The force differs from the one calculated by *lj* by the subtraction of the value of the force at r_{cut}, such that the force smoothly goes to zero at the cut-off. The potential is modified by a linear function. This potential can be used as a substitute for *lj*, when the exact analytical form of the latter is not required but a smaller cut-off radius is desired for computational efficiency. See Toxvaerd et. al. 2011 for a discussion of this potential.

$$\begin{aligned} V(r) = & 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \alpha\left(\frac{\sigma}{r}\right)^{6}\right] + \Delta V(r) & r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

$$\Delta V(r) = -(r - r_{\mathrm{cut}}) \frac{\partial V_{\mathrm{LJ}}}{\partial r}(r_{\mathrm{cut}})$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- σ - *sigma* (in distance units)
- α - *alpha* (unitless) - *optional*: defaults to 1.0
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
fslj = pair.force_shifted_lj(r_cut=1.5, nlist=nl)
fslj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log*=*True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: “none” (the default), “shift”, and “xplor”:

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**gauss**(*r_cut*, *nlist*, *name=None*)

Gaussian pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

gauss specifies that a Gaussian pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{gauss}}(r) = & \varepsilon \exp\left[-\frac{1}{2}\left(\frac{r}{\sigma}\right)^2\right] & r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- σ - *sigma* (in distance units)
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
gauss = pair.gauss(r_cut=3.0, nlist=nl)
gauss.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
gauss.pair_coeff.set('A', 'B', epsilon=2.0, sigma=1.0, r_cut=3.0, r_on=2.0);
gauss.pair_coeff.set(['A', 'B'], ['C', 'D'], epsilon=3.0, sigma=0.5)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**gb**(*r_cut*, *nlist*, *name=None*)

Gay-Berne anisotropic pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

gb computes the Gay-Berne potential between anisotropic particles.

This version of the Gay-Berne potential supports identical pairs of uniaxial ellipsoids, with orientation-independent energy-well depth.

The interaction energy for this anisotropic pair potential is (Allen et. al. 2006):

$$\begin{aligned} V_{\mathrm{GB}}(\vec{r}, \vec{e}_i, \vec{e}_j) = & \quad 4\varepsilon \left[\zeta^{-12} - \zeta^{-6}\right] & \zeta < \zeta_{\mathrm{cut}} \\ = & \quad 0 & \zeta \geq \zeta_{\mathrm{cut}} \end{aligned}$$

$$\zeta = \left(\frac{r - \sigma + \sigma_{\mathrm{min}}}{\sigma_{\mathrm{min}}}\right)$$

$$\sigma^{-2} = \frac{1}{2}\hat{\vec{r}} \cdot \vec{H^{-1}} \cdot \hat{\vec{r}}$$

$$\vec{H} = 2\ell_{\perp}^2 \vec{1} + (\ell_{\parallel}^2 - \ell_{\perp}^2)(\vec{e_i} \otimes \vec{e_i} + \vec{e_j} \otimes \vec{e_j})$$

with $\sigma_{\mathrm{min}} = 2\min(\ell_{\perp}, \ell_{\parallel})$.

The cut-off parameter r_{cut} is defined for two particles oriented parallel along the **long** axis, i.e. $\zeta_{\mathrm{cut}} = \left(\frac{r - \sigma_{\mathrm{max}} + \sigma_{\mathrm{min}}}{\sigma_{\mathrm{min}}}\right)$ where $\sigma_{\mathrm{max}} = 2\max(\ell_{\perp}, \ell_{\parallel})$.

The quantities $\ell_{\parallel}$ and $\ell_{\perp}$ denote the semi-axis lengths parallel and perpendicular to particle orientation.

Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- $\ell_{\perp}$ - *lperp* (in distance units)
- $\ell_{\parallel}$ - *lpar* (in distance units)

- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
gb = pair.gb(r_cut=2.5, nlist=nl)
gb.pair_coeff.set('A', 'A', epsilon=1.0, lperp=0.45, lpar=0.5)
gb.pair_coeff.set('A', 'B', epsilon=2.0, lperp=0.45, lpar=0.5, r_cut=2**(1.0/6.
↪0));
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff

valid values for mode are: "none" (the default) and "shift":

- *none* - No shifting is performed and potentials are abruptly cut off
- *shift* - A constant shift is applied to the entire potential so that it is 0 at the cutoff

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
```

class hoomd.md.pair.**lj**(*r_cut*, *nlist*, *name=None*)
Lennard-Jones pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

lj specifies that a Lennard-Jones pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{LJ}}(r) = & \quad 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \alpha \left(\frac{\sigma}{r}\right)^{6} \right] & r < r_{\mathrm{cut}} \\ = & \quad 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- σ - *sigma* (in distance units)
- α - *alpha* (unitless) - *optional*: defaults to 1.0
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
lj = pair.lj(r_cut=3.0, nlist=nl)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
lj.pair_coeff.set('A', 'B', epsilon=2.0, sigma=1.0, alpha=0.5, r_cut=3.0, r_on=2.
↪0);
lj.pair_coeff.set('B', 'B', epsilon=1.0, sigma=1.0, r_cut=2**(1.0/6.0), r_on=2.0);
lj.pair_coeff.set(['A', 'B'], ['C', 'D'], epsilon=1.5, sigma=2.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)

Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**lj1208**(*r_cut*, *nlist*, *name=None*)

Lennard-Jones 12-8 pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

lj1208 specifies that a Lennard-Jones pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{LJ}}(r) = & \quad 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \alpha \left(\frac{\sigma}{r}\right)^{8} \right] & r < r_{\mathrm{cut}} \\ = & \quad 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- σ - *sigma* (in distance units)
- α - *alpha* (unitless) - *optional*: defaults to 1.0
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

New in version 2.2.

Changed in version 2.2.

Example:

```
nl = nlist.cell()
lj1208 = pair.lj1208(r_cut=3.0, nlist=nl)
lj1208.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
lj1208.pair_coeff.set('A', 'B', epsilon=2.0, sigma=1.0, alpha=0.5, r_cut=3.0, r_
↪on=2.0);
lj1208.pair_coeff.set('B', 'B', epsilon=1.0, sigma=1.0, r_cut=2**(1.0/6.0), r_
↪on=2.0);
lj1208.pair_coeff.set(['A', 'B'], ['C', 'D'], epsilon=1.5, sigma=2.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \text{tags1}, j \in \text{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**mie**(*r_cut*, *nlist*, *name=None*)
Mie pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

mie specifies that a Mie pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{mie}}(r) = & \left(\frac{n}{n-m}\right)\left(\frac{n}{m}\right)^{\frac{m}{n-m}}\varepsilon\left[\left(\frac{\sigma}{r}\right)^{n} - \left(\frac{\sigma}{r}\right)^{m}\right] & r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- σ - *sigma* (in distance units)
- n - *n* (unitless)
- m - *m* (unitless)

- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
mie = pair.mie(r_cut=3.0, nlist=nl)
mie.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, n=12, m=6)
mie.pair_coeff.set('A', 'B', epsilon=2.0, sigma=1.0, n=14, m=7, r_cut=3.0, r_on=2.
↪0);
mie.pair_coeff.set('B', 'B', epsilon=1.0, sigma=1.0, n=15.1, m=6.5, r_cut=2**(1.0/
↪6.0), r_on=2.0);
mie.pair_coeff.set(['A', 'B'], ['C', 'D'], epsilon=1.5, sigma=2.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used

throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**moliere**(*r_cut*, *nlist*, *name=None*)
Moliere pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).

- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

moliere specifies that a Moliere type pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{Moliere}}(r) = & \frac{Z_i Z_j e^2}{4\pi\epsilon_0 r_{ij}} \left[0.35\exp\left(-0.3\frac{r_{ij}}{a_F}\right) + 0.55\exp\left(-1.2\frac{r_{ij}}{a_F}\right) + 0.10\exp\left(-6.0\frac{r_{ij}}{a_F}\right)\right] & r < r_{\mathrm{cut}} \\ = & 0 & r > r_{\mathrm{cut}} \end{aligned}$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- Z_i - *Z_i* - Atomic number of species i (unitless)
- Z_j - *Z_j* - Atomic number of species j (unitless)
- e - *elementary_charge* - The elementary charge (in charge units)
- a_0 - *a_0* - The Bohr radius (in distance units)
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
moliere = pair.moliere(r_cut = 3.0, nlist=nl)
moliere.pair_coeff.set('A', 'B', Z_i = 54.0, Z_j = 7.0, elementary_charge = 1.0,␣
↪a_0 = 1.0);
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: “none” (the default), “shift”, and “xplor”:

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff

- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**morse**(*r_cut*, *nlist*, *name=None*)

Morse pair potential.

morse specifies that a Morse pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{morse}}(r) = & \quad D_0 \left[\exp\left(-2\alpha\left(r - r_0\right)\right) - 2\exp\left(-\alpha\left(r - r_0\right)\right)\right] & r < r_{\mathrm{cut}} \\ = & \quad 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- D_0 - *D0*, depth of the potential at its minimum (in energy units)
- α - *alpha*, controls the width of the potential well (in units of 1/distance)
- r_0 - *r0*, position of the minimum (in distance units)
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
morse = pair.morse(r_cut=3.0, nlist=nl)
morse.pair_coeff.set('A', 'A', D0=1.0, alpha=3.0, r0=1.0)
morse.pair_coeff.set('A', 'B', D0=1.0, alpha=3.0, r0=1.0, r_cut=3.0, r_on=2.0);
morse.pair_coeff.set(['A', 'B'], ['C', 'D'], D0=1.0, alpha=3.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)

Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**pair**(*r_cut*, *nlist*, *name=None*)

Common pair potential documentation.

Users should not invoke *pair* directly. It is a base command that provides common features to all standard pair forces. Common documentation for all pair potentials is documented here.

All pair force commands specify that a given potential energy and force be computed on all non-excluded particle pairs in the system within a short range cutoff distance r_{cut}.

The force $\vec{F}$ applied between each pair of particles is:

$$\begin{aligned} \vec{F} = & -\nabla V(r) & r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

where $\vec{r}$ is the vector pointing from one particle to the other in the pair, and $V(r)$ is chosen by a mode switch (see *set_params()*):

$$\begin{aligned} V(r) = & V_{\mathrm{pair}}(r) & \text{mode is no\_shift} \\ = & V_{\mathrm{pair}}(r) - V_{\mathrm{pair}}(r_{\mathrm{cut}}) & \text{mode is shift} \\ = & S(r) \cdot V_{\mathrm{pair}}(r) & \text{mode is xplor and } r_{\mathrm{on}} < r_{\mathrm{cut}} \\ = & V_{\mathrm{pair}}(r) - V_{\mathrm{pair}}(r_{\mathrm{cut}}) & \text{mode is xplor and } r_{\mathrm{on}} \geq r_{\mathrm{cut}} \end{aligned}$$

$S(r)$ is the XPLOR smoothing function:

$$\begin{aligned} S(r) = & 1 & r < r_{\mathrm{on}} \\ = & \frac{(r_{\mathrm{cut}}^2 - r^2)^2 \cdot (r_{\mathrm{cut}}^2 + 2r^2 - 3r_{\mathrm{on}}^2)}{(r_{\mathrm{cut}}^2 - r_{\mathrm{on}}^2)^3} & r_{\mathrm{on}} \leq r \leq r_{\mathrm{cut}} \\ = & 0 & r > r_{\mathrm{cut}} \end{aligned}$$

and $V_{\mathrm{pair}}(r)$ is the specific pair potential chosen by the respective command.

Enabling the XPLOR smoothing function $S(r)$ results in both the potential energy and the force going smoothly to 0 at $r = r_{\mathrm{cut}}$, reducing the rate of energy drift in long simulations. r_{on} controls the point at which the smoothing starts, so it can be set to only slightly modify the tail of the potential. It is suggested that you plot your potentials with various values of r_{on} in order to find a good balance between a smooth potential function and minimal modification of the original $V_{\mathrm{pair}}(r)$. A good value for the LJ potential is $r_{\mathrm{on}} = 2 \cdot \sigma$.

The split smoothing / shifting of the potential when the mode is `xplor` is designed for use in mixed WCA / LJ systems. The WCA potential and it's first derivative already go smoothly to 0 at the cutoff, so there is no need to apply the smoothing function. In such mixed systems, set r_{on} to a value greater than r_{cut} for those pairs that interact via WCA in order to enable shifting of the WCA potential to 0 at the cutoff.

The following coefficients must be set per unique pair of particle types. See *hoomd.md.pair* for information on how to set coefficients:

- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on} - *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

When $r_{\mathrm{cut}} \leq 0$ or is set to False, the particle type pair interaction is excluded from the neighbor list. This mechanism can be used in conjunction with multiple neighbor lists to make efficient calculations in systems with large size disparity. Functionally, this is equivalent to setting $r_{\mathrm{cut}} = 0$ in the pair force because negative r_{cut} has no physical meaning.

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log*=*True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**reaction_field**(*r_cut*, *nlist*, *name=None*)
Onsager reaction field pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

reaction_field specifies that an Onsager reaction field pair potential should be applied between every non-excluded particle pair in the simulation.

Reaction field electrostatics is an approximation to the screened electrostatic interaction, which assumes that the medium can be treated as an electrostatic continuum of dielectric constant ϵ_{RF} outside the cutoff sphere of radius r_{cut}. See: Barker et. al. 1973.

$$V_{\mathrm{RF}}(r) = \varepsilon \left[\frac{1}{r} + \frac{(\epsilon_{RF} - 1)r^2}{(2\epsilon_{RF} + 1)r_c^3} \right]$$

By default, the reaction field potential does not require charge or diameter to be set. Two parameters, ε and ϵ_{RF} are needed. If $epsilon_{RF}$ is specified as zero, it will represent infinity.

If *use_charge* is set to True, the following formula is evaluated instead: .. math:

```
V_{\mathrm{RF}}(r) = q_i q_j \varepsilon \left[ \frac{1}{r} +
    \frac{(\epsilon_{RF}-1) r^2}{(2 \epsilon_{RF} + 1) r_c^3} \right]
```

where q_i and q_j are the charges of the particle pair.

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in units of energy*distance)
- ϵ_{RF} - *eps_rf* (dimensionless)
- r_{cut} - *r_cut* (in units of distance) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on} - *r_on* (in units of distance) - *optional*: defaults to the global r_cut specified in the pair command
- *use_charge* (boolean), evaluate potential using particle charges - *optional*: defaults to False

New in version 2.1.

Example:

```
nl = nlist.cell()
reaction_field = pair.reaction_field(r_cut=3.0, nlist=nl)
reaction_field.pair_coeff.set('A', 'A', epsilon=1.0, eps_rf=1.0)
reaction_field.pair_coeff.set('A', 'B', epsilon=-1.0, eps_rf=0.0)
reaction_field.pair_coeff.set('B', 'B', epsilon=1.0, eps_rf=0.0)
reaction_field.pair_coeff.set(system.particles.types, system.particles.types,␣
↪epsilon=1.0, eps_rf=0.0, use_charge=True)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**slj**(*r_cut*, *nlist*, *d_max=None*, *name=None*)
Shifted Lennard-Jones pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.
- **d_max** (*float*) – Maximum diameter particles in the simulation will have (in distance units)

slj specifies that a shifted Lennard-Jones type pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{SLJ}}(r) = & \quad 4\varepsilon \left[\left(\frac{\sigma}{r-\Delta} \right)^{12} - \left(\frac{\sigma}{r-\Delta} \right)^{6} \right] & r < (r_{\mathrm{cut}} + \Delta) \\ = & \quad 0 & r \geq (r_{\mathrm{cut}} + \Delta) \end{aligned}$$

where $\Delta = (d_i + d_j)/2 - 1$ and d_i is the diameter of particle i.

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- σ - *sigma* (in distance units) - *optional*: defaults to 1.0

- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Attention: Due to the way that pair.slj modifies the cutoff criteria, a shift_mode of *xplor* is not supported.

The actual cutoff radius for pair.slj is shifted by the diameter of two particles interacting. Thus to determine the maximum possible actual r_cut in simulation pair.sljmust know the maximum diameter of all the particles over the entire run, *d_max* . This value is either determined automatically from the initialization or can be set by the user and can be modified between runs with `hoomd.md.nlist.nlist.set_params()`. In most cases, the correct value can be identified automatically.

The specified value of *d_max* will be used to properly determine the neighbor lists during the following `hoomd.run()` commands. If not specified, `slj` will set d_max to the largest diameter in particle data at the time it is initialized.

If particle diameters change after initialization, it is **imperative** that *d_max* be the largest diameter that any particle will attain at any time during the following `hoomd.run()` commands. If *d_max* is smaller than it should be, some particles will effectively have a smaller value of *r_cut* then was set and the simulation will be incorrect. *d_max* can be changed between runs by calling `hoomd.md.nlist.nlist.set_params()`.

Example:

```
nl = nlist.cell()
slj = pair.slj(r_cut=3.0, nlist=nl, d_max = 2.0)
slj.pair_coeff.set('A', 'A', epsilon=1.0)
slj.pair_coeff.set('A', 'B', epsilon=2.0, r_cut=3.0);
slj.pair_coeff.set('B', 'B', epsilon=1.0, r_cut=2**(1.0/6.0));
slj.pair_coeff.set(['A', 'B'], ['C', 'D'], epsilon=2.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log*=*True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

See *pair.set_params()*.

Note: **xplor** is not a valid setting for *slj*.

class hoomd.md.pair.**square_density**(*r_cut*, *nlist*, *name=None*)
Soft potential for simulating a van-der-Waals liquid

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

square_density specifies that the three-body potential should be applied to every non-bonded particle pair in the simulation, that is harmonic in the local density.

The self energy per particle takes the form

$$\Psi^{ex} = B(\rho - A)^2 \tag{14.1}$$

which gives a pair-wise additive, three-body force

$$\vec{f}_{ij} = \{B(n_i - A) + B(n_j - A)\}\, w'_{ij}\vec{e}_{ij} \tag{14.2}$$

Here, w_{ij} is a quadratic, normalized weighting function,

$$w(x) = \frac{15}{2\pi r_{c,\mathrm{weight}}^3}(1 - r/r_{c,\mathrm{weight}})^2 \tag{14.3}$$

The local density at the location of particle i is defined as

$$n_i = \sum_{j \neq i} w_{ij}\left(\left|\vec{r}_i - \vec{r}_j\right|\right) \tag{14.4}$$

The following coefficients must be set per unique pair of particle types:

- A - *A* (in units of volume^-1) - mean density (*default*: 0)
- B - *B* (in units of energy*volume^2) - coefficient of the harmonic density term

Example:

```
nl = nlist.cell()
sqd = pair.van_der_waals(r_cut=3.0, nlist=nl)
sqd.pair_coeff.set('A', 'A', A=0.1)
sqd.pair_coeff.set('A', 'A', B=1.0)
```

For further details regarding this multibody potential, see

Warning: Currently HOOMD does not support reverse force communication between MPI domains on the GPU. Since reverse force communication is required for the calculation of multi-body potentials, attempting to use the square_density potential on the GPU with MPI will result in an error.

[1] P. B. Warren, "Vapor-liquid coexistence in many-body dissipative particle dynamics" Phys. Rev. E. Stat. Nonlin. Soft Matter Phys., vol. 68, no. 6 Pt 2, p. 066702, 2003.

compute_energy(*tags1*, *tags2*)
Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \text{tags1}, j \in \text{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**table**(*width*, *nlist*, *name=None*)
Tabulated pair potential.

Parameters

- **width** (*int*) – Number of points to use to interpolate V and F.
- **nlist** (*hoomd.md.nlist*) – Neighbor list (default of None automatically creates a global cell-list based neighbor list)
- **name** (*str*) – Name of the force instance

table specifies that a tabulated pair potential should be applied between every non-excluded particle pair in the simulation.

The force $\vec{F}$ is (in force units):

$$\begin{aligned} \vec{F}(\vec{r}) = & \quad 0 & r < r_{\mathrm{min}} \\ = & \quad F_{\mathrm{user}}(r)\hat{r} & r_{\mathrm{min}} \leq r < r_{\mathrm{max}} \\ = & \quad 0 & r \geq r_{\mathrm{max}} \end{aligned}$$

and the potential $V(r)$ is (in energy units)

$$\begin{aligned} V(r) = & 0 & r < r_{\mathrm{min}} \\ = & V_{\mathrm{user}}(r) & r_{\mathrm{min}} \leq r < r_{\mathrm{max}} \\ = & 0 & r \geq r_{\mathrm{max}} \end{aligned}$$

where $\vec{r}$ is the vector pointing from one particle to the other in the pair.

