HALMD ● HAL's MD package

Highly Accelerated Large-scale Molecular Dynamics

Documentation of HAL's MD package

Release 1.0-alpha1

Felix Höfling, Peter Colberg, Nicolas Höft, Michael Kopp

May 26, 2014

1 Introduction **1**
1.1 Physics applications . . . 1
1.2 Features . . . 1
1.3 Historical footnote . . . 2

2 Installation **3**
2.1 Downloading the source code . . . 3
2.2 Software prerequisites . . . 4
2.3 Building *HAL's MD package* . . . 10
2.4 Installation in Ubuntu . . . 12

3 Usage **15**
3.1 Getting started . . . 15
3.2 H5MD data files . . . 16
3.3 Plotting the results . . . 17
3.4 Multi-GPU machines . . . 19

4 Modules **21**
4.1 Simulation . . . 21
4.2 Numeric . . . 49
4.3 Input and Output . . . 50
4.4 Observables . . . 56
4.5 Random Numbers . . . 76
4.6 Utilities . . . 76

5 Simulation units **85**

6 Validation **87**
6.1 Simple fluids . . . 87
6.2 Binary mixtures . . . 88

7 Benchmarks **89**
7.1 Simple Lennard-Jones fluid in 3 dimensions . . . 89
7.2 Supercooled binary mixture (Kob-Andersen) . . . 90

8 FAQ **91**

9 Developer guide **93**
9.1 Development of *HAL's MD package* . . . 93
9.2 Debugging . . . 93

9.3 Floating-point precision . . . 93
9.4 Short guide on Luabind . . . 94
9.5 How to write a HALMD module . . . 98
9.6 Redmine . . . 98
9.7 Coding conventions . . . 99
9.8 Test suite . . . 99

10 Changelog **101**
10.1 Version 0.2.1 . . . 101
10.2 Version 0.2.0 . . . 101
10.3 Version 0.1.2 . . . 101
10.4 Version 0.1.2 . . . 101
10.5 Version 0.1.1 . . . 102
10.6 Version 0.1.0 . . . 102

11 Documentation archive **103**

12 Useful CMake cache variables **105**

13 Useful environment variables for CMake **107**

Python Module Index **109**

Index **111**

CHAPTER

ONE

INTRODUCTION

HAL's MD package is a high-precision molecular dynamics package for large-scale simulations of the complex dynamics in inhomogeneous liquids. It has been specifically designed to support acceleration through CUDA-enabled graphics processors.

HAL's MD package is maintained and developed by Felix Höfling and was initially written together with Peter Colberg. Special credit goes to Nicolas Höft for his many contributions.

Note: A description of the implementation, performance tests, numerical stability tests, and an application to the slow glassy dynamics of the Kob-Andersen mixture is found in the article by Peter H. Colberg and Felix Höfling, Highly accelerated simulations of glassy dynamics using GPUs: Caveats on limited floating-point precision, Comput. Phys. Commun. **182**, 1120 (2011) [arXiv:0912.3824].

1.1 Physics applications

HAL's MD package is designed to study

- the spatio-temporal dynamics of inhomogeneous and complex liquids
- both two- and three-dimensional systems
- particles interacting via many truncated and untruncated pair *potentials* (bonded and external potentials coming soon)
- microcanonical (NVE) and canonical (NVT) ensembles (*Integrators*)
- glass transition, liquid–vapour interfaces, demixing of binary fluids, confined fluids, porous media, ...

1.2 Features

HAL's MD package features

- GPU-acceleration: 1 NVIDIA Kepler K20Xm GPU comparable to 100 CPU cores (*Benchmarks*)
- high performance and excellent numerical long-time stability (e.g., energy conservation)
- user scripts, which define complex simulation protocols
- online evaluation of *observables* including dynamic correlation functions
- structured, compressed, and portable H5MD output files

- extensibility by the generic and modular design
- free availability under GPL license

1.2.1 Technical features

HAL's MD package brings

- an extensive automatic test suite using CMake
- double-single floating-point precision for numerically critical hot spots
- *C^2-smooth potential cutoffs* for improved energy conservation
- an integrated, lightweight Lua interpreter
- template-based C++ code taking advantage of C++11

1.3 Historical footnote

The name *HAL's MD package* was chosen in honour of the machine HAL at the Arnold Sommerfeld Center for Theoretical Physics of the Ludwigs-Maximilians-Universität München. HAL has been the project's first GPGPU machine in 2007, equipped initially with two NVIDIA GeForce 8800 Ultra. HAL survived a critical air condition failure in the server room.

CHAPTER

TWO

INSTALLATION

2.1 Downloading the source code

2.1.1 Cloning the repository

HALMD is maintained in a public Git repository

```
git clone --recursive http://git.halmd.org/halmd.git
```

which is redirected to GitHub. If you prefer the git protocol use

```
git clone --recursive git://github.com/fhoefling/halmd.git
```

This will create a directory `halmd`, which holds a hidden copy of the repository and a working copy of the source files.

2.1.2 Selecting a release version

By default, the above command yields the tip of the development branch. A specific release version is checked out by

```
git checkout TAG
```

where `TAG` is a valid release tag as listed from

```
git tag -n3
```

2.1.3 Git tutorials

If you are new to Git or version control in general, the Git tutorial will get you started.

Former Subversion users may also read the Git SVN Crash Course.

For in-depth documentation, see the Git User's Manual.

2.2 Software prerequisites

2.2.1 Automatic installation

This guide describes an easy and automated way of installing all packages required for building HALMD. To find out more about the required packages, see *Software prerequisites*; details of the installation process are given in *Manual installation*.

Quick Start Guide

Create and change to a new directory (preferably on a local disk):

```
mkdir /tmp/halmd_prerequisites && cd /tmp/halmd_prerequisites
```

Download, compile (with 4 processes) and install required packages to `~/opt`:

```
nice make -f .../halmd/examples/packages.mk CONCURRENCY_LEVEL=4 install
```

Add packages to shell environment and set the `CMAKE_PREFIX_PATH` variable in particular:

```
make -f .../halmd/examples/packages.mk env >> ~/.bashrc
```

That was easy!

A more thorough look at packages.mk

The makefile `packages.mk` provides many more rules than just `install`:

```
make -f .../halmd/examples/packages.mk TAB TAB
```

```
build              clean-luabind      env-boost          fetch-boost
build-boost        configure-boost    env-cmake          fetch-cmake
build-cmake        configure-cmake    env-hdf5           fetch-hdf5
build-hdf5         configure-hdf5     env-lua            fetch-lua
build-lua          distclean          env-luabind        fetch-luabind
build-luabind      distclean-boost    extract-boost      install
clean              distclean-cmake    extract-cmake      install-boost
clean-boost        distclean-hdf5     extract-hdf5       install-cmake
clean-cmake        distclean-lua      extract-lua        install-hdf5
clean-hdf5         distclean-luabind  extract-luabind    install-lua
clean-lua          env                fetch              install-luabind
```

You may choose to install only selected dependencies:

```
make -f .../halmd/examples/packages.mk install-boost install-cmake
```

To compile and install to a path other than `~/opt`:

```
make -f .../halmd/examples/packages.mk install PREFIX=~/pkg/debian6.0-x86_64
```

If you wish to first download all packages:

```
make -f .../halmd/examples/packages.mk fetch
```

To remove all package build directories:

```
make -f .../halmd/examples/packages.mk clean
```

Also remove downloaded tarballs and patches:

```
make -f .../halmd/examples/packages.mk distclean
```

To compile as a non-root user and install as root:

```
make -f .../halmd/examples/packages.mk PREFIX=/opt
sudo make -f .../halmd/examples/packages.mk install PREFIX=/opt
```

Troubleshooting

There are some requirements to ensure a smooth run of packages.mk:

- a recent C++ compiler (e.g., GCC)
- some standard tools: wget, tar, gzip, rm, cp, touch, patch
- Boost requires a development package of the bzip2 library (files `bzlib.h` and `libbz2.so`)

The bzip2 library is necessary for Boost.IOStreams. As HALMD does not make use of this library, you may opt to compile Boost without bzip2 support by prepending `NO_BZIP2=1` to the make command.

On a Debian system, the following packages are required:

```
apt-get install build-essential zlib1g-dev libbz2-dev unzip libreadline6-dev
```

2.2.2 Manual installation

This section is a step-by-step guide for manual installation of the required build dependencies of HALMD. Be sure to check if your distribution ships with any of these packages before attempting to compile them yourself. Before proceeding, be aware of the section *Automatic installation*.

Tip: When installing third-party packages, it is advisable to put them into separate directories. If you install software only for yourself, use package directories of the form `~/opt/PKGNAME-PKGVERSION`, for example `~/opt/boost-1.47.0` or `~/opt/Sphinx-1.0.4`. If you install software system-wide as the root user, use `/opt/PKGNAME-PKGVERSION`. This simple scheme allows you to have multiple versions of a package, or remove a package without impacting others.

When initially creating the CMake build tree, include all third-party package directories in the environment variable `CMAKE_PREFIX_PATH`. For example, if Boost and Lua are installed in your home directory, CUDA is installed system-wide, and the HALMD source is in `~/projects/halmd`, the initial cmake command might look like this

```
CMAKE_PREFIX_PATH=~/opt/boost_1_47_0:/opt/cuda-3.1:~/opt/lua-5.1.4 cmake ~/projects/halm
```

Instead of setting CMAKE_PREFIX_PATH manually, you would include the package directories in your ~/.bashrc (or your favourite shell's equivalent)

```
export CMAKE_PREFIX_PATH="${HOME}/opt/boost_1_47_0${CMAKE_PREFIX_PATH+:$CMAKE_PREFIX_PAT
export CMAKE_PREFIX_PATH="/opt/cuda-3.1${CMAKE_PREFIX_PATH+:$CMAKE_PREFIX_PATH}"
export CMAKE_PREFIX_PATH="${HOME}/opt/lua-5.1.4${CMAKE_PREFIX_PATH+:$CMAKE_PREFIX_PATH}"
```

GNU/Linux

CMake

The build process of HALMD depends on CMake, a cross-platform, open-source build system.

Get the latest CMake source package, currently CMake 2.8.5.

Get the latest CMake-CUDA patch, currently CMake-CUDA 2.8.5-0.

Extract the CMake source package, and apply the patches in the CMake source directory with

```
patch -p1 < ../cmake-cuda-2.8.5-0-gda84f60.patch
```

Prepare the CMake build with

```
./configure --prefix=$HOME/opt/cmake-2.8.5
```

Compile CMake with

```
make
```

Install CMake into your packages directory:

```
make install
```

Include CMake in your shell environment, by adding to ~/.bashrc:

```
export PATH="${HOME}/opt/cmake-2.8.5/bin${PATH+:$PATH}"
export MANPATH="${HOME}/opt/cmake-2.8.5/man${MANPATH+:$MANPATH}"
```

Boost C++ libraries

Get the latest Boost source package, currently Boost 1.47.0.

Get the latest Boost.Log source package from the upstream repository.

Note: The Boost.Log library is a proposed extension to the Boost C++ libraries. As a result of the formal review of Boost.Log in March 2010, the library has been accepted subject to several conditions. It is not shipped yet with upstream Boost.

We will build Boost and Boost.Log in a single step, therefore extract both source packages and copy the Boost.Log headers and library sources to the Boost source directory using

```
cp -r boost-log/boost/log boost_1_47_0/boost/
cp -r boost-log/libs/log boost_1_47_0/libs/
```

In the Boost source directory, bootstrap the build with

```
./bootstrap.sh
```

Compile Boost using

```
./bjam cxxflags=-fPIC
```

This compiles both dynamic and static libraries.

Note: By default, CMake uses the dynamically linked Boost libraries.

This is the recommended way of linking to Boost, as static linking of the unit test executables significantly increases the size of the build tree. If you wish to link statically nevertheless, for example to run a program on another machine without your Boost libraries, invoke cmake with `-DBoost_USE_STATIC_LIBS=True` on the *first* run.

Warning: Boost may require more than fifteen minutes to compile.
You are strongly advised to take a coffee break.

Install the Boost libraries into your packages directory:

```
./bjam cxxflags=-fPIC install --prefix=$HOME/opt/boost_1_47_0
```

Include Boost in your shell environment, by adding to ~/.bashrc:

```
export CMAKE_PREFIX_PATH="${HOME}/opt/boost_1_47_0${CMAKE_PREFIX_PATH+:$CMAKE_PREFIX_PAT
export LD_LIBRARY_PATH="${HOME}/opt/boost_1_47_0/lib${LD_LIBRARY_PATH+:$LD_LIBRARY_PATH}
```

Lua interpreter

Get the latest Lua source package from the Lua download page, currently Lua 5.1.4.

Get the Lua 5.1.4-2 patch fixing several bugs.

Extract the Lua source package, and apply the patch in the Lua source directory with

```
cd lua-5.1.4/src
patch < ../../patch-lua-5.1.4-2
```

The recommended way of embedding the Lua intepreter in an executable is to link the Lua library statically, which is the default mode of compilation.

On **32-bit platforms**, compile the Lua library with

```
make linux
```

On **64-bit platforms**, include the `-fPIC` flag using

```
make linux CFLAGS="-DLUA_USE_LINUX -fPIC -O2 -Wall"
```

Install the Lua library into your packages directory:

```
make install INSTALL_TOP=~/opt/lua-5.1.4
```

Include Lua in your shell environment, by adding to ~/.bashrc:

```
export CMAKE_PREFIX_PATH="${HOME}/opt/lua-5.1.4${CMAKE_PREFIX_PATH+:$CMAKE_PREFIX_PATH}"
export PATH="${HOME}/opt/lua-5.1.4/bin${PATH+:$PATH}"
export MANPATH="${HOME}/opt/lua-5.1.4/man${MANPATH+:$MANPATH}"
```

HDF5 library

Get the latest HDF5 source package, currently HDF5 1.8.6.

Prepare a statically linked build of the HDF5 C and C++ library with

```
CFLAGS=-fPIC CXXFLAGS=-fPIC ./configure --enable-cxx --enable-static --disable-shared --
```

Note: Compiling HDF5 with C++ support disables multi-threading.

Compile HDF5 using

```
make
```

Install the HDF5 libraries into your packages directory:

```
make install
```

Include HDF5 in your shell environment, by adding to ~/.bashrc:

```
export PATH="${HOME}/opt/hdf5-1.8.6/bin${PATH+:$PATH}"
export CMAKE_PREFIX_PATH="${HOME}/opt/hdf5-1.8.6${CMAKE_PREFIX_PATH+:$CMAKE_PREFIX_PATH}
```

Sphinx documentation generator

Get the latest Sphinx source package, currently Sphinx 1.0.7.

Query your Python version

```
python -V
```

Create a package directory for Sphinx using the Python major and minor version

```
mkdir -p $HOME/opt/Sphinx-1.0.7/lib/python2.5/site-packages
```

Add the package directory to the PYTHON_PATH environment variable

```
export PYTHONPATH="${HOME}/opt/Sphinx-1.0.7/lib/python2.5/site-packages${PYTHONPATH+:$PY
```

Install Sphinx into your packages directory

```
python setup.py install --prefix=$HOME/opt/Sphinx-1.0.7
```

Include Sphinx in your shell environment, by adding to ~/.bashrc:

```
export PATH="${HOME}/opt/Sphinx-1.0.7/bin${PATH+:$PATH}"
export PYTHONPATH="${HOME}/opt/Sphinx-1.0.7/lib/python2.5/site-packages${PYTHONPATH+:$PY
```

AIX

Boost

Compile and install Boost using

```
./bjam --toolset=vacpp address-model=64 cxxflags=-qrtti=all install --prefix=$HOME/opt/p
```

Lua

Compile the Lua library

```
make aix
```

HDF5

Prepare a statically linked build of the HDF5 C and C++ library with

```
CC=xlc_r CXX=xlC_r CXXFLAGS=-qrtti=all ./configure --enable-cxx --enable-static --disabl
```

The following software packages are required for building HALMD. For an automated installation procedure, refer to the next section, *Automatic installation*. A detailed step-by-step guide for manual installation is given in section *Manual installation*.

- NVIDIA CUDA toolkit $\geq$ 4.1

  Warning: CUDA driver $\leq$ 4.1 has a known security vulnerability (CVE-2012-0946). For this reason, we recommend CUDA 4.2 or later.

  Please refer to the installation instructions shipped with the toolkit.

- CMake $\geq$ 2.8.8 with a patch for native CUDA support

  The build process of HALMD depends on CMake, a cross-platform, open-source build system.

  Note: The CMake-CUDA patch adds *native* CUDA source file compilation and linking support to CMake and is not to be confused nor compatible with the CUDA module in CMake 2.8.

- Git $\geq$ 1.5.6.2

  The source code of HALMD is managed by Git, a fast and efficient, distributed version control system. Git is available for many operating systems and their flavours.

- Boost C++ Libraries $\geq$ 1.49.0

  The C++ part of HALMD uses of libraries in the Boost C++ collection.

- Lua interpreter $\geq$ 5.1

  Note: We recommend Lua 5.2 or later.

  A simulation with HALMD is setup and configured by means of the Lua scripting language. The fast and lightweight Lua interpreter is embedded in the HALMD executable.

- HDF5 C++ Library >= 1.8

  "HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data."

To optionally generate documentation in HTML and PDF format:

- Sphinx documentation generator $\geq$ 0.6.1

  "Sphinx is a tool that makes it easy to create intelligent and beautiful documentation."

- LaTeX including pdflatex and dvipng
- graphviz

2.3 Building *HAL's MD package*

2.3.1 Compilation and Installation

HALMD uses CMake to generate its make files, which is similar to the more commonly used Autotools suite recognisable by the `configure` script accompanying a software package, but much faster and much easier to develop with.

With cmake, out-of-tree compilation is preferred, so we generate the compilation or build tree in a separate directory. This allows one to have multiple builds of the same software at the same time, for example a release build with aggressive optimisation and a debug build including debugging symbols. Note that the build directory may be a subdirectory in the source tree.

Setting up the build tree

In the cloned HALMD repository, switch to a new build directory:

```
mkdir -p build/release && cd build/release
```

If the third-party packages are installed in standard locations, run

```
cmake ../..
```

This will detect all necessary software, and then generate the make files. If third-party packages are not found in standard locations, make sure to correctly set the environment variable `CMAKE_PREFIX_PATH`, see *Software prerequisites*.

Compilation

Compilation is done using make, which supports parallel builds:

```
nice make -j4
```

The default installation directory is `/usr/local`, which may be adjusted by invoking

```
cmake -DCMAKE_INSTALL_PREFIX=$HOME/opt/halmd-version ../..
```

For compilation and subsequent installation type:

```
nice make -j4 install
```

Further CMake configuration

Compilation flags may be configured via CMake's text mode interface:

```
ccmake .
```

To finish configuration, hit "c" and "g" to apply and recompile with make. Alternatively, you may use CMake's graphical interface:

```
cmake-gui .
```

The following switch displays the actual commands invoked by make:

```
CMAKE_VERBOSE_MAKEFILE        ON
```

An installation prefix may be specified as following:

```
CMAKE_INSTALL_PREFIX          /your/home/directory/usr
```

The compiled program is then installed into this tree by

```
nice make -j4 install
```

Updating the build tree

After checking out to a different version (or more recent Git commit), **switch to the build directory** (e.g., `build/release`) and run:

```
cmake .
```

This instructs CMake to regenerate the build tree using the configuration from the previous run of CMake. Then compile with `make` as usual.

Setting build parameters

Parameters may be passed to cmake as environment variables or cache variables.

Environment variables are prepended to the cmake command:

```
CXXFLAGS="-fPIC -Wall" cmake ../..
```

Useful environment variables for CMake

Cache variables are appended using the -D option:

```
cmake -DCMAKE_BUILD_TYPE=Release ../..
```

Useful CMake cache variables

The following example demonstrates how to compile separate, dynamically linked executables for each backend, which are statically linked to all libraries except the standard C and C++ libraries:

```
CXXFLAGS="-fPIC -Wall"
NVCCFLAGS="-Xcompiler -fPIC -Xptxas -v -arch sm_12" \
cmake \
    -DCMAKE_BUILD_TYPE=Release \
    ../..
```

The options given here correspond to the default values.

2.3.2 Testing

HALMD includes an extensive, preliminary test suite, which may be started in the build tree by

```
ctest
```

2.3.3 Supported compilers

HALMD requires a C++ compiler with sufficient C++11 support. It is known to compile with the following compilers.

- GCC 4.8
  - GCC 4.8.2 (upstream) on openSuSE 12.3 (x86_64)
- GCC 4.7
  - GCC 4.7.3 (upstream) on SuSE Linux 11 SP2 (x86_64)
  - GCC 4.7.2 on openSuSE 12.3 (x86_64)
- GCC 4.6
  - GCC 4.6.2 on openSuSE 12.1 (x86_64)
- Clang 3.2
  - Clang 3.2 (upstream)
- Intel C++ compiler
  - Intel C++ compiler 13.1

The following C++ compilers **fail** to compile HALMD.

- GCC $\leq$ 4.5
- Clang $\leq$ 2.7
  - Clang 2.7 on Debian GNU/Linux squeeze (x86_64)
- Intel C++ compiler $\leq$ 12.1
- XL C++

2.4 Installation in Ubuntu

In the following a quick step-by-step guide how to download and install HALMD in Ubuntu will be given.

2.4.1 Ubuntu 14.04 LTS

Prepare installation

First, make sure you have all required packages installed:

```
sudo apt-get install git build-essential libluajit-5.1-dev zlib1g-dev wget nvidia-cuda-t
```

Clone the HALMD source code repository:

```
git clone --recursive http://git.halmd.org/halmd.git
```

Build and install prerequisites

Then you will have to build boost from source using `examples/packages.mk`:

```
mkdir halmd-prerequisites
cd halmd-prerequisites
nice make CONCURRENCY_LEVEL=4 -f ../halmd/examples/packages.mk install-boost install-hdf
```

This step is required as using the according packages from the Ubuntu repository will not work—all these packages need special build options not provided in the official packages. Note that this installation will require some time but you need to do this only once. A more detailed description of the package installation can be found in *Automatic installation*.

After executing the above commands, the necessary packages will be installed in `~/opt`. In order to make these packages available for the subsequent build tools, run

```
source <(make -f ../halmd/examples/packages.mk env-boost env-hdf5 env-cmake)
```

Alternatively, you may append the output to your `~/.bashrc`. You can verify that everything went well by running `cmake --version` which should output something like `cmake version 2.8.12.1 with native CUDA support` (the important part is `with native CUDA support`).

Build and install HALMD

Now, we can start building HALMD. First, create a build directory (this can be anywhere, for convenience we create it in our home directory) and then run cmake to generate the necessary Makefiles

```
mkdir ~/halmd-build && cd ~/halmd-build
cmake ~/halmd -DCMAKE_INSTALL_DIRECTORY=~/opt/halmd/
```

There may be warnings now about missing packages (e.g. Sphinx) but this is not essential now as it is only required to build the manual page. If you are building with CUDA support, make sure that the CUDA compiler has been detected and works. If there was a problem and you were able to fix this, it may be necessary to remove the build directory completely and rerun `cmake ~/halmd` afterwards.

