HALMD ● HAL's MD package

Highly Accelerated Large-scale Molecular Dynamics

Documentation of HAL's MD package

Release 1.0.0

Felix Höfling, Peter Colberg, Nicolas Höft, Roya Ebrahimi Viand

Oct 13, 2021

CONTENTS

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 and Daniel Kirchner for their manifold contributions.

Note: A description of the implementation, performance tests, numerical stability tests, and an application to slow glassy dynamics can be found in the article by P. H. Colberg and F. 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 software under LGPL-3+ 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

The following release tarballs of *HAL's MD package* are available:

- latest testing release
- version 1.0.0

Developers' previews on version 1.0:

- version 1.0-alpha6
- version 1.0-alpha5
- version 1.0-alpha4
- version 1.0-alpha3
- version 1.0-alpha2
- version 1.0-alpha1
- version 0.2.1
- version 0.2.0
- version 0.1.3

Download and verify the integrity of the tar balls with the following command (in bash):

```
TAR="halmd-1.0.0.tar.bz2" URL="http://code.halmd.org/tar"; \
wget "$URL/$TAR" && openssl dgst -sha512 \
  -verify <(wget -qO- "$URL/cert.pem" | openssl x509 -noout -pubkey) \
  -signature <(wget -qO- "$URL/$TAR.sig") "$TAR"
```

The files are signed with an X.509 certificate issued for Felix Hoefling by MPG-CA. The SHA1 fingerprint of the certificate is BA:FB:6E:55:81:8E:C5:4C:DD:47:93:52:CF:EA:1D:FE:0B:4D:35:8A as shown by:

```
wget -qO- http://code.halmd.org/tar/cert.pem | openssl x509 -fingerprint -
↪noout
```

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 parallel 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 install-TAB TAB
```

```
install-boost     install-gcc       install-graphviz  install-lua       ␣
↪install-ninja
install-clang     install-gdb       install-halmd     install-luajit    ␣
↪install-nvcuda-tools
install-cmake     install-git       install-hdf5      install-luatrace  ␣
↪install-python-sphinx
```

Each `install-` rule depends on rules `fetch-`, `extract-`, `configure-`, and `build-`.

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 $\geq$ 4.7)
- some standard tools: wget, tar, gzip, rm, cp, touch, patch
- optionally, the bzip2 library for Boost.IOStreams

On a Debian system, install the following packages:

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

Boost C++ libraries

The compilation of Boost C++ requires GCC $\geq$ 4.7. If your distribution comes with GCC = 4.6 specify:

```
make -f .../halmd/examples/packages.mk BOOST_ABI=c++0x install-boost
```

Recent versions of non-GCC compilers may be used by setting `BOOST_TOOLSET` accordingly, e.g., `clang` for the Clang compiler, `intel` for Intel's C++ Compiler, or `pgi` for PGI's C++ compiler:

```
make -f .../halmd/examples/packages.mk BOOST_TOOLSET=clang install-boost
```

The bzip2 library is necessary to build Boost.IOStreams (files `bzlib.h` and `libbz2.so`). As *HAL's MD package* does not make use of this library, you may opt to compile the Boost C++ libraries without bzip2 support by prepending `NO_BZIP2=1` to the make command.

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-1.65.0` or `~/opt/Sphinx-1.6.3`. 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_65_0:/opt/cuda-8.5:~/opt/lua-5.2.3 cmake ~/
↪projects/halmd
```

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="${HOME}/opt/boost_1_65_0${CMAKE_PREFIX_PATH+:
↪$CMAKE_PREFIX_PATH}"
export CMAKE_PREFIX_PATH="/opt/cuda-8.5${CMAKE_PREFIX_PATH+:$CMAKE_PREFIX_
↪PATH}"
export CMAKE_PREFIX_PATH="${HOME}/opt/lua-5.2.3${CMAKE_PREFIX_PATH+:$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, currently CMake 3.5.2.

Extract the CMake source package, and prepare the CMake build with

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

Compile CMake with

```
make
```

Install CMake into your packages directory:

```
make install
```

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

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

Boost C++ libraries

Get the latest Boost source package, currently Boost 1.72.0 (1.55.0 is the minimum required version, some unit tests will be disabled in Boost versions less than 1.59.0).

To build Boost, extract the source package and bootstrap the build with

```
./bootstrap.sh
```

Compile Boost using

```
./b2 cxxflags="-fPIC -std=c++11"
```

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:

```
./b2 cxxflags="-fPIC -std=c++11" install --prefix=$HOME/opt/boost_1_72_0
```

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

```
export CMAKE_PREFIX_PATH="${HOME}/opt/boost_1_72_0${CMAKE_PREFIX_PATH+:
↪$CMAKE_PREFIX_PATH}"
export LD_LIBRARY_PATH="${HOME}/opt/boost_1_72_0/lib${LD_LIBRARY_PATH+:$LD_
↪LIBRARY_PATH}"
```

Lua interpreter

Get the latest Lua source package from the Lua download page, currently Lua 5.2.4.

Extract the Lua source package.

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-5.2.4
```

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

```
export CMAKE_PREFIX_PATH="${HOME}/opt/lua-5.2.4${CMAKE_PREFIX_PATH+:$CMAKE_
↪PREFIX_PATH}"
export PATH="${HOME}/opt/lua-5.2.4/bin${PATH+:$PATH}"
export MANPATH="${HOME}/opt/lua-5.2.4/man${MANPATH+:$MANPATH}"
```

HDF5 library

Get the latest HDF5 source package, currently HDF5 1.10.0-patch1.

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

```
CFLAGS=-fPIC CXXFLAGS=-fPIC ./configure --enable-cxx --enable-static --
↪prefix=$HOME/opt/hdf5-1.10.0-patch1
```

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="${HOME}/opt/hdf5-1.10.0-patch1/bin${PATH+:$PATH}"
export CMAKE_PREFIX_PATH="${HOME}/opt/hdf5-1.10.0-patch1${CMAKE_PREFIX_
↪PATH+:$CMAKE_PREFIX_PATH}"
```

Sphinx documentation generator

Get the latest Sphinx source package, currently Sphinx 1.2.3.

Query your Python version

```
python -V
```

Create a package directory for Sphinx using the Python major and minor version (here 2.7)

```
mkdir -p $HOME/opt/Sphinx-1.2.3/lib/python2.7/site-packages
```

Add the package directory to the PYTHON_PATH environment variable

```
export PYTHONPATH="${HOME}/opt/Sphinx-1.2.3/lib/python2.7/site-packages${
↪PYTHONPATH+:$PYTHONPATH}"
```

Install Sphinx into your packages directory

```
python setup.py install --prefix=$HOME/opt/Sphinx-1.2.3
```

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

```
export PATH="${HOME}/opt/Sphinx-1.2.3/bin${PATH+:$PATH}"
export PYTHONPATH="${HOME}/opt/Sphinx-1.2.3/lib/python2.7/site-packages$
↪{PYTHONPATH+:$PYTHONPATH}"
```

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*.

2.2.3 Build environment

- a *C++ compiler* with C++11 support, builds have been tested with GCC and Clang

  HALMD makes extensive use of C++11 features. Instructions for a semi-automated build of GCC are given in *Automatic installation*.

- CMake $\geq$ 2.8.12, latest version tested: 3.11.1

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

- NVIDIA CUDA toolkit $\geq$ 5.0, latest version tested: 8.5

  Please refer to the installation instructions shipped with the toolkit. The toolkit is not needed for builds without GPU acceleration.

2.2.4 Third-party libraries

- Boost C++ Libraries $\geq$ 1.55.0 (some unit tests will be disabled in Boost versions less than 1.59.0), latest version tested: 1.72.0

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

  Note: The the release of the Boost C++ library must not be newer than the CMake release used. Otherwise, indirect library dependencies may not be resolved properly, see *FAQ*.

  On older distributions based on GCC 4, the shipped Boost C++ Libraries can not be used due to an ABI incompatibility (C++98 vs. C++11). For instructions how to build Boost C++ with the C++11 ABI, see *Automatic installation*.

- Lua interpreter $\geq$ 5.1, $<$ 5.3 *or* Lua Just-In-Time compiler $\geq$ 2.0

  Note: Lua 5.3 is not supported. The Lua JIT compiler is recommended for advanced simulation scripts containing loops, user-defined correlation functions, etc.