$F_{\mathrm{user}}(r)$ and $V_{\mathrm{user}}(r)$ are evaluated on *width* grid points between r_{min} and r_{max}. Values are interpolated linearly between grid points. For correctness, you must specify the force defined by: $F = -\frac{\partial V}{\partial r}$.

The following coefficients must be set per unique pair of particle types:

- $V_{\mathrm{user}}(r)$ and $F_{\mathrm{user}}(r)$ - evaluated by `func` (see example)
- coefficients passed to `func` - *coeff* (see example)
- $_{\mathrm{min}}$ - *rmin* (in distance units)
- $_{\mathrm{max}}$ - *rmax* (in distance units)

Set table from a given function

When you have a functional form for V and F, you can enter that directly into python. *table* will evaluate the given function over *width* points between *rmin* and *rmax* and use the resulting values in the table:

```
def lj(r, rmin, rmax, epsilon, sigma):
    V = 4 * epsilon * ( (sigma / r)**12 - (sigma / r)**6);
    F = 4 * epsilon / r * ( 12 * (sigma / r)**12 - 6 * (sigma / r)**6);
    return (V, F)

nl = nlist.cell()
table = pair.table(width=1000, nlist=nl)
table.pair_coeff.set('A', 'A', func=lj, rmin=0.8, rmax=3.0, coeff=dict(epsilon=1.
↪5, sigma=1.0))
table.pair_coeff.set('A', 'B', func=lj, rmin=0.8, rmax=3.0, coeff=dict(epsilon=2.
↪0, sigma=1.2))
table.pair_coeff.set('B', 'B', func=lj, rmin=0.8, rmax=3.0, coeff=dict(epsilon=0.
↪5, sigma=1.0))
```

Set a table from a file

When you have no function for for *V* or *F*, or you otherwise have the data listed in a file, *table* can use the given values directly. You must first specify the number of rows in your tables when initializing pair.table. Then use *set_from_file()* to read the file:

```
nl = nlist.cell()
table = pair.table(width=1000, nlist=nl)
table.set_from_file('A', 'A', filename='table_AA.dat')
table.set_from_file('A', 'B', filename='table_AB.dat')
table.set_from_file('B', 'B', filename='table_BB.dat')
```

Note: For potentials that diverge near r=0, make sure to set *rmin* to a reasonable value. If a potential does not diverge near r=0, then a setting of *rmin=0* is valid.

---

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_from_file(*a*, *b*, *filename*)
Set a pair interaction from a file.

Parameters

- **a** (*str*) – Name of type A in pair
- **b** (*str*) – Name of type B in pair
- **filename** (*str*) – Name of the file to read

The provided file specifies V and F at equally spaced r values.

Example:

```
#r  V    F
1.0 2.0 -3.0
1.1 3.0 -4.0
1.2 2.0 -3.0
1.3 1.0 -2.0
1.4 0.0 -1.0
1.5 -1.0 0.0
```

The first r value sets *rmin*, the last sets *rmax*. Any line with # as the first non-whitespace character is is treated as a comment. The *r* values must monotonically increase and be equally spaced. The table is read directly into the grid points used to evaluate $F_{\mathrm{user}}(r)$ and $_{\mathrm{user}}(r)$.

class hoomd.md.pair.**tersoff**(*r_cut*, *nlist*, *name=None*)

Tersoff Potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

tersoff specifies that the Tersoff three-body potential should be applied to every non-bonded particle pair in the simulation. Despite the fact that the Tersoff potential accounts for the effects of third bodies, it is included in the pair potentials because the species of the third body is irrelevant. It can thus use type-pair parameters similar to those of the pair potentials.

The Tersoff potential is a bond-order potential based on the Morse potential that accounts for the weakening of individual bonds with increasing coordination number. It does this by computing a modifier to the attractive term of the potential. The modifier contains the effects of third-bodies on the bond energies. The potential also includes a smoothing function around the cutoff. The smoothing function used in this work is exponential in nature as opposed to the sinusoid used by Tersoff. The exponential function provides continuity up (I believe) the second derivative.

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**yukawa**(*r_cut*, *nlist*, *name=None*)

Yukawa pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

yukawa specifies that a Yukawa pair potential should be applied between every non-excluded particle pair in the simulation.

$$\begin{aligned} V_{\mathrm{yukawa}}(r) = & \varepsilon \frac{\exp(-\kappa r)}{r} & r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- κ - *kappa* (in units of 1/distance)
- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
yukawa = pair.lj(r_cut=3.0, nlist=nl)
yukawa.pair_coeff.set('A', 'A', epsilon=1.0, kappa=1.0)
yukawa.pair_coeff.set('A', 'B', epsilon=2.0, kappa=0.5, r_cut=3.0, r_on=2.0);
yukawa.pair_coeff.set(['A', 'B'], ['C', 'D'], epsilon=0.5, kappa=3.0)
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group

- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \text{tags1}, j \in \text{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*mode=None*)
Set parameters controlling the way forces are computed.

Parameters **mode** (*str*) – (if set) Set the mode with which potentials are handled at the cutoff.

Valid values for *mode* are: "none" (the default), "shift", and "xplor":

- **none** - No shifting is performed and potentials are abruptly cut off
- **shift** - A constant shift is applied to the entire potential so that it is 0 at the cutoff
- **xplor** - A smoothing function is applied to gradually decrease both the force and potential to 0 at the cutoff when ron < rcut, and shifts the potential to 0 at the cutoff when ron >= rcut.

See *pair* for the equations.

Examples:

```
mypair.set_params(mode="shift")
mypair.set_params(mode="no_shift")
mypair.set_params(mode="xplor")
```

class hoomd.md.pair.**zbl**(*r_cut*, *nlist*, *name=None*)
ZBL pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list
- **name** (*str*) – Name of the force instance.

zbl specifies that a Ziegler-Biersack-Littmark pair potential should be applied between every non-excluded particle pair in the simulation.

$$V_{\mathrm{ZBL}}(r) = \frac{Z_i Z_j e^2}{4\pi\epsilon_0 r_{ij}}\left[0.1818\exp\left(-3.2\frac{r_{ij}}{a_F}\right) + 0.5099\exp\left(-0.9423\frac{r_{ij}}{a_F}\right) + 0.2802\exp\left(-0.4029\frac{r_{ij}}{a_F}\right) + 0.02817\exp\left(-0.201\right.\right.$$
$$= 0,$$

See *pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- Z_i - *Z_i* - Atomic number of species i (unitless)
- Z_j - *Z_j* - Atomic number of species j (unitless)
- e - *elementary_charge* - The elementary charge (in charge units)
- a_0 - *a_0* - The Bohr radius (in distance units)

- r_{cut} - *r_cut* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command
- r_{on}- *r_on* (in distance units) - *optional*: defaults to the global r_cut specified in the pair command

Example:

```
nl = nlist.cell()
zbl = pair.zbl(r_cut = 3.0, nlist=nl)
zbl.pair_coeff.set('A', 'B', Z_i = 54.0, Z_j = 7.0, elementary_charge = 1.0, a_0␣
↪= 1.0);
```

compute_energy(*tags1*, *tags2*)

Compute the energy between two sets of particles.

Parameters

- **tags1** (ndarray<int32>) – a numpy array of particle tags in the first group
- **tags2** (ndarray<int32>) – a numpy array of particle tags in the second group

$$U = \sum_{i \in \mathrm{tags1}, j \in \mathrm{tags2}} V_{ij}(r)$$

where $V_{ij}(r)$ is the pairwise energy between two particles i and j.

Assumed properties of the sets *tags1* and *tags2* are:

- *tags1* and *tags2* are disjoint
- all elements in *tags1* and *tags2* are unique
- *tags1* and *tags2* are contiguous numpy arrays of dtype int32

None of these properties are validated.

Examples:

```
tags=numpy.linspace(0,N-1,1, dtype=numpy.int32)
# computes the energy between even and odd particles
U = mypair.compute_energy(tags1=numpy.array(tags[0:N:2]), tags2=numpy.
↪array(tags[1:N:2]))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_params(*coeff*)
zbl has no energy shift modes

14.12 md.update

Overview

md.update.constraint_ellipsoid	Constrain particles to the surface of a ellipsoid.
md.update.enforce2d	Enforces 2D simulation.
md.update.rescale_temp	Rescales particle velocities.
md.update.zero_momentum	Zeroes system momentum.
md.update.mueller_plathe_flow	Updater class for a shear flow according to an algorithm published by Mueller Plathe.:

Details

Update particle properties.

When an updater is specified, it acts on the particle system each time step to change it in some way. See the documentation of specific updaters to find out what they do.

class hoomd.md.update.**constraint_ellipsoid**(*group*, *r=None*, *rx=None*, *ry=None*, *rz=None*, *P=(0, 0, 0)*)

Constrain particles to the surface of a ellipsoid.

Parameters

- **group** (*hoomd.group*) – Group for which the update will be set
- **P** (*tuple*) – (x,y,z) tuple indicating the position of the center of the ellipsoid (in distance units).
- **rx** (*float*) – radius of an ellipsoid in the X direction (in distance units).
- **ry** (*float*) – radius of an ellipsoid in the Y direction (in distance units).
- **rz** (*float*) – radius of an ellipsoid in the Z direction (in distance units).
- **r** (*float*) – radius of a sphere (in distance units), such that r=rx=ry=rz.

constraint_ellipsoid specifies that all particles are constrained to the surface of an ellipsoid. Each time step particles are projected onto the surface of the ellipsoid. Method from: http://www.geometrictools.com/Documentation/DistancePointEllipseEllipsoid.pdf

Attention: For the algorithm to work, we must have $rx >= rz$, $ry >= rz$, $rz > 0$.

Note: This method does not properly conserve virial coefficients.

Note: random thermal forces from the integrator are applied in 3D not 2D, therefore they aren't fully accurate. Suggested use is therefore only for T=0.

Examples:

```
update.constraint_ellipsoid(P=(-1,5,0), r=9)
update.constraint_ellipsoid(rx=7, ry=5, rz=3)
```

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.md.update.**enforce2d**
Enforces 2D simulation.

Every time step, particle velocities and accelerations are modified so that their z components are 0: forcing 2D simulations when other calculations may cause particles to drift out of the plane. Using enforce2d is only allowed when the system is specified as having only 2 dimensions.

Examples:

```
update.enforce2d()
```

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.md.update.**mueller_plathe_flow**(*group*, *flow_target*, *slab_direction*, *flow_direction*, *n_slabs*, *max_slab=-1*, *min_slab=-1*)

Updater class for a shear flow according to an algorithm published by Mueller Plathe.:

> "Florian Mueller-Plathe. Reversing the perturbation in nonequilibrium molecular dynamics: An easy way to calculate the shear viscosity of fluids. Phys. Rev. E, 59:4894-4898, May 1999."

The simulation box is divided in a number of slabs. Two distinct slabs of those are chosen. The "max" slab searched for the max. velocity component in flow direction, the "min" is searched for the min. velocity component. Afterward, both velocity components are swapped.

This introduces a momentum flow, which drives the flow. The strength of this flow, can be controlled by the flow_target variant, which defines the integrated target momentum flow. The searching and swapping is repeated until the target is reached. Depending on the target sign, the "max" and "min" slap might be swapped.

Parameters

- **group** (*hoomd.group*) – Group for which the update will be set
- **flow_target** (*hoomd.variant*) – Integrated target flow. The unit is the in the natural units of the simulation: [flow_target] = [timesteps] x $\mathcal{M}$ x $\frac{\mathcal{D}}{\tau}$. The unit of [timesteps] is your discretization dt x τ.
- **slab_direction** (*X*, *Y*, or *Z*) – Direction perpendicular to the slabs..
- **flow_direction** (*X*, *Y*, or *Z*) – Direction of the flow..
- **n_slabs** (*int*) – Number of slabs. You want as many as possible for small disturbed volume, where the unphysical swapping is done. But each slab has to contain a sufficient number of particle.
- **max_slab** (*int*) – Id < n_slabs where the max velocity component is search for. If set < 0 the value is set to its default n_slabs/2.
- **min_slab** (*int*) – Id < n_slabs where the min velocity component is search for. If set < 0 the value is set to its default 0.

Attention:

- This updater has to be always applied every timestep.
- This updater works currently only with orthorhombic boxes.

New in version v2.1.

Examples:

```
#const integrated flow with 0.1 slope for max 1e8 timesteps
const_flow = hoomd.variant.linear_interp( [(0,0),(1e8,0.1*1e8)] )
#velocity gradient in z direction and shear flow in x direction.
update.mueller_plathe_flow(all,const_flow,md.update.mueller_plathe_flow.Z,md.
↪update.mueller_plathe_flow.X,100)
```

X = None
Direction Enum X for this class

Y = None
Direction Enum Y for this class

Z = None
Direction Enum Z for this class

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

get_flow_epsilon()
Get the tolerance between target flow and actual achieved flow.

get_max_slab()
Get the slab id of max velocity search.

get_min_slab()
Get the slab id of min velocity search.

get_n_slabs()
Get the number of slabs.

get_summed_exchanged_momentum()
Returned the summed up exchanged velocity of the full simulation.

has_max_slab()
Returns, whether this MPI instance is part of the max slab.

has_min_slab()
Returns, whether this MPI instance is part of the min slab.

restore_state()
Restore the state information from the file used to initialize the simulations

set_flow_epsilon(*epsilon*)
Set the tolerance between target flow and actual achieved flow.

Args: epsilon (float): New tolerance for the deviation of actual and achieved flow.

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.md.update.**rescale_temp**(*kT*, *period=1*, *phase=0*)

Rescales particle velocities.

Parameters

- **kT** (*hoomd.variant* or float) – Temperature set point (in energy units)
- **period** (*int*) – Velocities will be rescaled every *period* time steps
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.

Every *period* time steps, particle velocities and angular momenta are rescaled by equal factors so that they are consistent with a given temperature in the equipartition theorem

$$\langle 1/2 m v^2 \rangle = k_B T$$
$$\langle 1/2 I \omega^2 \rangle = k_B T$$

Attention: *rescale_temp* does **not** run on the GPU, and will significantly slow down simulations.

Examples:

```
update.rescale_temp(kT=1.2)
rescaler = update.rescale_temp(kT=0.5)
update.rescale_temp(period=100, kT=1.03)
update.rescale_temp(period=100, kT=hoomd.variant.linear_interp([(0, 4.0), (1e6, 1.
↪0)]))
```

disable()

Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()

Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()

Restore the state information from the file used to initialize the simulations

set_params(*kT=None*)

Change rescale_temp parameters.

Parameters **kT** (*hoomd.variant* or float) – New temperature set point (in energy units)

Examples:

```
rescaler.set_params(kT=2.0)
```

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.md.update.**zero_momentum**(*period=1*, *phase=0*)
Zeroes system momentum.

Parameters

- **period** (*int*) – Momentum will be zeroed every *period* time steps
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.

Every *period* time steps, particle velocities are modified such that the total linear momentum of the system is set to zero.

Examples:

```
update.zero_momentum()
zeroer= update.zero_momentum(period=10)
```

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Changes the updater period.

Parameters **period** (*int*) – New period to set.

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

14.13 md.wall

Geometry

md.wall.cylinder	Cylinder wall.
md.wall.group	Defines a wall group.
md.wall.plane	Plane wall.
md.wall.sphere	Sphere wall.

Overview

md.wall.force_shifted_lj	Force-shifted Lennard-Jones wall potential.
md.wall.gauss	Gaussian wall potential.
md.wall.lj	Lennard-Jones wall potential.
md.wall.mie	Mie potential wall potential.
md.wall.morse	Morse wall potential.
md.wall.slj	Shifted Lennard-Jones wall potential
md.wall.wallpotential	Generic wall potential.
md.wall.yukawa	Yukawa wall potential.

Details

Wall potentials.

Wall potentials add forces to any particles within a certain distance, r_{cut}, of each wall. In the extrapolated mode, all particles deemed outside of the wall boundary are included as well.

Wall geometries are used to specify half-spaces. There are two half spaces for each of the possible geometries included and each can be selected using the inside parameter. In order to fully specify space, it is necessary that one half space be closed and one open. Setting *inside=True* for closed half-spaces and *inside=False* for open ones. See *wallpotential* for more information on how the concept of half-spaces are used in implementing wall forces.

Attention: The current wall force implementation does not support NPT integrators.

Wall groups (*group*) are used to pass wall geometries to wall forces. By themselves, wall groups do nothing. Only when you specify a wall force (i.e. *lj*), are forces actually applied between the wall and the

class hoomd.md.wall.**cylinder**(*r=0.0*, *origin=(0.0, 0.0, 0.0)*, *axis=(0.0, 0.0, 1.0)*, *inside=True*)
Cylinder wall.

Parameters

- **r** (*float*) – Cylinder radius (in distance units)
- **origin** (*tuple*) – Cylinder origin (in x,y,z coordinates)
- **axis** (*tuple*) – Cylinder axis vector (in x,y,z coordinates)
- **inside** (*bool*) – Selects the half-space to be used (bool)

Define a cylindrical half-space:

- inside = True selects the space inside the radius of the cylinder and includes the cylinder surface.
- inside = False selects the space outside the radius of the cylinder.

Use in function calls or by reference in the creation or modification of wall groups.

For an example see *sphere*.

class hoomd.md.wall.**force_shifted_lj**(*walls*, *r_cut=False*, *name=''*)

Force-shifted Lennard-Jones wall potential.

Parameters

- **walls** (*group*) – Wall group containing half-space geometries for the force to act in.
- **r_cut** (*float*) – The global r_cut value for the force. Defaults to False or 0 if not specified.
- **name** (*str*) – The force name which will be used in the metadata and log files.

Wall force evaluated using the Force-shifted Lennard-Jones potential. See *hoomd.md.pair.force_shifted_lj* for force details and base parameters and *wallpotential* for generalized wall potential implementation.

Example:

```
walls=wall.group()
# add walls to interact with
wall_force_fslj=wall.force_shifted_lj(walls, r_cut=3.0)
wall_force_fslj.force_coeff.set('A', epsilon=1.0, sigma=1.0)
wall_force_fslj.force_coeff.set('B', epsilon=1.5, sigma=3.0, r_cut = 8.0)
wall_force_fslj.force_coeff.set(['C','D'], epsilon=1.0, sigma=1.0, alpha = 1.5)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.wall.**gauss**(*walls*, *r_cut=False*, *name=''*)
Gaussian wall potential.

Parameters

- **walls** (*group*) – Wall group containing half-space geometries for the force to act in.
- **r_cut** (*float*) – The global r_cut value for the force. Defaults to False or 0 if not specified.
- **name** (*str*) – The force name which will be used in the metadata and log files.