You are now ready to build HALMD. Execute

```
nice make -j4
```

and the build process will start.

Note: The build process is very memory hungry, consider reducing the number of parallel builds to a lower number (i.e. use `-j2` instead of `-j4` if you want 2 parallel builds) if you experience problems.

If successful, you can run `./halmd/halmd --version` from the build directory and this should give you simple version information about HALMD.

If you want to install HALMD, simply run `make install` from the build directory. In order to be able to run halmd from everywhere, run

```
echo "export PATH=\"${HOME}/opt/halmd/bin\${PATH+:\$PATH}\"" >> ~/.bashrc
```

and log out and in again.

You can test now a simple example by running

```
halmd ~/opt/halmd/share/doc/halmd/examples/liquid/lennard_jones_equilibration.lua -v
```

You may now clean-up the build directories `halmd-prerequisites` and `halmd-build`.

CHAPTER

THREE

USAGE

3.1 Getting started

HAL's MD package is configured and steered with customisable simulation scripts written in Lua 5. For a quick start refer to one of the "liquid" scripts found in `share/doc/halmd/examples` in the installation directory.

3.1.1 Program parameters

HAL's MD package brings a command line parser which allows one to define script parameters. The possible command line options are described in the help:

```
halmd script.lua --help
```

3.1.2 Example: a Lennard-Jones fluid

Let us consider a simple fluid with 20,000 Lennard-Jones particles at density $\rho^* = 0.8$. Equilibration is done with a Boltzmann thermostat at temperature $T^* = 2$ over 10,000 steps

```
halmd liquid/lennard_jones_equilibration.lua -v \
  --timestep 0.005 --time 50 \
  --density 0.8 --particles 20000 \
  --temperature 2 \
  --sampling state-vars=100
```

Many parameters have sensible default values and may be omitted, e.g, the collision rate of the thermostat (0.1), or the cutoff radius of the potential ($r_c = 2^{1/6}\sigma$ corresponding to a purely repulsive potential). The option `-v` makes the output more verbose, check that your CUDA device has beed detected properly.

The system state is written at the beginning and the end of the simulation if not specified differently. The initial configuration of the particles is an fcc lattice. The default output settings yield an H5MD file with a time stamp in its name, `lennard_jones_equilibration_%Y%m%d_%H%M%S.h5` and a corresponding log file.

We may now continue the simulation at constant energy by resuming from the H5MD file using the accompanying script

```
halmd liquid/lennard_jones.lua -v \
  --timestep 0.001 --time 100 \
  --trajectory output_from_previous_run.h5 \
  --sampling state-vars=1000
```

This will continue the simulation over 10^5 steps and write observables like thermodynamic state variables every 1000 steps (potential energy, instantaneous "temperature", pressure, ...)

3.1.3 Inspection of the results

If the HDF5 tools are properly installed, you may have a quick overview of the output file

```
h5ls output.h5
```

or look at a specific data set

```
h5dump -d observables/potential_energy output.h5 | less
```

For a more advanced inspection and analysis of the HDF5 output files, see *Plotting the results*. You may try the exemplary script

```
plotting/plot_h5md.py output.h5
```

You may also have a look at the H5MD tools, a collection of analysis and plot scripts.

3.2 H5MD data files

3.2.1 Why H5MD?

The H5MD file format presents a unique standard to store data for and from molecular simulations along with derived quantities such as physical observables.

H5MD builds on the technology of the "Hierarchical Data Format 5" (HDF5) , which is a well established scientific file format, with bindings for C, C++, Fortran, Python and support by Matlab, Mathematica, ... An excellent overview is found in the documentation of the project HDF5 for Python.

Note: The output files of *HAL's MD package* comply with H5MD version 1.0, published in

P. de Buyl, P. H. Colberg, and F. Höfling, H5MD: a structured, efficient, and portable file format for molecular data, Comput. Phys. Commun. (2014), in press, [arXiv:1308.6382]

3.2.2 Working with HDF5 files

Using the h5ls/h5dump tools

For a quick analysis of HDF5 data files, use the `h5ls` tool (bundled with the HDF5 library):

```
h5ls -v file
```

Alternatively, the structure of a file may be inspected with the `h5dump` tool:

```
h5dump -A file
```

The contents of individual groups or datasets may be displayed as follows:

```
h5dump -g /path/to/group file
h5dump -d /path/to/dataset file
```

Using Python and h5py

h5py is a Python module wrapping the HDF5 library. It is based on NumPy, which implements a MATLAB-like interface to multi-dimensional arrays. This is where the H5MD format reveals its true strength, as NumPy allows arbitrary transformations of HDF5 datasets, all while using a real programming language.

As a simple example, we open a H5MD file and print a dataset:

```
import h5py
f = h5py.File("file", "r")
d = f["path/to/dataset"]
print d
print d[0:5]
f.close()
```

Attributes may be read with the attrs class member:

```
print f["h5md"].attrs["version"]
if "observables" in f.keys():
    print f["observables"].attrs["dimension"]
```

For further information, refer to the Numpy and Scipy Documentation and the HDF5 for Python Documentation.

3.3 Plotting the results

Convenient and powerful access to the HDF5 output files is provided by the Python packages h5py together with PyLab. An examplary Python script for accessing, post-processing and plotting the output data of HALMD is provided in the sources at `examples/plotting/plot_h5md.py`; it requires H5Py $\geq$ 2.0.1.

The various aspects of the script are detailed in the following. It starts with loading some packages, defining command line options, and opening the HDF5 output file.

```
import argparse
import h5py
from numpy import *
from pylab import *

def main():
    # define and parse command line arguments
    parser = argparse.ArgumentParser(prog='plot_h5md.py')
    parser.add_argument('--range', type=int, nargs=2, help='select range of data points'
    parser.add_argument('--dump', metavar='FILENAME', help='dump plot data to filename')
    parser.add_argument('--no-plot', action='store_true', help='do not produce plots, bu
    parser.add_argument('input', metavar='INPUT', help='H5MD input file with data for st
    args = parser.parse_args()

    # evaluate option --range
    range = args.range or [0, -1]

    # open and read data file
    H5 = h5py.File(args.input, 'r')
    H5param = H5['parameters']
    H5obs = H5['observables']
```

The script shows how to extract some of the simulation parameters that are stored along with each HDF5 output file.

```
    # print some details
    print 'Particles: {0:s}'.format(H5param['box'].attrs['particles'])
    print 'Interaction potential: {0:s}'.format(H5param.attrs['potential'])
    if H5param.attrs['force'] == 'pair_trunc':
        print 'Potential cutoff: {0:s}'.format(H5param[H5param.attrs['potential']].attrs
    print 'Box size:',
    for x in H5param['box'].attrs['length']:
        print ' {0:g}'.format(x),
    print
    print 'Density: {0:g}'.format(float(H5param['box'].attrs['density']))
```

It illustrates how to computes the average temperature, pressure, and potential energy over the whole simulation run or just over a selected range of data points, i.e., a time window.

```
    # compute and print some averages
    # the simplest selection of a data set looks like this:
    #     temp, temp_err = compute_average(H5obs['temperature'], 'Temperature')
    # full support for slicing (th second pair of brackets) requires conversion to a Num
    temp, temp_err = compute_average(array(H5obs['temperature/sample'])[range[0]:range[1
    pressure, pressure_err = compute_average(array(H5obs['pressure/sample'])[range[0]:ra
    epot, epot_err = compute_average(array(H5obs['potential_energy/sample'])[range[0]:ra

def compute_average(data, label, nblocks = 10):
    """ compute and print average of data set
        The error is estimated from grouping the data in shorter blocks """

    # group data in blocks, discard excess data
    data = reshape(data[:(nblocks * (data.shape[0] / nblocks))], (nblocks, -1))

    # compute mean and error
    avg = mean(data)
    err = std(mean(data, axis=1)) / sqrt(nblocks - 1)
    print '{0:s}: {1:.4f} ± {2:.2g}'.format(label, avg, err)

    # return as tuple
    return avg, err
```

Eventually, the script can dump the fluctuating potential energy as function of time to a text file

```
    # select data for plotting the potential energy as function of time
    x = array(H5obs['potential_energy/time'])[range[0]:range[1]]
    y = array(H5obs['potential_energy/sample'])[range[0]:range[1]]

    # append plot data to file
    if args.dump:
        f = open(args.dump, 'a')
        print >>f, '# time   E_pot(t)'
        savetxt(f, array((x, y)).T)
        print >>f, '\n'
        f.close()
```

or directly generate a figure from these data

```
    # generate plot
    if not args.no_plot:
```

```
# plot potential energy versus time
plot(x, y, '-b', label=args.input)

# plot mean value for comparison
x = linspace(min(x), max(x), num=50)
y = zeros_like(x) + epot
plot(x, y, ':k')

# add axes labels and finalise plot
axis('tight')
xlabel(r'Time $t$')
ylabel(r'Potential energy $E_\mathrm{pot}$')
legend(loc='best', frameon=False)
show()
```

3.4 Multi-GPU machines

To distribute multiple HALMD processes among CUDA devices in a single machine, the CUDA devices have to be locked exclusively by the respective process. HALMD will then choose the first available CUDA device, or an available device in the subset given by the `--device` option.

3.4.1 nvidia-smi tool

If your NVIDIA driver version comes with the nvidia-smi tool, set all CUDA devices to *compute exclusive mode* to restrict use to one process per device:

```
sudo nvidia-smi --gpu=0 --compute-mode-rules=1
sudo nvidia-smi --gpu=1 --compute-mode-rules=1
```

Warning: Compute exclusive mode seems to work reliably only with NVIDIA Tesla devices. Although NVIDIA GeForce cards may be set to compute exclusive mode as well, doing so might occasionally cause a system crash.

3.4.2 nvlock tool

If your NVIDIA driver version does not support the nvidia-smi tool, or if you wish not to set the devices to compute exclusive mode, the `nvlock` tool may be used to exclusively assign a GPU to each process:

```
nvlock halmd [...]
```

You may also directly use the preload library:

```
LD_PRELOAD=libnvlock.so halmd [...]
```

nvlock is available at

```
https://github.com/fhoefling/nvcuda-tools
```

and is compiled with

```
cmake .
make
```

CHAPTER

FOUR

MODULES

4.1 Simulation

4.1.1 Binning

This module implements the method of cell lists. It splits up the simulation box into smaller boxes and assigns each particle into one sub-box. This enables faster look-up for particles that interact with a cut-off potential.

class halmd.mdsim.**binning**(*args*)

Construct binning module instance.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of halmd.mdsim.particle
- **args.box** – instance of halmd.mdsim.box
- **args.r_cut** (*table*) – cutoff radius matrix for the potentials
- **args.skin** (*number*) – neighbour list skin *(default: ''0.5'')*
- **args.occupancy** (*number*) – initial cell occupancy *(GPU variant only, default: ''0.5'')*

r_cut

Cut-off radius matrix for the particle interactions.

r_skin

"Skin" of the particle. This is an additional distance ratio added to the cutoff radius for the minimal edge lengths of the cells.

particle

Instance of halmd.mdsim.particle.

disconnect()

Disconnect binning module from profiler.

4.1.2 Box

The box module keeps the edge lengths of the simulation box, and implements periodic boundary conditions for use in other modules. At present the module supports cuboid box geometries.

Example:

```
local box = halmd.mdsim.box({length = {100, 100, 10}})
```

class halmd.mdsim.**box**(*args*)

Construct box.

Parameters

- **args** (*table*) – keyword arguments
- **args.edges** (*table*) – sequence of edge vectors (parallelepiped)
- **args.length** (*table*) – sequence of edge lengths (cuboid)

Returns instance of box

Warning: Non-cuboid geometries are not implemented, `edges` must be a diagonal matrix.

dimension

Space dimension of the simulation box as a number.

length

Edge lengths as a sequence.

volume

Box volume.

edges()

Returns the edge vectors as a matrix.

For a cuboid simulation domain, `edges` is equal to

```
{{length[1], 0, 0},
 {0, length[2], 0},
 {0, 0, length[3]}}
```

origin()

Returns the coordinates of the lowest corner of the box.

writer(*args*)

Write box specification to file.

http://nongnu.org/h5md/draft.html#box-specification

Parameters

- **args** (*table*) – keyword arguments
- **args.writer** – instance of group writer (optional)
- **args.file** – instance of H5MD file (optional)
- **args.location** (*string table*) – location within file (optional)

Returns instance of group writer

If the argument `writer` is present, a box varying in time is assumed and the box data are written as a time series upon every invocation of writer:write(). Sharing the same writer instance enables hard-linking of the datasets `step` and `time`.

Otherwise, a `file` instance along with a `location` must be given. The current box information is immediately written to the subgroup `"box"` as time-independent data.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings, the group name `"box"` is appended.

`halmd.mdsim.box.`**`reader`**(*args*)

Read edge vectors of simulation domain from H5MD file.

http://nongnu.org/h5md/draft.html#box-specification

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of `halmd.io.readers.h5md`
- **args.location** – location of box group within file

Returns

edge vectors

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings, the group name `"box"` is appended.

The return value may be used to restore the domain:

```
local file = readers.h5md({path = args.trajectory})
local edges = mdsim.box.reader({file = file, location = {"particles", "all"}})
local box = mdsim.box({edges = edges})
```

4.1.3 Clock

The clock tracks simulation step and time, and defines the integration time step.

Example:

```
local clock = require("halmd.mdsim.clock")
clock.on_set_timestep(function(timestep)
    print(("time step has changed to %g"):format(timestep))
end)
```

`halmd.mdsim.clock.`**`step`**

Current simulation step.

`halmd.mdsim.clock.`**`time`**

Current simulation time in MD units.

`halmd.mdsim.clock.`**`timestep`**

Integration time step in MD units.

`halmd.mdsim.clock.`**`set_timestep`**(*timestep*)

Define integration time step.

The value of the time step is propagated to all integrators.

`halmd.mdsim.clock.`**`on_set_timestep`**(*slot*)

Connect a unary slot that accepts the integration time step.

This slot is called after setting the time step with `set_timestep()`.

4.1.4 Core

The simulation core drives the MD step.

halmd.mdsim.core.**mdstep**()
Perform a single MD integration step.

This method is invoked by halmd.observables.sampler.run().

halmd.mdsim.core.**on_prepend_integrate**(*slot*)
Connect nullary slot to signal.

Returns signal connection

halmd.mdsim.core.**on_integrate**(*slot*)
Connect nullary slot to signal.

Returns signal connection

halmd.mdsim.core.**on_append_integrate**(*slot*)
Connect nullary slot to signal.

Returns signal connection

halmd.mdsim.core.**on_prepend_finalize**(*slot*)
Connect nullary slot to signal.

Returns signal connection

halmd.mdsim.core.**on_finalize**(*slot*)
Connect nullary slot to signal.

Returns signal connection

halmd.mdsim.core.**on_append_finalize**(*slot*)
Connect nullary slot to signal.

Returns signal connection

4.1.5 Maximum Displacement

This module monitors the maximum displacement of particles with regard to a certain start position. As this is only needed for the halmd.mdsim.neighbour module, it exports no functions and direct construction is not necessary.

class halmd.mdsim.**max_displacement**(*args*)
Construct Maximum Displacement module

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of halmd.mdsim.particle
- **args.box** – instance of halmd.mdsim.box

particle
Instance of halmd.mdsim.particle.

disconnect()
Disconnect module from profiler.

4.1.6 Neighbour List

This module provides the implementation for a Verlet neighbour list. It stores the neighbours for each particle that are within a certain radius to reduce the computational cost for the force calculation in each time step.

Due to its nature it can only work with finite interaction potentials.

class `halmd.mdsim.`**`neighbour`**(*args*)

Construct neighbour module.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance, or sequence of two instances, of `halmd.mdsim.particle`
- **args.box** – instance of `halmd.mdsim.box`
- **args.r_cut** (*table*) – matrix with elements $r_{\mathrm{c},ij}$
- **args.skin** (*number*) – neighbour list skin *(default: 0.5)*
- **args.algorithm** (*string*) – Preferred implementation of the neighbour list *(GPU variant only)*
- **args.occupancy** (*number*) – Desired cell occupancy. Defaults to `halmd.mdsim.defaults.occupancy()` *(GPU variant only)*
- **args.disable_binning** (*boolean*) – Disable use of binning module and construct neighbour lists from particle positions directly (*default: false*).
- **args.disable_sorting** (*boolean*) – Disable use of Hilbert sorting `halmd.mdsim.sorts.hilbert` (*default: false*).
- **args.displacement** – instance or two instances of `halmd.mdsim.max_displacement` *(optional)*
- **args.binning** – instance or two instances of `halmd.mdsim.binning` *(optional)*

If all elements in `r_cut` matrix are equal, a scalar value may be passed instead.

If `displacement` or `binning` is left unspecified, a default module of `halmd.mdsim.max_displacement` or `halmd.mdsim.binning` is constructed. Providing an instance of the respective module allows the reuse of the module (e.g. when different neighbour lists share the first instance of `particle`).

For the `host` implementation of the `particle` module with binning disabled, Hilbert sorting is disabled also.

Specifying `algorithm` will affect the GPU implementation of the neighbour list build when binning is enabled only. The available algorithms are `naive` and `shared_mem`, where the latter tends to be faster on older GPUs (i.e. $\leq$ Tesla C1060), but slower on at least GTX 260 and later. Note that the `shared_mem` algorithm works only when both binning modules have equal number of cells in each spatial direction.

`particle`

Sequence of the two instances of `halmd.mdsim.particle`.

displacement
Sequence of two instances of `halmd.mdsim.max_displacement`.

binning
Sequence of two instances of `halmd.mdsim.binning`. May be `nil` if binning was disabled.

cell_occupancy
Average cell occupancy. *Only available on GPU variant.*

r_skin
"Skin" of the particle. This is an additional distance ratio added to the cutoff radius. Particles within this extended sphere are stored as neighbours.

disconnect()
Disconnect neighbour module from core and profiler.

4.1.7 Particle

class `halmd.mdsim.particle`(*args*)
Construct particle instance.

Parameters

- **args** (*table*) – keyword arguments
- **args.particles** (*number*) – number of particles
- **args.species** (*number*) – number of species (*default:* 1)
- **args.dimension** (*number*) – dimension of space (*default:* 3)
- **args.memory** (*string*) – device where the particle information is stored *(optional)*
- **args.label** (*string*) – instance label (*default:* `all`)

The supported values for `memory` are "host" and "gpu". If `memory` is not specified, the memory location is selected according to the compute device.

nparticle
Number of particles.

nspecies
Number of particle species.

label
Instance label.

memory
Device where the particle memory resides.

> **Warning:** During simulation, particle arrays are reordered in memory according to a space-filling curve. To access particles in initial order, use `get_reverse_tag()` to retrieve the current particle indices.

get_position()
Returns sequence with particle positions.

set_position(*position*)
Set particle positions to given sequence.

get_image()
Returns sequence with particle images.

set_image(*image*)
Set particle images to given sequence.

get_velocity()
Returns sequence with particle velocities.

set_velocity(*velocity*)
Set particle velocities to given sequence.

get_tag()
Returns sequence with particle tags.

set_tag(*tag*)
Set particle tags to given sequence.

get_reverse_tag()
Returns sequence with particle reverse tags.

set_reverse_tag(*reverse_tag*)
Set particle reverse tags to given sequence.

get_species()
Returns sequence with particle species.

set_species(*species*)
Set particle species to given sequence.

get_mass()
Returns sequence with particle masses.

set_mass(*mass*)
Set particle masses to given sequence.

get_force()
Returns unordered sequence with particle forces.

get_potential_energy()
Returns unordered sequence with potential energies.

get_stress_pot()
Returns unordered sequence with potential parts of stress tensors.

shift_velocity(*vector*)
Shift all velocities by `vector`.

shift_velocity_group(*group*, *vector*)
Shift velocities of group by `vector`.

rescale_velocity(*scalar*)
Rescale magnitude of all velocities by `scalar`.

rescale_velocity_group(*group*, *scalar*)
Rescale magnitude of velocities of group by `scalar`.

shift_rescale_velocity(*vector*, *scalar*)
First shift, then rescale all velocities.

shift_rescale_velocity_group(*group*, *vector*, *scalar*)
First shift, then rescale velocities of group.

aux_enable()
Enable the computation of auxliliary variables in the next on_force() step. These are: `stress_pot` and `potential_energy` and derived properties (such as the internal energy or the virial). The auxiliary variables should be activated like this:

```
sampler:on_prepare(function() particle:aux_enable() end, every, start)
```

on_prepend_force(*slot*)
Connect nullary slot to signal.

Returns signal connection

on_force(*slot*)
Connect nullary slot to signal.

Returns signal connection

on_append_force(*slot*)
Connect nullary slot to signal.

Returns signal connection

__eq(*other*)

Parameters **other** – instance of `halmd.mdsim.particle`

Implements the equality operator `a = b` and returns true if the other `particle` instance is the same as this one.

4.1.8 Forces

Full Pair Force

class `halmd.mdsim.forces.`**pair_full**(*args*)
Construct full pair force.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance, or sequence of two instances, of `halmd.mdsim.particle`
- **args.box** – instance of `halmd.mdsim.box`
- **args.potential** – instance of `halmd.mdsim.potentials`
- **args.weight** (*number*) – weight of the auxiliary variables *(default: 1)*

The module computes the full potential forces (untruncated, in minimum image convention) exerted by the particles of the second *particle* instance on those of the first one. The two instances agree if only a single instance is passed. Recomputation is triggered by the signals *on_force* and *on_prepend_force* of *args.particle[1]*.

The argument `weight` determines the fraction of the potential energy and the stress tensor that that is added to by the interaction of this force module. A value of *1* is defined as adding the full potential energy and stress tensor of each interaction. This is especially useful when considering pair forces where the particle instances (*A* and *B*) are distinct and only *AB* but not *BA* interaction is calculated.

Note: If two different instances of `halmd.mdsim.particle` are passed, Newton's 3rd law is not obeyed. To restore such a behaviour, the module must be constructed a second time with the order of particle instances reversed.

---

potential
Instance of `halmd.mdsim.potentials`.

disconnect()
Disconnect force from profiler and particle module.

Truncated Pair Force

class halmd.mdsim.forces.**pair_trunc**(*args*)
Construct truncated pair force.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance, or sequence of two instances, of `halmd.mdsim.particle`
- **args.box** – instance of `halmd.mdsim.box`
- **args.potential** – instance of `halmd.mdsim.potentials`
- **args.trunc** – instance of `halmd.mdsim.forces.trunc` (optional)
- **args.neighbour** – instance of `halmd.mdsim.neighbour` (optional)
- **args.weight** (*number*) – weight of the auxiliary variables *(default: 1)*

The module computes the truncated potential forces excerted by the particles of the second *particle* instance on those of the first one. The two instances agree if only a single instance is passed. Recomputation is triggered by the signals *on_force* and *on_prepend_force* of *args.particle[1]*.

The argument `weight` determines the fraction of the potential energy and the stress tensor that that is added to by the interaction of this force module. A value of *1* is defined as adding the full potential energy and stress tensor of each interaction. This is especially useful when considering pair forces where the particle instances (*A* and *B*) are distinct and only *AB* but not *BA* interaction is calculated.