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

- HDF5 C++ Library $\geq$ 1.8.13, latest version tested: 1.10.1

  "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."

2.2.5 Documentation

Documentation is generated **optionally** in HTML and PDF format if these prerequisites are met:

- Sphinx documentation generator $\geq$ 1.4, latest version tested: 1.6.3

  "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*.

Note: CMake $\leq$ 3.0 may have problems locating a CUDA host compiler (especially if you use nvcc.profile to specify a compiler directory). If you experience errors like "-m64: No such file or directory" when compiling cuda sources, the problem can be solved by manually setting the CMake variable CUDA_HOST_COMPILER to the full path of the desired host compiler. CMake $\geq$ 3.1 fixes this problem.

The problem does not occur for builds without GPU acceleration.

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. Building has been tested with the following compilers:

- GCC
  - GCC 5.1.0 (upstream) on openSuSE 12.3 (x86_64) and CentOS 6.3 (x86_64)
  - GCC 4.9.2, 4.8.2 (upstream) on openSuSE 12.3 (x86_64) and CentOS 6.3 (x86_64)
  - GCC 4.7.3 (upstream) on CentOS 6.3 (x86_64)
  - GCC 4.7.2 on openSuSE 12.3 (x86_64)
  - GCC 4.6.2 on openSuSE 12.1 (x86_64)

  Note: Building with GCC $\geq$ 5.1 requires the macro definition `_GLIBCXX_USE_CXX11_ABI=0`, which may be passed via the environment variable `CXXFLAGS=-D ...`. For the background, see GCC Dual ABI.

- Clang
  - Clang 3.6.0 (upstream) on openSuSE 12.3 (x86_64)
  - Clang 3.5.0, 3.4, 3.3, 3.2 (upstream) on openSuSE 12.3 (x86_64) and CentOS 6.3 (x86_64)

The following C++ compilers **fail** to build HALMD:

- GCC $\leq$ 4.5

- Clang $\leq$ 2.7
  - Clang 2.7 on Debian GNU/Linux squeeze (x86_64)
- Intel C++ compiler $\leq$ 14.0
- 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-toolkit
```

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-hdf5 install-cmake
```

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. For further details about the simulation scripts, see *Simulation Scripts*.

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, $2\tau^{-1}$, 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 been 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 \
```

(continues on next page)

(continued from previous page)

```
  --input 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:

```
h5ls -d output.h5/observables/potential_energy | less
```

For 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
plotting/plot_h5md.py --no-plot --range 20 100 output.h5
```

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

3.2 Simulation Scripts

HAL's MD package is configured and steered with customisable simulation scripts written in Lua 5.

3.2.1 Structure

A simulation script must define a global method `main`, which sets up and runs the simulation. Optionally, a global method `define_args` can be defined to specify custom command line arguments, see *Program Options*.

halmd
: Predefined global that holds the *Modules*.

main (*args*)
: Main simulation entry point (defined by the simulation script).

  Parameters **args** (*table*) – command line arguments as returned from parser

define_args (*parser*)
: Optional entry point (defined by the simulation script) that, if present, is called prior to *main()* and can be used to define command line arguments.

  Parameters **parser** – instance of *halmd.utility.program_options.argument_parser*

3.2.2 Examples

Complete examples of simulation scripts can be found in `share/doc/halmd/examples` in the installation directory. A minimal simulation script is of the following form:

```
-- grab module namespaces
local log = halmd.io.log

function main(args)
    -- some output to logger
    log.info("Write 'Hello World!' to " .. args.output .. ".log")

    -- here: setup system and run simulation
end

function define_args(parser)
    parser:add_argument("output,o", {type = "string", action = parser.
↪substitute_date_time_action,
        default = "project_%Y%m%d_%H%M%S", help = "prefix of output files"}
↪)

    parser:add_argument("random-seed", {type = "integer", action = parser.
↪random_seed_action,
        help = "seed for random number generator"})

    return parser
end
```

3.2.3 Advanced

The `main` and `define_args` functions are called from the internal lua script `run.lua`, which is found in `share/halmd/lua` in the installation directory. This script defines a few standard command line arguments, calls the custom method `define_args` if present, executes the argument parser, and performs some initial setup, e.g., of the logger. Advanced users may modify this script to change the predefined behaviour.

Backwards compatibility with versions $0.3 < x < 1.0$ of *HAL´s MD package* is achieved by replacing the script `run.lua` with an empty script. In this case, the simulation script is run directly and all initial setup is in the responsibility of the user.

3.3 H5MD data files

3.3.1 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. **185**, 1546 (2014), [arXiv:1308.6382]

3.3.2 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.h5
```

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

```
h5dump -A file.h5
```

The contents of individual groups or datasets may be displayed either by

```
h5ls -v file.h5/path/to/group
h5ls -d file.h5/path/to/dataset
```

or this way:

```
h5dump -g /path/to/group file.h5
h5dump -d /path/to/dataset file.h5
```

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 an H5MD file and print a dataset:

```
import h5py
f = h5py.File("file.h5", "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.

3.4 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, but do the analysis')
    parser.add_argument('--group', help='particle group (default:
↪%(default)s)', default='all')
    parser.add_argument('input', metavar='INPUT', help='H5MD input file␣
↪with data for state variables')
    args = parser.parse_args()

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

    # open and read data file
    H5 = h5py.File(args.input, 'r')
    H5obs = H5['observables']
    H5particle = H5['particles'][args.group]
    H5box = H5particle['box']
```

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}'.format(H5obs['particle_number'][()])
    print 'Box size:',
    for x in diag(H5box['edges']):
        print ' {0:g}'.format(x),
    print
    print 'Density: {0:g}'.format(H5obs['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')
```

(continues on next page)

(continued from previous page)

```
    # full support for slicing (the second pair of brackets) requires␣
↪conversion to a NumPy array before
    temp, temp_err = compute_average(array(H5obs['temperature/value
↪'])[range[0]:range[1]], 'Temperature')
    pressure, pressure_err = compute_average(array(H5obs['pressure/value
↪'])[range[0]:range[1]], 'Pressure')
    epot, epot_err = compute_average(array(H5obs['potential_energy/value
↪'])[range[0]:range[1]], 'Potential energy')
```

```
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/value'])[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 $E_\mathrm{pot}$')
```

(continues on next page)

(continued from previous page)

```
        legend(loc='best', frameon=False)
        show()
```

3.5 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.

3.5.1 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
```

3.5.2 nvlock tool

Warning: The mechanism used by `nvlock` to make certain GPUs invisible to the NVIDIA driver does no longer work with recent drivers, e.g., later than version 346. Please use the environment variable *CUDA_VISIBLE_DEVICES* instead. (Caveat: with this, the reported GPU IDs are re-enumerated, always starting with 0.)

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/halmd-org/nvcuda-tools
```

and is compiled with

```
cmake .
make
```

3.6 Tutorial: diffusion in a simple liquid

3.6.1 Prerequisites

You need a proper installation of *HAL's MD package* and, for the analysis part, Python with the h5py package. The HDF5 tools (e.g., `h5ls`) are not required, but of advantage.

These simulation scripts are found in the folder `/share/doc/halmd/examples` relative to the installation path:

- liquid/lennard_jones_equilibration.lua
- liquid/lennard_jones.lua
- liquid/rescale_velocity.lua
- plotting/plot_h5md.py

3.6.2 Equilibration phase

Melting of the initial lattice configuration with an NVT simulation:

```
halmd lennard_jones_equilibration.lua -v \
    --output lj_thermalisation
    --density 0.7 \
    --temperature 1.3 \
    --cutoff 2.5 \
    --time 1e2 \
    --sampling state-vars=200
```

Inspect the output and determine the mean potential energy from the second half:

```
h5ls -d lj_thermalisation.h5/observables/potential_energy
plot_h5md.py lj_thermalisation.h5
plot_h5md.py --no-plot --range 50 -1 lj_thermalisation.h5
```

You should obtain a potential energy per particle of $u_{\text{pot}} \approx -3.957\epsilon$, from which we calculate the internal (or: total) energy as $u_{\text{int}} = u_{\text{pot}} + 3k_{\text{B}}T/2 \approx -2.007\epsilon$.

Continue the simulation in the NVE ensemble, i.e., truly Newtonian dynamics, for a similar period of time. Before, we slightly rescale the velocities (< 1%) to match the average internal energy. And some expensive observables are turned off:

```
halmd lennard_jones.lua -v \
    --output lj_equilibration \
    --input lj_thermalisation.h5 \
    --rescale-to-energy -2.007 \
    --cutoff 2.5 \
    --time 1e2 \
    --sampling structure=0 correlation=0
```

Note: It is crucial to specify the cutoff here again. A better way would be to define the potential in a small lua file, which is included by both simulation scripts.

Let us check some thermodynamic properties again:

```
plot_h5md.py --no-plot --range 50 -1 lj_equilibration.h5
```

3.6.3 Production phase

The same script is then used for production, with all sampling turned on:

```
halmd lennard_jones.lua -v \
    --output lj_production \
    --input lj_equilibration.h5 \
    --cutoff 2.5 \
    --time 1e3
```

No energy rescaling is applied here.

3.6.4 Diffusion analysis

The diffusion constant can be obtained from the mean-square displacement (MSD) data, $\delta r^2(t)$. The latter are found in the output file `lj_production.h5` of the last run at the following location within the data structure: `dynamics/all/mean_square_displacement`. The Python script provided at `examples/plotting/diffusion_constant.py` performs the analysis described in the following. An exemplary invocation that limits the fit range is:

```
diffusion_constant.py --range 0 -6 --min-time 100 lj_production.h5
```

The result for the diffusion constant should be in the range $D = (0.1495 \pm 0.0005)\sigma^2\tau^{-1}$.

For convenience, a command line interface is defined at the end of the analysis script. The result of the option parser is then passed to the `main` function.

```
def parse_args():
    import argparse

    # define and parse command line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('--group', default='all', help='particle group in␣
↪H5MD file')
    parser.add_argument('--dimension', type=int, default=3, help='space␣
↪dimension')
    parser.add_argument('--no-plot', action='store_true', help='do not␣
↪produce plots, but do the analysis')
    parser.add_argument('--rectify', action='store_true', help='rectify␣
↪plot by showing MSD(t) / t')
    parser.add_argument('--range', type=int, nargs=2, help='range of data␣
↪points to include in fit')
    parser.add_argument('--min-time', type=float, nargs=1, help='left end␣
↪of time interval used for fitting')
    parser.add_argument('input', metavar='INPUT', help='H5MD input file␣
↪with MSD data')
    return parser.parse_args()
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    args = parse_args()
```

The input file is passed as positional argument (without keyword). The `main` function begins by loading some libraries ...

```
def main(args):
    import h5py
    import numpy as np
    from scipy.optimize import curve_fit
```

... and reading the MSD data from the H5MD input file. The data are originally arranged in overlapping blocks of time grids of growing resolution. For our purpose, we flatten the arrays and bring them in chronological order first.

```
    with h5py.File(args.input, 'r') as H5:
        msd = H5['dynamics/{0}/mean_square_displacement'.format(args.
↪group)]

        # get relavant data for MSD calculation
        x = np.array(msd['time']).flatten()
        y = np.array(msd['value']).flatten()
        yerr = np.array(msd['error']).flatten()

    # bring data in chronological order
    idx = np.argsort(x)
    x, y, yerr = x[idx], y[idx], yerr[idx]
```

The diffusion law, $\delta r^2(t) \simeq 6Dt$, holds asymptotically only, and the data at long times are noisy. So we restrict the fit by evaluating the command-line options `--range` and `--min-time` and masking the data accordingly. The former specifies an index range `[start, stop)`, and the latter gives the minimal lag time of the correlation.

```
    mask = np.ones_like(x, dtype=bool)

    # deselect according to --range argument
    if args.range:
        mask[:args.range[0]] = False
        mask[args.range[1]:] = False

    # deselect according to --min-time argument, exclude t = 0 data
    min_time = args.min_time or 0
    mask[np.where(x <= min_time)] = False

    # split input data
    fit_x, no_fit_x = x[mask], x[~mask]
    fit_y, no_fit_y = y[mask], y[~mask]
    fit_yerr, no_fit_yerr = yerr[mask], yerr[~mask]
```

Eventually, we fit the selected data with a linear function that passes through the origin, and print the obtained diffusion constant along with an error estimate.

```
    linear_model = lambda x, D : 2 * args.dimension * D * x

```

(continues on next page)

(continued from previous page)

```
    popt, pcov = curve_fit(linear_model, fit_x, fit_y, sigma=fit_yerr)
    perr = np.sqrt(np.diag(pcov))

    print("D = {0:.5g} ± {1:.2g}".format(popt[0], perr[0]))
```

If not deselected, we generate a log-log plot of the MSD data along with the obtained diffusion law. The data used in the fit is blue, whereas the other data is green. As an option, a more sensitive representation shows $\delta r^2(t)/t$, rectifying the expected linear increase.

```
    if not args.no_plot:
        import matplotlib.pyplot as plt

        # time grid for reference lines
        t = np.logspace(np.log10(fit_x[0]) - 1, np.log10(fit_x[-1]) + .3)

        # plot MSD versus time
        if not args.rectify:
            plt.errorbar(fit_x, fit_y, yerr=fit_yerr, fmt='xb', label=
↪"data used in fit")
            plt.errorbar(no_fit_x, no_fit_y, yerr=no_fit_yerr, fmt='xg',␣
↪label="data not used in fit")
            plt.loglog(t, linear_model(t, *popt), '-k')
            ylabel = "Mean-square displacement"
        else:
            plt.errorbar(fit_x, fit_y / fit_x, yerr=fit_yerr / fit_x, fmt=
↪'xb', label="data used in fit")
            plt.errorbar(no_fit_x, no_fit_y / no_fit_x, yerr=no_fit_yerr /␣
↪no_fit_x, fmt='xg', label="data not used in fit")
            plt.semilogx(t, linear_model(t, *popt) / t, '-k')
            ylabel = "MSD(t) / t"

        # add axes labels and finalise plot
        plt.axis('tight')
        plt.xlabel('Time')
        plt.ylabel(ylabel)
        plt.legend(loc='best', frameon=False)
        plt.show()
```

CHAPTER

FOUR

RECIPES

This section provides recipes and code snippets for how to accomplish specific tasks in a simulation script.

Contributions are highly welcome.

4.1 Create initial state of a fluid mixture

For fluid mixtures, which have the same interaction laws for each component (but different coefficients), all fluid particles are collected in a single instance of *particle*. The different components are distinguished by their value for `species`, particles of the same species form a contiguous range in `tag` (which is given by the index in the particle array). This allows for the efficient selection of each component from a *range of tags*.

The setup procedure has the following steps:

- create system state
- sequentially assign particle species
- sequentially place particles on an fcc lattice
- randomly shuffle the positions
- assign random velocities according to a Maxwell–Boltzmann distribution

```
local mdsim   = halmd.mdsim
local numeric = halmd.numeric
local random  = halmd.random

local nparticle = {8000, 2000} -- particle numbers for each component
local length = {20, 20, 20}    -- cubic simulation box
local temperature = 1.5        -- temperature of Maxwell-Boltzmann␣
↪distribution

-- create system state
local box = mdsim.box({length = length})
local particle = mdsim.particle({dimension = #length, particles = numeric.
↪sum(nparticle), species = #nparticle})

-- assign particle species
local species = {}
for s = 1, #nparticle do
    for i = 1, nparticle[s] do
```

(continues on next page)

(continued from previous page)

```
        table.insert(species, s - 1)    -- species values are 0-based
    end
end
particle.data["species"] = species

-- set particle positions sequentially on an fcc lattice
mdsim.positions.lattice({box = box, particle = particle}):set()

-- shuffle positions randomly
local r = particle.data["position"]
r = random.generator({memory = "host"}):shuffle(r)
particle.data["position"] = r

-- set initial particle velocities
mdsim.velocities.boltzmann({particle = particle, temperature = temperature}
↪):set()
```

Note: The approach to randomly shuffle the particle positions is not very efficient performance-wise: in the current implementation, it involves 6 deep copies of the position array (assuming that the GPU is the compute device).

CHAPTER

FIVE

MODULES

5.1 Simulation

5.1.1 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.

5.1.2 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]}}
```

lowest_corner()