Wall force evaluated using the Gaussian potential. See *hoomd.md.pair.gauss* for force details and base parameters and *wallpotential* for generalized wall potential implementation

Example:

```
walls=wall.group()
# add walls to interact with
wall_force_gauss=wall.gauss(walls, r_cut=3.0)
wall_force_gauss.force_coeff.set('A', epsilon=1.0, sigma=1.0)
wall_force_gauss.force_coeff.set('A', epsilon=2.0, sigma=1.0, r_cut=3.0)
wall_force_gauss.force_coeff.set(['C', 'D'], epsilon=3.0, sigma=0.5)
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log*=*True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.wall.**group**(**walls*)
Defines a wall group.

Parameters **walls** (*list*) – Wall objects to be included in the group.

All wall forces use a wall group as an input so it is necessary to create a wall group object before any wall force can be created. Modifications of the created wall group may occur at any time before *hoomd.run()* is invoked. Current supported geometries are spheres, cylinder, and planes. The maximum number of each type of wall is 20, 20, and 60 respectively.

The **inside** parameter used in each wall geometry is used to specify the half-space that is to be used for the force implementation. See *wallpotential* for more general details and *sphere cylinder*, and *plane* for the definition of inside for each geometry.

An effective use of wall forces **requires** considering the geometry of the system. Walls are only evaluated in one simulation box and are not periodic. It is therefore important to ensure that the walls that intersect with a periodic boundary meet.

Note: The entire structure can easily be viewed by printing the group object.

Note: While all x,y,z coordinates can be given as a list or tuple, only origin parameters are points in x,y,z space. Normal and axis parameters are vectors and must have a nonzero magnitude.

Note: Wall structure modifications between *hoomd.run()* calls will be implemented in the next run. However, modifications must be done carefully since moving the wall can result in particles moving to a relative position which causes exceptionally high forces resulting in particles moving many times the box length in one move.

Example:

```
In[0]:
# Creating wall geometry definitions using convenience functions
wallstructure=wall.group()
wallstructure.add_sphere(r=1.0,origin=(0,1,3))
wallstructure.add_sphere(1.0,[0,-1,3],inside=False)
wallstructure.add_cylinder(r=1.0,origin=(1,1,1),axis=(0,0,3),inside=True)
wallstructure.add_cylinder(4.0,[0,0,0],(1,0,1))
wallstructure.add_cylinder(5.5,(1,1,1),(3,-1,1),False)
wallstructure.add_plane(origin=(3,2,1),normal=(2,1,4))
wallstructure.add_plane((0,0,0),(10,2,1))
wallstructure.add_plane((0,0,0),(0,2,1))
print(wallstructure)

Out[0]:
Wall_Data_Structure:
spheres:2{
[0:   Radius=1.0  Origin=(0.0, 1.0, 3.0)  Inside=True]
[1:   Radius=1.0  Origin=(0.0, -1.0, 3.0) Inside=False]}
cylinders:3{
[0:   Radius=1.0  Origin=(1.0, 1.0, 1.0)  Axis=(0.0, 0.0, 3.0)    Inside=True]
[1:   Radius=4.0  Origin=(0.0, 0.0, 0.0)  Axis=(1.0, 0.0, 1.0)    Inside=True]
[2:   Radius=5.5  Origin=(1.0, 1.0, 1.0)  Axis=(3.0, -1.0, 1.0)   Inside=False]}
planes:3{
[0:   Origin=(3.0, 2.0, 1.0)  Normal=(2.0, 1.0, 4.0)]
[1:   Origin=(0.0, 0.0, 0.0)  Normal=(10.0, 2.0, 1.0)]
[2:   Origin=(0.0, 0.0, 0.0)  Normal=(0.0, 2.0, 1.0)]}

In[1]:
# Deleting wall geometry definitions using convenience functions in all accepted␣
↪types
wallstructure.del_plane(range(3))
wallstructure.del_cylinder([0,2])
wallstructure.del_sphere(1)
print(wallstructure)

Out[1]:
```

(continues on next page)

(continued from previous page)

```
Wall_Data_Structure:
spheres:1{
[0:   Radius=1.0  Origin=(0.0, 1.0, 3.0)  Inside=True]}
cylinders:1{
[0:   Radius=4.0  Origin=(0.0, 0.0, 0.0)  Axis=(1.0, 0.0, 1.0)    Inside=True]}
planes:0{}

In[2]:
# Modifying wall geometry definitions using convenience functions
wallstructure.spheres[0].r=2.0
wallstructure.cylinders[0].origin=[1,2,1]
wallstructure.cylinders[0].axis=(0,0,1)
print(wallstructure)

Out[2]:
Wall_Data_Structure:
spheres:1{
[0:   Radius=2.0  Origin=(0.0, 1.0, 3.0)  Inside=True]}
cylinders:1{
[0:   Radius=4.0  Origin=(1.0, 2.0, 1.0)  Axis=(0.0, 0.0, 1.0)    Inside=True]}
planes:0{}
```

add(*wall*)

Generic wall add for wall objects.

Generic convenience function to add any wall object to the group. Accepts *sphere*, *cylinder*, *plane*, and lists of any combination of these.

add_cylinder(*r*, *origin*, *axis*, *inside=True*)

Adds a cylinder to the wall group.

Parameters

- **r** (*float*) – Cylinder radius (in distance units)
- **origin** (*tuple*) – Cylinder origin (in x,y,z coordinates)
- **axis** (*tuple*) – Cylinder axis vector (in x,y,z coordinates)
- **inside** (*bool*) – Selects the half-space to be used (bool)

Adds a cylinder with the specified parameters to the group.

add_plane(*origin*, *normal*, *inside=True*)

Adds a plane to the wall group.

Parameters

- **origin** (*tuple*) – Plane origin (in x,y,z coordinates)
- **normal** (*tuple*) – Plane normal vector (in x,y,z coordinates)
- **inside** (*bool*) – Selects the half-space to be used (bool)

Adds a plane with the specified parameters to the wallgroup.planes list.

add_sphere(*r*, *origin*, *inside=True*)

Adds a sphere to the wall group.

Parameters

- **r** (*float*) – Sphere radius (in distance units)

- **origin** (*tuple*) – Sphere origin (in x,y,z coordinates)
- **inside** (*bool*) – Selects the half-space to be used (bool)

Adds a sphere with the specified parameters to the group.

del_cylinder(**indexs*)
Deletes the cylinder or cylinders in index.

Parameters **index** (*list*) – The index of cylinder(s) desired to delete. Accepts int, range, and lists.

Removes the specified cylinder or cylinders from the wallgroup.cylinders list.

del_plane(**indexs*)
Deletes the plane or planes in index.

Parameters **index** (*list*) – The index of plane(s) desired to delete. Accepts int, range, and lists.

Removes the specified plane or planes from the wallgroup.planes list.

del_sphere(**indexs*)
Deletes the sphere or spheres in index.

Parameters **index** (*list*) – The index of sphere(s) desired to delete. Accepts int, range, and lists.

Removes the specified sphere or spheres from the wallgroup.spheres list.

class hoomd.md.wall.**lj**(*walls*, *r_cut=False*, *name=''*)
Lennard-Jones wall potential.

Parameters

- **walls** (*group*) – Wall group containing half-space geometries for the force to act in.
- **r_cut** (*float*) – The global r_cut value for the force. Defaults to False or 0 if not specified.
- **name** (*str*) – The force name which will be used in the metadata and log files.

Wall force evaluated using the Lennard-Jones potential. See *hoomd.md.pair.lj* for force details and base parameters and *wallpotential* for generalized wall potential implementation

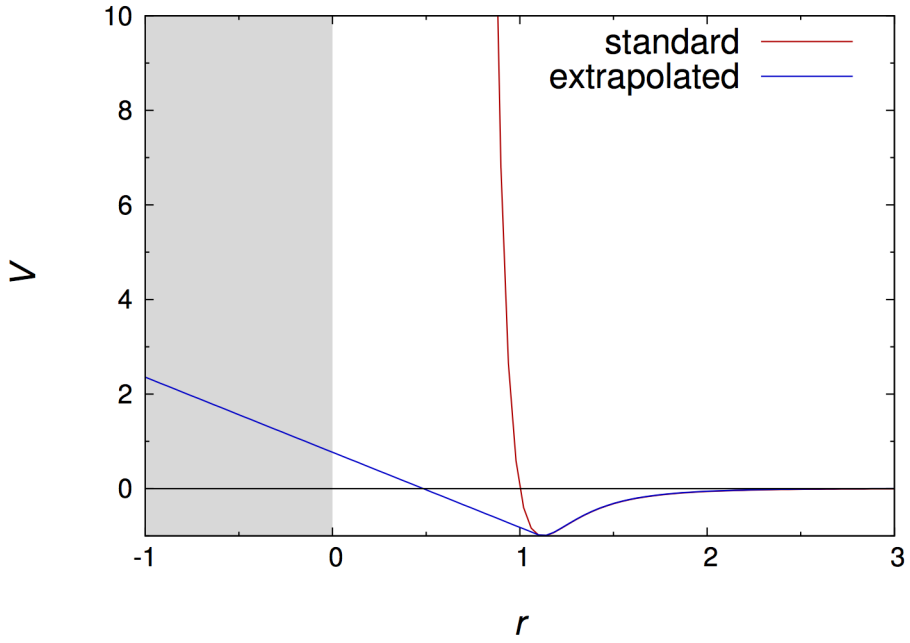
Standard mode:

```
walls=wall.group()
#add walls
lj=wall.lj(walls, r_cut=3.0)
lj.force_coeff.set('A', sigma=1.0,epsilon=1.0)  #plotted below in red
lj.force_coeff.set('B', sigma=1.0,epsilon=1.0, r_cut=2.0**(1.0/2.0))
lj.force_coeff.set(['A','B'], epsilon=2.0, sigma=1.0, alpha=1.0, r_cut=3.0)
```

Extrapolated mode:

```
walls=wall.group()
#add walls
lj_extrap=wall.lj(walls, r_cut=3.0)
lj_extrap.force_coeff.set('A', sigma=1.0,epsilon=1.0, r_extrap=1.1) #plotted in␣
↪blue below
```

V(r) plot:



disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.wall.**mie**(*walls*, *r_cut=False*, *name=''*)
Mie potential wall potential.

Parameters

- **walls** (*group*) – Wall group containing half-space geometries for the force to act in.
- **r_cut** (*float*) – The global r_cut value for the force. Defaults to False or 0 if not specified.
- **name** (*str*) – The force name which will be used in the metadata and log files.

Wall force evaluated using the Mie potential. See *hoomd.md.pair.mie* for force details and base parameters and *wallpotential* for generalized wall potential implementation

Example:

```
walls=wall.group()
# add walls to interact with
wall_force_mie=wall.mie(walls, r_cut=3.0)
wall_force_mie.force_coeff.set('A', epsilon=1.0, sigma=1.0, n=12, m=6)
wall_force_mie.force_coeff.set('A', epsilon=2.0, sigma=1.0, n=14, m=7, r_cut=3.0)
wall_force_mie.force_coeff.set('B', epsilon=1.0, sigma=1.0, n=15.1, m=6.5, r_
↪cut=2**(1.0/6.0))
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.wall.**morse**(*walls*, *r_cut=False*, *name=''*)
Morse wall potential.

Parameters

- **walls** (*group*) – Wall group containing half-space geometries for the force to act in.
- **r_cut** (*float*) – The global r_cut value for the force. Defaults to False or 0 if not specified.
- **name** (*str*) – The force name which will be used in the metadata and log files.

Wall force evaluated using the Morse potential. See *hoomd.md.pair.morse* for force details and base parameters and *wallpotential* for generalized wall potential implementation

Example:

```
walls=wall.group()
# add walls to interact with
wall_force_morse=wall.morse(walls, r_cut=3.0)
wall_force_morse.force_coeff.set('A', D0=1.0, alpha=3.0, r0=1.0)
wall_force_morse.force_coeff.set('A', D0=1.0, alpha=3.0, r0=1.0, r_cut=3.0)
wall_force_morse.force_coeff.set(['C', 'D'], D0=1.0, alpha=3.0)
```

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.wall.**plane**(*origin=(0.0, 0.0, 0.0), normal=(0.0, 0.0, 1.0), inside=True*)
Plane wall.

Parameters

- **origin** (*tuple*) – Plane origin (in x,y,z coordinates)n Default : (0.0, 0.0, 0.0)
- **normal** (*tuple*) – Plane normal vector (in x,y,z coordinates)n Default : (0.0, 0.0, 1.0)
- **inside** (*bool*) – Selects the half-space to be used (bool)n Default : True

Define a planar half space.

- inside = True selects the space on the side of the plane to which the normal vector points and includes the plane surface.

- inside = False selects the space on the side of the plane opposite the normal vector.

Use in function calls or by reference in the creation or modification of wall groups.

For an example see *sphere*.

class hoomd.md.wall.**slj**(*walls*, *r_cut=False*, *d_max=None*, *name=''*)

Shifted Lennard-Jones wall potential

Parameters

- **walls** (*group*) – Wall group containing half-space geometries for the force to act in.
- **r_cut** (*float*) – The global r_cut value for the force. Defaults to False or 0 if not specified.
- **name** (*str*) – The force name which will be used in the metadata and log files.

Wall force evaluated using the Shifted Lennard-Jones potential. Note that because slj is dependent upon particle diameters the following correction is necessary to the force details in the *hoomd.md.pair.slj* description.

$\Delta = d_i/2 - 1$ where d_i is the diameter of particle i. See *hoomd.md.pair.slj* for force details and base parameters and *wallpotential* for generalized wall potential implementation

Example:

```
walls=wall.group()
# add walls to interact with
wall_force_slj=wall.slj(walls, r_cut=3.0)
wall_force_slj.force_coeff.set('A', epsilon=1.0, sigma=1.0)
wall_force_slj.force_coeff.set('A', epsilon=2.0, sigma=1.0, r_cut=3.0)
wall_force_slj.force_coeff.set('B', epsilon=1.0, sigma=1.0, r_cut=2**(1.0/6.0))
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.wall.**sphere**(*r=0.0*, *origin=(0.0, 0.0, 0.0)*, *inside=True*)
Sphere wall.

Parameters

- **r** (*float*) – Sphere radius (in distance units)
- **origin** (*tuple*) – Sphere origin (in x,y,z coordinates)
- **inside** (*bool*) – Selects the half-space to be used

Define a spherical half-space:

- inside = True selects the space inside the radius of the sphere and includes the sphere surface.
- inside = False selects the space outside the radius of the sphere.

Use in function calls or by reference in the creation or modification of wall groups.

The following example is intended to demonstrate cylinders and planes as well. Note that the distinction between points and vectors is reflected in the default parameters.

Examples:

```
In[0]:
# One line initialization
one_line_walls=wall.group(wall.sphere(r=3,origin=(0,0,0)),wall.cylinder(r=2.5,
↪axis=(0,0,1),inside=True), wall.plane(normal=(1,0,0)))
print(one_line_walls)
full_wall_object=wall.group([wall.sphere()]*20,[wall.cylinder()]*20,[wall.
↪plane()]*60)
# Sharing wall group elements and access by reference
common_sphere=wall.sphere()
linked_walls1=wall.group(common_sphere,wall.plane(origin=(3,0,0),normal=(-1,0,0)))
linked_walls2=wall.group(common_sphere,wall.plane(origin=(-3,0,0),normal=(1,0,0)))
common_sphere.r=5.0
linked_walls1.spheres[0].origin=(0,0,1)
print(linked_walls1)
print(linked_walls2)
```

(continues on next page)

(continued from previous page)

```
Out[0]:
Wall_Data_Structure:
spheres:1{
[0:   Radius=3    Origin=(0.0, 0.0, 0.0)  Inside=True]}
cylinders:1{
[0:   Radius=2.5  Origin=(0.0, 0.0, 0.0)  Axis=(0.0, 0.0, 1.0)    Inside=True]}
planes:1{
[0:   Origin=(0.0, 0.0, 0.0)  Normal=(1.0, 0.0, 0.0)]}
Wall_Data_Structure:
spheres:1{
[0:   Radius=5.0  Origin=(0.0, 0.0, 1.0)  Inside=True]}
cylinders:0{}
planes:1{
[0:   Origin=(3.0, 0.0, 0.0)  Normal=(-1.0, 0.0, 0.0)]}
Wall_Data_Structure:
spheres:1{
[0:   Radius=5.0  Origin=(0.0, 0.0, 1.0)  Inside=True]}
cylinders:0{}
planes:1{
[0:   Origin=(-3.0, 0.0, 0.0) Normal=(1.0, 0.0, 0.0)]}
```

class hoomd.md.wall.**wallpotential**(*walls*, *r_cut*, *name=''*)

Generic wall potential.

wallpotential should not be used directly. It is a base class that provides common features to all standard wall potentials. Rather than repeating all of that documentation in many different places, it is collected here.

All wall potential commands specify that a given potential energy and potential be computed on all particles in the system within a cutoff distance, r_{cut}, from each wall in the given wall group. The force $\vec{F}$ is in the direction of $\vec{r}$, the vector pointing from the particle to the wall or half-space boundary and $V_{\mathrm{pair}}(r)$ is the specific pair potential chosen by the respective command. Wall forces are implemented with the concept of half-spaces in mind. There are two modes which are allowed currently in wall potentials: standard and extrapolated.

Standard Mode.

In the standard mode, when $r_{\mathrm{extrap}} \leq 0$, the potential energy is only applied to the half-space specified in the wall group. $V(r)$ is evaluated in the same manner as when the mode is shift for the analogous *pair* potentials within the boundaries of the half-space.

$$V(r) = V_{\mathrm{pair}}(r) - V_{\mathrm{pair}}(r_{\mathrm{cut}})$$

For inside=True (closed) half-spaces:

$$\begin{aligned} \vec{F} = & -\nabla V(r) & 0 \leq r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \\ = & 0 & r < 0 \end{aligned}$$

For inside=False (open) half-spaces:

$$\begin{aligned} \vec{F} = & -\nabla V(r) & 0 < r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \\ = & 0 & r \leq 0 \end{aligned}$$

The wall potential can be linearly extrapolated beyond a minimum separation from the wall r_{extrap} in the active half-space. This can be useful for bringing particles outside the half-space into the active half-space. It can also

be useful for typical wall force usages by effectively limiting the maximum force experienced by the particle due to the wall. The potential is extrapolated into **both** half-spaces and the cutoff r_{cut} only applies in the active half-space. The user should then be careful using this mode with multiple nested walls. It is intended to be used primarily for initialization.

The extrapolated potential has the following form:

$$\begin{aligned} V_{\mathrm{extrap}}(r) = & V(r) & , r > r_{\mathrm{extrap}} \\ = & V(r_{\mathrm{extrap}}) + (r_{\mathrm{extrap}} - r)\vec{F}(r_{\mathrm{extrap}}) \cdot \vec{n} & , r \leq r_{\mathrm{extrap}} \end{aligned}$$

where $\vec{n}$ is the normal into the active half-space. This gives an effective force on the particle due to the wall:

$$\begin{aligned} \vec{F}(r) = & \vec{F}_{\mathrm{pair}}(r) & , r > r_{\mathrm{extrap}} \\ = & \vec{F}_{\mathrm{pair}}(r_{\mathrm{extrap}}) & , r \leq r_{\mathrm{extrap}} \end{aligned}$$

where $\vec{F}_{\mathrm{pair}}$ is given by the gradient of the pair force

$$\begin{aligned} \vec{F}_{\mathrm{pair}}(r) = & -\nabla V_{\mathrm{pair}}(r) & , r < r_{\mathrm{cut}} \\ = & 0 & , r \geq r_{\mathrm{cut}} \end{aligned}$$

In other words, if r_{extrap} is chosen so that the pair force would point into the active half-space, the extrapolated potential will push all particles into the active half-space. See *lj* for a pictorial example.

To use extrapolated mode, the following coefficients must be set per unique particle types:

- All parameters required by the pair potential base for the wall potential
- r_{cut} - *r_cut* (in distance units) - *Optional: Defaults to global r_cut for the force if given or 0.0 if not*
- r_{extrap} - *r_extrap* (in distance units) - *Optional: Defaults to 0.0*

Generic Example

Note that the walls object below must be created before it is given as an argument to the force object. However, walls can be modified at any time before *hoomd.run()* is called and it will update itself appropriately. See *group* for more details about specifying the walls to be used:

```
walls=wall.group()
# Edit walls
my_force=wall.pairpotential(walls)
my_force.force_coeff.set('A', all required arguments)
my_force.force_coeff.set(['B','C'],r_cut=0.3, all required arguments)
my_force.force_coeff.set(['B','C'],r_extrap=0.3, all required arguments)
```

A specific example can be found in *lj*

Attention: The current wall force implementation does not support NPT integrators.

Note: The virial due to walls is computed, but the pressure and reported by *hoomd.analyze.log* is not well defined. The volume (area) of the box enters into the pressure computation, which is not correct in a confined system. It may not even be possible to define an appropriate volume with soft walls.

Note: An effective use of wall forces **requires** considering the geometry of the system. Each wall is only evaluated in one simulation box and thus is not periodic. Forces will be evaluated and added to all particles from

all walls in the wall group. Additionally there are no safeguards requiring a wall to exist inside the box to have interactions. This means that an attractive force existing outside the simulation box would pull particles across the periodic boundary where they would immediately cease to have any interaction with that wall. It is therefore up to the user to use walls in a physically meaningful manner. This includes the geometry of the walls, their interactions, and as noted here their location.

Note: When $r_{\mathrm{cut}} \leq 0$ or is set to False the particle type wall interaction is excluded.

Note: While wall potentials are based on the same potential energy calculations as pair potentials, Features of pair potentials such as specified neighborlists, and alternative force shifting modes are not supported.

---

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.wall.**yukawa**(*walls*, *r_cut=False*, *name=''*)

Yukawa wall potential.

Parameters

- **walls** (*group*) – Wall group containing half-space geometries for the force to act in.
- **r_cut** (*float*) – The global r_cut value for the force. Defaults to False or 0 if not specified.
- **name** (*str*) – The force name which will be used in the metadata and log files.

Wall force evaluated using the Yukawa potential. See *hoomd.md.pair.yukawa* for force details and base parameters and *wallpotential* for generalized wall potential implementation

Example:

```
walls=wall.group()
# add walls to interact with
wall_force_yukawa=wall.yukawa(walls, r_cut=3.0)
wall_force_yukawa.force_coeff.set('A', epsilon=1.0, kappa=1.0)
wall_force_yukawa.force_coeff.set('A', epsilon=2.0, kappa=0.5, r_cut=3.0)
wall_force_yukawa.force_coeff.set(['C', 'D'], epsilon=0.5, kappa=3.0)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

14.14 md.special_pair

Overview

md.special_pair.lj	LJ special pair potential.
md.special_pair.coulomb	Coulomb special pair potential.