Note: If two different instances of `halmd.mdsim.particle` are passed, Newton's 3rd law is not obeyed. To restore such a behaviour, the module must be constructed a second time with the order of particle instances reversed.

If `trunc` is not specified, the pair potential is C^0 continuous at the cutoff.

If `neighbour` is left unspecified, a default neighbour list module is constructed using the default parameters of `halmd.mdsim.neighbour`. If if a different value for, e.g., the `occupancy`

parameter is needed, the neighbour list module has to be provided explicitly.

potential
Instance of halmd.mdsim.potentials.

disconnect()
Disconnect force from profiler.

> **Warning:** Currently this does not disconnect particle sorting, binning and neighbour lists.

Smoothing Functions

Local r^4 smoothing

When passed to halmd.mdsim.forces.pair_trunc, this function transforms the potential $V(r)$ into a C^2-continuous function, and the force into a C^1-continuous function. The degree of smoothing is controlled with a dimensionless parameter. Before smoothing the potential is shifted to $V(r_c) = 0$ by the force module.

The smoothing function is

$$g(\xi) = \frac{\xi^4}{1+\xi^4}, \qquad \xi = \frac{r - r_c}{h r_c},$$

with smoothing parameter $h \ll 1$ and cutoff distance r_c, and its derivative

$$g'(\xi) = 4\frac{\xi^3}{(1+\xi^4)^2}.$$

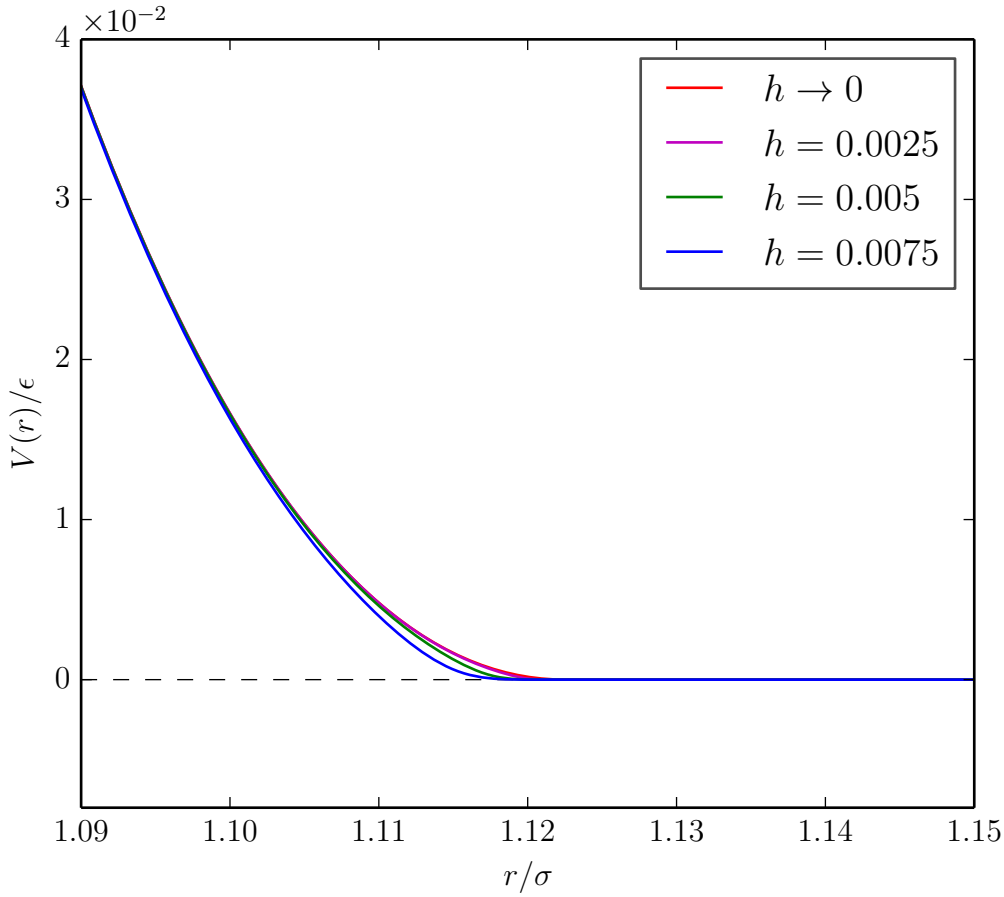
The C^2-continuous potential is

$$V_{C^2}(r) = V(r)\, g\left(\frac{r - r_c}{h r_c}\right),$$

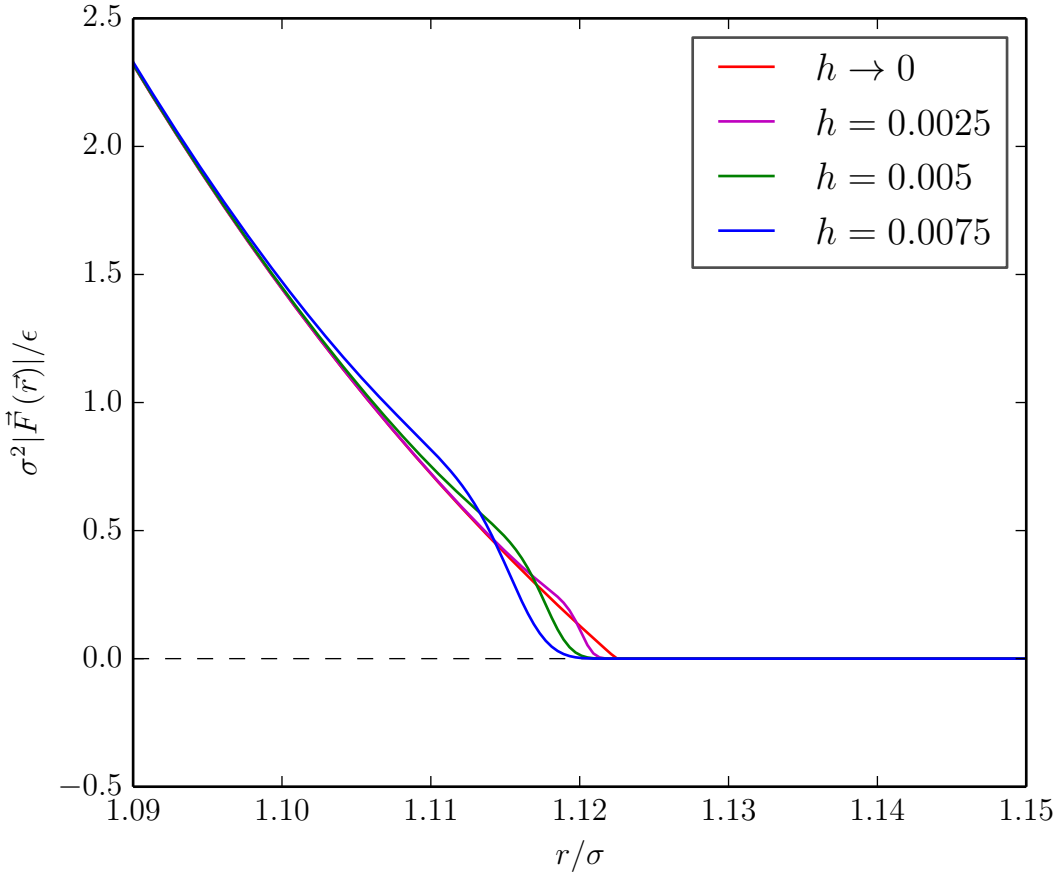
and the C^1-continuous force is

$$|\vec{F}_{C^1}(\vec{r})| = |\vec{F}(\vec{r})| g\left(\frac{r - r_c}{h r_c}\right) - \frac{1}{h r_c} V(r)\, g'\left(\frac{r - r_c}{h r_c}\right).$$

The following figure shows unmodified and C^2-smooth variants of the Weeks-Chandler-Andersen potential, a purely repulsive form of the Lennard-Jones potential with cutoff $r_c = \sqrt[6]{2}$.



The following figure shows the absolute value of the force.



class halmd.mdsim.forces.trunc.**local_r4**(*args*)

Construct smoothing function.

Parameters

- **args** (*table*) – keyword arguments
- **args.h** (*number*) – smoothing parameter

h

Smoothing parameter.

log(*logger*)

Output smoothing parameter to logger.

Parameters **logger** – instance of halmd.io.log.logger

4.1.9 Integrators

Euler

This integrator implements the explicit Euler method.

The algorithm propagates the positions in time as follows:

$$\vec{r}(t+\tau) = \vec{r}(t) + \tau\vec{v}(t)$$

where τ is the timestep.

class halmd.mdsim.integrators.**euler**(*args*)
Construct Euler integrator for given system of particles

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of halmd.mdsim.particle
- **args.box** – instance of halmd.mdsim.box
- **args.timestep** (*number*) – integration time step (defaults to halmd.mdsim.clock.timestep)

set_timestep(*timestep*)
Set integration time step in MD units.

Parameters **timestep** (*number*) – integration timestep

This method forwards to halmd.mdsim.clock.set_timestep(), to ensure that all integrators use an identical time step.

timestep
Integration time step in MD units.

disconnect()
Disconnect integrator from core and profiler.

integrate()
Calculate integration step

By default this function is connected to halmd.mdsim.core.on_integrate().

Velocity Verlet

This NVE-ensemble integrator implements the velocity-Verlet algorithm in J. Chem. Phys. 76, 637 (1982).

The algorithm consists of a first half-step

$$\vec{v}\left(t+\frac{\tau}{2}\right) = \vec{v}\left(t\right) + \frac{\tau}{2}\frac{\vec{F}\left(t\right)}{m}$$
$$\vec{r}\left(t+\tau\right) = \vec{r}\left(t\right) + \tau\vec{v}\left(t+\frac{\tau}{2}\right)$$

and a second half-step

$$\vec{v}\left(t+\tau\right) = \vec{v}\left(t+\frac{\tau}{2}\right) + \frac{\tau}{2}\frac{\vec{F}\left(t+\tau\right)}{m}$$

class halmd.mdsim.integrators.**verlet**(*args*)
Construct velocity-Verlet integrator for given system of particles.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of halmd.mdsim.particle
- **args.box** – instance of halmd.mdsim.box

- **args.timestep** (*number*) – integration time step (defaults to `halmd.mdsim.clock.timestep`)

set_timestep (*timestep*)
Set integration time step in MD units.

Parameters **timestep** (*number*) – integration timestep

This method forwards to `halmd.mdsim.clock.set_timestep()`, to ensure that all integrators use an identical time step.

timestep
Integration time step in MD units.

disconnect ()
Disconnect integrator from core and profiler.

integrate ()
Calculate first half-step.

By default this function is connected to `halmd.mdsim.core.on_integrate()`.

finalize ()
Calculate second half-step.

By default this function is connected to `halmd.mdsim.core.on_finalize()`.

Velocity Verlet with Andersen thermostat

This module implements the *Verlet algorithm* with the Andersen thermostat.

Warning: This integrator may cause a significant drift of the centre of mass velocity.
For heating or cooling a system to a nominal temperature before equilibration, we recommend the `velocity-Verlet with Boltzmann distribution` integrator.

class `halmd.mdsim.integrators.`**verlet_nvt_andersen** (*args*)
Construct velocity-Verlet with Andersen thermostat.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of `halmd.mdsim.particle`
- **args.box** – instance of `halmd.mdsim.box`
- **args.temperature** (*number*) – temperature of heat bath
- **args.rate** (*number*) – collision rate
- **args.timestep** (*number*) – integration timestep (defaults to `halmd.mdsim.clock.timestep`)

set_timestep (*timestep*)
Set integration time step in MD units.

Parameters **timestep** (*number*) – integration timestep

This method forwards to `halmd.mdsim.clock.set_timestep()`, to ensure that all integrators use an identical time step.

timestep
Integration time step.

set_temperature (*temperature*)
Set temperature of heat bath.

Parameters **temperature** (*number*) – temperature of heat bath

temperature
Temperature of heat bath.

collision_rate
Collision rate with the heat bath.

disconnect()
Disconnect integrator from core and profiler.

integrate()
First leapfrog half-step of velocity-Verlet algorithm.

By default this function is connected to `halmd.mdsim.core.on_integrate()`.

finalize()
Second leapfrog half-step of velocity-Verlet algorithm.

By default this function is connected to `halmd.mdsim.core.on_finalize()`.

Velocity Verlet with Boltzmann distribution

This integrator combines the `velocity-Verlet algorithm` and the `Boltzmann velocity distribution`. At a periodic interval, the velocities are assigned from a Boltzmann velocity distribution in the second integration half-step.

This integrator is especially useful for cooling or heating a system to a nominal temperature. After an assignment from the Boltzmann distribution, the centre of mass velocity is shifted to exactly zero, and the velocities are rescaled to exactly the nominal temperature.

class `halmd.mdsim.integrators.`**verlet_nvt_boltzmann** (*args*)
Construct integrator for given system of particles.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of `halmd.mdsim.particle`
- **args.box** – instance of `halmd.mdsim.box`
- **args.timestep** (*number*) – integration time step (defaults to `halmd.mdsim.clock.timestep`)
- **args.temperature** (*number*) – temperature of Boltzmann distribution
- **args.rate** (*number*) – nominal coupling rate

set_timestep (*timestep*)
Set integration time step in MD units.

Parameters **timestep** (*number*) – integration timestep

This method forwards to `halmd.mdsim.clock.set_timestep()`, to ensure that all integrators use an identical time step.

timestep
Integration time step in MD units.

temperature
Temperature of Boltzmann distribution in MD units.

interval
Coupling interval in steps.

The interval equals $\left\lfloor \frac{1}{\nu\tau} \right\rceil$, with nominal coupling rate ν and time-step τ.

rate
Effective coupling rate per time in MD units.

The effective coupling rate equals $\frac{1}{\Delta s\,\tau}$, with coupling interval Δs and time-step τ.

set_temperature (*temperature*)
Set the temperature of the Boltzmann distribution to the given value.

integrate()
Calculate first half-step.

By default this function is connected to `halmd.mdsim.core.on_integrate()`.

finalize()
Calculate second half-step, or assign velocities from Boltzmann distribution.

By default this function is connected to `halmd.mdsim.core.on_finalize()`.

disconnect()
Disconnect integrator from core and profiler.

Velocity Verlet with Nosé-Hoover thermostat

This NVT-ensemble integrator implements the *Verlet algorithm* algorithm with Nosé-Hoover chain thermostat with a chain length $M = 2$.

For reference and detailed description of the algorithm see the original papers by S. Nosé, W.G. Hoover and Martyna et al.:

- S. Nosé, J. Chem. Phys. 81, 511 (1984)
- W. G. Hoover, Phys. Rev. A 31, 1695 (1985)
- J. Martyna et al., J. Chem. Phys. 97, 2635 (1992)

class `halmd.mdsim.integrators.`**verlet_nvt_hoover** (*args*)
Construct velocity-Verlet integrator with Nosé-Hoover chain thermostat.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of `halmd.mdsim.particle`
- **args.force** – instance of `halmd.mdsim.forces`
- **args.box** – instance of `halmd.mdsim.box`

- **args.timestep** (*number*) – integration time step (defaults to `halmd.mdsim.clock.timestep`)
- **args.temperature** (*number*) – temperature of heat bath
- **args.resonance_frequency** (*number*) – Coupling rate of the thermostat
- **args.every** (*number*) – Number of time steps between writing of theromostat chain values

set_timestep (*timestep*)
Set integration time step in MD units.

Parameters **timestep** (*number*) – integration timestep

This method forwards to `halmd.mdsim.clock.set_timestep()`, to ensure that all integrators use an identical time step.

timestep
Integration time step in MD units.

set_temperature (*temperature*)
Set temperature of heat bath.

Parameters **temperature** (*number*) – temperature of heat bath

temperature
Temperature of heat bath.

resonance_frequency
Resonance frequency of the Nosé-Hoover thermostat, this is connected to the mass of the thermostat via $m_1 = dT/\Omega^2$ and $m_2 = T/\Omega^2$, where Ω is $2\pi \times$ Coupling frequency and d the dimension.

set_mass (*mass*)
Set mass of heat bath.

Parameters **mass** (*table*) – Sequence of masses m_1, m_2 for the heat bath coupling.

mass
Array of masses m_1, m_2 of heat bath, connected to the coupling strength of the thermostat.

disconnect ()
Disconnect integrator from core and profiler.

integrate ()
Calculate first half-step.

By default this function is connected to `halmd.mdsim.core.on_integrate()`.

finalize ()
Calculate second half-step.

By default this function is connected to `halmd.mdsim.core.on_finalize()`.

4.1.10 Particle Groups

All

A particle group represents a subset of particles, which is defined by an instance of particle together with a sequence of indices.

Example:

```
-- construct particle instance for given simulation domain
local system = halmd.mdsim.particle({particles = 10000})

-- select all particles
local group_all = halmd.mdsim.particle_groups.all({particle = particle})
```

class halmd.mdsim.particle_groups.**all**(*args*)

Construct particle group from all particles.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of halmd.mdsim.particle
- **args.label** (*string*) – group label (defaults to halmd.mdsim.particle.label)
- **args.global** (*boolean*) – particle group comprises the whole simulation world (*default:* true)

particle

Instance of halmd.mdsim.particle

size

Number of particles in group.

global

The value of args.global passed upon construction.

From Range

A particle group represents a subset of particles, which is defined by an instance of particle together with a sequence of indices.

Example:

```
-- construct particle instance for given simulation domain
local system = halmd.mdsim.particle({particles = 10000, species = 2})

-- select each species, assuming particles of a species have contiguous tags
local group_A = halmd.mdsim.particle_groups.from_range({particle = system, range = {1, 5
local group_B = halmd.mdsim.particle_groups.from_range({particle = system, range = {5001
```

class halmd.mdsim.particle_groups.**from_range**(*args*)

Construct particle group from tag range.

Parameters

- **args** (*table*) – keyword arguments

- **args.particle** – instance of `halmd.mdsim.particle`
- **args.range** (*table*) – particle tag range `{first, last}`
- **args.label** (*string*) – group label
- **args.global** (*boolean*) – particle group can comprise the whole simulation world (*default:* `false`)

Note: Particle tags are 1-based, i.e. the first particle has tag 1.

---

particle
Instance of `halmd.mdsim.particle`

size
Number of particles in group.

global
True if the particle group comprises the whole simulation world. This requires that `args.global` was set to true upon construction and that `size` equals the number of particles in `particle`.

4.1.11 Potentials

Lennard-Jones potential

This module implements the Lennard-Jones potential,

$$U_{\mathrm{LJ}}(r_{ij}) = 4\epsilon_{ij}\left(\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right)$$

for the interaction between two particles of species i and j.

class `halmd.mdsim.potentials.`**`lennard_jones`**(*args*)
Construct Lennard-Jones potential.

Parameters

- **args** (*table*) – keyword arguments
- **args.epsilon** (*table*) – matrix with elements ϵ_{ij} (*default:* `1`)
- **args.sigma** (*table*) – matrix with elements σ_{ij} (*default:* `1`)
- **args.cutoff** (*table*) – matrix with elements $r_{\mathrm{c},ij}$
- **args.species** (*number*) – number of particle species *(optional)*
- **args.memory** (*string*) – select memory location *(optional)*
- **args.label** (*string*) – instance label *(optional)*

If the argument `species` is omitted, it is inferred from the first dimension of the parameter matrices.

If all elements of a matrix are equal, a scalar value may be passed instead which is promoted to a square matrix of size given by the number of particle `species`.

The supported values for `memory` are "host" and "gpu". If `memory` is not specified, the memory location is selected according to the compute device.

Note: The cutoff is only relevant with `halmd.mdsim.forces.pair_trunc`.

epsilon
Matrix with elements ϵ_{ij}.

sigma
Matrix with elements σ_{ij}.

r_cut
Matrix with elements $r_{\mathrm{c},ij}$ in reduced units.

r_cut_sigma
Matrix with elements $r_{\mathrm{c},ij}$ in units of σ_{ij}.

description
Name of potential for profiler.

memory
Device where the particle memory resides.

Linearly truncated Lennard-Jones potential

This module implements the Lennard-Jones potential with a linear truncation scheme (which is equivalent to a shifted force),

$$U_{\mathrm{LJ}}\left(r_{ij}\right) = 4\epsilon_{ij}\left(\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6} + \left(r_{ij} - r_c\right)\cdot F_c\right)$$

for the interaction between two particles of species i and j. Here, r_c denotes the cutoff distance and $F_c = -U_{\mathrm{LJ}}\prime(r_c)$ the force at the cutoff for the untruncated potential.

This linear truncation scheme modifies the potential drastically and at all distances by adding a constant force, and we do not recommend it for future work. It is mainly provided for historical reasons to connect to existing data and publications. Please refer to the `local r^4 truncation` scheme for an alternative.

References

1. *S. K. Das, J. Horbach, K. Binder, M. E. Fisher, and J. V. Sengers*, J. Chem. Phys. 125, 024506 (2006)
2. *S. Toxvaerd and J. C. Dyre*, J. Chem. Phys. 134, 081102 (2011)
3. *S. Toxvaerd, O. J. Heilmann, and J. C. Dyre*, J. Chem. Phys. 136, 224106 (2012)

class `halmd.mdsim.potentials.`**`lennard_jones_linear`**(*args*)
Construct linearly truncated Lennard-Jones potential.

Parameters

- **args** (*table*) – keyword arguments
- **args.epsilon** (*table*) – matrix with elements ϵ_{ij} (*default:* `1`)

- **args.sigma** (*table*) – matrix with elements σ_{ij} (*default:* `1`)
- **args.cutoff** (*table*) – matrix with elements $r_{\mathrm{c},ij}$
- **args.species** (*number*) – number of particle species *(optional)*
- **args.memory** (*string*) – select memory location *(optional)*
- **args.label** (*string*) – instance label *(optional)*

If the argument `species` is omitted, it is inferred from the first dimension of the parameter matrices.

If all elements of a matrix are equal, a scalar value may be passed instead which is promoted to a square matrix of size given by the number of particle `species`.

The supported values for `memory` are "host" and "gpu". If `memory` is not specified, the memory location is selected according to the compute device.

Note: The cutoff is only relevant with `halmd.mdsim.forces.pair_trunc`.

---

epsilon
Matrix with elements ϵ_{ij}.

sigma
Matrix with elements σ_{ij}.

r_cut
Matrix with elements $r_{\mathrm{c},ij}$ in reduced units.

r_cut_sigma
Matrix with elements $r_{\mathrm{c},ij}$ in units of σ_{ij}.

description
Name of potential for profiler.

memory
Device where the particle memory resides.

Modified Lennard-Jones potential

This module implements a modified Lennard-Jones potential,

$$U_{\mathrm{LJ}}(r_{ij}) = 4\epsilon_{ij}\left(\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{m_{ij}} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{n_{ij}}\right)$$

for the interaction between two particles of species i and j.