Returns the coordinates of the lowest corner of the box.

writer(*args*)

Write box specification to file.

http://nongnu.org/h5md/h5md.html#simulation-box

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/h5md.html#simulation-box

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})
```

5.1.3 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.`**`advance`**()

Manually advance the clock by one step. The method should be used with care by experienced users only.

`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()*.

5.1.4 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

5.1.5 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.

5.1.6 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 $r_{c,ij}$
- **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.

5.1.7 Particle

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

Parameters

- **args** (*table*) – keyword arguments
- **args.dimension** (*number*) – dimension of space
- **args.particles** (*number*) – number of particles
- **args.species** (*number*) – number of species (*default:* 1)
- **args.memory** (*string*) – device where the particle information is stored *(optional)*
- **args.precision** (*string*) – floating point precision *(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.

The supported values for `precision` are `single` and `double-single` if `memory` equals `gpu`, and `single` for host memory. If `precision` is not specified, the highest available precision is used.

nparticle
Number of particles.

nspecies
Number of particle species.

memory
Device where the particle memory resides.

precision
Floating-point precision of the data stored.

label
Instance label.

data
Pseudo-table providing access to the particle data:

```
table = particle.data["position"]
particle.data["position"] = table
```

or using the equivalent syntax:

```
table = particle.data.position
particle.data.position = table
```

The following named arrays holding per-particle data are predefined:

- scalar integer data:
  - `id`: particle ID, unique within instance of *halmd.mdsim.particle*
  - `reverse_id`: maps a particle ID to its current array index in memory
- scalar floating-point data:
  - `mass`: inertial mass (e.g., for *halmd.mdsim.integrators*)
  - `potential_energy`: potential energy felt by each particle, see also *halmd.observables.thermodynamics*
  - `species`: atomic, chemical, or coarse-grained species (e.g., for *halmd.mdsim.potentials*)
- vector integer data:
  - `image`: periodic image of the simulation box that contains the given particle
- vector floating-point data:
  - `force`: force acting on each particle
  - `position`: position reduced to the original periodic simulation box
  - `velocity`: velocity of each particle
- other floating-point fields:
  - `potential_stress_tensor`: contribution to the potential part of the stress tensor. In d dimensions, d diagonal entries of this symmetric tensor are followed by $d(d-1)/2$ off-diagonal entries.

Warning: Accessing particle data using this table is equivalent to calling *get()* and *set()*, respectively, and involves a full copy of the data to a lua table.

Warning: During simulation, particle arrays may be reordered in memory according to a space-filling curve, see *halmd.mdsim.sorts*. To access particles in initial order, use `get_reverse_id()` to retrieve a map from particle IDs to current particle indices in memory.

get (*name*)
Returns particle data identified by the name of the particle array.

Parameters **name** (*string*) – identifier of the particle array

set (*name*, *data*)
Set particle data identified by the name of the particle array.

Parameters

- **name** (*string*) – identifier of the particle array
- **data** (*table*) – table containing the data

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.

5.1.8 Forces

External Potential Force

class halmd.mdsim.forces.**external**(*args*)

Construct external potential force.

Parameters

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

The module computes the force on the particles due to an external potential. Recomputation is triggered by the signals *on_force* and *on_prepend_force* of *args.particle*.

potential

Instance of *halmd.mdsim.potentials.external*.

disconnect()

Disconnect force from profiler and particle module.

on_prepend_apply(*slot*)

Connect nullary slot function to signal. The signal is emitted before the force computation.

Returns signal connection

on_append_apply(*slot*)

Connect nullary slot function to signal. The signal is emitted after the force computation.

Returns signal connection

Pair Force

class halmd.mdsim.forces.**pair**(*args*)

Construct 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.neighbour** – instance of *halmd.mdsim.neighbour* (optional)
- **args.weight** (*number*) – weight of the auxiliary variables *(default: 1)*

The module computes the 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 `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.

Note: The argument `neighbour` is only relevant for truncated potentials.

---

potential
Instance of *halmd.mdsim.potentials*.

disconnect()
Disconnect force from profiler.

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

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) 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.

---

potential
Instance of *halmd.mdsim.potentials*.

disconnect()
Disconnect force from profiler and particle module.

on_prepend_apply(*slot*)
Connect nullary slot function to signal. The signal is emitted before the force computation.

Returns signal connection

on_append_apply(*slot*)
Connect nullary slot function to signal. The signal is emitted after the force computation.

Returns signal connection

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.neighbour** – instance of *halmd.mdsim.neighbour* or a table of keyword arguments (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 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. In this situation, *weight* needs to be set to *0.5*.

If `neighbour` is left unspecified, a default neighbour list module is constructed using the default parameters of *halmd.mdsim.neighbour*. If a different value for, e.g., the `skin` parameter is needed, either the neighbour list module has to be provided explicitly or a keyword table with the non-default arguments is passed, which is forwarded to the default-constructed neighbour list module. For example:

```
mdsim.forces.pair({
    box = [...], particle = [...], potential = [...]
  , neighbour = {skin = 0.7}      -- override default skin width
})
```

potential
Instance of *halmd.mdsim.potentials*.

disconnect()
Disconnect force from profiler.

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

on_prepend_apply(*slot*)
Connect nullary slot function to signal. The signal is emitted before the force computation.

Returns signal connection

on_append_apply(*slot*)
Connect nullary slot function to signal. The signal is emitted after the force computation.

Returns signal connection

5.1.9 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.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 frequency of the thermostat

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 $m_1 = dNT/\Omega^2$ and $m_2 = T/\Omega^2$, where Ω is $2\pi \times$ resonance frequency, N the total number of (point) particles, and d the dimension of space.

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.

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()*.

position ()
Return current values of thermostat chain variables (which are generalised positions).

velocity ()
Return current "velocities" of the thermostat chain variables.

internal_energy ()
Return internal energy of thermostat variables divided by *particle.nparticle*.

disconnect ()
Disconnect integrator from core and profiler.

class writer(*args*)

Write heat bath variables (generalised positions and velocities) and derived quantities (internal energy) to file.

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., `{xi = "position", v_xi = "velocity"}`; the table form is interpreted as `{position = "position", ...}`. The keys denote the field names in the file and are appended to `location`. Valid values are `position`, `velocity`, `internal_energy`.

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

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

disconnect()

Disconnect Nosé–Hoover writer from observables sampler.

5.1.10 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`)
- **args.fluctuating** (*boolean*) – the number or identity of selected particles can vary as the simulation progresses (*default:* `false`)

The flags `global` and `fluctuating` are used, e.g., for the output of thermodynamic quantities via *halmd.observables.thermodynamics*.

label
Particle group label.

particle
Instance of *halmd.mdsim.particle*.

size
Number of particles in group.

global
The value of `args.global` passed upon construction.

fluctuating
The value of `args.fluctuating` passed upon construction.

ID 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 IDs
local group_A = halmd.mdsim.particle_groups.id_range({particle = system,␣
↪range = {1, 5000}, label = "A"})
local group_B = halmd.mdsim.particle_groups.id_range({particle = system,␣
↪range = {5001, 10000}, label = "B"})
```

class halmd.mdsim.particle_groups.**id_range**(*args*)
Construct particle group from ID range.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of *halmd.mdsim.particle*
- **args.range** (*table*) – particle ID range `{first, last}`
- **args.label** (*string*) – group label
- **args.global** (*boolean*) – particle group can comprise the whole simulation world (*default:* `false`)
- **args.fluctuating** (*boolean*) – the number or identity of selected particles can vary as the simulation progresses (*default:* `false`)

Note: Particle IDs are 1-based, i.e. the first particle has ID 1.

The flags `global` and `fluctuating` are used, e.g., for the output of thermodynamic quantities via *halmd.observables.thermodynamics*.

label
Particle group label.

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`.

fluctuating
The value of `args.fluctuating` passed upon construction.

to_particle(*args*)

Parameters

- **args** (*table*) – keyword argruments
- **args.particle** – instance of *halmd.mdsim.particle* *(optional)*
- **args.label** – label of the new particle instance *(optional)*

Returns instance of *halmd.mdsim.particle* with data from the particle group

Copy the particle group to a new particle instance. If no parameters given, a suitable particle instance will be constructed.