Details

Potentials between special pairs of particles

Special pairs are used to implement interactions between designated pairs of particles. They act much like bonds, except that the interaction potential is typically a pair potential, such as LJ.

By themselves, special pairs that have been specified in an initial configuration do nothing. Only when you specify an force (i.e. special_pairs.lj), are forces actually calculated between the listed particles.

class hoomd.md.special_pair.**coeff**

Define special_pair coefficients.

The coefficients for all special pair potentials are specified using this class. Coefficients are specified per pair type.

There are two ways to set the coefficients for a particular special_pair potential. The first way is to save the special_pair potential in a variable and call *set()* directly. See below for an example of this.

The second method is to build the coeff class first and then assign it to the special_pair potential. There are some advantages to this method in that you could specify a complicated set of special_pair potential coefficients in a separate python file and import it into your job script.

Example:

```
my_coeffs = hoomd.md.special_pair.coeff();
special_pair_force.pair_coeff.set('pairtype1', epsilon=1, sigma=1)
special_pair_force.pair_coeff.set('backbone', epsilon=1.2, sigma=1)
```

set (*type*, ***coeffs*)
Sets parameters for special_pair types.

Parameters

- **type** (*str*) – Type of special_pair (or a list of type names)
- **coeffs** – Named coefficients (see below for examples)

Calling *set()* results in one or more parameters being set for a special_pair type. Types are identified by name, and parameters are also added by name. Which parameters you need to specify depends on the special_pair potential you are setting these coefficients for, see the corresponding documentation.

All possible special_pair types as defined in the simulation box must be specified before executing run(). You will receive an error if you fail to do so. It is not an error, however, to specify coefficients for special_pair types that do not exist in the simulation. This can be useful in defining a potential field for many different types of special_pairs even when some simulations only include a subset.

Examples:

```
my_special_pair_force.special_pair_coeff.set('pair1', epsilon=1, sigma=1)
my_special_pair_force.pair_coeff.set('pair2', epsilon=0.5, sigma=0.7)
my_special_pair_force.pair_coeff.set(['special_pairA','special_pairB'],␣
↪epsilon=0, sigma=1)
```

Note: Single parameters can be updated. If both `k` and `r0` have already been set for a particle type, then executing `coeff.set('polymer', r0=1.0)` will update the value of `r0` and leave the other parameters as they were previously set.

class hoomd.md.special_pair.**coulomb**(*name=None*)
Coulomb special pair potential.

Parameters **name** (*str*) – Name of the special_pair instance.

coulomb specifies a Coulomb potential energy between the two particles in each defined pair.

This is useful for implementing e.g. special 1-4 interactions in all-atom force fields. It uses a standard Coulomb interaction with a scaling parameter. This allows for using this for scaled 1-4 interactions like in OPLS where both the 1-4 LJ and Coulomb interactions are scaled by 0.5.

$$\begin{aligned} V_{\mathrm{Coulomb}}(r) = & \quad \alpha \cdot \left[\frac{q_a q_b}{r}\right] & r < r_{\mathrm{cut}} \\ = & \quad 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

where $\vec{r}$ is the vector pointing from one particle to the other in the bond.

Coefficients:

- α - Coulomb scaling factor (defaults to 1.0)
- q_a - charge of particle a (in hoomd charge units)
- q_b - charge of particle b (in hoomd charge units)
- r_{cut} - *r_cut* (in distance units)

Example:

```
coul = special_pair.coulomb(name="myOPLS_style")
coul.pair_coeff.set('pairtype_1', alpha=0.5, r_cut=1.1)
```

Note: The energy of special pair interactions is reported in a log quantity **special_pair_coul_energy**, which is separate from those of other non-bonded interactions. Therefore, the total energy of non-bonded interactions is obtained by adding that of standard and special interactions.

New in version 2.2.

Changed in version 2.2.

disable(*log=False*)
Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()
Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

class hoomd.md.special_pair.**lj**(*name=None*)

LJ special pair potential.

Parameters **name** (*str*) – Name of the special_pair instance.

lj specifies a Lennard-Jones potential energy between the two particles in each defined pair.

This is useful for implementing e.g. special 1-4 interactions in all-atom force fields.

The pair potential uses the standard LJ definition.

$$\begin{aligned} V_{\mathrm{LJ}}(r) = & \quad 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \alpha \left(\frac{\sigma}{r}\right)^{6} \right] & r < r_{\mathrm{cut}} \\ = & \quad 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

where $\vec{r}$ is the vector pointing from one particle to the other in the bond.

Coefficients:

- ε - *epsilon* (in energy units)
- σ - *sigma* (in distance units)
- α - *alpha* (unitless) - *optional*: defaults to 1.0
- r_{cut} - *r_cut* (in distance units)

Example:

```
lj = special_pair.lj(name="my_pair")
lj.pair_coeff.set('pairtype_1', epsilon=5.4, sigma=0.47, r_cut=1.1)
```

Note: The energy of special pair interactions is reported in a log quantity **special_pair_lj_energy**, which is separate from those of other non-bonded interactions. Therefore, the total energy of nonbonded interactions is obtained by adding that of standard and special interactions.

New in version 2.1.

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

CHAPTER 15

mpcd

Overview

hoomd.mpcd.integrator	MPCD integrator

Details

Multiparticle collision dynamics.

Simulating complex fluids and soft matter using conventional molecular dynamics methods (*hoomd.md*) can be computationally demanding due to large disparities in the relevant length and time scales between molecular-scale solvents and mesoscale solutes such as polymers, colloids, and deformable materials like cells. One way to overcome this challenge is to simplify the model for the solvent while retaining its most important interactions with the solute. MPCD is a particle-based simulation method for resolving solvent-mediated fluctuating hydrodynamic interactions with a microscopically detailed solute model. This method has been successfully applied to a simulate a broad class of problems, including polymer solutions and colloidal suspensions both in and out of equilibrium.

Algorithm

In MPCD, the solvent is represented by point particles having continuous positions and velocities. The solvent particles propagate in alternating streaming and collision steps. During the streaming step, particles evolve according to Newton's equations of motion. Typically, no external forces are applied to the solvent, and streaming is straightforward with a large time step. Particles are then binned into local cells and undergo a stochastic multiparticle collision within the cell. Collisions lead to the build up of hydrodynamic interactions, and the frequency and nature of the collisions, along with the solvent properties, determine the transport coefficients. All standard collision rules conserve linear momentum within the cell and can optionally be made to enforce angular-momentum conservation. Currently, we have implemented the following collision rules with linear-momentum conservation only:

- *srd* – Stochastic rotation dynamics
- *at* – Andersen thermostat

Solute particles can be coupled to the solvent during the collision step. This is particularly useful for soft materials like polymers. Standard molecular dynamics integration can be applied to the solute. Coupling to the MPCD solvent introduces both hydrodynamic interactions and a heat bath that acts as a thermostat. In the future, fluid-solid coupling will also be introduced during the streaming step to couple hard particles and boundaries.

Details of this implementation of the MPCD algorithm for HOOMD-blue can be found in M. P. Howard et al. (2018).

Getting started

MPCD is intended to be used as an add-on to the standard MD methods in *hoomd.md*. To get started, take the following steps:

1. Initialize any solute particles using standard methods (*hoomd.init*).
2. Initialize the MPCD solvent particles using one of the methods in *mpcd.init*. Additional details on how to manipulate the solvent particle data can be found in *mpcd.data*.
3. Create an MPCD *integrator*.
4. Choose the appropriate streaming method from *mpcd.stream*.
5. Choose the appropriate collision rule from *mpcd.collide*, and set the collision rule parameters.
6. Setup an MD integrator and any interactions between solute particles.
7. Optionally, configure the sorting frequency to improve performance (see *update.sort*).
8. Run your simulation!

Example script for a pure bulk SRD fluid:

```
import hoomd
hoomd.context.initialize()
from hoomd import mpcd

# Initialize (empty) solute in box.
box = hoomd.data.boxdim(L=100.)
hoomd.init.read_snapshot(hoomd.data.make_snapshot(N=0, box=box))

# Initialize MPCD particles and set sorting period.
s = mpcd.init.make_random(N=int(10*box.get_volume()), kT=1.0, seed=7)
s.sorter.set_period(period=25)

# Create MPCD integrator with streaming and collision methods.
mpcd.integrator(dt=0.1)
mpcd.stream.bulk(period=1)
mpcd.collide.srd(seed=42, period=1, angle=130., kT=1.0)

hoomd.run(2000)
```

Stability

hoomd.mpcd is currently **stable**, but remains under development. When upgrading versions, existing job scripts may need to be need to be updated. Such modifications will be noted in the change log.

Maintainer: Michael P. Howard, Princeton University.

class hoomd.mpcd.**integrator**(*dt*, *aniso=None*)
MPCD integrator

Parameters

- **dt** (*float*) – Each time step of the simulation *hoomd.run()* will advance the real time of the system forward by *dt* (in time units).
- **aniso** (*bool*) – Whether to integrate rotational degrees of freedom (bool), default None (autodetect).

The MPCD integrator enables the MPCD algorithm concurrently with standard MD *integrate* methods. An integrator must be created in order for *stream* and *collide* methods to take effect. Embedded MD particles require the creation of an appropriate integration method. Typically, this will just be *nve*.

In MPCD simulations, *dt* defines the amount of time that the system is advanced forward every time step. MPCD streaming and collision steps can be defined to occur in multiples of *dt*. In these cases, any MD particle data will be updated every *dt*, while the MPCD particle data is updated asynchronously for performance. For example, if MPCD streaming happens every 5 steps, then the particle data will be updated as follows:

```
         0     1     2     3     4     5
MD:      |---->|---->|---->|---->|---->|
MPCD:    |---------------------------->|
```

If the MPCD particle data is accessed via the snapshot interface at time step 3, it will actually contain the MPCD particle data for time step 5. The MD particles can be read at any time step because their positions are updated every step.

Examples:

```
mpcd.integrator(dt=0.1)
mpcd.integrator(dt=0.01, aniso=True)
```

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*dt=None*, *aniso=None*)
Changes parameters of an existing integration mode.

Parameters

- **dt** (*float*) – New time step delta (if set) (in time units).
- **aniso** (*bool*) – Anisotropic integration mode (bool), default None (autodetect).

Examples:

```
integrator.set_params(dt=0.007)
integrator.set_params(dt=0.005, aniso=False)
```

Modules

15.1 mpcd.collide

Overview

at
srd

Details

MPCD collision methods

An MPCD collision method is required to update the particle velocities over time. It is meant to be used in conjunction with an *integrator* and streaming method (see *stream*). Particles are binned into cells based on their positions, and all particles in a cell undergo a stochastic collision that updates their velocities while conserving linear momentum. Collision rules can optionally be extended to also enforce angular-momentum conservation. The stochastic collision lead to a build up of hydrodynamic interactions, and the choice of collision rule and solvent properties determine the transport coefficients.

class hoomd.mpcd.collide.**at**(*seed*, *period*, *kT*, *group=None*)

Andersen thermostat method

Parameters

- **seed** (*int*) – Seed to the collision method random number generator (must be positive)
- **period** (*int*) – Number of integration steps between collisions
- **kT** (*hoomd.variant* or float) – Temperature set point for the thermostat (in energy units).
- **group** (*hoomd.group*) – Group of particles to embed in collisions

This class implements the Andersen thermostat collision rule for MPCD, as described by Allahyarov and Gompper. Every *period* steps, the particles are binned into cells. The size of the cell can be selected as a property of the MPCD system (see *data.system.set_params()*). New particle velocities are then randomly drawn from a Gaussian distribution (using *seed*) relative to the center-of-mass velocity for the cell. The random velocities are given zero-mean so that the cell momentum is conserved. This collision rule naturally imparts the canonical (NVT) ensemble consistent with *kT*. The properties of the AT fluid are tuned using *period*, *kT*, the underlying size of the MPCD cell list, and the particle density.

Note: The *period* must be chosen as a multiple of the MPCD *stream* period. Other values will result in an error when *hoomd.run()* is called.

When the total mean-free path of the MPCD particles is small, the underlying MPCD cell list must be randomly shifted in order to ensure Galilean invariance. Because the performance penalty from grid shifting is small, shifting is enabled by default in all simulations. Disable it using *set_params()* if you are sure that you do not want to use it.

HOOMD particles in *group* can be embedded into the collision step (see *embed()*). A separate integration method (*integrate*) must be specified in order to integrate the positions of particles in *group*. The recommended integrator is *nve*.

Examples:

```
collide.at(seed=42, period=1, kT=1.0)
collide.at(seed=77, period=50, kT=1.5, group=hoomd.group.all())
```

disable()

Disable the collision method

Examples:

```
method.disable()
```

Disabling the collision method removes it from the current MPCD system definition. Only one collision method can be attached to the system at any time, so use this method to remove the current collision method before adding another.

embed(*group*)
Embed a particle group into the MPCD collision

Parameters **group** (*hoomd.group*) – Group of particles to embed

The *group* is embedded into the MPCD collision step and cell properties. During collisions, the embedded particles are included in determining per-cell quantities, and the collisions are applied to the embedded particles.

No integrator is generated for *group*. Usually, you will need to create a separate method to integrate the embedded particles. The recommended (and most common) integrator to use is *nve*. It is generally **not** a good idea to use a thermostatting integrator for the embedded particles, since the MPCD particles themselves already act as a heat bath that will thermalize the embedded particles.

Examples:

```
polymer = hoomd.group.type('P')
md.integrate.nve(group=polymer)
method.embed(polymer)
```

enable()
Enable the collision method

Examples:

```
method.enable()
```

Enabling the collision method adds it to the current MPCD system definition. Only one collision method can be attached to the system at any time. If another method is already set, *disable()* must be called first before switching.

set_params(*shift=None*, *kT=None*)
Set parameters for the SRD collision method

Parameters

- **shift** (*bool*) – If True, perform a random shift of the underlying cell list.
- **kT** (*hoomd.variant* or *float*) – Temperature set point for the thermostat (in energy units).

Examples:

```
srd.set_params(shift=False)
srd.set_params(shift=True, kT=1.0)
srd.set_params(kT=hoomd.data.variant.linear_interp([[0,1.0],[100,5.0]]))
```

set_period(*period*)
Set the collision period.

Parameters **period** (*int*) – New collision period.

The MPCD collision period can only be changed to a new value on a simulation timestep that is a multiple of both the previous *period* and the new *period*. An error will be raised if it is not.

Examples:

```
# The initial period is 5.
# The period can be updated to 2 on step 10.
hoomd.run_upto(10)
method.set_period(period=2)

# The period can be updated to 4 on step 12.
hoomd.run_upto(12)
hoomd.set_period(period=4)
```

class hoomd.mpcd.collide.**srd**(*seed*, *period*, *angle*, *kT=False*, *group=None*)

Stochastic rotation dynamics method

Parameters

- **seed** (*int*) – Seed to the collision method random number generator (must be positive)
- **period** (*int*) – Number of integration steps between collisions
- **angle** (*float*) – SRD rotation angle (degrees)
- **kT** (*hoomd.variant* or float or bool) – Temperature set point for the thermostat (in energy units). If False (default), no thermostat is applied and an NVE simulation is run.
- **group** (*hoomd.group*) – Group of particles to embed in collisions

This class implements the classic stochastic rotation dynamics collision rule for MPCD as first proposed by Malevanets and Kapral. Every *period* steps, the particles are binned into cells. The size of the cell can be selected as a property of the MPCD system (see *data.system.set_params()*). The particle velocities are then rotated by *angle* around an axis randomly drawn from the unit sphere. The rotation is done relative to the average velocity, so this rotation rule conserves momentum and energy within each cell, and so also globally. The properties of the SRD fluid are tuned using *period*, *angle*, *kT*, the underlying size of the MPCD cell list, and the particle density.

Note: The *period* must be chosen as a multiple of the MPCD *stream* period. Other values will result in an error when *hoomd.run()* is called.

When the total mean-free path of the MPCD particles is small, the underlying MPCD cell list must be randomly shifted in order to ensure Galilean invariance. Because the performance penalty from grid shifting is small, shifting is enabled by default in all simulations. Disable it using *set_params()* if you are sure that you do not want to use it.

HOOMD particles in *group* can be embedded into the collision step (see *embed()*). A separate integration method (*integrate*) must be specified in order to integrate the positions of particles in *group*. The recommended integrator is *nve*.

The SRD method naturally imparts the NVE ensemble to the system comprising the MPCD particles and *group*. Accordingly, the system must be properly initialized to the correct temperature. (SRD has an H theorem, and so particles exchange momentum to reach an equilibrium temperature.) A thermostat can be applied in conjunction with the SRD method through the *kT* parameter. SRD employs a Maxwell-Boltzmann thermostat on the cell level, which generates the (correct) isothermal ensemble. The temperature is defined relative to the cell-average velocity, and so can be used to dissipate heat in nonequilibrium simulations. Under this thermostat, the SRD algorithm still conserves momentum, but energy is of course no longer conserved.

Examples:

```
collide.srd(seed=42, period=1, angle=130.)
collide.srd(seed=77, period=50, angle=130., group=hoomd.group.all())
collide.srd(seed=1991, period=10, angle=90., kT=1.5)
```

disable()
Disable the collision method

Examples:

```
method.disable()
```

Disabling the collision method removes it from the current MPCD system definition. Only one collision method can be attached to the system at any time, so use this method to remove the current collision method before adding another.

embed(*group*)
Embed a particle group into the MPCD collision

Parameters **group** (*hoomd.group*) – Group of particles to embed

The *group* is embedded into the MPCD collision step and cell properties. During collisions, the embedded particles are included in determining per-cell quantities, and the collisions are applied to the embedded particles.

No integrator is generated for *group*. Usually, you will need to create a separate method to integrate the embedded particles. The recommended (and most common) integrator to use is *nve*. It is generally **not** a good idea to use a thermostatting integrator for the embedded particles, since the MPCD particles themselves already act as a heat bath that will thermalize the embedded particles.

Examples:

```
polymer = hoomd.group.type('P')
md.integrate.nve(group=polymer)
method.embed(polymer)
```

enable()
Enable the collision method

Examples:

```
method.enable()
```

Enabling the collision method adds it to the current MPCD system definition. Only one collision method can be attached to the system at any time. If another method is already set, *disable()* must be called first before switching.

set_params(*angle=None*, *shift=None*, *kT=None*)
Set parameters for the SRD collision method

Parameters

- **angle** (*float*) – SRD rotation angle (degrees)
- **shift** (*bool*) – If True, perform a random shift of the underlying cell list
- **kT** (*hoomd.variant* or float or bool) – Temperature set point for the thermostat (in energy units). If False, any set thermostat is removed and an NVE simulation is run.