The difference to `halmd.mdsim.potentials.lennard_jones` is that the exponents of repulsion, m_{ij}, and of attraction, n_{ij}, can be specified explicitly.

class `halmd.mdsim.potentials.`**`modified_lennard_jones`**(*args*)
Construct modified Lennard-Jones potential.

Parameters

- **args** (*table*) – keyword arguments
- **args.epsilon** (*table*) – matrix with elements ϵ_{ij} (*default:* `1`)

- **args.sigma** (*table*) – matrix with elements σ_{ij} (*default:* `1`)
- **args.cutoff** (*table*) – matrix with elements $r_{c,ij}$
- **args.index_repulsion** (*table*) – exponent of repulsive part, m_{ij}
- **args.index_attraction** (*table*) – exponent of attractive part, , n_{ij}
- **args.species** (*number*) – number of particle species *(optional)*
- **args.memory** (*string*) – select memory location *(optional)*
- **args.label** (*string*) – instance label *(optional)*

If the argument `species` is omitted, it is inferred from the first dimension of the parameter matrices.

If all elements of a matrix are equal, a scalar value may be passed instead which is promoted to a square matrix of size given by the number of particle `species`.

The supported values for `memory` are "host" and "gpu". If `memory` is not specified, the memory location is selected according to the compute device.

Note: The cutoff is only relevant with `halmd.mdsim.forces.pair_trunc`.

---

epsilon
Matrix with elements ϵ_{ij}.

sigma
Matrix with elements σ_{ij}.

r_cut
Matrix with elements $r_{c,ij}$ in reduced units.

r_cut_sigma
Matrix with elements $r_{c,ij}$ in units of σ_{ij}.

index_m
Matrix with exponents m_{ij} of repulsive part of the potential

index_n
Matrix with exponents n_{ij} of attractive part of the potential

description
Name of potential for profiler.

memory
Device where the particle memory resides.

Morse potential

This module implements the Morse potential,

$$U_{\text{Morse}}(r_{ij}) = \epsilon_{ij}\left(1 - e^{r_{ij}/\sigma_{ij} - r_{\min,ij}}\right)^2 - \epsilon_{ij}$$

for the interaction between two particles of species i and j.

class `halmd.mdsim.potentials.`**morse**(*args*)
Construct Morse potential.

Parameters

- **args** (*table*) – keyword arguments
- **args** – keyword arguments
- **args.epsilon** (*table*) – matrix with elements ϵ_{ij} (*default:* `1`)
- **args.sigma** (*table*) – matrix with elements σ_{ij} (*default:* `1`)
- **args.minimum** (*table*) – Minimum/equilibrium distance $r_{\min}$ in units of σ_{ij}
- **args.cutoff** (*table*) – matrix with elements $r_{\mathrm{c},ij}$
- **args.species** (*number*) – number of particle species *(optional)*
- **args.memory** (*string*) – select memory location *(optional)*
- **args.label** (*string*) – instance label *(optional)*

If the argument `species` is omitted, it is inferred from the first dimension of the parameter matrices.

If all elements of a matrix are equal, a scalar value may be passed instead which is promoted to a square matrix of size given by the number of particle `species`.

The supported values for `memory` are "host" and "gpu". If `memory` is not specified, the memory location is selected according to the compute device.

Note: The cutoff is only relevant with `halmd.mdsim.forces.pair_trunc`.

`epsilon`
Matrix with elements ϵ_{ij}.

`sigma`
Matrix with elements σ_{ij}.

`r_cut`
Matrix with elements $r_{\mathrm{c},ij}$ in reduced units.

`r_cut_sigma`
Matrix with elements $r_{\mathrm{c},ij}$ in units of σ_{ij}.

`r_min_sigma`
Equilibrium distance $r_{\mathrm{min},ij}$ in units of σ_{ij}.

`description`
Name of potential for profiler.

`memory`
Device where the particle memory resides.

Power-law potential

This module implements the (inverse) power-law potential,

$$U\left(r_{ij}\right) = \epsilon_{ij}\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{n_{ij}},$$

for the interaction between two particles of species i and j with the power-law index n_{ij}.

class halmd.mdsim.potentials.**power_law**(*args*)

Construct power-law potential.

Parameters

- **args** (*table*) – keyword arguments
- **args.epsilon** (*table*) – matrix with elements ϵ_{ij} (*default:* 1)
- **args.sigma** (*table*) – matrix with elements σ_{ij} (*default:* 1)
- **args.index** (*table*) – power-law index n_{ij} (*default:* 12)
- **args.cutoff** (*table*) – matrix with elements $r_{\mathrm{c},ij}$
- **args.species** (*number*) – number of particle species *(optional)*
- **args.memory** (*string*) – select memory location *(optional)*
- **args.label** (*string*) – instance label *(optional)*

If the argument species is omitted, it is inferred from the first dimension of the parameter matrices.

If all elements of a matrix are equal, a scalar value may be passed instead which is promoted to a square matrix of size given by the number of particle species.

The supported values for memory are "host" and "gpu". If memory is not specified, the memory location is selected according to the compute device.

Note: The cutoff is only relevant with halmd.mdsim.forces.pair_trunc.

epsilon
Matrix with elements ϵ_{ij}.

sigma
Matrix with elements σ_{ij}.

r_cut
Matrix with elements $r_{\mathrm{c},ij}$ in reduced units.

r_cut_sigma
Matrix with elements $r_{\mathrm{c},ij}$ in units of σ_{ij}.

index
Matrix with power-law indices n_{ij}

description
Name of potential for profiler.

memory
Device where the particle memory resides.

Power-law with hard core potential

This module implements the (inverse) power-law potential,

$$U\left(r_{ij}\right) = \epsilon_{ij}\left(\frac{\sigma_{ij}}{r_{ij} - r_{\mathrm{core},ij}}\right)^{n_{ij}},$$

for the interaction between two particles of species i and j with the power-law index n_{ij} and the core size $r_{\text{core, ij}}$.

class halmd.mdsim.potentials.**power_law_with_core**(*args*)

Construct power-law with hard core potential.

Parameters

- **args** (*table*) – keyword arguments
- **args.epsilon** (*table*) – matrix with elements ϵ_{ij} (*default:* 1)
- **args.sigma** (*table*) – matrix with elements σ_{ij} (*default:* 1)
- **args.core** (*table*) – matrix with elements $r_{\text{core},ij}$
- **args.index** (*table*) – power-law index n_{ij} (*default:* 12)
- **args.cutoff** (*table*) – matrix with elements $r_{\text{c},ij}$
- **args.species** (*number*) – number of particle species *(optional)*
- **args.memory** (*string*) – select memory location *(optional)*
- **args.label** (*string*) – instance label *(optional)*

If the argument species is omitted, it is inferred from the first dimension of the parameter matrices.

If all elements of a matrix are equal, a scalar value may be passed instead which is promoted to a square matrix of size given by the number of particle species.

The supported values for memory are "host" and "gpu". If memory is not specified, the memory location is selected according to the compute device.

Note: The cutoff is only relevant with halmd.mdsim.forces.pair_trunc.

---

epsilon
Matrix with elements ϵ_{ij}.

sigma
Matrix with elements σ_{ij}.

r_cut
Matrix with elements $r_{\text{c},ij}$ in reduced units.

r_cut_sigma
Matrix with elements $r_{\text{c},ij}$ in units of σ_{ij}.

r_core_sigma
Matrix with elements $r_{\text{core},ij}$ in units of σ_{ij}.

index
Matrix with power-law indices n_{ij}

description
Name of potential for profiler.

memory
Device where the particle memory resides.

4.1.12 Velocities

Boltzmann distribution

This module initialises particle velocities from a Boltzmann distribution.

The velocity distribution per degree of freedom is a Gaussian with mean $\mu_v = 0$ and width $\sigma_v = \sqrt{\frac{kT}{m}}$,

$$f\left(v\right) = \sqrt{\frac{m}{2\pi kT}} \exp\left(\frac{-mv^2}{2kT}\right)$$

To account for the finite size of the system, the velocities are shifted,

$$\vec{v}_{\text{shifted}} \equiv \vec{v} - \vec{V}_{\text{cm}}$$

to yield a centre of mass velocity of zero, and scaled,

$$\vec{v}_{\text{scaled}} \equiv \vec{v}_{\text{shifted}} \sqrt{\frac{\frac{1}{2}kTfN}{E_{\text{kin}} - \frac{1}{2}M\left(\vec{V}_{\text{cm}}\right)^2}}$$

to yield the temperature T for a system with f positional degrees of freedom, using centre of mass velocity and kinetic energy,

$$\vec{V}_{\text{cm}} = \frac{\vec{P}_{\text{cm}}}{M} = \frac{\sum_{n=1}^{N} m_n \vec{v}_n}{\sum_{n=1}^{N} m_n}$$

$$E_{\text{kin}} = \frac{1}{2}\sum_{n=1}^{N} m_n v_n^2$$

class `halmd.mdsim.velocities.`**`boltzmann`**(*args*)

Construct boltzmann module.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of `halmd.mdsim.particle`
- **args.temperature** (*number*) – temperature of distribution

`set`()

Initialise particle velocities from Boltzmann distribution.

`temperature`

Temperature of the distribution in reduced units. The value can be changed by assignment.

4.1.13 Positions

Excluded volume

This module implements a method to place a number of spheres that have no overlap.

This example shows use of the excluded volume with the placement of 1000 particles at random positions with a diameter of `1` in a cubic box with each edge being `50`:

```
local random = math.random
math.randomseed(os.time())

local edge = 50
local box = halmd.mdsim.box{length = {edge, edge, edge}}
local excluded = halmd.mdsim.positions.excluded_volume{box = box, cell_length = 1}

local obstacles = {}
local diameter = 1
local repeats = 50
for i = 1, 1000 do
    for j = 1, repeats do
        local r = {edge * random(), edge * random(), edge * random()}
        if excluded:place_sphere(r, diameter) then
            obstacles[i] = r
            excluded:exclude_sphere(r, diameter)
            break
        end
    end
    if not obstacles[i] then
        error(("cannot place obstacle %d after %d repeats"):format(i, repeats))
    end
end

local particle = halmd.mdsim.particle{dimension = box.dimension, particles = #obstacles}
particle:set_position(obstacles)
```

Note: If one uses the random number generator from Lua , this should be done in conjunction with LuaJIT, only. Standard Lua uses the OS-dependent `rand()` function.

See http://luajit.org/extensions.html#math_random

---

class `halmd.mdsim.positions.`**`excluded_volume`**(*args*)

Construct excluded volume instance

Parameters

- **args** (*table*) – keyword arguments
- **args.box** (*number*) – instance of `halmd.mdsim.box`
- **args.cell_length** (*number*) – cell length for internal binning (must not be smaller than largest sphere diameter)

`exclude_sphere`(*centre*, *diameter*)

Place a single sphere at `centre` with a diameter of `diameter`

`exclude_spheres`(*centres*, *diameters*)

Place a set of spheres with their respective centres and diameters

`place_spheres`(*centre*, *diameter*)

Test if a sphere at `centre` with diameter `diameter` can be placed without overlap with any other previously set sphere

Lattice

This module places particles on a face-centered cubic (fcc) lattice. Optionally, the lattice may be restricted to a cuboid ("slab") centred and aligned with the simulation box.

class halmd.mdsim.positions.**lattice**(*args*)

Construct Lattice module

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – Instance of halmd.mdsim.particle.
- **args.box** – Instance of halmd.mdsim.box.
- **args.slab** (*table*) – Vector specifying the fraction of the box size to be filled *(optional)*.

slab

Restrict the lattice to a slab of given extents relative to the box size, the slab is centred within the simulation box. More generally, the lattice may be restricted to a cuboid aligned with the box since each direction may be less than unity. The default is *{1, ... 1}*, i.e., no restriction.

set()

Set all particle positions on an fcc lattice.

disconnect()

Disconnect module from profiler.

4.1.14 Sort algorithms

Hilbert sort

This module re-orders the particle data in halmd.mdsim.particle according to a space-filling Hilbert curve. The idea behind is that interacting particles, being close in space, are also close in memory for better cache efficiency. The module is used and constructed internally by halmd.mdsim.neighbour, i.e. manual construction is not needed.

For details see:

- S. Aluru and F. Sevilgen, *Parallel domain decomposition and load balancing using space-filling curves,* Proc. 4th IEEE Int. Conf. High Performance Computing, p. 230 (1997)
- J. Anderson, C. D. Lorenz, and A. Travesset, *General purpose molecular dynamics simulations fully implemented on graphics processing units,* J. Comp. Phys. **227**, 5342 (2008)

class halmd.mdsim.sorts.**hilbert**(*args*)

Construct Hilbert sort module.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of halmd.mdsim.particle
- **args.box** – instance of halmd.mdsim.box
- **args.binning** – instance of halmd.mdsim.binning *(see below)*

If `particle` instance resides in GPU memory (i.e. `particle.memory` is `gpu`), a `binning` instance is not required for construction of the Hilber sort module.

order()
Sort the particles according to a space-filling Hilbert curve.

disconnect()
Disconnect neighbour module from core and profiler.

4.2 Numeric

This module provides simple numeric routines in Lua.

halmd.numeric.**sum**(*t*)
Compute the sum of the indexed elements of a table.

Parameters **t** (*table*) – input table

Returns sum over all indexed elements in `t`

halmd.numeric.**prod**(*t*)
Compute the product of the indexed elements of a table.

Parameters **t** (*table*) – input table

Returns product over all indexed elements in `t`

halmd.numeric.**find_comp**(*t*, *comp*)
Find the last value of a table that satisfies comp(a,b)

Parameters

- **t** (*table*) – input table
- **comp** – callable that takes two elements of `t` and returns `true` or `false`

Returns last element in `t` that satisfied `comp(a,b)`

halmd.numeric.**max**(*t*)
Find the maximum value in a table

Parameters **t** (*table*) – input table

Returns maximum value in `t`

halmd.numeric.**min**(*t*)
Find the minimum value in a table

Parameters **t** (*table*) – input table

Returns minmum value in `t`

halmd.numeric.**scalar_matrix**(*rows*, *columns*, *value*)
Create matrix of given size with scalar value

Parameters

- **rows** (*number*) – number of rows
- **columns** (*number*) – number of columns
- **value** – value for each element of the matrix

Returns matrix of dimension rows × columns with each element set to `value`

`halmd.numeric.`**`trans`**(*m*)
Calculate transpose of matrix

Parameters **m** (*matrix*) – input matrix

Returns transpose of `m`

`halmd.numeric.`**`diag`**(*m*)
Return diagonal elements of n×n matrix

Parameters **m** (*matrix*) – input square matrix

Returns table of diagonal elements of `m`

`halmd.numeric.`**`offset_to_multi_index`**(*offset*, *dims*)
Convert one-dimensional offset to multi-dimensional index

Assumes contiguous storage of the array data in row-major order.

Parameters

- **offset** (*number*) – 1-based one-dimensional offset
- **dims** (*table*) – dimensions (shape) of multi-dimensional array

Returns 1-based multi-dimensional index of array element at `offset`

`halmd.numeric.`**`multi_index_to_offset`**(*index*, *dims*)
Convert multi-dimensional index to one-dimensional offset

Assumes contiguous storage of the array data in row-major order.

Parameters

- **index** (*table*) – 1-based multi-dimensional index
- **dims** (*table*) – dimensions (shape) of multi-dimensional array

Returns 1-based offset of array element at `index`

4.3 Input and Output

4.3.1 Logging

This module provides logging to HALMD scripts and modules.

Script Logger

The script logger may be used in HALMD scripts.

A message is logged with one of the following severity levels.

Severity	Description
error	An error has occurred, and HALMD will abort
warning	Warn user, e.g. when using single precision
info	Normal logging level, e.g. for parameters
debug	Low frequency debugging messages
trace	High frequency debugging messages

This example shows use of the script logger:

```
local halmd = require("halmd")

local log = halmd.io.log

function my_simulation(args)
    log.info("box edge lengths: %s", table.concat(args.length, " "))

    log.info("equilibrate system for %d steps", args.equilibrate)

    log.info("measure mean-square displacement for %d steps", args.steps)
end
```

halmd.io.log.**error**(*format*, ...)
Log message with severity error.

Parameters **format** (*string*) – see string.format

halmd.io.log.**warning**(*format*, ...)
Log message with severity warning.

Parameters **format** (*string*) – see string.format

halmd.io.log.**info**(*format*, ...)
Log message with severity info.

Parameters **format** (*string*) – see string.format

halmd.io.log.**debug**(*format*, ...)
Log message with severity debug.

Parameters **format** (*string*) – see string.format

halmd.io.log.**trace**(*format*, ...)
Log message with severity trace.

Parameters **format** (*string*) – see string.format

Module Logger

This class provides module loggers for Lua and C++ modules.

A logger may optionally have a label, which is prepended to messages to distinguish output of a module from that of other modules.

This example shows use of a logger in a HALMD module:

```
local log    = require("halmd.io.log")
local module = require("halmd.utility.module")

-- C++ class
```

```
local my_potential = assert(libhalmd.mdsim.potentials.my_potential)

-- use the same logger for all instances
local logger = log.logger({label = "my_potential"})

local M = module(function(args)
    -- logs message with prefix "my_potential: "
    logger:info("parameters: %g %g %g", 1.0, 0.88, 0.8)

    -- pass module logger to C++ constructor
    local self = my_potential(..., logger)

    return self
end)

return M
```

class `halmd.io.log.`**`logger`**(*args*)

Construct a logger instance, optionally with given label.

Parameters

- **args** (*table*) – keyword arguments (optional)
- **args.label** (*string*) – logging prefix (optional)

`error`(*format*, ...)

Log message with severity `error`.

Parameters **format** (*string*) – see string.format

`warning`(*format*, ...)

Log message with severity `warning`.

Parameters **format** (*string*) – see string.format

`info`(*format*, ...)

Log message with severity `info`.

Parameters **format** (*string*) – see string.format

`debug`(*format*, ...)

Log message with severity `debug`.

Parameters **format** (*string*) – see string.format

`trace`(*format*, ...)

Log message with severity `trace`.

Parameters **format** (*string*) – see string.format

Logging Setup

The following functions setup available logging sinks.

By default, messages are logged to console with severity `warning`.

This example shows logging setup in a HALMD script:

```
local halmd = require("halmd")

-- log warning and error messages to console
halmd.io.log.open_console({severity = "warning"})
-- log anything except trace messages to file
halmd.io.log.open_file("kob_andersen.log", {severity = "debug"})
```

halmd.io.log.**open_console**(*args*)

Log messages with equal or higher `severity` to console.

If `severity` is not specified, it is set to `info`.

Parameters

- **args** (*table*) – keyword arguments (optional)
- **args.severity** (*string*) – log severity level (optional)

halmd.io.log.**close_console**()

Disable logging to console.

halmd.io.log.**open_file**(*filename*, *args*)

Log messages with equal or higher `severity` to file.

If `severity` is not specified, it is set to `info`.

If a file with `filename` exists, it is truncated.

Parameters

- **filename** (*string*) – log filename
- **args** (*table*) – keyword arguments (optional)
- **args.severity** (*string*) – log severity level (optional)

halmd.io.log.**close_file**()

Close log file.

4.3.2 Readers

H5MD Reader

This module provides a file reader for the H5MD format.

http://nongnu.org/h5md/

class halmd.io.readers.**h5md**(*args*)

Construct H5MD reader.

Parameters

- **args** (*table*) – keyword arguments
- **args.path** (*string*) – pathname of input file

reader(*self*, *args*)

Construct a group reader.

Parameters

- **args** (*table*) – keyword arguments
- **args.location** (*table*) – sequence with group's path
- **args.mode** (*string*) – read mode ("append" or "truncate")

Returns instance of group reader

close (*self*)
Close file.

root
HDF5 root group of the file.

path
Filename of the file.

version
H5MD major and minor version of file.

creator
Name of the program that created the file.

creator_version
Version of the program that created the file.

creation_time
Creation time of the file in seconds since the Unix epoch.

This time stamp may be converted to a human-readable time using os.date:

```
halmd.log.info(("file created at %s"):format(os.date("%c", file.creation_time))
```

author
Name of author of the file.

halmd.io.readers.h5md.**check** (*path*)
Check whether file is a valid H5MD file.

Parameters **path** – filename

Returns `true` if the file is a valid H5MD file, `false` if not, or `nil` if the file does not exist

An error message is emitted if the return value is not `true`.

The function is useful to validate a command-line argument:

```
local parser = halmd.utility.program_options.argument_parser()

parser:add_argument("trajectory", {
    help = "H5MD trajectory file"
  , type = "string"
  , required = true
  , action = function(args, key, value)
        halmd.io.readers.h5md.check(value)
        args[key] = value
    end
})
```

4.3.3 Writers

H5MD Writer

This module provides a file writer for the H5MD format.

http://nongnu.org/h5md/

class `halmd.io.writers.`**`h5md`**(*args*)

Construct H5MD writer.

Parameters

- **args** (*table*) – keyword arguments
- **args.path** (*string*) – pathname of output file
- **args.email** (*string*) – email address of file author *(optional)*

Returns instance of file writer

Create the output file and writes the H5MD metadata.

http://nongnu.org/h5md/draft.html#global-attributes

The author name is retrieved from the password file entry for the real user id of the calling process.

Warning: The output file will be truncated if it exists.

The file may be flushed to disk by sending the USR2 signal to the process:

```
killall -USR2 halmd
```

This yields a consistent snapshot on disk (until the next sample is written), which is useful to peek at output data during the simulation.

`writer`(*self*, *args*)

Construct a group writer.

Parameters

- **args** (*table*) – keyword arguments
- **args.location** (*table*) – sequence with group's path
- **args.mode** (*string*) – write mode ("append" or "truncate")

Returns instance of group writer

Example for creating and using a truncate writer:

```
local writer = file:writer({location = {"particles", "box"}, mode = "truncate"}
writer:on_write(box.origin, {"offset"})
writer:on_write(box.edges, {"edges"})

local sampler = require("halmd.observables.sampler")
sampler:on_start(writer.write)
```

Example for creating and using an append writer:

```
local writer = file:writer({location = {"observables"}, mode = "append"})
writer:on_prepend_write(observable.sample)
writer:on_write(observable.en_pot, {"potential_energy"})
writer:on_write(observable.en_kin, {"kinetic_energy"})
writer:on_write(observable.en_tot, {"internal_energy"})

local sampler = require("halmd.observables.sampler")
sampler:on_start(writer.write)
```

flush()
Flush the output file to disk.

root
HDF5 root group of the file.

path
Filename of the file.

halmd.io.writers.h5md.**version**()
Returns sequence of integers with major and minor H5MD version.

4.4 Observables

4.4.1 Density mode

The module computes the complex Fourier modes of the particle density field,

$$\rho(\vec{k}) = \sum_{n=1}^{N} \exp(\mathrm{i}\vec{k} \cdot \vec{r}_n).$$

The auxiliary module halmd.observables.utility.wavevector provides suitable wavevectors that are compatible with the reciprocal lattice of the periodic simulation box.

class halmd.observables.**density_mode**(*args*)
Construct instance of halmd.observables.density_mode.