Note: Only positions, mass, species and velocity are copied to the particle instance. Other data (e.g. ID, force) will not be copied.

If `args.particle` is present, the particle group will be copied into the given particle instance. Otherwise a new suitable particle instance with the label *args.label* will be created. If `args.label` is not given, it defaults to the group label. `species` of the new particle instance will be initialized with `particle.species`.

Note: `args.particle` must reside in the same memory as the group and the number of particles must be equal to `size`.

Region

A particle group represents a subset of particles, which is defined by an instance of particle together with a region in the simulation domain.

Example:

```
-- construct particle instance for given simulation domain
local system = halmd.mdsim.particle({particles = 10000})
local geometry = halmd.mdsim.geometries.cuboid({lowest_corner = {0,0,0},␣
↪length = {1, 1, 1}})

-- select particles from within this sub-box (region)
local group_cuboid = halmd.mdsim.particle_groups.region({
    particle = system, label = "subbox"
  , geometry = geometry, selection = "included"
})
```

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

Construct particle group from region and species.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of *halmd.mdsim.particle*
- **args.geometry** (*table*) – geometry of the region, instance of *halmd.mdsim.geometries*
- **args.selection** (*string*) – select particles within or outside the geometry. Allowed values are `included` and `excluded`.
- **args.label** (*string*) – group label
- **args.global** (*boolean*) – particle group can comprise the whole simulation world (*default:* `false`)
- **args.fluctuating** (*boolean*) – the number or identity of selected particles can vary as the simulation progresses (*default:* `true`)

The flags `global` and `fluctuating` are used, e.g., for the output of thermodynamic quantities via *halmd.observables.thermodynamics*.

particle

Instance of *halmd.mdsim.particle*

size

Number of particles in group.

label

Instance label.

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`.

fluctuating

The value of `args.fluctuating` passed upon construction.

to_particle(*args*)

Parameters

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

- **args.particle** – instance of *halmd.mdsim.particle* *(optional)*
- **args.label** – label of the new particle instance *(optional)*

Returns instance of *halmd.mdsim.particle* with data from the particle group

Copy the particle group to a new particle instance. If no parameters given, a suitable particle instance will be constructed.

Note: Only positions, mass, species and velocity are copied to the particle instance. Other data (e.g. id, force) will not be copied.

If `args.particle` is present, the particle group will be copied into the given particle instance. Otherwise a new suitable particle instance with the label *args.label* will be created. If `args.label` is not given, it defaults to the group label. `species` of the new particle instance will be initialized with `particle.species`.

Note: `args.particle` must reside in the same memory as the group and the number of particles must be equal to `size`.

---

disconnect()
Disconnect region module from profiler.

Region and Species

A particle group represents a subset of particles, which is defined by an instance of particle together with a region_species in the simulation domain.

Example:

```
-- construct particle instance for given simulation domain
local system = halmd.mdsim.particle({particles = 10000, species = 2})
local geometry = halmd.mdsim.geometries.cuboid({lowest_corner = {0,0,0},␣
↪length = {1, 1, 1}})

-- select particles of species 0 and from within this sub-box (region)
local group_cuboid = halmd.mdsim.particle_groups.region_species({
    particle = system, label = "subbox"
  , geometry = geometry, selection = "included"
  , species = 0
})
```

class halmd.mdsim.particle_groups.**region_species**(*args*)
Construct particle group from region.

Parameters

- **args** (*table*) – keyword arguments
- **args.particle** – instance of *halmd.mdsim.particle*
- **args.geometry** (*table*) – geometry of the region, instance of *halmd.mdsim.geometries*

- **args.selection** (*string*) – select particles within or outside the geometry. Allowed values are `included` and `excluded`.
- **args.species** (*string*) – 0-based particle species
- **args.label** (*string*) – group label
- **args.global** (*boolean*) – particle group can comprise the whole simulation world (*default:* `false`)
- **args.fluctuating** (*boolean*) – the number or identity of selected particles can vary as the simulation progresses (*default:* `true`)

The flags `global` and `fluctuating` are used, e.g., for the output of thermodynamic quantities via *halmd.observables.thermodynamics*.

particle
Instance of *halmd.mdsim.particle*

size
Number of particles in group.

label
Instance label.

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`.

fluctuating
The value of `args.fluctuating` passed upon construction.

to_particle(*args*)

Parameters

- **args** (*table*) – keyword argruments
- **args.particle** – instance of *halmd.mdsim.particle* *(optional)*
- **args.label** – label of the new particle instance *(optional)*

Returns instance of *halmd.mdsim.particle* with data from the particle group

Copy the particle group to a new particle instance. If no parameters given, a suitable particle instance will be constructed.

Note: Only positions, mass, species and velocity are copied to the particle instance. Other data (e.g. id, force) will not be copied.

If `args.particle` is present, the particle group will be copied into the given particle instance. Otherwise a new suitable particle instance with the label *args.label* will be created. If `args.label` is not given, it defaults to the group label. `species` of the new particle instance will be initialized with `particle.species`.

Note: `args.particle` must reside in the same memory as the group and the number of particles must be equal to `size`.

disconnect()
Disconnect region_species module from profiler.

5.1.11 Potentials

External potentials

Harmonic potential

This module implements the harmonic potential,

$$U_i(\vec{r}) = \frac{1}{2} K_i (\vec{r} - \vec{r}_{0,i})^2$$

for the potential energy of a particle of species i.

class halmd.mdsim.potentials.external.**harmonic**(*args*)
Construct the harmonic potential.

Parameters

- **args** (*table*) – keyword arguments
- **args.stiffness** (*table*) – sequence of stiffness coefficients K_i
- **args.offset** (*table*) – sequence of offset vectors $\vec{r}_{0,i}$
- **args.species** (*number*) – number of particle species *(optional)*
- **args.memory** (*string*) – select memory location *(optional)*
- **args.label** (*string*) – instance label *(optional)*

If all elements of a parameter sequence are equal, a single value may be passed instead. In this case, `species` must be specified.

If the argument `species` is omitted, it is inferred from the length of the parameter sequences.

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

stiffness
Sequence with stiffness coefficients K_i.

offset
Sequence with offset vectors $\vec{r}_{0,i}$.

description
Name of potential for profiler.

memory
Device where the particle memory resides.

Planar wall potential

This module implements the external potential for a collection of planar walls.

Position vector $\vec{R}_i$ of the wall i is given by $\hat{n}_i \cdot \vec{R}_i = r_{0,i}$, where $\hat{n} = \vec{n}/|\vec{n}|$ is the normalized outward normal vector to the wall surface. $d_i = (\vec{r}_i \cdot \hat{n}_i - r_{0,i})$ is the distance of a particle, at position $\vec{r}$, from the wall i.

$$U\left(d_i\right) = \varepsilon_i^\alpha \left(\frac{2}{15} \left(\frac{\sigma_i^\alpha}{d_i} \right)^9 - w_i^\alpha \left(\frac{\sigma_i^\alpha}{d_i} \right)^3 \right)$$

is the interaction energy of a particle of species α due to the wall i. While computing $U\left(d_i\right)$ we always make d_i positive.

The potential is truncated at a cutoff distance r_c and further transformed to a C^2 continuous function using *halmd.mdsim.forces.pair_trunc*.

The parameters ε, σ, w and r_c depend on both the wall and the species. For example, $\varepsilon[i][\alpha]$ contains ε_i^α, where indices i, α run over the wall and species respectively.

class halmd.mdsim.potentials.external.**planar_wall**(*args*)

Construct the planar wall module.

Parameters

- **args** (*table*) – keyword arguments.
- **args.offset** (*table*) – positions of the walls $r_{0,i}$ in MD units.
- **args.surface_normal** (*number*) – outward normal vectors to the wall surfaces $\vec{n}$ in MD units.
- **args.epsilon** (*matrix*) – interaction strengths ε_i^α in MD units.
- **args.sigma** (*matrix*) – interaction ranges σ_i^α in MD units.
- **args.wetting** (*matrix*) – wetting parameters w_i^α in MD units.
- **args.cutoff** (*matrix*) – cutoff lengths $r_{c,i}^\alpha$ in MD units.
- **args.smoothing** (*number*) – smoothing parameter h for the C^2 continuous truncation in MD units.
- **args.memory** (*string*) – select memory location *(optional)*.
- **args.label** (*string*) – instance label *(optional)*.