Examples:

```
srd.set_params(angle=90.)
srd.set_params(shift=False)
srd.set_params(angle=130., shift=True, kT=1.0)
srd.set_params(kT=hoomd.data.variant.linear_interp([[0,1.0],[100,5.0]]))
srd.set_params(kT=False)
```

set_period(*period*)
Set the collision period.

Parameters **period** (*int*) – New collision period.

The MPCD collision period can only be changed to a new value on a simulation timestep that is a multiple of both the previous *period* and the new *period*. An error will be raised if it is not.

Examples:

```
# The initial period is 5.
# The period can be updated to 2 on step 10.
hoomd.run_upto(10)
method.set_period(period=2)

# The period can be updated to 4 on step 12.
hoomd.run_upto(12)
hoomd.set_period(period=4)
```

15.2 mpcd.data

Overview

snapshot	MPCD system snapshot
system	MPCD system data
make_snapshot	Creates an empty MPCD system snapshot

Details

MPCD data structures

MPCD and HOOMD

MPCD data is currently initialized in a secondary step from HOOMD using a snapshot interface. Even if only MPCD particles are present in the system, an empty HOOMD system must first be created. Once the HOOMD system has been initialized (see *hoomd.init*), an MPCD snapshot can be created using:

```
>>> snap = mpcd.data.make_snapshot(100)
```

The MPCD system can then initialized from the snapshot (see *hoomd.mpcd.init*):

```
>>> mpcd.init.read_snapshot(snap)
```

Because the MPCD data is stored separately from the HOOMD data, special care must be taken when using certain commands that operate on the HOOMD system data. For example, the HOOMD box size is not permitted to be changed after the MPCD particle data has been initialized (see *hoomd.mpcd.init*), so any resizes or replications of the HOOMD system must occur before then:

```
>>> hoomd_sys.replicate(2,2,2)
>>> snap.replicate(2,2,2)
>>> mpcd_sys = mpcd.init.read_snapshot(snap)
```

(continues on next page)

(continued from previous page)

```
>>> hoomd_sys.replicate(2,1,1)
**ERROR**
```

Similarly, *box_resize* will also fail after the MPCD system has been initialized.

During a simulation, the MPCD particle data can be read, modified, and restored using *take_snapshot()* and *restore_snapshot()*:

```
snap = mpcd_sys.take_snapshot()
# modify snapshot
mpcd_sys.restore_snapshot(snap)
```

MPCD and MPI

MPCD supports MPI parallelization through domain decomposition. The MPCD data in the snapshot is only valid on rank 0, and is distributed to all ranks through the snapshot collective calls.

Particle data

All MPCD particle data is accessible through the *particles* snapshot property. The size of the MPCD particle data N can be resized:

```
>>> snap.particles.resize(200)
>>> print(snap.particles.N)
200
```

Because the number of MPCD particles in a simulation is large, fewer particle properties are tracked per particle than for standard HOOMD particles. All particle data can be set as for standard snapshots using numpy arrays. Each particle is assigned a tag from 0 to N (exclusive) that is tracked. The following particle properties are recorded:

- Particle positions are stored as an Nx3 numpy array:

  ```
  >>> snap.particles.position[4] = [1., 2., 3.]
  >>> print(snap.particles.position[4])
  [ 1. 2. 3.]
  ```

  By default, all positions are initialized with zeros.

- Particle velocities can similarly be manipulated as an Nx3 numpy array:

  ```
  >>> snap.particles.velocity[2] = [0.5, 1.5, -0.25]
  >>> print(snap.particles.velocity[2])
  [0.5 1.5 -0.25]
  ```

  By default, all velocities are initialized with zeros. It is important to reassign these to a sensible value consistent with the temperature of the system.

- Each particle can be assigned a type (a name for the kind of the particle). First, a list of possible types for the system should be set:

  ```
  >>> snap.particles.types = ['A','B']
  print(snapshot.particles.types)
  ```

  Then, an index is assigned to each particle corresponding to the type:

```
>>> snap.particles.typeid[1] = 1 # B
>>> snap.particles.typeid[2] = 0 # A
```

By default, all particles are assigned a type index of 0, and no types are set. If no types are specified, type A is created by default at initialization.

- All MPCD particles have the same mass, which can be accessed or set:

```
>>> snap.particles.mass = 1.5
>>> print(snap.mass)
1.5
```

By default, all particles are assigned unit mass.

hoomd.mpcd.data.**make_snapshot**(*N=0*)

Creates an empty MPCD system snapshot

Parameters **N** (*int*) – Number of MPCD particles in the snapshot

Returns MPCD snapshot

Return type snap (*hoomd.mpcd.data.snapshot*)

Examples:

```
snap = mpcd.data.make_snapshot()
snap = mpcd.data.make_snapshot(N=50)
```

Notes

The HOOMD system **must** be initialized **before** the MPCD snapshot is taken, or an error will be raised.

class hoomd.mpcd.data.**snapshot**(*sys_snap*)

MPCD system snapshot

Parameters **sys_snap** (*object*) – The C++ representation of the system data snapshot

The MPCD system snapshot must be initialized after the HOOMD system.

This class is not intended to be initialized directly by the user, but rather returned by *make_snapshot()* or *take_snapshot()*.

particles

MPCD particle data snapshot

replicate(*nx=1*, *ny=1*, *nz=1*)

Replicate the MPCD system snapshot

Parameters

- **nx** (*int*) – Number of times to replicate snapshot in x
- **ny** (*int*) – Number of times to replicate snapshot in y
- **nz** (*int*) – Number of times to replicate snapshot in z

Examples:

```
snap.replicate(nx=2,ny=1,nz=3)
```

This method is intended only to be used with *hoomd.data.system_data.replicate()* **prior** to initialization of the MPCD system. The MPCD snapshot must be replicated to a size consistent with the system at the time of initialization. An error will be raised otherwise.

class hoomd.mpcd.data.**system**(*sysdata*)

MPCD system data

Parameters **sysdata** (*object*) – C++ representation of the MPCD system data

This class is not intended to be initialized by the user, but is the result returned by *hoomd.mpcd.init*.

restore_snapshot(*snapshot*)

Replaces the current MPCD system state

Parameters **snapshot** (*hoomd.mpcd.data.snapshot*) – MPCD system snapshot

The MPCD system data is replaced by the contents of *snapshot*.

Examples:

```
snap = mpcd_sys.take_snapshot()
snap.particles.typeid[2] = 1
mpcd_sys.restore_snapshot(snap)
```

set_params(*cell=None*)

Set parameters of the MPCD system

Parameters **cell** (*float*) – Edge length of an MPCD cell.

Every MPCD system is given a cell list for binning particles (see *mpcd.collide*). The size of the cell list sets the length scale over which hydrodynamic interactions are resolved. By default, the system is given a cell size of 1.0, i.e., the cell sets the unit of length, which is typical for most use cases. If your simulation has a different fundamental unit of length, you can adjust the cell size, but be aware that this will also change the fluid properties.

take_snapshot(*particles=True*)

Takes a snapshot of the current state of the MPCD system

Parameters **particles** (*bool*) – If true, include particle data in snapshot

Examples:

```
snap = mpcd_sys.take_snapshot()
```

15.3 mpcd.init

Overview

make_random	Initialize particles randomly
read_snapshot	Initialize from a snapshot

Details

MPCD system initialization

Commands to initialize the MPCD system data. Currently, random initialization and snapshot initialization (see *hoomd.mpcd.data*) are supported. Random initialization is useful for large systems where a snapshot is impracti-

cal. Snapshot initialization is useful when you require fine control over the particle properties and initial configuration.

hoomd.mpcd.init.**make_random**(*N*, *kT*, *seed*)

Initialize particles randomly

Parameters

- **N** (*int*) – Total number of MPCD particles
- **kT** (*float*) – Temperature of MPCD particles (in energy units)
- **seed** (*int*) – Random seed for initialization

Returns Initialized MPCD system data (*hoomd.mpcd.data.system*)

MPCD particles are randomly initialized into the simulation box. An MPCD system can be randomly initialized only **after** the HOOMD system is first initialized (see *hoomd.init*). The system can only be initialized one time. The total number of particles *N* is evenly divided between all domains. Random positions are then drawn uniformly within the (local) box. Particle velocities are drawn from a Maxwell-Boltzmann distribution consistent with temperature *kT*. All MPCD particles are given unit mass and type A.

Examples:

```
mpcd.init.make_random(N=1250000000, kT=1.0, seed=42)
```

Notes

Random number generation is performed using C++11 `mt19937` seeded by *seed* plus the rank number in MPI simulations. This random number generator is separate from other generators used in MPCD, so *seed* can be reasonably recycled elsewhere.

hoomd.mpcd.init.**read_snapshot**(*snapshot*)

Initialize from a snapshot

Parameters **snapshot** (*hoomd.mpcd.data.snapshot*) – MPCD system data snapshot

Returns Initialized MPCD system data (*hoomd.mpcd.data.system*)

An MPCD system can be initialized from a snapshot **after** the HOOMD system is first initialized (see *hoomd.init*). The system can only be initialized one time. If no type is specified in the snapshot, a default type *A* will be assigned to the MPCD particles.

Examples:

```
snap = mpcd.data.make_snapshot(N=10)
snap.particles.position[:] = L * np.random.random((10,3))
mpcd.init.read_snapshot(snap)
```

Notes

It is expected that the snapshot has the same box size as the HOOMD system. By default, this is how a new snapshot is initialized. If the HOOMD system is resized after the MPCD snapshot is created and before initialization from the MPCD snapshot, an error will be raised if the MPCD snapshot is not properly resized.

15.4 mpcd.stream

Overview

bulk	Streaming method for bulk geometry.

Details

MPCD streaming methods

An MPCD streaming method is required to update the particle positions over time. It is meant to be used in conjunction with an *integrator* and collision method (see *collide*). Particle positions are propagated ballistically according to Newton's equations (without any acceleration) for a time Δt:

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t$$

where **r** and **v** are the particle position and velocity, respectively.

class hoomd.mpcd.stream.**bulk**(*period*)

Streaming method for bulk geometry.

Parameters **period** (*int*) – Number of integration steps between collisions.

bulk performs the streaming step for MPCD particles in a fully periodic geometry (2D or 3D). This geometry is appropriate for modeling bulk fluids. The streaming time Δt is equal to *period* steps of the *integrator*. For a pure MPCD fluid, typically *period* should be 1. When particles are embedded in the MPCD fluid through the collision step, *period* should be equal to the MPCD collision *period* for best performance. The MPCD particle positions will be updated every time the simulation timestep is a multiple of *period*. This is equivalent to setting a *phase* of 0 using the terminology of other periodic *update* methods.

Example for pure MPCD fluid:

```
mpcd.integrator(dt=0.1)
mpcd.collide.srd(seed=42, period=1, angle=130.)
mpcd.stream.bulk(period=1)
```

Example for embedded particles:

```
mpcd.integrator(dt=0.01)
mpcd.collide.srd(seed=42, period=10, angle=130., group=hoomd.group.all())
mpcd.stream.bulk(period=10)
```

disable()

Disable the streaming method

Examples:

```
method.disable()
```

Disabling the streaming method removes it from the current MPCD system definition. Only one streaming method can be attached to the system at any time, so use this method to remove the current streaming method before adding another.

enable()

Enable the streaming method

Examples:

```
method.enable()
```

Enabling the streaming method adds it to the current MPCD system definition. Only one streaming method can be attached to the system at any time. If another method is already set, *disable()* must be called first before switching. Streaming will occur when the timestep is the next multiple of *period*.

set_period(*period*)
Set the streaming period.

Parameters **period** (*int*) – New streaming period.

The MPCD streaming period can only be changed to a new value on a simulation timestep that is a multiple of both the previous *period* and the new *period*. An error will be raised if it is not.

Examples:

```
# The initial period is 5.
# The period can be updated to 2 on step 10.
hoomd.run_upto(10)
method.set_period(period=2)

# The period can be updated to 4 on step 12.
hoomd.run_upto(12)
hoomd.set_period(period=4)
```

15.5 mpcd.update

Overview

sort	Sorts MPCD particles in memory to improve cache coherency.

Details

MPCD particle updaters

Updates properties of MPCD particles.

class hoomd.mpcd.update.**sort**(*system*)
Sorts MPCD particles in memory to improve cache coherency.

Parameters **system** (*hoomd.mpcd.data.system*) – MPCD system to create sorter for

Warning: Do not create *hoomd.mpcd.update.sort* explicitly in your script. HOOMD creates a sorter by default.

Every *period* time steps, particles are reordered in memory based on the cell list generated at the current timestep. Sorting can significantly improve performance of all other cell-based steps of the MPCD algorithm. The efficiency of the sort operation obviously depends on the number of particles, and so the *period* should be tuned to give the maximum performance.

Note: The *period* should be no smaller than the MPCD collision period, or unnecessary cell list builds will

occur.

Essentially all MPCD systems benefit from sorting, and so a sorter is created by default with the MPCD system. To disable it or modify parameters, save the system and access the sorter through it:

```
s = mpcd.init.read_snapshot(snap)
# the sorter is only available after initialization
s.sorter.set_period(period=5)
s.sorter.disable()
```

disable()
Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any *hoomd.run()* command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with *enable()*

enable()
Enables the updater.

Examples:

```
updater.enable()
```

See also:

disable()

restore_state()
Restore the state information from the file used to initialize the simulations

set_period(*period*)
Change the sorting period.

Parameters **period** (*int*) – New period to set.

Examples:

```
sorter.set_period(100)
sorter.set_period(1)
```

While the simulation is running, the action of each updater is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

tune(*start*, *stop*, *step*, *tsteps*, *quiet=False*)
Tune the sorting period.

Parameters

- **start** (*int*) – Start of tuning interval to scan (inclusive).
- **stop** (*int*) – End of tuning interval to scan (inclusive).
- **step** (*int*) – Spacing between tuning points.
- **tsteps** (*int*) – Number of timesteps to run at each tuning point.
- **quiet** (*bool*) – Quiet the individual run calls.

Returns The optimal sorting period from the scanned range.

Return type int

The optimal sorting period for the MPCD particles is determined from a sequence of short runs. The sorting period is first set to *start*. The TPS value is determined for a run of length *tsteps*. This run is repeated 3 times, and the median TPS of the runs is saved. The sorting period is then incremented by *step*, and the process is repeated until *stop* is reached. The period giving the fastest TPS is determined, and the sorter period is updated to this value. The results of the scan are also reported as output, and the fastest sorting period is also returned.

Note: A short warmup run is **required** before calling *tune()* in order to ensure the runtime autotuners have found optimal kernel launch parameters.

Examples:

```
# warmup run
hoomd.run(5000)

# tune sorting period
sorter.tune(start=5, stop=50, step=5, tsteps=1000)
```

CHAPTER 16

dem

Details

Simulate rounded, faceted shapes in molecular dynamics.

The DEM component provides forces which apply short-range, purely repulsive interactions between contact points of two shapes. The resulting interaction is consistent with expanding the given polygon or polyhedron by a disk or sphere of a particular rounding radius.

The pair forces located in *hoomd.dem.pair* behave like other hoomd pair forces, computing forces and torques for each particle based on its interactions with its neighbors. Also included are geometric helper utilities in *hoomd.dem.utils*.

16.1 Initialization

When initializing systems, be sure to set the inertia tensor of DEM particles. Axes with an inertia tensor of 0 (the default) will not have their rotational degrees of freedom integrated. Because only the three principal components of inertia are given to hoomd, particle vertices should also be specified in the principal reference frame so that the inertia tensor is diagonal.

Example:

```
snap = hoomd.data.make_snapshot(512, box=hoomd.data.boxdim(L=10))
snap.particles.moment_inertia[:] = (10, 10, 10)
system = hoomd.init.read_snapshot(snap)
```

16.2 Integration

To allow particles to rotate, use integrators which can update rotational degrees of freedom:

- *hoomd.md.integrate.nve*

- *hoomd.md.integrate.nvt*
- *hoomd.md.integrate.npt*
- *hoomd.md.integrate.langevin*
- *hoomd.md.integrate.brownian*

Note that the Nosé-Hoover thermostats used in *hoomd.md.integrate.nvt* and *hoomd.md.integrate.npt* work by rescaling momenta and angular momenta. This can lead to instabilities in the start of the simulation if particles are initialized with 0 angular momentum and no neighbor interactions. Two easy fixes for this problem are to initialize each particle with some angular momentum or to first run for a few steps with *hoomd.md.integrate.langevin* or *hoomd.md.integrate.brownian*.

16.3 Data Storage

To store trajectories of DEM systems, use a format that knows about anisotropic particles, such as:

- *hoomd.dump.getar*
- *hoomd.dump.gsd*

Stability

hoomd.dem is **stable**. When upgrading from version 2.x to 2.y (y > x), existing job scripts that follow *documented* interfaces for functions and classes will not require any modifications. **Maintainer:** Matthew Spellings.

Modules

16.4 dem.pair

Overview

WCA
SWCA

Details

DEM pair potentials.

class hoomd.dem.pair.**SWCA**(*nlist*, *radius=1.0*, *d_max=None*)

Specify a purely repulsive Weeks-Chandler-Andersen DEM force with a particle-varying rounding radius.

Parameters

- **nlist** (*hoomd.md.nlist*) – Neighbor list to use
- **radius** (*float*) – Unshifted rounding radius r to apply to the shape vertices
- **d_max** (*float*) – maximum rounding diameter among all particles in the system

The SWCA potential enables simulation of particles with heterogeneous rounding radii. The effect is as if a *hoomd.md.pair.slj* interaction with $r_{cut} = 2^{1/6}\sigma$ and $\sigma = 2 \cdot r$ were applied between the contact points

of each pair of particles.

Examples:

```
# 2D system of squares
squares = hoomd.dem.pair.SWCA(radius=.5)
squares.setParams('A', [[1, 1], [-1, 1], [-1, -1], [1, -1]])
# 3D system of rounded square plates
squarePlates = hoomd.dem.pair.SWCA(radius=.5)
squarePlates.setParams('A',
    vertices=[[1, 1, 0], [-1, 1, 0], [-1, -1, 0], [1, -1, 0]],
    faces=[[0, 1, 2, 3]], center=False)
# 3D system of some convex shape specified by vertices
(vertices, faces) = hoomd.dem.utils.convexHull(vertices)
shapes = hoomd.dem.pair.SWCA(radius=.5)
shapes.setParams('A', vertices=vertices, faces=faces)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

get_type_shapes()
Returns a list of shape descriptions with one element for each unique particle type in the system. Currently assumes that all 3D shapes are convex.

setParams2D(*type*, *vertices*, *center=False*)
Set the vertices for a given particle type.

Parameters

- **type** (*str*) – Name of the type to set the shape of
- **vertices** (*list*) – List of (2D) points specifying the coordinates of the shape
- **center** (*bool*) – If True, subtract the center of mass of the shape from the vertices before setting them for the shape

Shapes are specified as a list of 2D coordinates. Edges will be made between all adjacent pairs of vertices, including one between the last and first vertex.

setParams3D(*type*, *vertices*, *faces*, *center=False*)
Set the vertices for a given particle type.

Parameters

- **type** (*str*) – Name of the type to set the shape of
- **vertices** (*list*) – List of (3D) points specifying the coordinates of the shape
- **faces** (*list*) – List of lists of indices specifying which coordinates comprise each face of a shape.
- **center** (*bool*) – If True, subtract the center of mass of the shape from the vertices before setting them for the shape

Shapes are specified as a list of coordinates (*vertices*) and another list containing one list for each polygonal face (*faces*). The elements of each list inside *faces* are integer indices specifying which vertex in *vertices* comprise the face.

update_coeffs()
Noop for this potential

class hoomd.dem.pair.**WCA**(*nlist*, *radius=1.0*)
Specify a purely repulsive Weeks-Chandler-Andersen DEM force with a constant rounding radius.

Parameters

- **nlist** (*hoomd.md.nlist*) – Neighbor list to use
- **radius** (*float*) – Rounding radius r to apply to the shape vertices

The effect is as if a *hoomd.md.pair.lj* interaction with $r_{cut} = 2^{1/6}\sigma$ and $\sigma = 2 \cdot r$ were applied between the contact points of each pair of particles.

Examples:

```
# 2D system of squares
squares = hoomd.dem.pair.WCA(radius=.5)
squares.setParams('A', [[1, 1], [-1, 1], [-1, -1], [1, -1]])
# 3D system of rounded square plates
squarePlates = hoomd.dem.pair.WCA(radius=.5)
squarePlates.setParams('A',
    vertices=[[1, 1, 0], [-1, 1, 0], [-1, -1, 0], [1, -1, 0]],
    faces=[[0, 1, 2, 3]], center=False)
# 3D system of some convex shape specified by vertices
(vertices, faces) = hoomd.dem.utils.convexHull(vertices)
shapes = hoomd.dem.pair.WCA(radius=.5)
shapes.setParams('A', vertices=vertices, faces=faces)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

get_type_shapes()
Returns a list of shape descriptions with one element for each unique particle type in the system. Currently assumes that all 3D shapes are convex.

setParams2D(*type*, *vertices*, *center=False*)
Set the vertices for a given particle type.