Parameters

- **args** (*table*) – keyword arguments
- **args.group** – instance of halmd.mdsim.particle_groups
- **args.wavevector** – instance of halmd.observables.utility.wavevector

Returns instance of density mode sampler

disconnect()
Disconnect density mode sampler from profiler.

wavevector
The wavevector instance passed upon construction.

label
The label of the underlying particle group.

count
The particle count N of the underlying particle group.

class `writer`(*args*)
Write time series of density modes to file.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file *(optional)*
- **args.every** (*number*) – sampling interval

Returns instance of density mode writer

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `{"structure", self.label, "density_mode"}`.

`disconnect`()
Disconnect density mode writer from observables sampler.

4.4.2 Phase Space

A phase_space sampler acquires particle coordinates from an instance of particle or particle_group. The sampler can copy particle data from host to host, gpu to host, or gpu to gpu memory. The particles are ordered by tag, which guarantees that a particle has the same array index over the course of the simulation.

class `halmd.observables.phase_space`(*args*)
Construct phase_space sampler.

Parameters

- **args** (*table*) – keyword arguments
- **args.group** – instance of `halmd.mdsim.particle_group`
- **args.box** – instance of `halmd.mdsim.box`

Note: The sample will be updated at most *once* per step, so you can reuse the same sampler with multiple observable modules for optimal performance.

`acquire`(*args*)
Returns data slot to acquire phase space sample.

Parameters

- **args** (*table*) – keyword arguments (optional)
- **args.memory** (*string*) – memory location of phase space sample (optional)

The supported values for `memory` are "host" and "gpu". If `memory` is not specified, the memory location is selected according to the compute device.

`position`()
Returns data slot that acquires phase space sample and returns position array.

Returns data slot that returns position array in host memory

velocity()
Returns data slot that acquires phase space sample and returns velocity array.

Returns data slot that returns velocity array in host memory

species()
Returns data slot that acquires phase space sample and returns species array.

Returns data slot that returns species array in host memory

mass()
Returns data slot that acquires phase space sample and returns mass array.

Returns data slot that returns mass array in host memory

set(*sample*)
Set positions, velocities, species and masses from phase space sample.

disconnect()
Disconnect phase_space sampler from profiler.

group
The particle group used by the sampler.

class writer(*args*)
Write trajectory of particle group to file.

http://nongnu.org/h5md/draft.html#particles-group

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.fields** (*table*) – data field names to be written
- **args.location** (*string table*) – location within file (optional)
- **args.every** (*number*) – sampling interval (optional)

Returns instance of group writer

The table `fields` specifies which data fields are written. It may either be passed as an indexed table, e.g. `{"position", "velocity"}`, or as a dictionary, e.g., `{r = "position", v = "velocity"}`; the table form is interpreted as `{position = "position", ...}`. The keys denote the field names in the file and are appended to `location`. The values specify the methods of the `phase_space` module, valid values are `position`, `velocity`, `species`, `mass`.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. If omitted it defaults to `{"particles", group.label}`.

If `every` is not specified or 0, a phase space sample will be written at the start and end of the simulation.

disconnect()
Disconnect phase_space writer from observables sampler.

class halmd.observables.phase_space.**reader**(*args*)
Construct reader for given particles group.

http://nongnu.org/h5md/draft.html#particles-group

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file reader, e.g, `halmd.io.readers.h5md`
- **args.fields** (*string table*) – data field names to be read
- **args.location** (*string table*) – location within file
- **args.memory** (*string*) – memory location of phase space sample (optional)

The supported values for `memory` are "host" and "gpu". If `memory` is not specified, the memory location is selected according to the compute device.

Returns a group reader, and a phase space sample.

The table `fields` specifies which data fields are read, valid values are `position`, `velocity`, `species`, `mass`. See `halmd.observables.phase_space:writer()` for details.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings, for example `{"particles", group label}`.

Construction of the reader module opens the file for inspection of the space dimension and particle number, which are then used to allocate a phase space sample in host memory. The sample is only filled upon calling, e.g., `read_at_step()`.

Example:

```
local file = halmd.io.readers.h5md({path = "input.h5"})
local reader, sample = halmd.observables.phase_space.reader({
   file = file, fields = {"position"}, location = {"particles", "all"}
})
reader:read_at_step(0)
local nparticle = assert(sample.size)
local dimension = assert(sample.dimension)
```

The returned group reader has these methods.

read_at_step (*step*)
Read sample at given step.

If `step` is negative, seek backward from last (`-1`) sample.

read_at_time (*time*)
Read sample at given time in MD units.

If `time` is negative, seek backward from last (`-0`) sample.

The returned phase space sample has these attributes.

nparticle
Returns number of particles.

nspecies
Returns number of species.

Note: This attribute is determined from the maximum element of the species array.

---

dimension
Returns dimension of positional coordinates.

4.4.3 Runtime Estimate

Given the number of steps, this module estimates the remaining runtime.

Example:

```
-- setup simulation box
halmd.observables.sampler:setup()

-- number of MD steps
local steps = 1000000

-- calculate remaining runtime every minute, and log every 15 minutes
local runtime = halmd.observables.runtime_estimate({steps = steps, first = 10, interval

-- run simulation
halmd.observables.sampler:run(steps)
```

A runtime estimate may be triggered by sending the process signal `USR1`:

```
killall -USR1 halmd
```

class `halmd.observables.`**`runtime_estimate`**(*args*)
Construct runtime_estimate instance.

Parameters

- **args** (*table*) – keyword arguments
- **steps** (*number*) – length of simulation run
- **first** (*number*) – time to first estimate in seconds
- **interval** (*number*) – frequency of estimates in seconds
- **sample** (*number*) – frequency of sampling in seconds

4.4.4 Sampler

The sampler concerts the sampling of observables.

Example:

```
local sampler = require("halmd.observables.sampler")
sampler:sample()
sampler:run(1000)
```

`halmd.observables.sampler.`**`sample`**()
Sample current state.

`halmd.observables.sampler.`**`run`**(*steps*)
Run simulation for given number of steps.

This method invokes `halmd.mdsim.core.mdstep()`.

halmd.observables.sampler.**on_prepare**(*slot*, *interval*, *start*)
Connect slot to signal emitted just before sampling current state. `slot()` will be executed every `interval` timesteps with the first time being executed at timestep `start`. `start` must be greater or equal to zero.

halmd.observables.sampler.**on_sample**(*slot*, *interval*, *start*)
Connect slot to signal emitted to sample current state. `slot()` will be executed every `interval` timesteps with the first time being executed at timestep `start`. `start` must be greater or equal to zero.

Returns signal connection

halmd.observables.sampler.**on_start**(*slot*)
Connect slot to signal emitted before starting simulation run.

Returns signal connection

halmd.observables.sampler.**on_finish**(*slot*)
Connect slot to signal emitted after finishing simulation run.

Returns signal connection

4.4.5 Static structure factor

The module computes the static structure factor

$$S_{(\alpha\beta)}(\vec{k}) = \frac{1}{N}\langle \rho_\alpha(\vec{k})^* \rho_\beta(\vec{k})\rangle$$

from the Fourier modes of a given pair of (partial) density fields,

$$\rho_\alpha(\vec{k}) = \sum_{n=1}^{N_\alpha} \exp(\mathrm{i}\vec{k}\cdot\vec{r}_n),$$

and the total number of particles N. The result is averaged over wavevectors of similar magnitude according to the shells defined by `halmd.observables.utility.wavevector`.

For details see, e.g., Hansen & McDonald: Theory of simple liquids, chapter 4.1.

class halmd.observables.**ssf**(*args*)
Construct instance of `halmd.observables.ssf`.

Parameters

- **args** (*table*) – keyword arguments
- **args.density_mode** – instance(s) of `halmd.observables.density_mode`
- **args.norm** (*number*) – normalisation factor
- **args.label** (*string*) – module label *(optional)*

Returns instance of static structure factor module

The argument `density_mode` is an instance or a table of 1 or 2 instances of `halmd.observables.density_mode` yielding the partial density modes $\rho_\alpha(\vec{k})$ and $\rho_\beta(\vec{k})$. They must have been constructed with the same instance of `halmd.observables.utility.wavevector`. Passing only one instance implies $\alpha = \beta$.

The optional argument `label` defaults to `density_mode[1].label .. "/" .. density_mode[2].label`.

disconnect()
Disconnect static structure factor module from profiler.

sampler
Callable that yields the static structure factor from the current density modes.

label
The module label passed upon construction or derived from the density modes.

class writer(*args*)
Write time series of static structure factor to file.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file *(optional)*
- **args.every** (*number*) – sampling interval

Returns instance of density mode writer

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `{"structure", self.label, "static_structure_factor"}`.

disconnect()
Disconnect static structure factor writer from observables sampler.

4.4.6 Thermodynamics

This module calculates the instantaneous values of thermodynamic state variables for the particles of a given group.

class halmd.observables.**thermodynamics**(*args*)
Construct thermodynamics module.

Parameters

- **args** (*table*) – keyword arguments
- **args.group** – instance of `halmd.mdsim.particle_groups`
- **args.box** – instance of `halmd.mdsim.box`

particle_number()
Returns the number of particles N selected by `args.group`.

density()
Returns the number density $\rho = N/V$ using the volume from `args.box`.

kinetic_energy()
Returns the mean kinetic energy per particle: $u_{\text{kin}} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} m_i \vec{v}_i^2$.

potential_energy()
Returns the mean potential energy per particle: $u_{\text{pot}} = \frac{1}{N}\sum_{i=1}^{N} U_{\text{tot}}(\vec{r}_i)$, where U_{tot} denotes the sum of external and pair potentials.

internal_energy()
Returns the mean internal energy per particle: $u_{\text{int}} = u_{\text{kin}} + u_{\text{pot}}$.

pressure()
Returns the pressure computed from the virial: $p = \rho(k_B T - \mathcal{V}/d)$.

temperature()
Returns the instantaneous temperature as given by the kinetic energy: $k_B T = 2u_{\text{kin}}/d$.

center_of_mass_velocity()
Returns the centre-of-mass velocity: $\vec{v}_{\text{cm}} = \sum_{i=1}^{N} m_i \vec{v}_i / \sum_{i=1}^{N} m_i$.

center_of_mass()
Returns the centre of mass: $\vec{r}_{\text{cm}} = \sum_{i=1}^{N} m_i \vec{r}'_i / \sum_{i=1}^{N} m_i$, where $\vec{r}'_i$ refers to absolute particle positions, i.e., extended by their image vectors for periodic boundary conditions.

mean_mass()
Returns the mean particle mass: $\bar{m} = \frac{1}{N}\sum_{i=1}^{N} m_i$.

virial()
Returns mean virial per particle as computed from the trace of the potential part of the stress tensor:

$$\mathcal{V} = \frac{1}{2N} \sum_{i \neq j} r_{ij}\, U'(r_{ij})\,.$$

stress_tensor()
Returns the elements of the stress tensor $\Pi_{\alpha\beta}$ as a vector. The first d (= *dimension*) elements contain the diagonal followed by $d(d-1)/2$ off-diagonal elements $\Pi_{xy}, \Pi_{xz}, \ldots, \Pi_{yz}, \ldots$ The stress tensor is computed as

$$\Pi_{\alpha\beta} = \sum_{i=1}^{N} \left[m_i v_{i\alpha} v_{i\beta} + \frac{1}{2} \sum_{j \neq i} \frac{r_{ij\alpha} r_{ij\beta}}{r_{ij}}\, U'(r_{ij}) \right],$$

where $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ in nearest image convention.

dimension
Space dimension d of the simulation box as a number.

group
Instance of `halmd.mdsim.particle_groups` used to construct the module.

writer(*args*)
Write state variables to a file.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.every** (*number*) – sampling interval
- **args.location** (*string table*) – location within file (optional)

- **args.fields** (*table*) – data fields to be written (optional)

Returns instance of group writer

The optional argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `{"observables", group.global and nil or group.label}`.

The optional table `fields` specifies which data fields are written. It may either be passed as an indexed table, e.g. `{"pressure"}`, or as a dictionary, e.g., `{p = "pressure"}`; the table form is interpreted as `{pressure = "pressure", ...}`. The keys denote the field names in the file and are appended to `location`. The values specify the data methods of the `thermodynamics` module, i.e., all methods described above except for `dimension` and `group`. The default is `{"potential_energy", "pressure", "temperature", "center_of_mass_velocity"}`.

disconnect()
Disconnect thermodynamics writer from observables sampler.

4.4.7 Dynamics

Blocking Scheme

class halmd.observables.dynamics.**blocking_scheme**(*args*)
Construct blocking scheme.

Parameters

- **args** – keyword arguments
- **args.max_lag** (*number*) – maximum lag time in MD units
- **args.every** (*number*) – sampling interval of lowest coarse-graining level in integration steps
- **args.size** (*number*) – size of each block, determines coarse-graining factor
- **args.shift** (*number*) – coarse-graining shift between odd and even levels (*default:* $\lfloor\sqrt{\text{size}}\rfloor$)
- **args.separation** (*number*) – minimal separation of samples for time averages in sampling steps (*default:* – every $\times$ size)
- **args.flush** (*number*) – interval in seconds for flushing the accumulated results to the file (*default:* 900)

disconnect()
Disconnect blocking scheme from sampler.

class correlation(*args*)
Compute time correlation function.

Parameters

- **args** (*table*) – keyword arguments
- **args.tcf** – time correlation function
- **args.file** – instance of `halmd.io.writers.h5md`

- **args.location** (*string table*) – location within file *(optional)*

The argument `tcf` specifies the time correlation function. It is expected to provide the attributes `acquire` (1 or 2 callables that yield the samples to be correlated) and `desc` (module description) as well as a method `writer` (file writer). Suitable modules are found in `halmd.observables.dynamics`, see there for details.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. If omitted it is defined by the time correlation function, typically {`"dynamics"`, particle group, name of correlation function}.

disconnect()
Disconnect correlation function from blocking scheme.

Correlation Function

This module permits the implementation of a user-defined time correlation function within the Lua simulation script.

The following example shows the use of this module together with `halmd.observables.dynamics.blocking_scheme` to determine the mean-square displacement of the centre of mass of a certain particle group. From this, the collective self-diffusion constant may be obtained. The centre of mass is computed efficiently by `halmd.observables.thermodynamics.center_of_mass()`, the squared displacement is then computed by the script function passed as `correlate`.

```
local msv = observables.thermodynamics({box = box, group = group, force = force})
local collective_msd = dynamics.correlation({
    -- acquire centre of mass
    acquire = function()
        return msv:center_of_mass()
    end
    -- correlate centre of mass at first and second point in time
    , correlate = function(first, second)
        local result = 0
        for i = 1, #first do
            result = result + math.pow(second[i] - first[i], 2)
        end
        return result
    end
    -- module description
    , desc = "collective mean-square displacement of AA particles"
})

local blocking_scheme = dynamics.blocking_scheme({
    max_lag = max_lag
    , every = 100
    , size = 10
    , separation = separation
})
blocking_scheme:correlation({
    tcf = collective_msd, file = file
  , location = {"dynamics", "AA", "collective_mean_square_displacement"}
})
```

class `halmd.observables.dynamics.`**correlation**(*args*)
Construct user-defined correlation function.

Parameters

- **args** – keyword arguments
- **args.acquire** – callable(s) that return a value
- **args.correlate** – callable that accepts two values and returns a `number`
- **args.location** (*string table*) – default location within file
- **args.desc** (*string*) – module description

The argument `acquire` is a callable or a table of up to 2 callables that yield the samples to be correlated.

The argument `location` defines the default value of `writer()`. For H5MD files, it obeys the structure {`"dynamics"`, particle group, name of correlation function}.

`acquire``()`
Acquire sample(s).

Returns sample

`correlate`(*first*, *second*)
Correlate two samples.

Parameters

- **first** – first sample
- **second** – second sample

Returns value

`desc`
Module description.

class `writer`(*args*)
Construct file writer.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file

Returns file writer as returned by `file:writer()`.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `args.location` passed upon construction of the correlation module.

Helfand Moment

This module calculates mean-square difference of the Helfand moment for the stress tensor [1,2],

$$\delta G^2_{\alpha\beta}(t) := \frac{1}{N}\left\langle [G_{\alpha\beta}(t) - G_{\alpha\beta}(0)]^2 \right\rangle \qquad \alpha, \beta \in \{x, y, z\}\,,$$

where the Helfand moment $G_{\alpha\beta}(t)$ is defined as the time integral of the stress tensor $\Pi_{\alpha\beta}(t)$,

$$G_{\alpha\beta}(t) = \int_0^t \Pi_{\alpha\beta}(t')\,\mathrm{d}t' \approx \sum_{k=0}^{n-1} \Pi_{\alpha\beta}(k\delta t)\,\delta t \qquad t = n\,\delta t\,.$$

The normalisation with the particle number N renders $\delta G^2_{\alpha\beta}(t)$ finite in the thermodynamic limit. The stress tensor is obtained from `halmd.observables.thermodynamics.stress_tensor()`, and the integral is computed numerically over discrete time intervals δt using `halmd.observables.utility.accumulator`.

The shear viscosity η is obtained from $\delta G^2_{\alpha\beta}(t)$ by virtue of the Einstein–Helfand relation

$$\eta = \frac{\rho}{k_B T} \lim_{t\to\infty} \frac{\mathrm{d}}{2\mathrm{d}t} \delta G^2_{\alpha\beta}(t)\,.$$

Note: The module returns the sum over all off-diagonal elements, $\sum_{\alpha<\beta} \delta G^2_{\alpha\beta}(t)$ analogously to `halmd.observables.dynamics.mean_square_displacement`.

---

class `halmd.observables.dynamics.`**`helfand_moment`**(*args*)

Construct Helfand moment

This module implements a `halmd.observables.dynamics.correlation` module.

Parameters

- **args** – keyword arguments
- **args.thermodynamics** – instance of `halmd.observables.thermodynamics`
- **args.interval** (*number*) – time interval for the integration of the stress tensor in simulation steps

`acquire`()

Acquire stress tensor

Returns Stress tensor sample

`correlate`(*first*, *second*)

Correlate two stress tensor samples.

Parameters

- **first** – first phase space sample
- **second** – second phase space sample

Returns mean-square integral of the off-diagonal elements of the stress tensor

[1] B. J. Alder, D. M. Gass, and T. E. Wainwright, *Studies in molecular dynamics. VIII. The transport coefficients for a hard-sphere fluid*, J. Chem. Phys. **53**, 3813 (1970) [Link].

[2] S. Viscardy and P. Gaspard, *Viscosity in molecular dynamics with periodic boundary conditions*, Phys. Rev. E **68**, 041204 (2003) [Link].

desc
Module description.

disconnect()
Disconnect module from core.

class writer(*args*)
Construct file writer.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file *(optional)*

Returns file writer as returned by `file:writer()`.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `{"dynamics", self.label, "mean_square_helfand_moment"}`.

Intermediate scattering function

The module computes the intermediate scattering function

$$S_{(\alpha\beta)}(\vec{k},t) = \frac{1}{N}\langle \rho_\alpha(\vec{k},t)^* \rho_\beta(\vec{k},0)\rangle$$

from the Fourier modes of a given pair of (partial) density fields,

$$\rho_\alpha(\vec{k},t) = \sum_{n=1}^{N_\alpha} \exp\left(\mathrm{i}\vec{k}\cdot\vec{r}_n(t)\right),$$

and the total number of particles N. The result is averaged over wavevectors of similar magnitude according to the shells defined by `halmd.observables.utility.wavevector`.

For details see, e.g., Hansen & McDonald: Theory of simple liquids, chapter 7.4.

class `halmd.observables.dynamics.`**intermediate_scattering_function**(*args*)
Construct instance of `halmd.observables.dynamics.intermediate_scattering_function`.

Parameters

- **args** (*table*) – keyword arguments
- **args.density_mode** (*table*) – instance(s) of `halmd.observables.density_mode`
- **args.norm** (*number*) – normalisation factor
- **args.label** (*string*) – module label *(optional)*

Returns instance of intermediate scattering function module

The argument `density_mode` is an instance or a table of up to 2 instances of `halmd.observables.density_mode` yielding the partial density modes $\rho_\alpha(\vec{k},t)$ and $\rho_\beta(\vec{k},t)$. They must have been constructed with the same instance of `halmd.observables.utility.wavevector`. Passing only one instance implies $\alpha = \beta$.

The optional argument `label` defaults to `density_mode[1].label .. "/" .. density_mode[2].label`.

disconnect()
Disconnect module from profiler.

acquire()
Acquire density mode sample.

Returns density mode sample

label
The module label passed upon construction or derived from the density modes.

desc
Module description.

class writer(*args*)
Construct file writer and output wavenumbers.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file *(optional)*

Returns file writer as returned by `file:writer()`.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `{"dynamics", self.label, "intermediate_scattering_function"}`.

Mean-Quartic Displacement

class `halmd.observables.dynamics.`**mean_quartic_displacement**(*args*)
Construct mean-quartic displacement.

Parameters

- **args** – keyword arguments
- **args.phase_space** – instance of `halmd.observables.phase_space`

acquire()
Acquire phase space sample.

Returns phase space sample

correlate(*first*, *second*)
Correlate two phase space samples.

Parameters

- **first** – first phase space sample
- **second** – second phase space sample

Returns mean-quartic displacement between samples

desc
Module description.

class writer (*args*)
Construct file writer.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file *(optional)*

Returns file writer as returned by `file:writer()`.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `{"dynamics", self.label, "mean_quartic_displacement"}`.

Mean-Square Displacement

class `halmd.observables.dynamics.`**mean_square_displacement** (*args*)
Construct mean-square displacement.

Parameters

- **args** – keyword arguments
- **args.phase_space** – instance of `halmd.observables.phase_space`

acquire()
Acquire phase space sample.

Returns phase space sample

correlate (*first*, *second*)
Correlate two phase space samples.

Parameters

- **first** – first phase space sample
- **second** – second phase space sample

Returns mean-square displacement between samples

desc
Module description.

class writer (*args*)
Construct file writer.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file *(optional)*

Returns file writer as returned by `file:writer()`.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `{"dynamics", self.label, "mean_square_displacement"}`.

Stress Tensor Autocorrelation Function

This module calculates the autocorrelation of the off-diagonal elements of the stess tensor $\Pi_{\alpha\beta}$:

$$C_{\alpha\beta}(t) = \frac{1}{N}\langle \Pi_{\alpha\beta}(t)\Pi_{\alpha\beta}(0)\rangle \,.$$

By normalisation with the particle number N, the result remains finite in the thermodynamic limit. The stress tensor is obtained from `halmd.observables.thermodynamics.stress_tensor()`.

The shear viscosity η is found from this autocorrelation via the Green–Kubo relation

$$\eta = \frac{\rho}{k_B T}\int_0^\infty C_{\alpha\beta}(t)\,\mathrm{d}t \,.$$

Note: The module returns the sum over all off-diagonal elements, $\sum_{\alpha<\beta} C_{\alpha\beta}(t)$, analogously to `halmd.observables.dynamics.mean_square_displacement`.

---

class `halmd.observables.dynamics.`**`stress_tensor_autocorrelation`**(*args*)

Construct stress tensor autocorrelation function.