If all elements of a parameter sequence are equal, a single value may be passed instead. In this case, `species` must be specified.

If the argument `species` is omitted, it is inferred from the length of the parameter sequences.

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

offset

Sequence with the wall position $r_{0,i}$.

surface_normal

Sequence with outward normal vector to the wall surface $\vec{n}_i$.

epsilon
Sequence with interaction strength ε_i^α.

sigma
Sequence with interaction range σ_i^α.

wetting
Sequence with wetting parameter w_i^α.

cutoff
Sequence with cutoff length $r_{c,i}^\alpha$.

smoothing
Sequence with smoothing parameter h for the C^2 continuous truncation.

description
Name of potential for profiler.

memory
Device where the particle memory resides.

Pair potentials

Lennard-Jones potential

This module implements the Lennard-Jones potential,

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

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

class halmd.mdsim.potentials.pair.**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.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.

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}.

description
Name of potential for profiler.

memory
Device where the particle memory resides.

truncate(*args*)
Truncate potential. See *Potential truncations* for available truncations.

Parameters

- **args** (*table*) – keyword argument
- **args[1]** (*string*) – name of truncation type
- **cutoff** (*table*) – matrix with elements $r_{c,ij}$
- **args.*** (*any*) – additional arguments depend on the truncation type

Returns truncated potential

Example:

```
potential = potential:truncate({"smooth_r4", cutoff = 5, h = 0.05}
↪)
```

modify(*args*)
Apply potential modification. See *Potential modifications* for available modifications.

Parameters

- **args** (*table*) – keyword argument
- **args[1]** (*string*) – name of modification type
- **args.*** (*any*) – additional arguments depend on the modification type

Returns modified potential

Example:

```
potential = potential:modify({"hard_core", radius = 0.5})
```

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.pair.lennard_jones* is that the exponents of repulsion, m_{ij}, and of attraction, n_{ij}, can be specified explicitly.

class halmd.mdsim.potentials.pair.**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.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.

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.

truncate(*args*)
Truncate potential. See *Potential truncations* for available truncations.

Parameters

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

- **args[1]** (*string*) – name of truncation type
- **cutoff** (*table*) – matrix with elements $r_{c,ij}$
- **args.*** (*any*) – additional arguments depend on the truncation type

Returns truncated potential

Example:

```
potential = potential:truncate({"smooth_r4", cutoff = 5, h = 0.05}
↪)
```

modify(*args*)

Apply potential modification. See *Potential modifications* for available modifications.

Parameters

- **args** (*table*) – keyword argument
- **args[1]** (*string*) – name of modification type
- **args.*** (*any*) – additional arguments depend on the modification type

Returns modified potential

Example:

```
potential = potential:modify({"hard_core", radius = 0.5})
```

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.pair.**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.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.

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.

truncate(*args*)
Truncate potential. See *Potential truncations* for available truncations.

Parameters

- **args** (*table*) – keyword argument
- **args[1]** (*string*) – name of truncation type
- **cutoff** (*table*) – matrix with elements $r_{\mathrm{c},ij}$
- **args.*** (*any*) – additional arguments depend on the truncation type

Returns truncated potential

Example:

```
potential = potential:truncate({"smooth_r4", cutoff = 5, h = 0.05}
↪)
```

modify(*args*)
Apply potential modification. See *Potential modifications* for available modifications.

Parameters

- **args** (*table*) – keyword argument
- **args[1]** (*string*) – name of modification type
- **args.*** (*any*) – additional arguments depend on the modification type

Returns modified potential

Example:

```
potential = potential:modify({"hard_core", radius = 0.5})
```

Power-law potential

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

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

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

class halmd.mdsim.potentials.pair.**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.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.

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
Matrix with power-law indices n_{ij}

description
Name of potential for profiler.

memory
Device where the particle memory resides.

truncate(*args*)

Truncate potential. See *Potential truncations* for available truncations.

Parameters

- **args** (*table*) – keyword argument
- **args[1]** (*string*) – name of truncation type
- **cutoff** (*table*) – matrix with elements $r_{c,ij}$
- **args.*** (*any*) – additional arguments depend on the truncation type

Returns truncated potential

Example:

```
potential = potential:truncate({"smooth_r4", cutoff = 5, h = 0.05}
↪)
```

modify(*args*)

Apply potential modification. See *Potential modifications* for available modifications.

Parameters

- **args** (*table*) – keyword argument
- **args[1]** (*string*) – name of modification type
- **args.*** (*any*) – additional arguments depend on the modification type

Returns modified potential

Example:

```
potential = potential:modify({"hard_core", radius = 0.5})
```

Adapters for pair potentials

Every potential defines `modify` and `truncate` methods to create modifications of a potential. The returned object is again a pair potential.

Potential modifications

hard_core

Add a hard core of radius r_{core} to point particles by shifting the potential radially outwards:

$$\tilde{U}(r) = U(r - r_{\text{core}}), \qquad r > r_{\text{core}}.$$

Example:

```
potential = potential:modify({"hard_core", radius = 0.5})
```

Potential truncations

sharp

Create sharp truncation by setting the potential to zero for particle distances r larger than the cutoff distance r_c, i.e., by multiplying with a step function:

$$\tilde{U}(r) = U(r)\Theta(r_c - r)$$

Both potential and force are discontinuous at the cutoff.

Example:

```
potential = potential:truncate({"sharp", cutoff = 4})
```

shifted

Amend the sharp truncation by an energy shift such that the potential is continuous at the cutoff distance r_c:

$$\tilde{U}(r) = [U(r) - U(r_c)]\Theta(r_c - r)$$

The force is not affected by this and remains discontinuous at the cutoff.

Example:

```
potential = potential:truncate({"shifted", cutoff = 2.5})
```

force_shifted

Amend the energy-shifted potential by a linear term, effectively shifting the force to zero at the cutoff distance r_c:

$$\tilde{U}(r) = [U(r) - U(r_c) - (r - r_c)U'(r_c)]\Theta(r_c - r)$$

Both energy and force are continuous at the cutoff. Note that this modification globally tilts the potential with possible physical implications for, e.g., phase diagrams.

Example:

```
potential = potential:truncate({"force_shifted", cutoff = 2.5})
```

smooth_r4

Truncate the potential $U(r)$ such that it remains a C^2-continuous function at the cutoff, i.e., the force being continuously differentiable. As a consequence, momentum and energy drift are drastically diminished even from long runs using symplectic integrators such as *halmd.mdsim.integrators.verlet*.

The truncation is implemented by multiplication of the energy-shifted potential with the local smoothing function

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

where r_c is the cutoff distance, and the parameter $h \ll \sigma$, which has the dimension of a length, controls the the range of smoothing. The C^2-continuous truncated potential then reads

$$\tilde{U}(r) = [U(r) - U(r_c)]\, g\left(\frac{r - r_c}{h}\right) \Theta(r_c - r),$$