Parameters

- **type** (*str*) – Name of the type to set the shape of
- **vertices** (*list*) – List of (2D) points specifying the coordinates of the shape
- **center** (*bool*) – If True, subtract the center of mass of the shape from the vertices before setting them for the shape

Shapes are specified as a list of 2D coordinates. Edges will be made between all adjacent pairs of vertices, including one between the last and first vertex.

setParams3D(*type*, *vertices*, *faces*, *center=False*)
Set the vertices for a given particle type.

Parameters

- **type** (*str*) – Name of the type to set the shape of
- **vertices** (*list*) – List of (3D) points specifying the coordinates of the shape
- **faces** (*list*) – List of lists of indices specifying which coordinates comprise each face of a shape.
- **center** (*bool*) – If True, subtract the center of mass of the shape from the vertices before setting them for the shape

Shapes are specified as a list of coordinates (*vertices*) and another list containing one list for each polygonal face (*faces*). The elements of each list inside *faces* are integer indices specifying which vertex in *vertices* comprise the face.

update_coeffs()
Noop for this potential

16.5 dem.utils

Overview

Details

Various helper utilities for geometry.

hoomd.dem.utils.**area**(*vertices*, *factor=1.0*)
Computes the signed area of a polygon in 2 or 3D.

Parameters

- **vertices** (*list*) – (x, y) or (x, y, z) coordinates for each vertex
- **factor** (*float*) – Factor to scale the resulting area by

hoomd.dem.utils.**center**(*vertices*, *faces=None*)

Centers shapes in 2D or 3D.

Parameters

- **vertices** (*list*) – List of (x, y) or (x, y, z) coordinates in 2D or 3D, respectively
- **faces** (*list*) – List of vertex indices for 3D polyhedra, or None for 2D. Faces should be in right-hand order.

Returns a list of vertices shifted to have the center of mass of the given points at the origin. Shapes should be specified in right-handed order. If the input shape has no mass, return the input.

Warning: All faces should be specified in right-handed order.

hoomd.dem.utils.**convexHull**(*vertices*, *tol=1e-06*)

Compute the 3D convex hull of a set of vertices and merge coplanar faces.

Parameters

- **vertices** (*list*) – List of (x, y, z) coordinates
- **tol** (*float*) – Floating point tolerance for merging coplanar faces

Returns an array of vertices and a list of faces (vertex indices) for the convex hull of the given set of vertice.

Note: This method uses scipy's quickhull wrapper and therefore requires scipy.

hoomd.dem.utils.**massProperties**(*vertices*, *faces=None*, *factor=1.0*)

Compute the mass, center of mass, and inertia tensor of a polygon or polyhedron

Parameters

- **vertices** (*list*) – List of (x, y) or (x, y, z) coordinates in 2D or 3D, respectively
- **faces** (*list*) – List of vertex indices for 3D polyhedra, or None for 2D. Faces should be in right-hand order.
- **factor** (*float*) – Factor to scale the resulting results by

Returns (mass, center of mass, moment of inertia tensor in (xx, xy, xz, yy, yz, zz) order) specified by the given list of vertices and faces. Note that the faces must be listed in a consistent order so that normals are all pointing in the correct direction from the face. If given a list of 2D vertices, return the same but for the 2D polygon specified by the vertices.

Warning: All faces should be specified in right-handed order.

The computation for the 3D case follows "Polyhedral Mass Properties (Revisited) by David Eberly, available at:

http://www.geometrictools.com/Documentation/PolyhedralMassProperties.pdf

hoomd.dem.utils.**rmax**(*vertices*, *radius=0.0*, *factor=1.0*)

Compute the maximum distance among a set of vertices

Parameters

- **vertices** (*list*) – list of (x, y) or (x, y, z) coordinates
- **factor** (*float*) – Factor to scale the result by

hoomd.dem.utils.**spheroArea**(*vertices*, *radius=1.0*, *factor=1.0*)

Computes the area of a spheropolygon.

Parameters

- **vertices** (*list*) – List of (x, y) coordinates, in right-handed (counterclockwise) order
- **radius** (*float*) – Rounding radius of the disk to expand the polygon by
- **factor** (*float*) – Factor to scale the resulting area by

CHAPTER 17

cgcmm

Details

CGCMM

Coarse grained CGCMM potential.

Stability

hoomd.cgcmm is **unstable**. When upgrading from version 2.x to 2.y (y > x), existing job scripts may need to be updated. **Maintainer needed!** This package is not maintained.

Modules

17.1 cgcmm.angle

Overview

cgcmm.angle.cgcmm

Details

CGCMM angle potentials.

class hoomd.cgcmm.angle.**cgcmm**
CGCMM angle potential.

The command angle.cgcmm defines a regular harmonic potential energy between every defined triplet of particles in the simulation, but in addition in adds the repulsive part of a CGCMM pair potential between the first

and the third particle.

B. Levine et. al. 2011 describes the CGCMM implementation details in HOOMD-blue. Cite it if you utilize the CGCMM potential in your work.

The total potential is thus:

$$V(\theta) = \frac{1}{2}k\left(\theta - \theta_0\right)^2$$

where θ is the current angle between the three particles and either:

$$V_{\mathrm{LJ}}(r_{13}) - V_{\mathrm{LJ}}(r_c) \text{ with } \quad V_{\mathrm{LJ}}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \quad \text{for } r <= r_c \quad r_c = \sigma \cdot 2^{\frac{1}{6}}$$

$$V_{\mathrm{LJ}}(r_{13}) - V_{\mathrm{LJ}}(r_c) \text{ with } \quad V_{\mathrm{LJ}}(r) = \frac{27}{4}\varepsilon\left[\left(\frac{\sigma}{r}\right)^{9} - \left(\frac{\sigma}{r}\right)^{6}\right] \quad \text{for } r <= r_c \quad r_c = \sigma \cdot \left(\frac{3}{2}\right)^{\frac{1}{3}}$$

$$V_{\mathrm{LJ}}(r_{13}) - V_{\mathrm{LJ}}(r_c) \text{ with } \quad V_{\mathrm{LJ}}(r) = \frac{3\sqrt{3}}{2}\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{4}\right] \quad \text{for } r <= r_c \quad r_c = \sigma \cdot 3^{\frac{1}{8}}$$

with r_{13} being the distance between the two outer particles of the angle.

Coefficients:

- θ_0 - rest angle `t0` (in radians)
- k - potential constant `k` (in units of energy/radians^2)
- ε - strength of potential `epsilon` (in energy units)
- σ - distance of interaction `sigma` (in distance units)

Coefficients k, θ_0, ε‘, and σ and Lennard-Jones exponents pair must be set for each type of angle in the simulation using *set_coeff()*.

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)
Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)
Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

set_coeff(*angle_type*, *k*, *t0*, *exponents*, *epsilon*, *sigma*)
Sets the CG-CMM angle coefficients for a particular angle type.

Parameters

- **angle_type** (*str*) – Angle type to set coefficients for
- **k** (*float*) – Coefficient k (in units of energy/radians^2)
- **t0** (*float*) – Coefficient θ_0 (in radians)
- **exponents** (*str*) – is the type of CG-angle exponents we want to use for the repulsion.
- **epsilon** (*float*) – is the 1-3 repulsion strength (in energy units)
- **sigma** (*float*) – is the CG particle radius (in distance units)

Examples:

```
cgcmm.set_coeff('polymer', k=3.0, t0=0.7851, exponents=126, epsilon=1.0,␣
↪sigma=0.53)
cgcmm.set_coeff('backbone', k=100.0, t0=1.0, exponents=96, epsilon=23.0,␣
↪sigma=0.1)
cgcmm.set_coeff('residue', k=100.0, t0=1.0, exponents='lj12_4', epsilon=33.0,␣
↪sigma=0.02)
cgcmm.set_coeff('cg96', k=100.0, t0=1.0, exponents='LJ9-6', epsilon=9.0,␣
↪sigma=0.3)
```

17.2 cgcmm.pair

Overview

cgcmm.pair.cgcmm	CMM coarse-grain model pair potential.

Details

CGCMM pair potentials.

class hoomd.cgcmm.pair.**cgcmm**(*r_cut*, *nlist*)

CMM coarse-grain model pair potential.

Parameters

- **r_cut** (*float*) – Default cutoff radius (in distance units).
- **nlist** (*hoomd.md.nlist*) – Neighbor list

cgcmm specifies that a special version of Lennard-Jones pair force should be added to every non-bonded particle pair in the simulation. This potential version is used in the CMM coarse grain model and uses a combination of Lennard-Jones potentials with different exponent pairs between different atom pairs.

B. Levine et. al. 2011 describes the CGCMM implementation details in HOOMD-blue. Cite it if you utilize the CGCMM potential in your work.

Multiple potential functions can be selected:

$$V_{\mathrm{LJ}}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \alpha\left(\frac{\sigma}{r}\right)^{6}\right]$$

$$V_{\mathrm{LJ}}(r) = \frac{27}{4}\varepsilon\left[\left(\frac{\sigma}{r}\right)^{9} - \alpha\left(\frac{\sigma}{r}\right)^{6}\right]$$

$$V_{\mathrm{LJ}}(r) = \frac{3\sqrt{3}}{2}\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \alpha\left(\frac{\sigma}{r}\right)^{4}\right]$$

See *hoomd.md.pair.pair* for details on how forces are calculated and the available energy shifting and smoothing modes. Use *pair_coeff.set* to set potential coefficients.

The following coefficients must be set per unique pair of particle types:

- ε - *epsilon* (in energy units)
- σ - *sigma* (in distance units)
- α - *alpha* (unitless) - *optional*: defaults to 1.0
- exponents, the choice of LJ-exponents, currently supported are 12-6, 9-6, and 12-4.

We support three keyword variants 124 (native), lj12_4 (LAMMPS), LJ12-4 (MPDyn).

Example:

```
nl = nlist.cell()
cg = pair.cgcmm(r_cut=3.0, nlist=nl)
cg.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=1.0, exponents='LJ12-6')
cg.pair_coeff.set('W', 'W', epsilon=3.7605, sigma=1.285588, alpha=1.0, exponents=
↪'lj12_4')
cg.pair_coeff.set('OA', 'OA', epsilon=1.88697479, sigma=1.09205882, alpha=1.0,␣
↪exponents='96')
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

CHAPTER 18

deprecated

Details

Deprecated functionalities

Commands in the *hoomd.deprecated* package are leftovers from previous versions of HOOMD-blue that are kept temporarily for users whose workflow depends on them. Deprecated features may be removed in a future version.

Stability

hoomd.deprecated is **deprecated**. When upgrad from version 2.x to 2.y (y > x), functions and classes in the package may be removed. Continued support for features in this package is not provided. These legacy functions will remain as long as they require minimal code modifications to maintain. **Maintainer:** *not maintained*.

Modules

18.1 deprecated.analyze

Overview

deprecated.analyze.msd	Mean-squared displacement.

Details

Deprecated analyzers.

class hoomd.deprecated.analyze.**msd**(*filename*, *groups*, *period*, *header_prefix=''*, *r0_file=None*, *overwrite=False*, *phase=0*)

Mean-squared displacement.

Parameters

- **filename** (*str*) – File to write the data to.
- **groups** (*list*) – List of groups to calculate the MSDs of.
- **period** (*int*) – Quantities are logged every *period* time steps.
- **header_prefix** (*str*) – (optional) Specify a string to print before the header.
- **r0_file** (*str*) – hoomd_xml file specifying the positions (and images) to use for $\vec{r}_0$.
- **overwrite** (*bool*) – set to True to overwrite the file *filename* if it exists.
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.

Deprecated since version 2.0: analyze.msd will be replaced by a more general system capable of window averaging in a future release.

msd can be given any number of groups of particles. Every *period* time steps, it calculates the mean squared displacement of each group (referenced to the particle positions at the time step the command is issued at) and prints the calculated values out to a file.

The mean squared displacement (MSD) for each group is calculated as:

$$\langle |\vec{r} - \vec{r}_0|^2 \rangle$$

and values are correspondingly written in units of distance squared.

The file format is the same convenient delimited format used by :py:class‘hoomd.analyze.log‘.

msd is capable of appending to an existing msd file (the default setting) for use in restarting in long jobs. To generate a correct msd that does not reset to 0 at the start of each run, save the initial state of the system in a hoomd_xml file, including position and image data at a minimum. In the continuation job, specify this file in the *r0_file* argument to analyze.msd.

Examples:

```
msd = analyze.msd(filename='msd.log', groups=[group1, group2],
                  period=100)

analyze.msd(groups=[group1, group2, group3], period=1000,
            filename='msd.log', header_prefix='#')

analyze.msd(filename='msd.log', groups=[group1], period=10,
            header_prefix='Log of group1 msd, run 5\n')
```

A group variable (*groupN* above) can be created by any number of group creation functions. See group for a list.

By default, columns in the file are separated by tabs, suitable for importing as a tab-delimited spreadsheet. The delimiter can be changed to any string using *set_params()*.

The *header_prefix* can be used in a number of ways. It specifies a simple string that will be printed before the header line of the output file. One handy way to use this is to specify header_prefix='#' so that `gnuplot` will ignore the header line automatically. Another use-case would be to specify a descriptive line containing details of the current run. Examples of each of these cases are given above.

If *r0_file* is left at the default of None, then the current state of the system at the execution of the analyze.msd command is used to initialize $\vec{r}_0$.

disable()
Disable the analyzer.

Examples:

```
my_analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any *hoomd.run()* command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with *enable()*.

enable()
Enables the analyzer

Examples:

```
my_analyzer.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*delimiter=None*)
Change the parameters of the msd analysis

Parameters **delimiter** (*str*) – New delimiter between columns in the output file (if specified).

Examples:

```
msd.set_params(delimiter=',');
```

set_period(*period*)
Changes the period between analyzer executions

Parameters **period** (*int*) – New period to set (in time steps)

Examples:

```
analyzer.set_period(100)
analyzer.set_period(1)
```

While the simulation is running (*hoomd.run()*, the action of each analyzer is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

18.2 deprecated.dump

Overview

deprecated.dump.pos	Writes simulation snapshots in the POS format
deprecated.dump.xml	Writes simulation snapshots in the HOOMD XML format.

Details

Deprecated trajectory file writers.

class hoomd.deprecated.dump.**pos**(*filename*, *period=None*, *unwrap_rigid=False*, *phase=0*, *addInfo=None*)

Writes simulation snapshots in the POS format

Parameters

- **filename** (*str*) – File name to write
- **period** (*int*) – (optional) Number of time steps between file dumps
- **unwrap_rigid** (*bool*) – When False, (the default) individual particles are written inside the simulation box which breaks up rigid bodies near box boundaries. When True, particles belonging to the same rigid body will be unwrapped so that the body is continuous. The center of mass of the body remains in the simulation box, but some particles may be written just outside it.
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where *(step + phase) % period == 0*.
- **addInfo** (*callable*) – A user-defined python function that returns a string of additional information when it is called. This information will be printed in the pos file beneath the shape definitions. The information returned by addInfo may dynamically change over the course of the simulation; addInfo is a function of the simulation timestep only.

Deprecated since version 2.0.

The file is opened on initialization and a new frame is appended every a period steps.

Warning: *pos* is not restart compatible. It always overwrites the file on initialization.

Examples:

```
dump.pos(filename="dump.pos", period=1000)
pos = dump.pos(filename="particles.pos", period=1e5)
```

disable()

Disable the analyzer.

Examples:

```
my_analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any *hoomd.run()* command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with *enable()*.

enable()

Enables the analyzer

Examples:

```
my_analyzer.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

set_def(*typ*, *shape*)
Set a pos def string for a given type

Parameters

- **typ** (*str*) – Type name to set shape def
- **shape** (*str*) – Shape def string to set

set_period(*period*)
Changes the period between analyzer executions

Parameters **period** (*int*) – New period to set (in time steps)

Examples:

```
analyzer.set_period(100)
analyzer.set_period(1)
```

While the simulation is running (*hoomd.run()*, the action of each analyzer is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

class hoomd.deprecated.dump.**xml**(*group*, *filename='dump'*, *period=None*, *time_step=None*, *phase=0*, *restart=False*, ***params*)
Writes simulation snapshots in the HOOMD XML format.

Parameters

- **group** (*hoomd.group*) – Group of particles to dump
- **filename** (*str*) – (optional) Base of the file name
- **period** (*int*) – (optional) Number of time steps between file dumps
- **params** – (optional) Any number of parameters that set_params() accepts
- **time_step** (*int*) – (optional) Time step to write into the file (overrides the current simulation step). time_step is ignored for periodic updates
- **phase** (*int*) – When -1, start on the current time step. When >= 0, execute on steps where &(step + phase) % period == 0*.
- **restart** (*bool*) – When True, write only *filename* and don't save previous states.

Deprecated since version 2.0: GSD is the new default file format for HOOMD-blue. It can store everything that an XML file can in an efficient binary format that is easy to access. See *hoomd.dump.gsd*.

Every *period* time steps, a new file will be created. The state of the particles in *group* at that time step is written to the file in the HOOMD XML format. All values are written in native HOOMD-blue units, see *Units* for more information.

If you only need to store a subset of the system, you can save file size and time spent analyzing data by specifying a group to write out. *xml* will write out all of the particles in *group* in ascending tag order. When the group is not *hoomd.group.all()*, *xml* will not write the topology fields (bond, angle, dihedral, improper, constraint).

Examples:

```
deprecated.dump.xml(group=group.all(), filename="atoms.dump", period=1000)
xml = deprecated.dump.xml(group=group.all(), filename="particles", period=1e5)
xml = deprecated.dump.xml(group=group.all(), filename="test.xml", vis=True)
```

(continues on next page)

(continued from previous page)

```
xml = deprecated.dump.xml(group=group.all(), filename="restart.xml", all=True,␣
↪restart=True, period=10000, phase=0);
xml = deprecated.dump.xml(group=group.type('A'), filename="A", period=1e3)
```

If period is set and restart is False, a new file will be created every *period* steps. The time step at which the file is created is added to the file name in a fixed width format to allow files to easily be read in order. I.e. the write at time step 0 with `filename="particles"` produces the file `particles.0000000000.xml`.

If period is set and restart is True, *xml* will write a temporary file and then move it to *filename*. This stores only the most recent state of the simulation in the written file. It is useful for writing jobs that are restartable - see *Restartable jobs*. Note that this causes high metadata traffic on lustre filesystems and may cause your account to be blocked at some supercomputer centers. Use *hoomd.dump.gsd* for efficient restart files.

By default, only particle positions are output to the dump files. This can be changed with set_params(), or by specifying the options in the *xml* command.

If *period* is not specified, then no periodic updates will occur. Instead, the file *filename* is written immediately. *time_step* is passed on to write()

disable()
Disable the analyzer.

Examples:

```
my_analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any *hoomd.run()* command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with *enable()*.