This module implements a `halmd.observables.dynamics.correlation` module.

Parameters

- **args** – keyword arguments
- **args.thermodynamics** – instance of `halmd.observables.thermodynamics`

`acquire`()

Acquire stress tensor

Returns Stress tensor sample

`correlate`(*first*, *second*)

Correlate two stress tensor samples.

Parameters

- **first** – first phase space sample
- **second** – second phase space sample

Returns stress tensor autocorrelation function between two samples.

`desc`

Module description.

`connect`(*args*)

Parameters

- **args** (*table*) – keyword arguments
- **args.every** – sampling interval

Returns sequence of signal connections

Internal use only. This function is called upon registration by `blocking_scheme:correlation()`.

Connect `msv.group.particle:aux_enable()` to the signal `on_prepend_force` of `halmd.observables.sampler` using the interval `every`.

class **`writer`** (*args*)
Construct file writer.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file *(optional)*

Returns file writer as returned by `file:writer()`.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `{"dynamics", self.label, "stress_tensor_autocorrelation"}`.

Velocity Autocorrelation Function

class `halmd.observables.dynamics.`**`velocity_autocorrelation`**(*args*)
Construct velocity autocorrelation function.

Parameters

- **args** – keyword arguments
- **args.phase_space** – instance of `halmd.observables.phase_space`

`acquire`()
Acquire phase space sample.

Returns phase space sample

`correlate`(*first*, *second*)
Correlate two phase space samples.

Parameters

- **first** – first phase space sample
- **second** – second phase space sample

Returns velocity autocorrelation function between samples

`desc`
Module description.

class **`writer`** (*args*)
Construct file writer.

Parameters

- **args** (*table*) – keyword arguments

- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file *(optional)*

Returns file writer as returned by `file:writer()`.

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings. It defaults to `{"dynamics", self.label, "velocity_autocorrelation"}`.

4.4.8 Auxiliary modules

Accumulator

This module accumulates values (e.g., the pressure) over the course of the simulation and returns statistical measures (e.g., sum, mean, and variance).

class `halmd.observables.utility.`**`accumulator`**(*args*)

Construct accumulator module.

Parameters

- **args** – keyword arguments
- **args.acquire** – callable that returns a number
- **args.every** (*number*) – interval for aquiring the value
- **args.start** (*number*) – start step for aquiring the value (*default:* `halmd.mdsim.clock.step`)
- **args.desc** (*string*) – profiling description
- **args.aux_enable** (*table*) – sequence of `halmd.mdsim.particle` instances *(optional)*

The parameter `aux_enable` is useful if `acquire()` depends on one of the auxiliary force variables, see `halmd.mdsim.particle.aux_enable()` for details. In sampling steps of the accumulator, each `particle` instance listed in `aux_enable` is notified to update the auxiliary variables before the integration step. Thereby, redundant force calculations can be avoided.

`sample`()

Sample next value by calling `args.acquire`.

`sum`()

Sum of accumulated values. Calculated as `mean × count`.

`mean`()

Mean of accumulated values.

`error_of_mean`()

Standard error of mean of accumulated values.

`variance`()

Variance of accumulated values.

`count`()

Number of samples accumulated.

reset()
Reset the accumulator.

disconnect()
Disconnect accumulator from core.

desc
Profiler description.

writer(*file*, *args*)
Write statistical measures to a file.

Parameters

- **args** (*table*) – keyword arguments
- **args.file** – instance of file writer
- **args.location** (*string table*) – location within file
- **args.every** (*number*) – sampling interval
- **args.reset** (*boolean*) – Reset accumulator after writing if true (disabled by default).

Returns instance of group writer

The argument `location` specifies a path in a structured file format like H5MD given as a table of strings, for example `{"observables", "averaged_pressure"}`.

Semi-logarithmic grid

Construct a semi-logarithmically spaced grid. The grid consists of a concatenation of linearly spaced grids and starts with multiples of the smallest value. After a given number of points, the grid is "decimated" by doubling the spacing until a maximum value is reached. A logarithmic grid is obtained by `decimation=1`, decimation is disabled by default.

Example:

```
-- construct grid from 0.1 to 4, double spacing every 3 points
local grid = semilog_grid({start=0.1, stop=4, decimation=3})

-- print the result
for i,x in pairs(grid.value) do
    io.write(x .. " ")
end
io.write("\n")
```

The result is a grid of 12 points: `0.1 0.2 0.3 0.4 0.6 0.8 1.0 1.4 1.8 2.2 3.0 3.8`.

`halmd.observables.utility.semilog_grid.`**value**
Return array of grid points.

class `halmd.observables.utility.`**semilog_grid**(*args*)
Construct instance of semilog_grid module.

Parameters

- **args** (*table*) – keyword arguments

- **args.start** (*number*) – first grid point, corresponds to initial spacing
- **args.stop** (*number*) – upper limit on grid points (not included)
- **args.decimation** (*integer*) – decimation parameter: 0=disabled *(default)*, 1=logarithmic, ...

halmd.observables.utility.semilog_grid.**add_options**(*parser*, *defaults*)

Add module options `maximum` and `decimation` to command line parser.

Parameters

- **parser** – instance of halmd.utility.program_options.argument_parser
- **defaults** (*dictionary*) – default values for the options

Wavevector

The module constructs a set of wavevector shells compatible with the reciprocal space of a periodic simulation box and a list of wavenumbers. The wavevectors are of the form

$$\vec{k} = (k_x, k_y, \dots) = \left(n_x \frac{2\pi}{L_x}, n_y \frac{2\pi}{L_y}, \dots \right)$$

where $n_x, n_y, \dots$ are integers and $L_x, L_y, \dots$ denote the edge length of the cuboid box. A wavevector "shell" is defined by its wavenumber, i.e. the magnitude $k = |\vec{k}|$, including a tolerance. The number of wavevectors within each shell may be limited to avoid excessively large shells for large wavenumber.

The list of wavenumbers may be constructed using halmd.observables.utility.semilog_grid, where the smallest wavenumber is given by $2\pi / \max(L_x, L_y, \dots)$.

Example:

```
local numeric = halmd.numeric
local utility = halmd.observables.utility

local box = halmd.mdsim.box({length={5,10,20}})

local qmin = 2 * math.pi / numeric.max(box.length)
local grid = utility.semilog_grid({start=qmin, stop=5 * math.pi, decimation=10})
local wavevector = utility.wavevector({box = box, wavenumber = grid.value, tolerance=0.0
```

class halmd.observables.utility.**wavevector**(*args*)

Construct instance of wavevector module.

Parameters

- **args** (*table*) – keyword arguments
- **args.box** – instance of halmd.mdsim.box
- **args.wavenumber** – list of wavenumbers
- **args.tolerance** (*number*) – relative tolerance on wavevector magnitude
- **args.max_count** (*integer*) – maximum number of wavevectors per wavenumber shell

wavenumber()

Returns data slot that yiels the wavenumber grid.

value()
Returns data slot that yiels the list of wavevectors grouped by their magnitude in ascending order.

__eq(*other*)

Parameters **other** – instance of halmd.observables.utility.wavevector

Implements the equality operator `a = b` and returns true if the other `wavevector` instance is the same as this one.

halmd.observables.utility.wavevector.**add_options**(*parser*, *defaults*)
Add module options to command line parser: `wavenumbers`, `tolerance`, `max-count`.

Parameters

- **parser** – instance of halmd.utility.program_options.argument_parser
- **defaults** (*dictionary*) – default values for the options

4.5 Random Numbers

This module provides pseudo-random number generators and random distributions.

halmd.random.**generator**(*args*)
Get pseudo-random number generator.

Parameters

- **args** (*table*) – keyword arguments
- **args.memory** (*string*) – `host` or `gpu` (*default:* compute device)
- **args.seed** (*number*) – initial seed value *(optional)*

Returns pseudo-random number generator

The first call for each memory argument constructs a singleton instance of the pseudo-random number generator, which is returned in subsequent calls.

If the argument `seed` is omitted, the initial seed is obtained from the system's random device, e.g., `/dev/urandom` on Linux.

halmd.random.**seed**(*seed*)
Set (or reset) the seed of the pseudo-random number generator.

4.6 Utilities

4.6.1 Device management

The device module selects a GPU from the pool of available GPUs. It allocates a CUDA context on that device, which will remain active till the program exits. Diagnostic information is logged about CUDA driver and runtime versions, and GPU capabilities.

halmd.utility.device.gpu may be used to query whether the GPU is being used:

```
local device = require("halmd.utility.device")
if device.gpu then
    -- using GPU
else
    -- using host
end
```

To select a specific GPU, you may use the nvlock tool:

```
CUDA_DEVICE=0 nvlock halmd liquid.lua
```

nvlock will lock the CUDA device for other processes using nvlock, similar to compute prohibitive mode. This allows scheduling one process per GPU.

halmd.utility.device.**gpu**
: Ordinal number of the CUDA device.

4.6.2 Module Definition

class halmd.utility.**module**(*new*)
: Define new module.

  Parameters **new** (*function*) – module constructor

  Returns module table

  Example:

```
local module = require("halmd.utility.module")

local M = module(function(args)
    -- create and return instance
end)

return M
```

halmd.utility.module.**loader**(*name*)
: This function provides a lazy module loader, which may be used to load submodules on demand. For a namespace, one defines a loader module:

```
-- halmd/mdsim/potentials/init.lua

local module = require("halmd.utility.module")

return module.loader("halmd.mdsim.potentials")
```

  The loader module then loads submodules upon use:

```
local potentials = require("halmd.mdsim.potentials")

-- This loads the lennard_jones module.
local lennard_jones = potentials.lennard_jones
```

  If a submodule cannot be loaded, the loader raises an error.

  Parameters **name** (*string*) – fully qualified name of module

  Returns module table with metatable containing module loader

4.6.3 POSIX Signal

The POSIX signal handler intercepts process signals.

halmd.utility.posix_signal.**on_hup**(*slot*)
Connect slot to invoke on signal HUP.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**on_int**(*slot*)
Connect slot to invoke on signal INT.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**on_alrm**(*slot*)
Connect slot to invoke on signal ALRM.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**on_term**(*slot*)
Connect slot to invoke on signal TERM.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**on_usr1**(*slot*)
Connect slot to invoke on signal USR1.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**on_usr2**(*slot*)
Connect slot to invoke on signal USR2.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**on_cont**(*slot*)
Connect slot to invoke on signal CONT.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**on_tstp**(*slot*)
Connect slot to invoke on signal TSTP.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**on_ttin**(*slot*)
Connect slot to invoke on signal TTIN.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**on_ttou**(*slot*)
Connect slot to invoke on signal TTOU.

Parameters **slot** – unary slot that accepts signal number

Returns connection

halmd.utility.posix_signal.**wait**()
Block process until signal is received.

halmd.utility.posix_signal.**poll**()
Poll signal queue, and returns true if signal was handled.

4.6.4 Profiler

The profiler collects and logs profiling times.

halmd.utility.profiler.**profile**()
Log and reset runtime accumulators.

halmd.utility.profiler.**on_profile**(*acc*, *desc*)
Connect accumulator for profiling.

Parameters

- **acc** – runtime accumulator
- **desc** (*string*) – description of runtime accumulator

Returns connection

halmd.utility.profiler.**on_prepend_profile**(*slot*)
Connect slot to signal.

Parameters **slot** – nullary function

Returns connection

halmd.utility.profiler.**on_append_profile**(*slot*)
Connect slot to signal.

Parameters **slot** – nullary function

Returns connection

4.6.5 Program Options

This module allows the use of command-line options in HALMD scripts.

Example:

```
halmd liquid.lua --lennard-jones epsilon=2 sigma=2 --disable-gpu
```

class halmd.utility.program_options.**argument_parser**
Create new command-line parser.

Example:

```
local options = require("halmd.utility.program_options")

local parser = options.argument_parser()
```

add_argument (*name*, *args*)
Add argument to parser.

Parameters

- **name** (*string*) – long name, and (optionally) short name separated by comma
- **args** (*table*) – keyword arguments
- **args.type** (*string*) – value type of option
- **args.dtype** (*string*) – element type of `vector` or `matrix` (optional)
- **args.help** (*string*) – description of option for –help (optional)
- **args.composing** (*boolean*) – allow multiple occurences with single value (default: `false`)
- **args.multitoken** (*boolean*) – allow multiple occurences with multiple values (default: `false`)
- **args.required** (*boolean*) – require at minimum one occurence (default: `false`)
- **args.choices** (*table*) – allowed values with descriptions (optional)
- **args.action** (*function*) – argument handler function (optional)
- **args.default** – default option value (optional)
- **args.implicit** – implicit option value (optional)

The following value types are supported:

Type	Description
boolean	Boolean
string	String
accumulate	Increment integer
vector	1-dimensional array of type `dtype`
matrix	2-dimensional array of type `dtype`

These integral and floating-point value types are supported:

Type	Description
number	Double-precision floating-point
integer	Signed 64-bit integer
int32	Signed 32-bit integer
int64	Signed 64-bit integer
uint32	Unsigned 32-bit integer
uint64	Unsigned 64-bit integer
float32	Single-precision floating-point
float64	Double-precision floating-point

Example:

```
parser:add_argument("disable-gpu", {type = "boolean", help = "disable GPU accel
```

An optional table `choices` may be used to constrain the value of an argument:

```
parser.add_argument("ensemble", {type = "string", choices = {
    nve = "Constant NVE",
    nvt = "Constant NVT",
    npt = "Constant NPT",
}, help = "statistical ensemble"})
```

Note that only arguments of `type` `string` are supported.

The optional `action` function receives the following arguments:

Parameters

- **args** (*table*) – parsed arguments
- **key** (*string*) – args table key of this argument
- **value** – parsed value of this argument

Note that if you specify `action`, the argument value will *not* be stored in the table returned by `parse_args()`, i.e. the argument handler function has to store a value in `args[key]` itself.

Example:

```
parser:add_argument("output", {type = "string", action = function(args, key, va
   -- substitute current time
   args[key] = os.date(value)
end, default = "halmd_%Y%m%d_%H%M%S.h5", help = "H5MD output file"})
```

add_argument_group (*name*, *args*)

Add argument group.

Parameters

- **name** – name of argument group
- **args** (*table*) – keyword arguments (optional)
- **args.help** (*string*) – description of argument group for –help (optional)

Returns argument group

Example:

```
local group = parser:add_argument_group("lennard-jones")
group:add_argument("epsilon", {type = "number", help = "potential well depths"}
group:add_argument("sigma", {type = "number", help = "collision diameter"})
```

set_defaults (*defaults*)

Set default option values.

Parameters **defaults** (*table*) – argument names with default values

Example:

```
parser:set_defaults({particles = {9000, 1000}, number_density = 0.8})
```

parse_args (*args*)

Parse arguments.

Parameters **args** (*table*) – sequence of arguments (optional)

Returns parsed arguments

If `args` is not specified, the command-line arguments are parsed.

Example:

```
local args = parser:parse_args()
```

4.6.6 HALMD Signal

`halmd.utility.signal.`**`disconnect`**(*conn*, *desc*, *level*)

Returns a callable that disconnects a sequence of signal connections.

Parameters

- **conn** – sequence of signal connections
- **name** (*string*) – module name to appear in the error message
- **level** (*number*) – call stack level for error message *(default: 2)*

4.6.7 Timer Service

The timer service emits periodic signals with intervals in real time.

`halmd.utility.timer_service.`**`on_periodic`**(*slot*, *interval*)

Connect function to call periodically at given interval.

Parameters

- **slot** – nullary function
- **interval** (*number*) – frequency of calls to function in seconds

Returns connection

`halmd.utility.timer_service.`**`on_periodic`**(*slot*, *interval*, *start*)

Connect function to call periodically at given interval.

Parameters

- **slot** – nullary function
- **interval** (*number*) – frequency of calls to function in seconds
- **start** (*number*) – time of first call in seconds

Returns connection

`halmd.utility.timer_service.`**`process`**()

Process timer event queue.

4.6.8 Version Information

`halmd.utility.version.`**`prologue`**()

Log HALMD version, build flags, command line and host name, and load `halmd.utility.profiler` and `halmd.utility.device`.

4.6.9 Functions on tables

halmd.utility.**empty**(*t*)
Test if table is empty.

halmd.utility.**keys**(*t*)
Returns table with sorted keys of table t as values.

halmd.utility.**sorted**(*t*)
Returns iterator over pairs of table t sorted by key.

halmd.utility.**reverse**(*t*)
Returns table with keys as values and values as keys.

4.6.10 Assertions

halmd.utility.**assert_kwarg**(*args*, *key*, *level*)
Assert keyword argument of table and return its value.

Parameters

- **args** (*table*) – argument table
- **key** (*string*) – parameter key
- **level** (*number*) – call stack level for error message *(default: 2)*

Returns *args[key]*

halmd.utility.**assert_type**(*var*, *name*, *level*)
Assert type of variable.

Parameters

- **var** – variable to check
- **name** (*string*) – Lua type name
- **level** (*number*) – call stack level for error message *(default: 2)*

Returns *var*

CHAPTER

FIVE

SIMULATION UNITS

Most physical quantities carry a dimension, and their numeric values are meaningful only in conjuction with a suitable unit. A computer, on the other hand, processes just plain numbers. The interpretation of such a numeric value as physical quantity depends on the—completely arbitrary—specification of the associated unit. Within a given simulation, the only constraint is that all units are derived from the same set of base units, e.g., for length, time, mass, temperature, and current/charge.

For example, an interaction range "$\sigma = 1$" of the Lennard-Jones potential may be interpreted as $\sigma = 1\,\mathrm{m}$, $\sigma = 1\,\mathrm{pm}$, or even $\sigma = 3.4$ (for argon). Another more abstract interpretation of "$\sigma = 1$" is that all lengths are measured relative to σ.

Typical choices for base units along with some derived units are given in the table:

physical dimension	symbol	SI base units	cgs system	abstract units (Lennard-Jones potential)
length	L	metre	centimetre	σ
time	T	second	second	$\tau = \sqrt{m\sigma^2/\epsilon}$
mass	M	kilogram	gram	m
temperature	Θ	kelvin		ϵ/k_B
current	I	ampère	franklin / second	q/τ
energy	$\mathrm{M}\times\mathrm{L}^2\times\mathrm{T}^{-2}$	joule	erg	ϵ
force	$\mathrm{M}\times\mathrm{L}\times\mathrm{T}^{-2}$	newton	dyne	$\epsilon/\sigma = m\sigma/\tau^2$
pressure	$\mathrm{M}\times\mathrm{L}^{-1}\times\mathrm{T}^{-2}$	pascal	barye	ϵ/σ^3
dynamic viscosity	$\mathrm{M}\times\mathrm{L}^{-1}\times\mathrm{T}^{-1}$	pascal × second	poise	$\sqrt{m\epsilon}/\sigma^2 = m/\sigma\tau$
charge	$\mathrm{I}\times\mathrm{T}$	ampère × second	franklin	q

CHAPTER

SIX

VALIDATION

The simulation package is regularly run against various tests which reproduce results from physics literature.

6.1 Simple fluids

6.1.1 Thermodynamics

Lennard–Jones potential

values for the truncated and shifted Lennard–Jones potential in three dimensions:

	cutoff radius r_c^*	density ρ^*	temperature T^*	pressure P^*	potential energy per particle U^*	isochoric specific heat c_V/k_B	isothermal compressibility $\chi_T \rho k_B T$
[1]	4.0	0.3	3.0	1.023(2)	-1.673(2)		
[2]	4.0	0.3	3.0	1.0245	-1.6717		0.654(20)
[*]	4.0	0.3	3.0	1.0234(3)	-1.6731(4)	1.648(1)	0.67(2)
[1]	4.0	0.6	3.0	3.69(1)	-3.212(3)		
[2]	4.0	0.6	3.0	3.7165	-3.2065		0.183(2)
[*]	4.0	0.6	3.0	3.6976(8)	-3.2121(2)	1.863(4)	0.184(5)

[1] Molecular dynamics simulations, J. K. Johnson, J. A. Zollweg, and K. E. Gubbins, *The Lennard-Jones equation of state revisited*, Mol. Phys. 78, 591 (1993).

[2] Integral equations theory, A. Ayadim, M. Oettel, and S Amokrane, *Optimum free energy in the reference functional approach for the integral equations theory*, J. Phys.: Condens. Matter 21, 115103 (2009).

[*] Result obtained with *HAL's MD package* (4000 particles, NVT ensemble with Nosé–Hoover chain)

6.1.2 Transport

Weeks–Chandler–Andersen potential

[1] Molecular dynamics simulations, D. Levesque and W. T. Ashurst, *Long-Time Behavior of the Velocity Autocorrelation Function for a Fluid of Soft Repulsive Particles*, Phys. Rev. Lett. 33, 277 (1974).

6.2 Binary mixtures

6.2.1 Transport

Kob–Andersen mixture

[1] Molecular dynamics simulations, P. Bordat, F. Affouard, M. Descamps, and F. Müller-Plathe, *The breakdown of the Stokes–Einstein relation in supercooled binary liquids*, J. Phys.: Condens. Matter 15, 5397 (2003).

CHAPTER

SEVEN

BENCHMARKS

The benchmark results were produced by the scripts in `examples/benchmarks`, e.g.:

```
examples/benchmarks/generate_configuration.sh lennard_jones
examples/benchmarks/run_benchmark.sh lennard_jones
```

7.1 Simple Lennard-Jones fluid in 3 dimensions

Parameters:

- 64,000 particles, number density $\rho = 0.4\sigma^3$
- force: lennard_jones ($r_c = 3\sigma, r_{\text{skin}} = 0.7\sigma$)
- integrator: verlet (NVE, $\delta t^* = 0.002$)

Hardware	time per MD step and particle	steps per second	FP precision	compilation details
Intel Xeon E5620	1.40 µs	11.2	double	GCC 4.4.1, -O3
NVIDIA Tesla S1070	58.6 ns	267	double-single	CUDA 4.2, -arch compute_12
	54.3 ns	288	single	CUDA 4.2, -arch compute_12
NVIDIA Tesla C2050	40.5 ns	386	double-single	CUDA 4.2, -arch compute_12
	34.6 ns	452	single	CUDA 4.2, -arch compute_12
NVIDIA Tesla S2050	46.4 ns	337	double-single	CUDA 4.2, -arch compute_20
	39.6 ns	395	single	CUDA 4.2, -arch compute_12
NVIDIA Tesla M2090	37.8 ns	414	double-single	CUDA 4.2, -arch compute_20
	33.2 ns	470	single	CUDA 4.2, -arch compute_12

Results were obtained from 1 independent measurement based on release version 0.2.0. Each run consisted of NVT equilibration at $T^* = 1.2$ over $\Delta t^* = 100$ (10^4 steps), followed by benchmarking 5 times 10^4 NVE steps in a row.