and the C^1-continuous force is

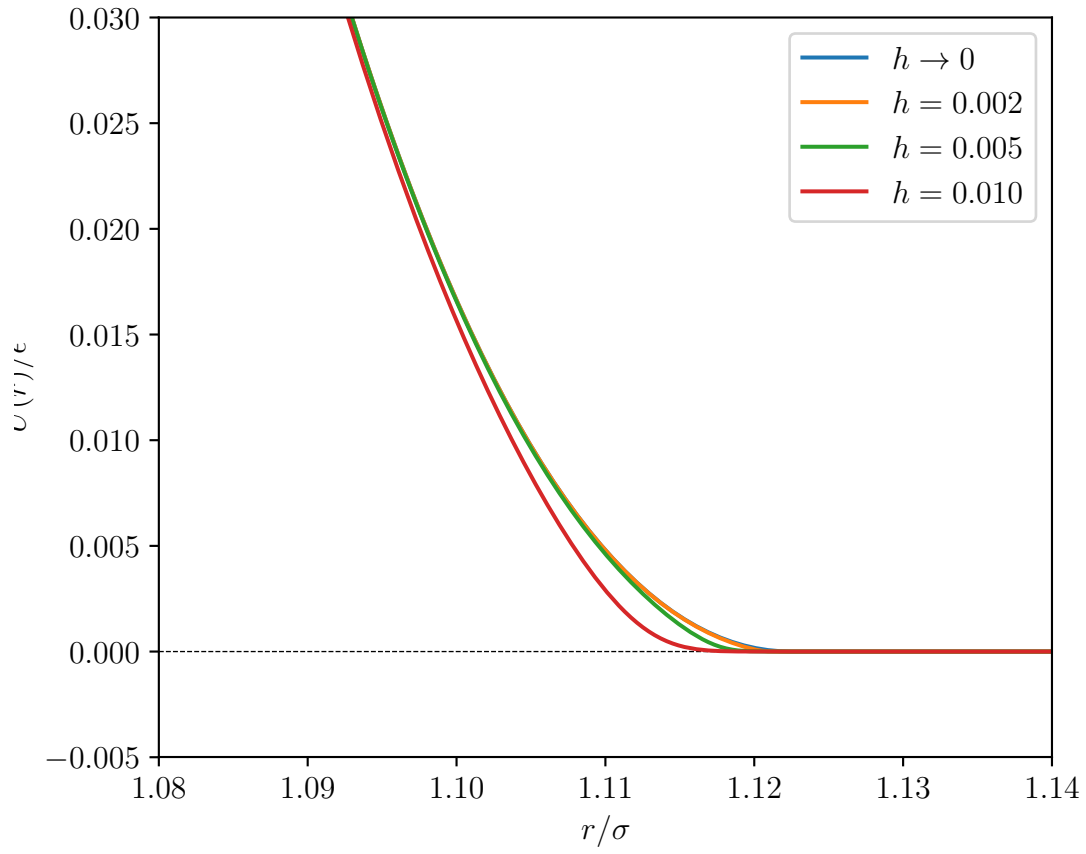
$$|\tilde{\vec{F}}(\vec{r})| = |\vec{F}(\vec{r})| g\left(\frac{r - r_c}{h}\right) - \frac{1}{h} U(r)\, g'\left(\frac{r - r_c}{h}\right) \Theta(r_c - r)$$

with the derivative of the smoothing function $g'(\xi) = 4\xi^3\left(1+\xi^4\right)^{-2}$.

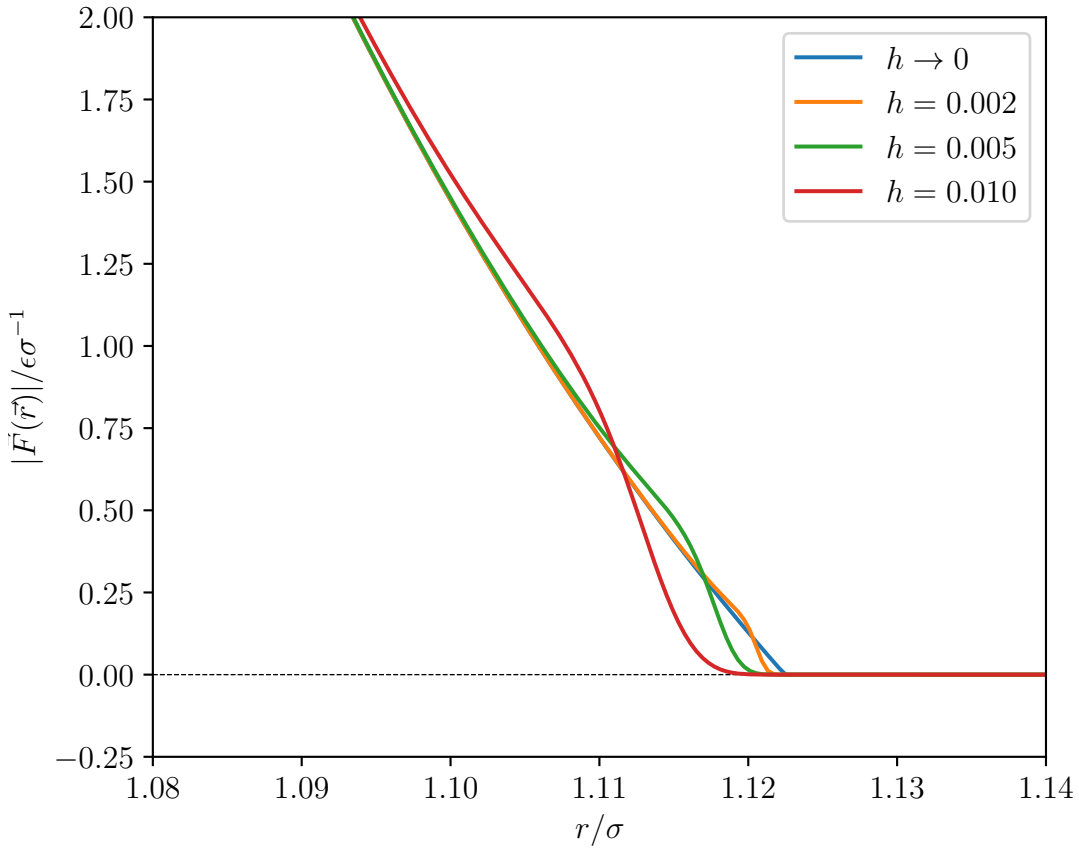
Example:

```
potential = potential:truncate({"smooth_r4", cutoff = 2.5, h = 0.005})
```

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



The following figure shows the absolute value of the force.


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
1.08
1.09
1.10
1.11
1.12
1.13
1.14
r/σ
$|\vec{F}(\vec{r})|/\epsilon\sigma^{-1}$
$h \to 0$
$h = 0.002$
$h = 0.005$
$h = 0.010$

5.1.12 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 $\sigma_v = \sqrt{\frac{kT}{m}}$,

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

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.

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.

5.1.13 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 = mdsim.box{length = {edge, edge, edge}}
local excluded = 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
```

(continues on next page)

(continued from previous page)

```
        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 = mdsim.particle{dimension = box.dimension, particles =
↪#obstacles}
particle.data["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-centred 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.

5.1.14 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.

5.1.15 Geometric selectors

A collection of geometric selectors for the `regions module`.

Cuboid

Defines a cuboid domain in space and a predicate whether a given position is inside or outside of the domain.

class halmd.mdsim.geometries.**cuboid**(*args*)

Construct cuboid geometry instance.

Parameters

- **args** (*table*) – keyword arguments
- **args.lowest_corner** (*table*) – coordinates of the lower left corner
- **args.length** (*table*) – cuboid edge lengths
- **args.precision** (*string*) – floating point precision *(optional)*

  The `lowest_corner` specifies the coordinates of the lower left corner of the cuboid, i.e., the minimal coordinates.

  The supported values for `precision` are `single` and `double`. If `precision` is not specified, the precision is selected according to the compute device: `single` for GPU computing and `single` otherwise.

Note: This module does not perform a validation of the meaningfulness of the domain, i.e. it does not test whether the geometry is placed outside the simulation domain.

---

lowest_corner

Coordinates of lower left corner.

length

Edge lengths of cuboid.

Sphere

Defines a spherical domain and a predicate whether a given position is inside or outside of the domain.

class halmd.mdsim.geometries.**sphere**(*args*)

Construct sphere geometry instance.

Parameters

- **args** (*table*) – keyword arguments
- **args.centre** (*table*) – sphere centre
- **args.radius** (*number*) – sphere radius
- **args.precision** (*string*) – floating point precision *(optional)*

The supported values for `precision` are `single` and `double`. If `precision` is not specified, the precision is selected according to the compute device: `single` for GPU computing and `single` otherwise.

Note: This module does not perform a validation of the meaningfulness of the domain, i.e. it does not test whether the geometry is placed outside the simulation domain.

---

centre
Coordinates of sphere centre.

radius
Sphere radius.

5.2 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_vector**(*size*, *value*)
Create vector of given size with scalar value

Parameters

- **size** (*number*) – number of elements
- **value** – value for each element of the vector

Returns vector of length size with each element set to `value`

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`

5.3 Input and Output

5.3.1 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.