enable()
Enables the analyzer

Examples:

```
my_analyzer.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

set_params(*all=None*, *vis=None*, *position=None*, *image=None*, *velocity=None*, *mass=None*, *diameter=None*, *type=None*, *body=None*, *bond=None*, *angle=None*, *dihedral=None*, *improper=None*, *constraint=None*, *acceleration=None*, *charge=None*, *orientation=None*, *angmom=None*, *inertia=None*, *vizsigma=None*)
Change xml write parameters.

Parameters

- **all** (*bool*) – (if True) Enables the output of all optional parameters below
- **vis** (*bool*) – (if True) Enables options commonly used for visualization. - Specifically, vis=True sets position, mass, diameter, type, body, bond, angle, dihedral, improper, charge
- **position** (*bool*) – (if set) Set to True/False to enable/disable the output of particle positions in the xml file
- **image** (*bool*) – (if set) Set to True/False to enable/disable the output of particle images in the xml file

- **velocity** (*bool*) – (if set) Set to True/False to enable/disable the output of particle velocities in the xml file
- **mass** (*bool*) – (if set) Set to True/False to enable/disable the output of particle masses in the xml file
- **diameter** (*bool*) – (if set) Set to True/False to enable/disable the output of particle diameters in the xml file
- **type** (*bool*) – (if set) Set to True/False to enable/disable the output of particle types in the xml file
- **body** (*bool*) – (if set) Set to True/False to enable/disable the output of the particle bodies in the xml file
- **bond** (*bool*) – (if set) Set to True/False to enable/disable the output of bonds in the xml file
- **angle** (*bool*) – (if set) Set to True/False to enable/disable the output of angles in the xml file
- **dihedral** (*bool*) – (if set) Set to True/False to enable/disable the output of dihedrals in the xml file
- **improper** (*bool*) – (if set) Set to True/False to enable/disable the output of impropers in the xml file
- **constraint** (*bool*) – (if set) Set to True/False to enable/disable the output of constraints in the xml file
- **acceleration** (*bool*) – (if set) Set to True/False to enable/disable the output of particle accelerations in the xml
- **charge** (*bool*) – (if set) Set to True/False to enable/disable the output of particle charge in the xml
- **orientation** (*bool*) – (if set) Set to True/False to enable/disable the output of particle orientations in the xml file
- **angmom** (*bool*) – (if set) Set to True/False to enable/disable the output of particle angular momenta in the xml file
- **inertia** (*bool*) – (if set) Set to True/False to enable/disable the output of particle moments of inertia in the xml file
- **vizsigma** (*bool*) – (if set) Set to a floating point value to include as vizsigma in the xml file

Examples:

```
xml.set_params(type=False)
xml.set_params(position=False, type=False, velocity=True)
xml.set_params(type=True, position=True)
xml.set_params(bond=True)
xml.set_params(all=True)
```

Attention: The simulation topology (bond, angle, dihedral, improper, constraint) cannot be output when the group for *xml* is not *hoomd.group.all*. An error will be raised.

set_period(*period*)

Changes the period between analyzer executions

Parameters **period** (*int*) – New period to set (in time steps)

Examples:

```
analyzer.set_period(100)
analyzer.set_period(1)
```

While the simulation is running (*hoomd.run()*, the action of each analyzer is executed every *period* time steps. Changing the period does not change the phase set when the analyzer was first created.

write(*filename*, *time_step=None*)

Write a file at the current time step.

Parameters

- **filename** (*str*) – File name to write to
- **time_step** (*int*) – (if set) Time step value to write out to the file

The periodic file writes can be temporarily overridden and a file with any file name written at the current time step.

When *time_step* is None, the current system time step is written to the file. When specified, *time_step* overrides this value.

Examples:

```
xml.write(filename="start.xml")
xml.write(filename="start.xml", time_step=0)
```

write_restart()

Write a restart file at the current time step.

This only works when dump.xml() is in **restart** mode. write_restart() writes out a restart file at the current time step. Put it at the end of a script to ensure that the system state is written out before exiting.

18.3 deprecated.init

Overview

deprecated.init.create_random	Generates N randomly positioned particles of the same type.
deprecated.init.create_random_polymers	Generates any number of randomly positioned polymers of configurable types.
deprecated.init.read_xml	## Reads initial system state from an XML file

Details

Deprecated initialization routines.

hoomd.deprecated.init.**create_random**(*N*, *phi_p=None*, *name='A'*, *min_dist=0.7*, *box=None*, *seed=1*, *dimensions=3*)

Generates N randomly positioned particles of the same type.

Parameters

- **N** (*int*) – Number of particles to create.

- **phi_p** (*float*) – Packing fraction of particles in the simulation box (unitless).
- **name** (*str*) – Name of the particle type to create.
- **min_dist** (*float*) – Minimum distance particles will be separated by (in distance units).
- **box** (*hoomd.data.boxdim*) – Simulation box dimensions.
- **seed** (*int*) – Random seed.
- **dimensions** (*int*) – The number of dimensions in the simulation.

Deprecated since version 2.0: Random initialization is best left to specific methods tailored by the user for their work.

Either *phi_p* or *box* must be specified. If *phi_p* is provided, it overrides the value of *box*.

Examples:

```
init.create_random(N=2400, phi_p=0.20)
init.create_random(N=2400, phi_p=0.40, min_dist=0.5)
system = init.create_random(N=2400, box=data.boxdim(L=20))
```

When *phi_p* is set, the dimensions of the created box are such that the packing fraction of particles in the box is *phi_p*. The number density e n is related to the packing fraction by $n = 2d/\pi \cdot \phi_P$, where d is the dimension, and assumes the particles have a radius of 0.5. All particles are created with the same type, given by *name*.

The result of *hoomd.deprecated.init.create_random()* can be saved in a variable and later used to read and/or change particle properties later in the script. See *hoomd.data* for more information.

hoomd.deprecated.init.**create_random_polymers**(*box*, *polymers*, *separation*, *seed=1*)

Generates any number of randomly positioned polymers of configurable types.

Parameters

- **box** (*hoomd.data.boxdim*) – Simulation box dimensions
- **polymers** (*list*) – Specification for the different polymers to create (see below)
- **separation** (*dict*) – Separation radii for different particle types (see below)
- **seed** (*int*) – Random seed to use

Deprecated since version 2.0: Random initialization is best left to specific methods tailored by the user for their work.

Any number of polymers can be generated, of the same or different types, as specified in the argument *polymers*. Parameters for each polymer include bond length, particle type list, bond list, and count.

The syntax is best shown by example. The below line specifies that 600 block copolymers A6B7A6 with a bond length of 1.2 be generated:

```
polymer1 = dict(bond_len=1.2, type=['A']*6 + ['B']*7 + ['A']*6,
                bond="linear", count=600)
```

Here is an example for a second polymer, specifying just 100 polymers made of 5 B beads bonded in a branched pattern:

```
polymer2 = dict(bond_len=1.2, type=['B']*5,
                bond=[(0, 1), (1,2), (1,3), (3,4)] , count=100)
```

The *polymers* argument can be given a list of any number of polymer types specified as above. *count* randomly generated polymers of each type in the list will be generated in the system.

In detail:

- bond_len defines the bond length of the generated polymers. This should not necessarily be set to the equilibrium bond length! The generator is dumb and doesn't know that bonded particles can be placed closer together than the separation (see below). Thus bond_len must be at a minimum set at twice the value of the largest separation radius. An error will be generated if this is not the case.
- type is a python list of strings. Each string names a particle type in the order that they will be created in generating the polymer.
- bond can be specified as "linear" in which case the generator connects all particles together with bonds to form a linear chain. bond can also be given a list if python tuples (see example above). - Each tuple in the form of c (a,b) specifies that particle c a of the polymer be bonded to particle c b. These bonds are given the default type name of 'polymer' to be used when specifying parameters to bond forces such as bond.harmonic. - A tuple with three elements (a,b,type) can be used as above, but with a custom name for the bond. For example, a simple branched polymer with different bond types on each branch could be defined like so:

```
bond=[(0,1), (1,2), (2,3,'branchA'), (3,4,'branchA), (2,5,'branchB'), (5,6,
↪'branchB')]
```

separation must contain one entry for each particle type specified in polymers ('A' and 'B' in the examples above). The value given is the separation radius of each particle of that type. The generated polymer system will have no two overlapping particles.

Examples:

```
init.create_random_polymers(box=data.boxdim(L=35),
                            polymers=[polymer1, polymer2],
                            separation=dict(A=0.35, B=0.35));

init.create_random_polymers(box=data.boxdim(L=31),
                            polymers=[polymer1],
                            separation=dict(A=0.35, B=0.35), seed=52);

# create polymers in an orthorhombic box
init.create_random_polymers(box=data.boxdim(Lx=18,Ly=10,Lz=25),
                            polymers=[polymer2],
                            separation=dict(A=0.35, B=0.35), seed=12345);

# create a triclinic box with tilt factors xy=0.1 xz=0.2 yz=0.3
init.create_random_polymers(box=data.boxdim(L=18, xy=0.1, xz=0.2, yz=0.3),
                            polymers=[polymer2],
                            separation=dict(A=0.35, B=0.35));
```

With all other parameters the same, create_random_polymers will always create the same system if seed is the same. Set a different seed (any integer) to create a different random system with the same parameters. Note that different versions of HOOMD e may generate different systems even with the same seed due to programming changes.

Note: For relatively dense systems (packing fraction 0.4 and higher) the simple random generation algorithm may fail to find room for all the particles and print an error message. There are two methods to solve this. First, you can lower the separation radii allowing particles to be placed closer together. Then setup integrate.nve with the limit option set to a relatively small value. A few thousand time steps should relax the system so that the simulation can be continued without the limit or with a different integrator. For extremely troublesome systems, generate it at a very low density and shrink the box with the command update.box_resize to the desired final size.

Note: The polymer generator always generates polymers as if there were linear chains. If you provide a non-linear bond topology, the bonds in the initial configuration will be stretched significantly. This normally doesn't pose a problem for harmonic bonds (bond.harmonic) as the system will simply relax over a few time steps, but can cause the system to blow up with FENE bonds (bond.fene).

hoomd.deprecated.init.**read_xml**(*filename*, *restart=None*, *time_step=None*, *wrap_coordinates=False*)

Reads initial system state from an XML file

Parameters

- **filename** (*str*) – File to read
- **restart** (*str*) – If it exists, read *restart* instead of *filename*.
- **time_step** (*int*) – (if specified) Time step number to use instead of the one stored in the XML file
- **wrap_coordinates** (*bool*) – Wrap input coordinates back into the box

Deprecated since version 2.0: GSD is the new default file format for HOOMD-blue. It can store everything that an XML file can in an efficient binary format that is easy to access. See *hoomd.init.read_gsd*.

Examples:

```
deprecated.init.read_xml(filename="data.xml")
deprecated.init.read_xml(filename="init.xml", restart="restart.xml")
deprecated.init.read_xml(filename="directory/data.xml")
deprecated.init.read_xml(filename="restart.xml", time_step=0)
system = deprecated.init.read_xml(filename="data.xml")
```

All particles, bonds, etc... are read from the given XML file, setting the initial condition of the simulation. After this command completes, the system is initialized allowing other commands in hoomd to be run.

For restartable jobs, specify the initial condition in *filename* and the restart file in *restart*. init.read_xml will read the restart file if it exists, otherwise it will read *filename*.

All values are read in native units, see *Units* for more information.

If *time_step* is specified, its value will be used as the initial time step of the simulation instead of the one read from the XML file.

If *wrap_coordinates* is set to True, input coordinates will be wrapped into the box specified inside the XML file. If it is set to False, out-of-box coordinates will result in an error.

CHAPTER 19

jit

Details

JIT

The JIT module provides *experimental* support to to JIT (just in time) compile C++ code and call it during the simulation. Compiled C++ code will execute at full performance unlike interpreted python code.

Stability

hoomd.jit is **unstable**. When upgrading from version 2.x to 2.y (y > x), existing job scripts may need to be updated.
Maintainer: Joshua A. Anderson, University of Michigan

New in version 2.3.

Modules

19.1 jit.external

Overview

jit.external.user

Details

class hoomd.jit.external.**user**(*mc*, *code=None*, *llvm_ir_file=None*, *clang_exec=None*)
Define an external field imposed on all particles in the system.

Parameters

- **code** (*str*) – C++ code to compile
- **llvm_ir_fname** (*str*) – File name of the llvm IR file to load.
- **clang_exec** (*str*) – The Clang executable to use

Potentials in jit.external behave similarly to external fields assigned via hpmc.field.callback. Potentials added using external.user are added to the total energy calculation in *hpmc* integrators. The *user* external field takes C++ code, JIT compiles it at run time and executes the code natively in the MC loop at with full performance. It enables researchers to quickly and easily implement custom energetic interactions without the need to modify and recompile HOOMD.

C++ code

Supply C++ code to the *code* argument and *user* will compile the code and call it to evaluate forces. Compilation assumes that a recent `clang` installation is on your PATH. This is convenient when the energy evaluation is simple or needs to be modified in python. More complex code (i.e. code that requires auxiliary functions or initialization of static data arrays) should be compiled outside of HOOMD and provided via the *llvm_ir_file* input (see below).

The text provided in *code* is the body of a function with the following signature:

```
float eval(const BoxDim& box,
unsigned int type_i,
const vec3<Scalar>& r_i,
const quat<Scalar>& q_i
Scalar diameter,
Scalar charge
)
```

- `vec3` and `quat` are is defined in HOOMDMath.h.
- *box* is the system box.
- *type_i* is the particle type.
- *r_i* is the particle position
- *q_i* the particle orientation.
- *diameter* the particle diameter.
- *charge* the particle charge.
- Your code *must* return a value.

Once initialized, the following log quantities are provided to analyze.log:

- **external_field_jit** – total energy of the field

Example:

```
gravity = """return r_i.z + box.getL().z/2;"""
external = hoomd.jit.external.user(mc=mc, code=gravity)
```

LLVM IR code

You can compile outside of HOOMD and provide a direct link to the LLVM IR file in *llvm_ir_file*. A compatible file contains an extern "C" eval function with this signature:

```
float eval(const BoxDim& box, unsigned int type_i, const vec3<Scalar>& r_i, const␣
↪quat<Scalar>& q_i, Scalar diameter, Scalar charge)
```

`vec3` and `quat` is defined in HOOMDMath.h.

Compile the file with clang: `clang -O3 --std=c++11 -DHOOMD_LLVMJIT_BUILD -I /path/to/hoomd/include -S -emit-llvm code.cc` to produce the LLVM IR in `code.ll`.

New in version 2.5.

compile_user(*code*, *clang_exec*, *fn=None*)
Helper function to compile the provided code into an executable

Parameters

- **code** (*str*) – C++ code to compile
- **clang_exec** (*str*) – The Clang executable to use
- **fn** (*str*) – If provided, the code will be written to a file.

New in version 2.3.

disable()
Disables the compute.

Examples:

```
c.disable()
```

Executing the disable command will remove the compute from the system. Any *hoomd.run()* command executed after disabling a compute will not be able to log computed values with *hoomd.analyze.log*.

A disabled compute can be re-enabled with *enable()*.

enable()
Enables the compute.

Examples:

```
c.enable()
```

See *disable()*.

restore_state()
Restore the state information from the file used to initialize the simulations

19.2 jit.patch

Overview

jit.patch.user	Define an arbitrary patch energy.
jit.patch.user_union	Define an arbitrary patch energy on a union of particles

Details

class hoomd.jit.patch.**user**(*mc*, *r_cut*, *code=None*, *llvm_ir_file=None*, *clang_exec=None*)

Define an arbitrary patch energy.

Parameters

- **r_cut** (*float*) – Particle center to center distance cutoff beyond which all pair interactions are assumed 0.
- **code** (*str*) – C++ code to compile
- **llvm_ir_fname** (*str*) – File name of the llvm IR file to load.
- **clang_exec** (*str*) – The Clang executable to use

Patch energies define energetic interactions between pairs of shapes in *hpmc* integrators. Shapes within a cutoff distance of *r_cut* are potentially interacting and the energy of interaction is a function the type and orientation of the particles and the vector pointing from the *i* particle to the *j* particle center.

The *user* patch energy takes C++ code, JIT compiles it at run time and executes the code natively in the MC loop at with full performance. It enables researchers to quickly and easily implement custom energetic interactions without the need to modify and recompile HOOMD.

C++ code

Supply C++ code to the *code* argument and *user* will compile the code and call it to evaluate patch energies. Compilation assumes that a recent `clang` installation is on your PATH. This is convenient when the energy evaluation is simple or needs to be modified in python. More complex code (i.e. code that requires auxiliary functions or initialization of static data arrays) should be compiled outside of HOOMD and provided via the *llvm_ir_file* input (see below).

The text provided in *code* is the body of a function with the following signature:

```
float eval(const vec3<float>& r_ij,
           unsigned int type_i,
           const quat<float>& q_i,
           float d_i,
           float charge_i,
           unsigned int type_j,
           const quat<float>& q_j,
           float d_j,
           float charge_j)
```

- `vec3` and `quat` are defined in HOOMDMath.h.
- *r_ij* is a vector pointing from the center of particle *i* to the center of particle *j*.
- *type_i* is the integer type of particle *i*
- *q_i* is the quaternion orientation of particle *i*
- *d_i* is the diameter of particle *i*
- *charge_i* is the charge of particle *i*
- *type_j* is the integer type of particle *j*
- *q_j* is the quaternion orientation of particle *j*
- *d_j* is the diameter of particle *j*

- *charge_j* is the charge of particle *j*
- Your code *must* return a value.
- When |r_ij| is greater than *r_cut*, the energy *must* be 0. This *r_cut* is applied between the centers of the two particles: compute it accordingly based on the maximum range of the anisotropic interaction that you implement.

Example:

```
square_well = """float rsq = dot(r_ij, r_ij);
                    if (rsq < 1.21f)
                        return -1.0f;
                    else
                        return 0.0f;
                """
patch = hoomd.jit.patch.user(mc=mc, r_cut=1.1, code=square_well)
```

LLVM IR code

You can compile outside of HOOMD and provide a direct link to the LLVM IR file in *llvm_ir_file*. A compatible file contains an extern "C" eval function with this signature:

```
float eval(const vec3<float>& r_ij,
           unsigned int type_i,
           const quat<float>& q_i,
           float d_i,
           float charge_i,
           unsigned int type_j,
           const quat<float>& q_j,
           float d_j,
           float charge_j)
```

`vec3` and `quat` are defined in HOOMDMath.h.

Compile the file with clang: `clang -O3 --std=c++11 -DHOOMD_LLVMJIT_BUILD -I /path/to/hoomd/include -S -emit-llvm code.cc` to produce the LLVM IR in `code.ll`.

New in version 2.3.

compile_user(*code*, *clang_exec*, *fn=None*)

Helper function to compile the provided code into an executable

Parameters

- **code** (*str*) – C++ code to compile
- **clang_exec** (*str*) – The Clang executable to use
- **fn** (*str*) – If provided, the code will be written to a file.

New in version 2.3.

class hoomd.jit.patch.**user_union**(*mc*, *r_cut*, *code=None*, *llvm_ir_file=None*, *r_cut_iso=None*, *code_iso=None*, *llvm_ir_file_iso=None*, *clang_exec=None*)

Define an arbitrary patch energy on a union of particles

Parameters

- **r_cut** (*float*) – Constituent particle center to center distance cutoff beyond which all pair interactions are assumed 0.

- **r_cut_iso** (float, **optional**) – Cut-off for isotropic interaction between centers of union particles
- **code** (*str*) – C++ code to compile
- **code_iso** (str, **optional**) – C++ code for isotropic part
- **llvm_ir_fname** (*str*) – File name of the llvm IR file to load.
- **llvm_ir_fname_iso** (str, **optional**) – File name of the llvm IR file to load for isotropic interaction

Example:

```
square_well = """float rsq = dot(r_ij, r_ij);
                    if (rsq < 1.21f)
                        return -1.0f;
                    else
                        return 0.0f;
                """
patch = hoomd.jit.patch.user_union(r_cut=1.1, code=square_well)
patch.set_params('A',positions=[(0,0,-5.),(0,0,.5)], typeids=[0,0])
```

Example with added isotropic interactions:

```
# square well attraction on constituent spheres
square_well = """float rsq = dot(r_ij, r_ij);
                    if (rsq < 1.21f)
                        return -1.0f;
                    else
                        return 0.0f;
                """