7.2 Supercooled binary mixture (Kob-Andersen)

Parameters:

- 256,000 particles, number density $\rho = 1.2\sigma^3$
- force: lennard_jones with 2 particle species (80% A, 20% B)

  ($\epsilon_{AA} = 1$, $\epsilon_{AB} = .5$, $\epsilon_{BB} = 1.5$, $\sigma_{AA} = 1$, $\sigma_{AB} = .88$, $\sigma_{BB} = .8$, $r_c = 2.5\sigma$, $r_{\text{skin}} = 0.5\sigma$)
- integrator: verlet (NVE, $\delta t^* = 0.001$)

Hardware	time per MD step and particle	steps per second	FP precision	compilation details
Intel Xeon E5620	1.96 µs	2.00	double	GCC 4.4.1, -O3
NVIDIA Tesla S1070	68.7 ns	56.9	double-single	CUDA 4.2, -arch compute_12
	68.4 ns	57.3	single	CUDA 4.2, -arch compute_12
NVIDIA Tesla C2050	41.9 ns	93.3	double-single	CUDA 4.2, -arch compute_12
	35.0 ns	112	single	CUDA 4.2, -arch compute_12
NVIDIA Tesla S2050	44.4 ns	88.0	double-single	CUDA 4.2, -arch compute_20
	38.0 ns	103	single	CUDA 4.2, -arch compute_12
NVIDIA Tesla M2090	35.8 ns	109	double-single	CUDA 4.2, -arch compute_20
	29.6 ns	132	single	CUDA 4.2, -arch compute_12

Results were obtained from 1 independent measurement and are based on release version 0.2.0. Each run consisted of NVT equilibration at $T^* = 0.7$ over $\Delta t^* = 100$ (2×10^4 steps), followed by benchmarking 5 times 10^4 NVE steps in a row.

CHAPTER

EIGHT

FAQ

With HALMD, I could do simulations of breath-taking quality and have obtained new scientific insight. How may I thank you?

Please acknowledge the use of HALMD in your publications by citing our article:

> P. H. Colberg and F. Höfling, *Highly accelerated simulations of glassy dynamics using GPUs: Caveats on limited floating-point precision,* Comp. Phys. Comm. *182*, 1120 (2011) [Link].

Why does HALMD abort with "[ERROR] potential energy diverged"?

An infinite potential energy sum of one or more particles indicates that the integration time-step is too large.

Try lowering the `--timestep` value.

Linking fails with: undefined reference to 'boost::filesystem3::detail::copy_file(...)'

```
Check that the ABI version of the installed Boost C++ library conforms to
C++11. This can be achieved by building Boost C++ with the GCC option
``--std=c++11``.
```

nvcc fails with 'cudafe++' died due to signal 11 (Invalid memory reference)

This is due to a bug in the CUDA compiler, which may be circumvented by including `--host-compilation=c` in the `NVCCFLAGS` environment variable passed to cmake, or in CMAKE_CUDA_FLAGS using ccmake.

nvcc fails with error: inline function '__signbit' cannot be declared weak

CUDA 3.0 (or less) is not compatible with GCC 4.4. As a work around install GCC 4.3 and place symlinks in the default CUDA compiler directory, e.g. if the CUDA toolkit is located in `/opt/cuda`, symlink `/opt/cuda/bin/{gcc,g++}` to `/usr/bin/{gcc,g++}-4.3`, respectively. The compiler directory may be overriden with the `--compiler-bindir` option.

CHAPTER

NINE

DEVELOPER GUIDE

9.1 Development of *HAL's MD package*

Your contribution to the development of *HAL's MD package* is highly appreciated. There are several ways doing so:

- The source code is available from GitHub.
- Patches can be submitted and discussed via the mailing list. (GitHub pull requests are fine, too).
- Bugs can be reported and viewed on Redmine. There is a rough roadmap available, which indicates what is planned in the next releases.
- Automatic test results can be viewed on the CDash Dashboard. If you want to contribute to the *test suite* with your own setup, get in contact with the developers and we can provide you the necessary information.

If you want to know about more about the development of *HAL's MD package*, you may continue reading the *Developer guide*.

9.2 Debugging

9.2.1 Debugging C++ exceptions

When debugging exceptions, instruct the GNU debugger to catch exceptions by executing "catch throw" before executing the program with "run".

9.2.2 Invalid device function

If you get a CUDA error "invalid device function" upon kernel launch, check whether the kernel function has a __global__ attribute.

9.3 Floating-point precision

Most of the classes in halmd/mdsim carry a template parameter `float_type`, which allows to choose between single and double floating-point precision. (This is not fully implemented yet.) In most cases, single precision will be sufficient, notable exceptions are situations where many values are accumulated. In particular, this is the case for the position and velocity variables in the velocity-Verlet integrator, which a higher precision (double or double-single) should always be used for.

The distinction of different floating-point precision ends at the level of the phase space data, i.e., after the particle positions etc. have been copied to `observables::samples::phase_space`. All derived observables are usually accumulated quantities (e.g. mean kinetic energy or density modes) and shall be computed with double precision. Although the relative statistical fluctuations of these quantities will be much larger than 10^{-7}, we shall stick to standard usage and employ double precision for the subsequent analysis. Note that still many classes in halmd/observables will carry the template parameter `float_type`, which is, however, only used to specify the precision of the input data, the results shall be in double precision in all cases.

At the level of file output (halmd/io), one may consider to write the final results in single precision as an optimisation measure to save disk space. (The high precision bits are mostly unneeded for questions in science, and such a random noise is not compressed very well.) Again, all intermediate computations from the particle coordinates down to the final results shall be done in double precision to avoid potential artifacts.

9.4 Short guide on Luabind

The *Luabind* library is used to export C++ classes to the Lua scripting interface.

The typical wrapper pattern is illustrated with code from the Verlet integrator. Let us start with the declaration of the abstract base class `halmd::mdsim::integrator`. It is qualified for Lua export by the static method `luaopen`.

```
// from file halmd/mdsim/integrator.hpp

#include <lua.hpp>

template <int dimension>
class integrator
{
public:
    static void luaopen(lua_State* L);

    integrator() {}
    virtual ~integrator() {}
    virtual void integrate() = 0;
    virtual void finalize() = 0;
    virtual double timestep() const = 0;
    virtual void timestep(double timestep) = 0;
};
```

The static method `luaopen` defines all entities that should be visible to Lua by means of *Luabind*. It is called by the library constructor, i.e., before programme execution starts.

```
// from file halmd/mdsim/integrator.cpp

#include <halmd/utility/lua/lua.hpp>

template <int dimension>
void integrator<dimension>::luaopen(lua_State* L)
{
    using namespace luabind;
    // dimension-dependent class name: integrator_2_, integrator_3_
    static std::string class_name("integrator_" + boost::lexical_cast<std::string>(dimen
    // register a new Lua module
```

```
    module(L)
    [
        namespace_("libhalmd")
        [
            namespace_("mdsim")
            [
                class_<integrator, boost::shared_ptr<integrator> >(class_name.c_str())
                    // no constructor, this is an abstract base class
                    .property("timestep", (double (integrator::*)() const) &integrator::
                    .def("integrate", &integrator::integrate)
                    .def("finalize", &integrator::finalize)
            ]
        ]
    ];
}

HALMD_LUA_API int luaopen_libhalmd_mdsim_integrator()
{
    integrator<3>::luaopen(L);
    integrator<2>::luaopen(L);
    return 0;
}
```

FIXME explain the template arguments of `class_`

FIXME explain the differences between property, def, def_readonly, add comments in code sample

FIXME explain `HALMD_LUA_API`

The actual class for the Verlet module derives from its abstract interface class. Again, it has a static method `luaopen`. Its constructor describes the dependencies from other modules (particle, box) and specific parameters (timestep). Further, it is good practice to define a type `_Base` pointing at the base class.

The class contains another static method `module_name`, which is used by **FIXME** some Lua script to select the integrator via a global command line option.

```
// from file halmd/mdsim/host/integrators/verlet.hpp

template <int dimension, typename float_type>
class verlet
  : public mdsim::integrator<dimension>
{
public:
    typedef mdsim::integrator<dimension> _Base;
    typedef host::particle<dimension, float_type> particle_type;
    typedef mdsim::box<dimension> box_type;

    static char const* module_name() { return "verlet"; }

    boost::shared_ptr<particle_type> particle;
    boost::shared_ptr<box_type> box;

    static void luaopen(lua_State* L);

    verlet(
        boost::shared_ptr<particle_type> particle
      , boost::shared_ptr<box_type> box
```

```
      , double timestep
    );
    virtual void integrate();
    virtual void finalize();
    virtual void timestep(double timestep);
    virtual double timestep() const;
};
```

Export to Lua is similar as for the base class. The main difference is that a constructor is defined using `def` and that a wrapper is needed for the static method `module_name`.

```
// from file halmd/mdsim/host/integrators/verlet.hpp

template <int dimension, typename float_type>
static char const* module_name_wrapper(verlet<dimension, float_type> const&)
{
    return verlet<dimension, float_type>::module_name();
}

template <int dimension, typename float_type>
void verlet<dimension, float_type>::luaopen(lua_State* L)
{
    using namespace luabind;
    // dimension-dependent class name: verlet_2_, verlet_3_
    static string class_name(module_name() + ("_" + lexical_cast<string>(dimension) + "_
    // register a new Lua module
    module(L)
    [
        namespace_("libhalmd")
        [
            namespace_("mdsim")
            [
                namespace_("host")
                [
                    namespace_("integrators")
                    [
                        class_<verlet, shared_ptr<_Base>, bases<_Base> >(class_name.c_st
                            .def(constructor<
                                shared_ptr<particle_type>
                              , shared_ptr<box_type>
                              , double>()
                            )
                            .property("module_name", &module_name_wrapper<dimension, flo
                    ]
                ]
            ]
        ]
    ];
}

HALMD_LUA_API int luaopen_libhalmd_mdsim_integrators_verlet()
{
    verlet<3, double>::luaopen(L);
    verlet<2, double>::luaopen(L);
    return 0;
}
```

FIXME explain the three template arguments of `class_`

FIXME explain why we need a wrapper for `module_name`. Is it due to a deficiency of luabind?

FIXME add some Lua code that exemplifies the usage of the exported module

```
require("halmd.mdsim.integrator")
integrator = assert(libhalmd.mdsim.host.integrators.verlet_2_)
print(integrator.module_name())
instance = integrator(particle, box, 0.001)
instance:integrate()
instance:finalize()
```

FIXME when do you use a `.` and when a `:` for member access? Like `core:run()` but `integrator.module_name()`?

9.4.1 Lua properties

When an object is created from a C++ class registered with Luabind, Luabind actually creates a C++ object representation object that wraps this C++ object. This means Luabind C++ objects may be extended in Lua with arbitrary member functions or variables. One method of extending a C++ object is with Luabind's `property()` function, which works analogous to Luabind's C++ `.property()`. Properties may be read-only or read-write.

In the first example, we create an object from the C++ class `potential_module`, and add a read-only Lua property `potential.name`. This is done by calling `property()` with a function as its first argument, where the function itself receives the object (`self`) and returns the property value (`"Lennard Jones"`). Note how we do not give this getter function a name, but conveniently define an unnamed function within the `property()` call.

```
local potential = libhalmd.potential_module()

-- set read-only Lua property
potential.name = property(function(self)
    return "Lennard Jones"
end)
```

In the second example, we add a read-write Lua property. We declare a local variable `name`, which is referenced by the local functions `get_name` and `set_name`. In C++ language terms, you may consider `name` a private member variable. To add the read-write property, we pass the getter and setter functions to `property()` as first and second argument, respectively.

```
-- set read-write Lua property
local name
local function get_name(self)
    return name
end
local function set_name(self, value)
    name = value
end
potential.name = property(get_name, set_name)
```

9.4.2 Debugging C++ types with class_info

Luabind provides a function `class_info`, which queries the class type of a Lua value. This is especially useful to debug `No matching overload found` errors, where the Lua value provided as an argument to a C++ function does not match the function signature(s).

`class_info` returns an object with the properties `name`, `methods` and `attributes`. In this example, we inspect a thermodynamics object:

```
local thermodynamics = halmd.observables.thermodynamics{}
local c = class_info(thermodynamics)
print(c.name)                 -- thermodynamics_3_
print(c.methods)              -- table: 0x1637390
print(c.attributes)           -- table: 0x16373e0
```

The thermodynamics class only exports a constructor function:

```
for k, v in pairs(class_info(thermodynamics).methods) do
    print(k, v)
end
-- __init  function: 0x10fd410
```

Its object provides signal slots and read-only data slots:

```
for k, v in pairs(class_info(thermodynamics).attributes) do
    print(k, v, class_info(thermodynamics[v]).name)
end
-- 1        en_kin          function<double ()>
-- 2        en_tot          function<double ()>
-- 3        prepare         signal<void ()>::slot_function_type
-- 4        en_pot          function<double ()>
-- 5        virial          function<double ()>
-- 6        pressure        function<double ()>
-- 7        sample          signal<void ()>::slot_function_type
-- 8        temp            function<double ()>
-- 9        hypervirial     function<double ()>
-- 10       v_cm            function<fixed_vector<double, 3> ()>
```

9.5 How to write a HALMD module

9.5.1 Implementation

9.5.2 Testing

9.5.3 Documentation

9.6 Redmine

9.6.1 Ticketing system

Our project development software automatically processes incoming commits. To relate to Redmine tickets in a commit message, include `refs #no` to reference a ticket or `closes #no` to close it, for example

```
This commit refs #1, #2 and fixes #3.
```

http://www.redmine.org/wiki/redmine/RedmineSettings#Referencing-issues-in-commit-messages

9.7 Coding conventions

1. New modules undergo rigorous peer reviewing before entering the branch of a release candidate.
2. Code design should observe the guidelines of H. Sutter and A. Alexandrescu in "C++ Coding Standards" (Addison Wesely, 2005).
3. TODO examplatory code fragments, formatting guidelines

```
for (unsigned int i = 0; i < count; ++i) {     //< use pre-increment
    // loop body
}
```

4. Naming rules
   - all identifiers are in lower case throughout, long names may be grouped by underscore, type identifiers end in `_type`
   - private class attributes and methods end with an underscore
   - names of containers and collections are singular
5. Use exception-safe containers and pointers

Todo

RAII, STL containers, boost::shared_ptr

9.8 Test suite

9.8.1 Structure

The test suite is divided into the following components.

test/integration

Integration tests. These tests run HALMD for a given set of command-line parameters or Lua input script, and verify the obtained results. The test execution is done with CMake scripts, the result verification with Boost Test.

test/lua

Lua unit tests. Unit tests of pure Lua components.

test/performance

Performance tests of individual HALMD components or the HALMD executable.

test/tools

Testing tools, e.g. test fixtures used in multiple tests.

test/unit

C++ unit tests. These minimal tests verify individual HALMD components.

9.8.2 Naming conventions

- Set **BOOST_TEST_MODULE** to the basename of the test source file

```
#define BOOST_TEST_MODULE lattice
```

- Use full path (with "test") to the test in the **executable name**

```
add_executable(test_unit_mdsim_positions_lattice
    lattice.cpp
)
```

- Use full path (without "test") to the test in the **CMake test name**

```
add_test(unit/mdsim/positions/lattice
    test_unit_mdsim_positions_lattice --log_level=test_suite
)
```

CHAPTER

TEN

CHANGELOG

10.1 Version 0.2.1

Improvements

- improve performance of force kernel for truncated pair interactions by about 10% due to inefficient use of the texture cache

Bug fixes

- fix regex benchmark scripts
- fix build failure with Boost C++ 1.53.0
- fix build failure with nvcc option -arch=sm_20 and CMake switch VERLET_DSFUN=FALSE

10.2 Version 0.2.0

Version 0.2.0 is a complete rewrite of branch 0.1.x, aiming at a modular code base. Most algorithms, in particular the actual MD simulation algorithms, have been kept.

This version features a slightly larger choice of potentials and NVT integrators, but it brings only rudimentary support for dynamic correlations functions.

10.3 Version 0.1.2

Improvements

- fully support mobility filters for the VACF

10.4 Version 0.1.2

Improvements

- revise documentation

Bug fixes

- fix build failure with Boost C++ 1.46

10.5 Version 0.1.1

New features

- computation of shear viscosity
- displacement/mobility filters for dynamic correlation functions

Bug fixes

- fix build failure with CUDA 3.2
- fix build failure with Boost C++ 1.42

10.6 Version 0.1.0

The first release of HAL's MD package, forming the basis for the preprint at http://arxiv.org/abs/0912.3824, later published in Comput. Phys. Commun. **182**, 1120 (2011).

CHAPTER

ELEVEN

DOCUMENTATION ARCHIVE

- cutting edge snapshot
- latest testing release
- latest stable release
- version 0.2.1
- version 0.2.0
- version 0.1.3

CHAPTER

TWELVE

USEFUL CMAKE CACHE VARIABLES

Cache variables are passed as options to CMake using `-D...`

CMAKE_BUILD_TYPE CMake build type.

For production builds with -O3 optimisation enabled, use `-DCMAKE_BUILD_TYPE=Release`.

For debugging builds with -O2 optimisation and debug symbols enabled, use `-DCMAKE_BUILD_TYPE=RelWithDebInfo`.

For builds using CUDA device emulation, use `-DCMAKE_BUILD_TYPE=DeviceEmu`.

CMAKE_PREFIX_PATH Path to third-party libraries, e.g. `-DCMAKE_PREFIX_PATH=$HOME/usr`.

This variable is only needed if libraries are installed in non-standard paths.

HALMD_USE_STATIC_LIBS Compile separate, statically linked executable for each backend.

Boost_USE_STATIC_LIBS Link to Boost libraries statically.

Recommended value is `-DBoost_USE_STATIC_LIBS=TRUE`.

HDF5_USE_STATIC_LIBS Link to HDF5 libraries statically.

Recommended value is `-DHDF5_USE_STATIC_LIBS=TRUE`.

LUA_USE_STATIC_LIBS Link to Lua libraries statically.

Recommended value is `-DLUA_USE_STATIC_LIBS=TRUE`.

HALMD_WITH_GPU Forcibly enable or disable GPU support.

By default, GPU support is enabled or disabled depending on whether CUDA is available. If HALMD_WITH_GPU is explicitly set to TRUE, CMake will fail if CUDA is not available. If HALMD_WITH_GPU is explicitly set to FALSE, GPU support will be disabled even if CUDA is available.

HALMD_POTENTIALS Semicolon-separated list of potential modules that shall be instantiated. By default, all available potentials are enabled.

HALMD_VARIANT_FORCE_DSFUN Use double-single precision functions in force summation (GPU backend only).

Default value is `TRUE`.

HALMD_VARIANT_HILBERT_ALT_3D Use alternative vertex rules for the 3D Hilbert curve used for particle ordering (GPU backend only).

Default value is `FALSE`.

HALMD_VARIANT_HOST_SINGLE_PRECISION Use single-precision math in host implementation (host backend only).

Default value is `FALSE`.

This option requires SSE, which is enabled by default on x86_64.

HALMD_VARIANT_VERLET_DSFUN Use double-single precision functions in Verlet integrator (GPU backend only).

Default value is `TRUE`.

CHAPTER

THIRTEEN

USEFUL ENVIRONMENT VARIABLES FOR CMAKE

CMAKE_PREFIX_PATH Colon-separated list with installation prefixes of third-party libraries.

This flag is useful with third-party libraries installed in non-system directories.

Example:

```
export CMAKE_PREFIX_PATH=$HOME/opt/rhel6-x86_64/lua-5.2.0:$HOME/opt/rhel6-x86_64/hd
```

CXX Path to C++ compiler.

Override default C++ compiler `c++`.

Example:

```
CXX=clang++ cmake ...
```

CXXFLAGS Compilation flags for C++ compiler.

These flags *extend* the default C++ compiler flags.

For developers, recommended value is `CXXFLAGS=-Werror` to treat warnings as errors.

Example:

```
CXXFLAGS=-Werror cmake ...
```

CUDACC Path to CUDA compiler.

Override default CUDA compiler `nvcc`.

CUDAFLAGS Compilation flags for CUDA compiler.

These flags *extend* the default CUDA compiler flags.

h

halmd, 20
halmd.io, 50
halmd.io.log, 50
halmd.io.readers, 53
halmd.io.writers, 54
halmd.mdsim, 21
halmd.mdsim.clock, 23
halmd.mdsim.core, 23
halmd.mdsim.forces, 28
halmd.mdsim.forces.trunc, 30
halmd.mdsim.integrators, 32
halmd.mdsim.particle_groups, 37
halmd.mdsim.positions, 46
halmd.mdsim.potentials, 39
halmd.mdsim.sorts, 48
halmd.mdsim.velocities, 45
halmd.numeric, 49
halmd.observables, 56
halmd.observables.dynamics, 64
halmd.observables.sampler, 60
halmd.observables.utility, 73
halmd.random, 76
halmd.utility, 76
halmd.utility.device, 76
halmd.utility.posix_signal, 77
halmd.utility.profiler, 79
halmd.utility.program_options, 79
halmd.utility.signal, 82
halmd.utility.timer_service, 82
halmd.utility.version, 82

Symbols

__eq() (halmd.mdsim.particle method), 28
__eq() (halmd.observables.utility.wavevector method), 76

A

accumulator (class in halmd.observables.utility), 73
acquire() (halmd.observables.dynamics.correlation method), 66
acquire() (halmd.observables.dynamics.helfand_moment method), 67
acquire() (halmd.observables.dynamics.intermediate_scattering_function method), 69
acquire() (halmd.observables.dynamics.mean_quartic_displacement method), 69
acquire() (halmd.observables.dynamics.mean_square_displacement method), 70
acquire() (halmd.observables.dynamics.stress_tensor_autocorrelation method), 71
acquire() (halmd.observables.dynamics.velocity_autocorrelation method), 72
acquire() (halmd.observables.phase_space method), 57
add_argument() (halmd.utility.program_options.argument_parser method), 80
add_argument_group() (halmd.utility.program_options.argument_parser method), 81
add_options() (in module halmd.observables.utility.semilog_grid), 75
add_options() (in module halmd.observables.utility.wavevector), 76
all (class in halmd.mdsim.particle_groups), 38
argument_parser (class in halmd.utility.program_options), 79
assert_kwarg() (in module halmd.utility), 83
assert_type() (in module halmd.utility), 83
author (halmd.io.readers.h5md attribute), 54
aux_enable() (halmd.mdsim.particle method), 28

B

binning (class in halmd.mdsim), 21
binning (halmd.mdsim.neighbour attribute), 26
blocking_scheme (class in halmd.observables.dynamics), 64
blocking_scheme.correlation (class in halmd.observables.dynamics), 64
boltzmann (class in halmd.mdsim.velocities), 46
Boost_USE_STATIC_LIBS, **105**
box (class in halmd.mdsim), 22

C

cell_occupancy (halmd.mdsim.neighbour attribute), 26
center_of_mass() (halmd.observables.thermodynamics method), 63
center_of_mass_velocity() (halmd.observables.thermodynamics method), 63
check() (in module halmd.io.readers.h5md), 54
close() (halmd.io.readers.h5md method), 54
close_console() (in module halmd.io.log), 53
close_file() (in module halmd.io.log), 53
CMAKE_BUILD_TYPE, **105**
CMAKE_PREFIX_PATH, **105**, **107**
collision_rate (halmd.mdsim.integrators.verlet_nvt_andersen attribute), 35
connect() (halmd.observables.dynamics.stress_tensor_autocorrelati method), 71
correlate() (halmd.observables.dynamics.correlation method), 66
correlate() (halmd.observables.dynamics.helfand_moment method), 67
correlate() (halmd.observables.dynamics.mean_quartic_displaceme method), 69
correlate() (halmd.observables.dynamics.mean_square_displaceme method), 70
correlate() (halmd.observables.dynamics.stress_tensor_autocorrelat method), 71

correlate() (halmd.observables.dynamics.velocity_autocorrelation method), 72
correlation (class in halmd.observables.dynamics), 65
correlation.writer (class in halmd.observables.dynamics), 66
count (halmd.observables.density_mode attribute), 56
count() (halmd.observables.utility.accumulator method), 73
creation_time (halmd.io.readers.h5md attribute), 54
creator (halmd.io.readers.h5md attribute), 54
creator_version (halmd.io.readers.h5md attribute), 54
CUDACC, **107**
CUDAFLAGS, **107**
CXX, **107**
CXXFLAGS, **107**

D

debug() (halmd.io.log.logger method), 52
debug() (in module halmd.io.log), 51
density() (halmd.observables.thermodynamics method), 62
density_mode (class in halmd.observables), 56
density_mode.writer (class in halmd.observables), 56
desc (halmd.observables.dynamics.correlation attribute), 66
desc (halmd.observables.dynamics.helfand_moment attribute), 68
desc (halmd.observables.dynamics.intermediate_scattering_function attribute), 69
desc (halmd.observables.dynamics.mean_quartic_displacement attribute), 69
desc (halmd.observables.dynamics.mean_square_displacement attribute), 70
desc (halmd.observables.dynamics.stress_tensor_autocorrelation attribute), 71
desc (halmd.observables.dynamics.velocity_autocorrelation attribute), 72
desc (halmd.observables.utility.accumulator attribute), 74
description (halmd.mdsim.potentials.lennard_jones attribute), 40
description (halmd.mdsim.potentials.lennard_jones_linear attribute), 41
description (halmd.mdsim.potentials.modified_lennard_jones attribute), 42
description (halmd.mdsim.potentials.morse attribute), 43
description (halmd.mdsim.potentials.power_law attribute), 44
description (halmd.mdsim.potentials.power_law_with_core attribute), 45
diag() (in module halmd.numeric), 50
dimension (halmd.mdsim.box attribute), 22
dimension (halmd.observables.phase_space.reader attribute), 59
dimension (halmd.observables.thermodynamics attribute), 63
disconnect() (halmd.mdsim.binning method), 21
disconnect() (halmd.mdsim.forces.pair_full method), 29
disconnect() (halmd.mdsim.forces.pair_trunc method), 30
disconnect() (halmd.mdsim.integrators.euler method), 33
disconnect() (halmd.mdsim.integrators.verlet method), 34
disconnect() (halmd.mdsim.integrators.verlet_nvt_andersen method), 35
disconnect() (halmd.mdsim.integrators.verlet_nvt_boltzmann method), 36
disconnect() (halmd.mdsim.integrators.verlet_nvt_hoover method), 37
disconnect() (halmd.mdsim.max_displacement method), 24
disconnect() (halmd.mdsim.neighbour method), 26
disconnect() (halmd.mdsim.positions.lattice method), 48
disconnect() (halmd.mdsim.sorts.hilbert method), 49
disconnect() (halmd.observables.density_mode method), 56
disconnect() (halmd.observables.density_mode.writer method), 57
disconnect() (halmd.observables.dynamics.blocking_scheme method), 64
disconnect() (halmd.observables.dynamics.blocking_scheme.correl method), 65
disconnect() (halmd.observables.dynamics.helfand_moment method), 68
disconnect() (halmd.observables.dynamics.intermediate_scattering_ method), 69
disconnect() (halmd.observables.phase_space method), 58
disconnect() (halmd.observables.phase_space.writer method), 58
disconnect() (halmd.observables.ssf method), 62

disconnect() (halmd.observables.ssf.writer method), 62
disconnect() (halmd.observables.thermodynamics method), 64
disconnect() (halmd.observables.utility.accumulator method), 74
disconnect() (in module halmd.utility.signal), 82
displacement (halmd.mdsim.neighbour attribute), 25

E

edges() (halmd.mdsim.box method), 22
empty() (in module halmd.utility), 83
epsilon (halmd.mdsim.potentials.lennard_jones attribute), 40
epsilon (halmd.mdsim.potentials.lennard_jones_linear attribute), 41
epsilon (halmd.mdsim.potentials.modified_lennard_jones attribute), 42
epsilon (halmd.mdsim.potentials.morse attribute), 43
epsilon (halmd.mdsim.potentials.power_law attribute), 44
epsilon (halmd.mdsim.potentials.power_law_with_core attribute), 45
error() (halmd.io.log.logger method), 52
error() (in module halmd.io.log), 51
error_of_mean() (halmd.observables.utility.accumulator method), 73
euler (class in halmd.mdsim.integrators), 32
exclude_sphere() (halmd.mdsim.positions.excluded_volume method), 47
exclude_spheres() (halmd.mdsim.positions.excluded_volume method), 47
excluded_volume (class in halmd.mdsim.positions), 47

F

finalize() (halmd.mdsim.integrators.verlet method), 34
finalize() (halmd.mdsim.integrators.verlet_nvt_andersen method), 35
finalize() (halmd.mdsim.integrators.verlet_nvt_boltzmann method), 36
finalize() (halmd.mdsim.integrators.verlet_nvt_hoover method), 37
find_comp() (in module halmd.numeric), 49
flush() (halmd.io.writers.h5md method), 56
from_range (class in halmd.mdsim.particle_groups), 38

G

generator() (in module halmd.random), 76
get_force() (halmd.mdsim.particle method), 27
get_image() (halmd.mdsim.particle method), 27
get_mass() (halmd.mdsim.particle method), 27
get_position() (halmd.mdsim.particle method), 26
get_potential_energy() (halmd.mdsim.particle method), 27
get_reverse_tag() (halmd.mdsim.particle method), 27
get_species() (halmd.mdsim.particle method), 27
get_stress_pot() (halmd.mdsim.particle method), 27
get_tag() (halmd.mdsim.particle method), 27
get_velocity() (halmd.mdsim.particle method), 27
global (halmd.mdsim.particle_groups.all attribute), 38
global (halmd.mdsim.particle_groups.from_range attribute), 39
gpu (in module halmd.utility.device), 77
group (halmd.observables.phase_space attribute), 58
group (halmd.observables.thermodynamics attribute), 63

H

h (halmd.mdsim.forces.trunc.local_r4 attribute), 32
h5md (class in halmd.io.readers), 53
h5md (class in halmd.io.writers), 55
halmd (module), 20
halmd.io (module), 50
halmd.io.log (module), 50
halmd.io.readers (module), 53
halmd.io.writers (module), 54
halmd.mdsim (module), 21
halmd.mdsim.clock (module), 23
halmd.mdsim.core (module), 23
halmd.mdsim.forces (module), 28
halmd.mdsim.forces.trunc (module), 30
halmd.mdsim.integrators (module), 32
halmd.mdsim.particle_groups (module), 37
halmd.mdsim.positions (module), 46
halmd.mdsim.potentials (module), 39
halmd.mdsim.sorts (module), 48
halmd.mdsim.velocities (module), 45
halmd.numeric (module), 49
halmd.observables (module), 56
halmd.observables.dynamics (module), 64
halmd.observables.sampler (module), 60
halmd.observables.utility (module), 73

halmd.random (module), 76
halmd.utility (module), 76
halmd.utility.device (module), 76
halmd.utility.posix_signal (module), 77
halmd.utility.profiler (module), 79
halmd.utility.program_options (module), 79
halmd.utility.signal (module), 82
halmd.utility.timer_service (module), 82
halmd.utility.version (module), 82
HALMD_POTENTIALS, **105**
HALMD_USE_STATIC_LIBS, **105**
HALMD_VARIANT_FORCE_DSFUN, **105**
HALMD_VARIANT_HILBERT_ALT_3D, **105**
HALMD_VARIANT_HOST_SINGLE_PRECISION, **106**
HALMD_VARIANT_VERLET_DSFUN, **106**
HALMD_WITH_GPU, **105**
HDF5_USE_STATIC_LIBS, **105**
helfand_moment (class in halmd.observables.dynamics), 67
helfand_moment.writer (class in halmd.observables.dynamics), 68
hilbert (class in halmd.mdsim.sorts), 48

I

index (halmd.mdsim.potentials.power_law attribute), 44
index (halmd.mdsim.potentials.power_law_with_core attribute), 45
index_m (halmd.mdsim.potentials.modified_lennard_jones attribute), 42
index_n (halmd.mdsim.potentials.modified_lennard_jones attribute), 42
info() (halmd.io.log.logger method), 52
info() (in module halmd.io.log), 51
integrate() (halmd.mdsim.integrators.euler method), 33
integrate() (halmd.mdsim.integrators.verlet method), 34
integrate() (halmd.mdsim.integrators.verlet_nvt_andersen method), 35
integrate() (halmd.mdsim.integrators.verlet_nvt_boltzmann method), 36
integrate() (halmd.mdsim.integrators.verlet_nvt_hoover method), 37
intermediate_scattering_function (class in halmd.observables.dynamics), 68
intermediate_scattering_function.writer (class in halmd.observables.dynamics), 69
internal_energy() (halmd.observables.thermodynamics method), 63
interval (halmd.mdsim.integrators.verlet_nvt_boltzmann attribute), 36

K

keys() (in module halmd.utility), 83
kinetic_energy() (halmd.observables.thermodynamics method), 62

L

label (halmd.mdsim.particle attribute), 26
label (halmd.observables.density_mode attribute), 56
label (halmd.observables.dynamics.intermediate_scattering_functio attribute), 69
label (halmd.observables.ssf attribute), 62
lattice (class in halmd.mdsim.positions), 48
length (halmd.mdsim.box attribute), 22
lennard_jones (class in halmd.mdsim.potentials), 39
lennard_jones_linear (class in halmd.mdsim.potentials), 40
loader() (in module halmd.utility.module), 77
local_r4 (class in halmd.mdsim.forces.trunc), 32
log() (halmd.mdsim.forces.trunc.local_r4 method), 32
logger (class in halmd.io.log), 52
LUA_USE_STATIC_LIBS, **105**

M

mass (halmd.mdsim.integrators.verlet_nvt_hoover attribute), 37
mass() (halmd.observables.phase_space method), 58
max() (in module halmd.numeric), 49
max_displacement (class in halmd.mdsim), 24
mdstep() (in module halmd.mdsim.core), 24
mean() (halmd.observables.utility.accumulator method), 73
mean_mass() (halmd.observables.thermodynamics method), 63
mean_quartic_displacement (class in halmd.observables.dynamics), 69
mean_quartic_displacement.writer (class in halmd.observables.dynamics), 70
mean_square_displacement (class in halmd.observables.dynamics), 70
mean_square_displacement.writer (class in halmd.observables.dynamics), 70
memory (halmd.mdsim.particle attribute), 26
memory (halmd.mdsim.potentials.lennard_jones attribute), 40

memory (halmd.mdsim.potentials.lennard_jones_linear attribute), 41
memory (halmd.mdsim.potentials.modified_lennard_jones attribute), 42
memory (halmd.mdsim.potentials.morse attribute), 43
memory (halmd.mdsim.potentials.power_law attribute), 44
memory (halmd.mdsim.potentials.power_law_with_cor attribute), 45
min() (in module halmd.numeric), 49
modified_lennard_jones (class in halmd.mdsim.potentials), 41
module (class in halmd.utility), 77
morse (class in halmd.mdsim.potentials), 42
multi_index_to_offset() (in module halmd.numeric), 50

N

neighbour (class in halmd.mdsim), 25
nparticle (halmd.mdsim.particle attribute), 26
nparticle (halmd.observables.phase_space.reader attribute), 59
nspecies (halmd.mdsim.particle attribute), 26
nspecies (halmd.observables.phase_space.reader attribute), 59

O

offset_to_multi_index() (in module halmd.numeric), 50
on_alrm() (in module halmd.utility.posix_signal), 78
on_append_finalize() (in module halmd.mdsim.core), 24
on_append_force() (halmd.mdsim.particle method), 28
on_append_integrate() (in module halmd.mdsim.core), 24
on_append_profile() (in module halmd.utility.profiler), 79
on_cont() (in module halmd.utility.posix_signal), 78
on_finalize() (in module halmd.mdsim.core), 24
on_finish() (in module halmd.observables.sampler), 61
on_force() (halmd.mdsim.particle method), 28
on_hup() (in module halmd.utility.posix_signal), 78
on_int() (in module halmd.utility.posix_signal), 78
on_integrate() (in module halmd.mdsim.core), 24
on_periodic() (in module halmd.utility.timer_service), 82
on_prepare() (in module halmd.observables.sampler), 60
on_prepend_finalize() (in module halmd.mdsim.core), 24
on_prepend_force() (halmd.mdsim.particle method), 28
on_prepend_integrate() (in module halmd.mdsim.core), 24
on_prepend_profile() (in module halmd.utility.profiler), 79
on_profile() (in module halmd.utility.profiler), 79
on_sample() (in module halmd.observables.sampler), 61
on_set_timestep() (in module halmd.mdsim.clock), 23
on_start() (in module halmd.observables.sampler), 61
on_term() (in module halmd.utility.posix_signal), 78
on_tstp() (in module halmd.utility.posix_signal), 78
on_ttin() (in module halmd.utility.posix_signal), 78
on_ttou() (in module halmd.utility.posix_signal), 79
on_usr1() (in module halmd.utility.posix_signal), 78
on_usr2() (in module halmd.utility.posix_signal), 78
open_console() (in module halmd.io.log), 53
open_file() (in module halmd.io.log), 53
order() (halmd.mdsim.sorts.hilbert method), 49
origin() (halmd.mdsim.box method), 22

P

pair_full (class in halmd.mdsim.forces), 28
pair_trunc (class in halmd.mdsim.forces), 29
parse_args() (halmd.utility.program_options.argument_parser method), 81
particle (class in halmd.mdsim), 26
particle (halmd.mdsim.binning attribute), 21
particle (halmd.mdsim.max_displacement attribute), 24
particle (halmd.mdsim.neighbour attribute), 25
particle (halmd.mdsim.particle_groups.all attribute), 38
particle (halmd.mdsim.particle_groups.from_range attribute), 39
particle_number() (halmd.observables.thermodynamics method), 62
path (halmd.io.readers.h5md attribute), 54

path (halmd.io.writers.h5md attribute), 56
phase_space (class in halmd.observables), 57
phase_space.writer (class in halmd.observables), 58
place_spheres() (halmd.mdsim.positions.excluded_volume method), 47
poll() (in module halmd.utility.posix_signal), 79
position() (halmd.observables.phase_space method), 57
potential (halmd.mdsim.forces.pair_full attribute), 29
potential (halmd.mdsim.forces.pair_trunc attribute), 30
potential_energy() (halmd.observables.thermodynamics method), 62
power_law (class in halmd.mdsim.potentials), 43
power_law_with_core (class in halmd.mdsim.potentials), 45
pressure() (halmd.observables.thermodynamics method), 63
process() (in module halmd.utility.timer_service), 82
prod() (in module halmd.numeric), 49
profile() (in module halmd.utility.profiler), 79
prologue() (in module halmd.utility.version), 82

R

r_core_sigma (halmd.mdsim.potentials.power_law_with_core attribute), 45
r_cut (halmd.mdsim.binning attribute), 21
r_cut (halmd.mdsim.potentials.lennard_jones attribute), 40
r_cut (halmd.mdsim.potentials.lennard_jones_linear attribute), 41
r_cut (halmd.mdsim.potentials.modified_lennard_jones attribute), 42
r_cut (halmd.mdsim.potentials.morse attribute), 43
r_cut (halmd.mdsim.potentials.power_law attribute), 44
r_cut (halmd.mdsim.potentials.power_law_with_core attribute), 45
r_cut_sigma (halmd.mdsim.potentials.lennard_jones attribute), 40
r_cut_sigma (halmd.mdsim.potentials.lennard_jones_linear attribute), 41
r_cut_sigma (halmd.mdsim.potentials.modified_lennard_jones attribute), 42
r_cut_sigma (halmd.mdsim.potentials.morse attribute), 43
r_cut_sigma (halmd.mdsim.potentials.power_law attribute), 44
r_cut_sigma (halmd.mdsim.potentials.power_law_with_core attribute), 45
r_sigma (halmd.mdsim.potentials.morse attribute), 43
r_skin (halmd.mdsim.binning attribute), 21
r_skin (halmd.mdsim.neighbour attribute), 26
rate (halmd.mdsim.integrators.verlet_nvt_boltzmann attribute), 36
read_at_step() (halmd.observables.phase_space.reader method), 59
read_at_time() (halmd.observables.phase_space.reader method), 59
reader (class in halmd.observables.phase_space), 58
reader() (halmd.io.readers.h5md method), 53
reader() (in module halmd.mdsim.box), 23
rescale_velocity() (halmd.mdsim.particle method), 27
rescale_velocity_group() (halmd.mdsim.particle method), 27
reset() (halmd.observables.utility.accumulator method), 73
resonance_frequency (halmd.mdsim.integrators.verlet_nvt_hoover attribute), 37
reverse() (in module halmd.utility), 83
root (halmd.io.readers.h5md attribute), 54
root (halmd.io.writers.h5md attribute), 56
run() (in module halmd.observables.sampler), 60
runtime_estimate (class in halmd.observables), 60

S

sample() (halmd.observables.utility.accumulator method), 73
sample() (in module halmd.observables.sampler), 60
sampler (halmd.observables.ssf attribute), 62
scalar_matrix() (in module halmd.numeric), 49
seed() (in module halmd.random), 76
semilog_grid (class in halmd.observables.utility), 74
set() (halmd.mdsim.positions.lattice method), 48
set() (halmd.mdsim.velocities.boltzmann method), 46
set() (halmd.observables.phase_space method), 58
set_defaults() (halmd.utility.program_options.argument_parser method), 81
set_image() (halmd.mdsim.particle method), 27

set_mass() (halmd.mdsim.integrators.verlet_nvt_hoover method), 37
set_mass() (halmd.mdsim.particle method), 27
set_position() (halmd.mdsim.particle method), 26
set_reverse_tag() (halmd.mdsim.particle method), 27
set_species() (halmd.mdsim.particle method), 27
set_tag() (halmd.mdsim.particle method), 27
set_temperature() (halmd.mdsim.integrators.verlet_nvt_andersen method), 35
set_temperature() (halmd.mdsim.integrators.verlet_nvt_boltzmann method), 36
set_temperature() (halmd.mdsim.integrators.verlet_nvt_hoover method), 37
set_timestep() (halmd.mdsim.integrators.euler method), 33
set_timestep() (halmd.mdsim.integrators.verlet method), 34
set_timestep() (halmd.mdsim.integrators.verlet_nvt_andersen method), 34
set_timestep() (halmd.mdsim.integrators.verlet_nvt_boltzmann method), 35
set_timestep() (halmd.mdsim.integrators.verlet_nvt_hoover method), 37
set_timestep() (in module halmd.mdsim.clock), 23
set_velocity() (halmd.mdsim.particle method), 27
shift_rescale_velocity() (halmd.mdsim.particle method), 27
shift_rescale_velocity_group() (halmd.mdsim.particle method), 28
shift_velocity() (halmd.mdsim.particle method), 27
shift_velocity_group() (halmd.mdsim.particle method), 27
sigma (halmd.mdsim.potentials.lennard_jones attribute), 40
sigma (halmd.mdsim.potentials.lennard_jones_linear attribute), 41
sigma (halmd.mdsim.potentials.modified_lennard_jones attribute), 42
sigma (halmd.mdsim.potentials.morse attribute), 43
sigma (halmd.mdsim.potentials.power_law attribute), 44
sigma (halmd.mdsim.potentials.power_law_with_core attribute), 45
size (halmd.mdsim.particle_groups.all attribute), 38
size (halmd.mdsim.particle_groups.from_range attribute), 39
slab (halmd.mdsim.positions.lattice attribute), 48
sorted() (in module halmd.utility), 83
species() (halmd.observables.phase_space method), 58
ssf (class in halmd.observables), 61
ssf.writer (class in halmd.observables), 62
step (in module halmd.mdsim.clock), 23
stress_tensor() (halmd.observables.thermodynamics method), 63
stress_tensor_autocorrelation (class in halmd.observables.dynamics), 71
stress_tensor_autocorrelation.writer (class in halmd.observables.dynamics), 72
sum_over (halmd.observables.utility.accumulator method), 73
sum() (in module halmd.numeric), 49

T

temperature (halmd.mdsim.integrators.verlet_nvt_andersen attribute), 35
temperature (halmd.mdsim.integrators.verlet_nvt_boltzmann attribute), 36
temperature (halmd.mdsim.integrators.verlet_nvt_hoover attribute), 37
temperature (halmd.mdsim.velocities.boltzmann attribute), 46
temperature() (halmd.observables.thermodynamics method), 63
thermodynamics (class in halmd.observables), 62
time (in module halmd.mdsim.clock), 23
timestep (halmd.mdsim.integrators.euler attribute), 33
timestep (halmd.mdsim.integrators.verlet attribute), 34
timestep (halmd.mdsim.integrators.verlet_nvt_andersen attribute), 34
timestep (halmd.mdsim.integrators.verlet_nvt_boltzmann attribute), 36
timestep (halmd.mdsim.integrators.verlet_nvt_hoover attribute), 37
timestep (in module halmd.mdsim.clock), 23
trace() (halmd.io.log.logger method), 52
trace() (in module halmd.io.log), 51
trans() (in module halmd.numeric), 50

V

value (in module halmd.observables.utility.semilog_grid), 74
value() (halmd.observables.utility.wavevector method), 75
variance() (halmd.observables.utility.accumulator method), 73

velocity() (halmd.observables.phase_space method), 57
velocity_autocorrelation (class in halmd.observables.dynamics), 72
velocity_autocorrelation.writer (class in halmd.observables.dynamics), 72
verlet (class in halmd.mdsim.integrators), 33
verlet_nvt_andersen (class in halmd.mdsim.integrators), 34
verlet_nvt_boltzmann (class in halmd.mdsim.integrators), 35
verlet_nvt_hoover (class in halmd.mdsim.integrators), 36
version (halmd.io.readers.h5md attribute), 54
version() (in module halmd.io.writers.h5md), 56
virial() (halmd.observables.thermodynamics method), 63
volume (halmd.mdsim.box attribute), 22

W

wait() (in module halmd.utility.posix_signal), 79
warning() (halmd.io.log.logger method), 52
warning() (in module halmd.io.log), 51
wavenumber() (halmd.observables.utility.wavevector method), 75
wavevector (class in halmd.observables.utility), 75
wavevector (halmd.observables.density_mode attribute), 56
writer() (halmd.io.writers.h5md method), 55
writer() (halmd.mdsim.box method), 22
writer() (halmd.observables.thermodynamics method), 63
writer() (halmd.observables.utility.accumulator method), 74