# soft repulsion between centers of unions
soft_repulsion = """float rsq = dot(r_ij, r_ij);
                    if (rsq < 6.25f)
                        return 1.0f;
                    else
                        return 0.0f;
                """

patch = hoomd.jit.patch.user_union(r_cut=1.1, code=square_well, r_cut_iso=5, code_
↪iso=soft_repulsion)
patch.set_params('A',positions=[(0,0,-5.),(0,0,.5)], typeids=[0,0])
```

New in version 2.3.

compile_user(*code*, *clang_exec*, *fn=None*)

Helper function to compile the provided code into an executable

Parameters

- **code** (*str*) – C++ code to compile
- **clang_exec** (*str*) – The Clang executable to use
- **fn** (*str*) – If provided, the code will be written to a file.

New in version 2.3.

CHAPTER 20

metal

Details

Metal potentials.

Stability

hoomd.metal is **unstable**. When upgrading from version 2.x to 2.y (y > x), existing job scripts may need to be updated. **Maintainer:** Lin Yang, Alex Travesset, Iowa State University.

Modules

20.1 metal.pair

Overview

metal.pair.eam	EAM pair potential.

Details

Metal pair potentials.

class hoomd.metal.pair.**eam**(*file*, *type*, *nlist*)
EAM pair potential.

Parameters

- **file** (*str*) – File name with potential tables in Alloy or FS format
- **type** (*str*) – Type of file potential ('Alloy', 'FS')

- **nlist** (*hoomd.md.nlist*) – Neighbor list (default of None automatically creates a global cell-list based neighbor list)

eam specifies that a EAM (embedded atom method) pair potential should be applied between every non-excluded particle pair in the simulation.

No coefficients need to be set for *eam*. All specifications, including the cutoff radius, form of the potential, etc. are read in from the specified file.

Particle type names must match those referenced in the EAM potential file.

Particle mass (in atomic mass) **must** be set in the input script, users are allowed to set different mass values other than those in the potential file.

Two file formats are supported: *Alloy* and *FS*. They are described in LAMMPS documentation (commands eam/alloy and eam/fs) here: http://lammps.sandia.gov/doc/pair_eam.html and are also described here: http://enpub.fulton.asu.edu/cms/potentials/submain/format.htm

Attention: EAM is **NOT** supported in MPI parallel simulations.

Example:

```
nl = nlist.cell()
eam = pair.eam(file='name.eam.fs', type='FS', nlist=nl)
eam = pair.eam(file='name.eam.alloy', type='Alloy', nlist=nl)
```

disable(*log=False*)

Disable the force.

Parameters **log** (*bool*) – Set to True if you plan to continue logging the potential energy associated with this force.

Examples:

```
force.disable()
force.disable(log=True)
```

Executing the disable command will remove the force from the simulation. Any *hoomd.run()* command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with *enable()*.

By setting *log* to True, the values of the force can be logged even though the forces are not applied in the simulation. For forces that use cutoff radii, setting *log=True* will cause the correct *r_cut* values to be used throughout the simulation, and therefore possibly drive the neighbor list size larger than it otherwise would be. If *log* is left False, the potential energy associated with this force will not be available for logging.

enable()

Enable the force.

Examples:

```
force.enable()
```

See *disable()*.

get_energy(*group*)

Get the energy of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the energy for.

Returns The last computed energy for the members in the group.

Examples:

```
g = group.all()
energy = force.get_energy(g)
```

get_net_force(*group*)

Get the force of a particle group.

Parameters **group** (*hoomd.group*) – The particle group to query the force for.

Returns The last computed force for the members in the group.

Examples

g = group.all() force = force.get_net_force(g)

CHAPTER 21

License

```
HOOMD-blue Open Source Software License Copyright (c) 2009-2019 The Regents of
the University of Michigan All rights reserved.

HOOMD-blue may contain modifications ("Contributions") provided, and to which
copyright is held, by various Contributors who have granted The Regents of the
University of Michigan the right to modify and/or distribute such Contributions.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
   may be used to endorse or promote products derived from this software without
   specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

CHAPTER 22

Credits

22.1 HOOMD-blue Developers

The following people contributed to the *hoomd* and *hoomd.md* packages.

Joshua Anderson, University of Michigan - **Lead developer**

Alex Travesset, Iowa State University and Ames Laboratory

Rastko Sknepnek, Northwestern

- integrate.npt
- pair.morse

Carolyn Phillips, University of Michigan

- dihedral.table
- angle.table
- bond.table
- pair.dpdlj
- pair.dpd
- pair.dpd_conservative
- integrate.langevin
- bond.fene
- pair.slj
- Initial testing and debugging of HOOMD on Mac OS X systems

Aaron Keys, University of Michigan

- update.enforce2d and other updates enabling to 2D simulations
- hoomd c++ compilation helper script

- binary restart files
- integrate.mode_minimize_fire

Axel Kohlmeyer, David LeBard, Ben Levine, from the ICMS group at Temple University

- pair.cgcmm
- angle.harmonic
- angle.cgcmm
- dihedral.harmonic
- improper.harmonic
- numerous other small contributions enhancing the usability of HOOMD

Igor Morozov, Andrey Kazennov, Roman Bystryi, Joint Institute for High Temperatures of RAS (Moscow, Russia)

- pair.eam (original implementation)

Philipp Mertmann, Ruhr University Bochum

- charge.pppm
- pair.ewald

Stephen Barr, Princeton University

- charge.pppm
- pair.ewald

Greg van Anders, Benjamin Schultz, University of Michigan

- refactoring of ForceCompute

Eric Irrgang, University of Michigan

- RPM packaging and daily builds

Ross Smith, University of Michigan

- Deb packaging and daily builds

Peter Palm, Jens Glaser, Leipzig University

- group functionality in force.constant
- misc bug fixes
- conversion of bond forces to template evaluator implementation

Jens Glaser, University of Michigan

- integrate.npt anisotropic integration (mkt)
- pair.force_shifted_lj
- Dynamic addition/removal of bonds
- Computation of virial and pressure tensor
- integrate.nph
- Framework for external potentials
- external.periodic

- ParticleData refactoring
- MPI communication
- Optimization of MPI communication for strong scaling
- Neighborlist and pair force performance improvements (multiple threads per particle)
- Enable cell based neighbor list on small boxes
- Testing of angle.table and dihedral.table
- Replicate command
- metadata output
- anisotropic particle integrators
- Gay-Berne pair potential
- pair.reaction_field
- Rewrite of rigid body framework
- Multi-GPU electrostatics (PPPM)
- pair.van_der_waals
- hpmc interaction_matrix
- special_pair framework
- TBB support
- randomize integrator variables
- GPUArray refactoring

Pavani Medapuram, University of Minnesota

- Framework for external potentials
- external.periodic

Brandon D. Smith, University of Michigan

- full double precision compile time option
- integrate.berendsen
- pair.tersoff

Trung Dac Nguyen, University of Michigan

- integrate.nve_rigid
- integrate.bdnvt_rigid
- integrate.nvt_rigid
- integrate.npt_rigid
- integrate.mode_minimize_rigid_fire
- associated rigid body data structures and helper functions
- integrate.nph_rigid

Ryan Marson, University of Michigan

- unwrap_rigid option to dump.dcd

Kevin Silmore, Princeton University

- OPLS dihedral

David Tarjan, University of Virginia

- performance tweaks to the neighbor list and pair force code

Sumedh R. Risbud, James W. Swan, Massachusetts Institute of Technology

- bug fixes for rigid body virial corrections

Michael P. Howard, Princeton University

- Automatic citation list generator
- Neighbor list memory footprint reduction
- Bounding volume hierarchy (tree) neighbor lists
- Stenciled cell list (stencil) neighbor lists
- Per-type MPI ghost layer communication
- Dynamic load balancing
- Wall potentials extrapolated mode
- XML dump by particle group
- Fix references when disabling/enabling objects
- Misc. bug fixes
- CUDA9+V100 compatibility

James Antonaglia, University of Michigan

- pair.mie

Carl Simon Adorf, University of Michigan

- Analyzer callback
- metadata output
- Frenkel-Ladd bug fixes

Paul Dodd, University of Michigan

- pair.compute_energy

Erin Teich, University of Michigan

- addInfo callback to dump.pos

Joseph Berleant, University of Michigan

- fix python 3.4 segfault

Matthew Spellings, University of Michigan

- anisotropic particle integrators
- Gay-Berne, dipole pair potentials
- GTAR file format
- External components in hoomd 2.x

James Proctor, University of Michigan

- Refactor external potential framework
- Wall potentials
- boost python to pybind11 conversion
- boost unit_test to upp11 conversion
- boost signals to Nano::Signals conversion
- Removal of misc boost library calls

Chengyu Dai, University of Michigan

- Rewrite integrate.brownian with 3D rotational updates
- Rewrite integrate.langevin with 3D rotational updates

Isaac Bruss, Chengyu Dai, University of Michigan

- force.active
- update.constraint_ellipsoid

Vyas Ramasubramani, University of Michigan

- init.read_gsd bug fixes
- Reverse communication for MPI

Nathan Horst

- Language and figure clarifying the dihedral angle definition.

Bryan VanSaders, University of Michigan

- constrain.oneD
- Constant stress mode to integrate.npt.
- map_overlaps() in hpmc.
- Torque options to force.constant and force.active

Ludwig Schneider, Georg-August Univeristy Goettingen

- Constant stress flow: hoomd.md.update.mueller_plathe_flow
- Matrix logging and hdf5 logging: hoomd.hdf5.log

Bjørnar Jensen, University of Bergen

- Add Lennard-Jones 12-8 pair potential
- Add Buckingham/exp-6 pair potential
- Add special_pair Coulomb 1-4 scaling

Lin Yang, Alex Travesset, Iowa State University

- metal.pair.eam - reworked implementation

Tim Moore, Vanderbilt University

- angle.cosinesq
- Documentation fixes

Bradley Dice, Avisek Das, University of Michigan

- integrator.randomize_velocities()

Bradley Dice, Simon Adorf, University of Michigan

- SSAGES support

Bradley Dice, University of Michigan

- Documentation improvements

Peter Schwendeman, Jens Glaser, University of Michigan

- NVLINK optimized multi-GPU execution

Alyssa Travitz, University of Michigan

- *get_net_force* implementation

Mike Henry, Boise State University

- Documentation improvements

22.2 HPMC developers

The following people contributed to the `hoomd.hpmc` package.

Joshua Anderson, University of Michigan - Lead developer

- Vision
- Initial design
- Code review
- NVT trial move processing (CPU / GPU)
- Sphere shape
- Polygon shape
- Spheropolygon shape
- Simple polygon shape
- Ellipsoid shape - adaptation of Michael's Ellipsoid overlap check
- 2D Xenocollide implementation
- 2D GJKE implementation
- MPI parallel domain decomposition
- Scale distribution function pressure measurement
- POS writer integration
- Bounding box tree generation, query, and optimizations
- BVH implementation of trial move processing
- SSE and AVX intrinsics
- *jit.patch.user* user defined patchy interactions with LLVM runtime compiled code

Eric Irrgang, University of Michigan

- NPT updater
- Convex polyhedron shape

- Convex spheropolyhedron shape
- 3D Xenocollide implementation
- 3D GJKE implementation
- Move size autotuner (in collaboration with Ben Schultz)
- Densest packing compressor (in collaboration with Ben Schultz)
- POS file utilities (in collaboration with Ben Schultz)
- Shape union low-level implementation
- Sphere union shape (in collaboration with Khalid Ahmed)

Ben Schultz, University of Michigan

- Frenkel-Ladd free energy determination
- Move size autotuner (in collaboration with Eric Irrgang)
- Densest packing compressor (in collaboration with Eric Irrgang)
- POS file utilities (in collaboration with Eric Irrgang)
- Assign move size by particle type
- Ellipsoid overlap check bug fixes

Jens Glaser, University of Michigan

- Patchy sphere shape
- General polyhedron shape
- BVH implementation for countOverlaps
- Hybrid BVH/small box trial move processing
- Helped port the Sphinx overlap check
- Dynamic number of particle types support
- Implicit depletants
- *jit.patch.user_union* user defined patchy interactions with LLVM runtime compiled code
- Geometric Cluster Algorithm implementation
- *convex_spheropolyhedron_union* shape class
- *test_overlap* python API

Eric Harper, University of Michigan

- Misc bug fixes to move size by particle type feature
- Initial code for MPI domain decomposition

Khalid Ahmed, University of Michigan

- Ported the Sphinx overlap check
- Sphere union shape (in collaboration with Eric Irrgang)

Elizabeth R Chen, University of Michigan

- Developed the Sphinx overlap check

Carl Simon Adorf, University of Michigan

- meta data output

Samanthule Nola, University of Michigan

- Run time determination of max_verts

Paul Dodd, Erin Teich, University of Michigan

- External potential framework
- Wall overlap checks
- Lattice external potential

Erin Teich, University of Michigan * Convex polyhedron union particle type

Vyas Ramasubramani, University of Michigan

- hpmc.util.tune fixes for tuning by type
- hpmc.update.boxmc fixes for non-orthorhombic box volume moves
- *jit.external.user* implementation

William Zygmunt, Luis Rivera-Rivera, University of Michigan

- Patchy interaction support in HPMC CPU integrators

22.3 DEM developers

The following people contributed to the `hoomd.dem` package.

Matthew Spellings, University of Michigan - Lead developer Ryan Marson, University of Michigan

22.4 MPCD developers

The following people contributed to the `hoomd.mpcd` package.

Michael P. Howard, Princeton University - Lead developer

- Design
- Cell list and properties
- Particle and cell communication
- Basic streaming method
- SRD and AT collision rules

22.5 Source code

HOOMD: HOOMD-blue is a continuation of the HOOMD project (http://www.ameslab.gov/hoomd/). The code from the original project is used under the following license:

```
Highly Optimized Object-Oriented Molecular Dynamics (HOOMD) Open
Source Software License
Copyright (c) 2008 Ames Laboratory Iowa State University
All rights reserved.

Redistribution and use of HOOMD, in source and binary forms, with or
without modification, are permitted, provided that the following
conditions are met:

* Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

* Neither the name of the copyright holder nor the names HOOMD's
contributors may be used to endorse or promote products derived from this
software without specific prior written permission.

Disclaimer

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND
CONTRIBUTORS ``AS IS''  AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.

IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS  BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
THE POSSIBILITY OF SUCH DAMAGE.
```

Sockets code from VMD is used for the IMDInterface to VMD (http://www.ks.uiuc.edu/Research/vmd/) - Used under the UIUC Open Source License.

Molfile plugin code from VMD is used for generic file format reading and writing - Used under the UIUC Open Source License:

```
University of Illinois Open Source License
Copyright 2006 Theoretical and Computational Biophysics Group,
All rights reserved.

Developed by: Theoretical and Computational Biophysics Group
              University of Illinois at Urbana-Champaign
              http://www.ks.uiuc.edu/

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the Software), to deal with
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to
do so, subject to the following conditions:
```

(continues on next page)

(continued from previous page)

```
Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimers.

Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimers in the documentation
and/or other materials provided with the distribution.

Neither the names of Theoretical and Computational Biophysics Group,
University of Illinois at Urbana-Champaign, nor the names of its contributors
may be used to endorse or promote products derived from this Software without
specific prior written permission.

THE SOFTWARE IS PROVIDED AS IS, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL
THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS WITH THE SOFTWARE.
```

XML parsing is performed with XML.c from http://www.applied-mathematics.net/tools/xmlParser.html - Used under the BSD License:

```
Copyright (c) 2002, Frank Vanden Berghen<br>
All rights reserved.<br>
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

 - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
 - Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
 - Neither the name of the Frank Vanden Berghen nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY
EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE REGENTS AND CONTRIBUTORS BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Saru is used for random number generation - Used under the following license:

```
Copyright (c) 2008 Steve Worley < m a t h g e e k@(my last name).com >

Permission to use, copy, modify, and distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.
```

(continues on next page)

(continued from previous page)

```
THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

Some **CUDA API headers** are included in the HOOMD-blue source code for code compatibility in CPU only builds - Used under the following license:

```
Copyright 1993-2008 NVIDIA Corporation.  All rights reserved.

NOTICE TO USER:

This source code is subject to NVIDIA ownership rights under U.S. and
international Copyright laws.  Users and possessors of this source code
are hereby granted a nonexclusive, royalty-free license to use this code
in individual and commercial software.

NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS,  WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
OR OTHER TORTIOUS ACTION,  ARISING OUT OF OR IN CONNECTION WITH THE USE
OR PERFORMANCE OF THIS SOURCE CODE.

U.S. Government End Users.   This source code is a "commercial item" as
that term is defined at  48 C.F.R. 2.101 (OCT 1995), consisting  of
"commercial computer  software"  and "commercial computer software
documentation" as such terms are  used in 48 C.F.R. 12.212 (SEPT 1995)
and is provided to the U.S. Government only as a commercial end item.
Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
source code with only those rights set forth herein.

Any use of this source code in individual and commercial software must
include, in the user documentation and internal comments to the code,
the above Disclaimer and U.S. Government End Users Notice.
```

FFTs on the CPU reference implementation of PPPM are performed using **kissFFT** from http://sourceforge.net/projects/kissfft/, used under the following license:

```
Copyright (c) 2003-2010 Mark Borgerding

All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
list of conditions and the following disclaimer.
```

(continues on next page)

(continued from previous page)

```
* Redistributions in binary form must reproduce the above copyright notice, this
list of conditions and the following disclaimer in the documentation and/or
other materials provided with the distribution.

* Neither the author nor the names of any contributors may be used to endorse or
promote products derived from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

ModernGPU source code is embedded in HOOMD's package and is used for various tasks: http://nvlabs.github.io/moderngpu/:

```
Copyright (c) 2013, NVIDIA CORPORATION.  All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
    * Neither the name of the NVIDIA CORPORATION nor the
    names of its contributors may be used to endorse or promote products
    derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL NVIDIA CORPORATION BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

CUB 1.4.1 source code is embedded in HOOMD's package and is used for various tasks: http://nvlabs.github.io/cub/:

```
Copyright (c) 2011, Duane Merrill.  All rights reserved.
Copyright (c) 2011-2015, NVIDIA CORPORATION.  All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
```

(continues on next page)

(continued from previous page)

```
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of the NVIDIA CORPORATION nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL NVIDIA CORPORATION BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Eigen 3.2.5 (http://eigen.tuxfamily.org/) is embedded in HOOMD's package and is made available under the Mozilla Public License v.2.0 (http://mozilla.org/MPL/2.0/). Its linear algebra routines are used for dynamic load balancing. Source code is available through the [downloads](http://glotzerlab.engin.umich.edu/hoomd-blue/download.html).

A constrained least-squares problem is solved for dynamic load balancing using **BVLSSolver**, which is embedded in HOOMD's package and is made available under the following license:

```
Copyright (c) 2015, Michael P. Howard. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    1. Redistributions of source code must retain the above copyright
       notice, this list of conditions and the following disclaimer.

    2. Redistributions in binary form must reproduce the above copyright
       notice, this list of conditions and the following disclaimer in the
       documentation and/or other materials provided with the distribution.

    3. Neither the name of the copyright holder nor the names of its
       contributors may be used to endorse or promote products derived from
       this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
OF SUCH DAMAGE.
```

libgetar is used to read and write GTAR files. Used under the MIT license:

```
Copyright (c) 2014-2016 The Regents of the University of Michigan

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
```

(continues on next page)

(continued from previous page)

```
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

pybind11 is used to provide python bindings for C++ classes. Used under the BSD license:

```
Copyright (c) 2016 Wenzel Jakob <wenzel.jakob@epfl.ch>, All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or
upgrades to the features, functionality or performance of the source code
("Enhancements") to anyone; however, if you choose to make your Enhancements
available either publicly, or directly to the author of this software, without
imposing a separate written license agreement for such Enhancements, then you
hereby grant the following license: a non-exclusive, royalty-free perpetual
license to install, use, modify, prepare derivative works, incorporate into
other computer software, distribute, and sublicense such enhancements or
derivative works thereof, in binary and source code form.
```

cereal is used to serialize C++ objects for transmission over MPI. Used under the BSD license:

```
Copyright (c) 2014, Randolph Voorhies, Shane Grant
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of cereal nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL RANDOLPH VOORHIES OR SHANE GRANT BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

22.6 Libraries

HOOMD-blue links to the following libraries:

- python - Used under the Python license (http://www.python.org/psf/license/)
- cuFFT - Used under the NVIDIA CUDA toolkit license (http://docs.nvidia.com/cuda/eula/index.html)

CHAPTER 23

Index

- genindex
- modindex

Python Module Index

h

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z