5.3.2 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
})
```

5.3.3 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)*
- **args.overwrite** (*boolean*) – if true, overwrite existing file *(default: false)*

Returns instance of file writer

Create the output file and writes the H5MD metadata.

http://nongnu.org/h5md/h5md.html#h5md-metadata

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.lowest_corner, {"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.

5.4 Observables

5.4.1 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.

5.4.2 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 ID, 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 only be updated when the underlying particle data has changed, so you can reuse the same sampler with multiple observable modules for optimal performance.

acquire(*name*)
Returns data slot to acquire phase space sample for the given particle array.

Parameters **name** (*string*) – identifier of the particle array to be sampled

The memory location of the sample depends on the type of the particle array. GPU particle arrays (e.g. "g_position") will be sampled to GPU memory, host wrappers (e.g. "position") will be sampled to Host memory.

acquire_position()
Returns data slot to acquire position data. The memory type is inferred from the particle instance (for GPU particles this samples the "g_position" data, for Host particles the "position" data).

acquire_velocity()
Returns data slot to acquire velocity data. The memory type is inferred from the particle instance (for GPU particles this samples the "g_velocity" data, for Host particles the "velocity" data).

acquire_species()
Returns data slot to acquire species data. This always samples to host memory, as species data is packed together with position data on GPU memory. acquire("g_position") can be used to obtain a GPU sample of both position and species.

acquire_mass()
Returns data slot to acquire mass data. This always samples to host memory, as mass data is packed together with velocity data on GPU memory. acquire("g_velocity") can be used to obtain a GPU sample of both velocity and mass.

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(*samples*)
Sets particle data from phase space samples.

Parameters **samples** (*table*) – List of samples to be set. The keys of the table contain the identifiers for the particle array, the values the 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/h5md.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/h5md.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 is a table mapping identifiers to the individual data samples and has the following additional 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.

5.4.3 Runtime Estimate

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

Example:

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

(continues on next page)

(continued from previous page)

```
-- 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 = 900, sample = 60})

-- 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

5.4.4 Sampler

The sampler concerts the sampling of observables.

Example:

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

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.**start**()

Initialise simulation modules connected to the signal `on_start`.

The function is called by *run()* upon its first invocation.

halmd.observables.sampler.**finish**()

Finalise the simulation by emitting the signal *on_finish*.

The function is called by the default simulation engine (in `lua/halmd/run.lua`) after `main()` from the user script has returned.

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 by *start()* before the simulation run starts.

Returns signal connection

halmd.observables.sampler.**on_finish**(*slot*)
Connect slot to signal emitted by *finish()*, which is called after the `main()` routine has returned.

Returns signal connection

5.4.5 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.

5.4.6 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: $u_{\mathrm{kin}} = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} m_i \vec{v}_i^2$.

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: $k_B T = 2u_{\text{kin}}/d$.

total_force()
Returns the total force: $\vec{F}_{\text{tot}} = \sum_{i=1}^{N} \vec{F}_i$.

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.

5.4.7 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: 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({
```

(continues on next page)

(continued from previous page)

```
    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 (multi-dimensional) value
- **args.correlate** – callable that accepts two values and returns a `number` or a numeric `table`
- **args.shape** (*number table*) – array shape of the result (*optional*)
- **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 `shape` specifies the array shape of the outcomes of `correlate`. It is only required if the result is a `table`. Currently, only 1-dimensional arrays are supported, for which `shape = { size }`.

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

[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].

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

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

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

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 position sample.

Returns phase space position 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 position sample.

Returns phase space position 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 velocity 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"}`.

5.4.8 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
```

(continues on next page)

(continued from previous page)

```
    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 wavevectors $\vec{k}$ compatible with the reciprocal space of the periodic simulation box and grouped into shells according to their wavenumber, $|\vec{k}|$. 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.

Two modes are supported:

1) a *sparse sampling* of thin shells according to a predefined set of wavenumbers $\{k_i\}$, allowing for a relative deviation of the wavenumber. The number of wavevectors per shell may be limited to avoid excessively large shells for large wavenumber.

2) a *dense grid* of wavevectors from all points of the reciprocal lattice up to the maximum wavenumber given, $|\vec{k}| < \max\{k_i\}$. The result is grouped into shells according to the wavenumbers given. In this case, shells are half-open sets, $k_{i-1} \le |\vec{k}| < k_i$, provided that $k_i < k_j$ for all $i < j$.

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=0.05, max_count=7})
```

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** (*table*) – list of wavenumbers
- **args.dense** (*boolean*) – dense grid of wavevectors (*default:* false)
- **args.tolerance** (*number*) – relative tolerance on wavevector magnitude
- **args.max_count** (*integer*) – maximum number of wavevectors per wavenumber shell
- **args.filter** (*table*) – filter on wavevectors (*default:* {1, ..., 1})

If dense is true, a dense grid of wavevectors is created. Otherwise, the arguments tolerance and max_count are required and a sparse sampling of wavevectors is returned.

The argument filter contains 0 or 1 for each Cartesian coordinate, 0 deletes the respective wavevector component.

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

5.5 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.

`halmd.random.shuffle`(*sequence*)

Return randomly shuffled sequence. The sequence is given as a Lua table.

The method is only available if the random number generator was constructed with `memory = "host"`.

5.6 Utilities

5.6.1 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_VISIBLE_DEVICES=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.

Warning: The `nvlock` tool is broken with recent NVIDIA drivers, e.g., later than version 346.

`halmd.utility.device.`**`gpu`**
Ordinal number of the CUDA device.

5.6.2 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

5.6.3 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.

5.6.4 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

5.6.5 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()
```

Note: Typically, the argument parser is not created directly. Instead a simulation script may define a global method `define_args`, which receives an argument parser as its argument, see *Simulation Scripts*.

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 acceleration"})
```

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, value)
   -- substitute current time
   args[key] = os.date(value)
end, default = "halmd_%Y%m%d_%H%M%S", help = "prefix of output␣
↪files"})
```

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()
```

class halmd.utility.program_options.argument_parser.**action**

substitute_date_time (*args*, *key*, *value*)

Substitute date/time helper action.

Parameters

- **args** (*table*) – parsed arguments
- **key** (*string*) – args table key of this argument
- **value** – parsed value of this argument

Helper function that can be used as action argument for argument_parser:add_argument and substitutes day and time in the parsed value of the argument.

Example:

```
parser:add_argument("output,o", {type = "string", action = parser.
↪action.substitute_date_time,
    default = "lennard_jones_%Y%m%d_%H%M%S", help = "prefix of␣
↪output files"})
```

class halmd.utility.program_options.argument_parser.**action**

substitute_environment(*args*, *key*, *value*)
Action that substitutes environment variables.

Parameters

- **args** (*table*) – parsed arguments
- **key** (*string*) – args table key of this argument
- **value** – parsed value of this argument

Helper function that can be used as action argument for argument_parser:add_argument and substitutes environment variables in the parsed value of the argument. Variable names must be enclosed by '%' characters and are obtained from *os.getenv*.

Example:

```
parser:add_argument("output,o", {type = "string", action = parser.
↪action.substitute_environment,
    default = "%HOME%/halmd_%PID%", help = "prefix of output files
↪"})
```

random_seed(*args*, *key*, *value*)
Random seed helper action.

Parameters

- **args** (*table*) – parsed arguments
- **key** (*string*) – args table key of this argument
- **value** – parsed value of this argument

Helper function that can be used as action argument for argument_parser:add_argument and initializes the RNG with a given seed.

Example:

```
parser:add_argument("random-seed", {type = "integer", action =␣
↪parser.action.random_seed,
    help = "seed for random number generator"})
```

5.6.6 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)*

5.6.7 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.

5.6.8 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*.

5.6.9 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.

5.6.10 Functions on strings

halmd.utility.**interp**(*s*, *t*)
Interpolate strings supporting Pythonic formatting specifications (see http://lua-users.org/wiki/StringInterpolation, example by RiciLake)

Parameters

- **s** (*string*) – formatting string
- **t** (*table*) – (key, value) pairs

Returns interpolated copy of string 's'

Occurrences of '{key:fmt}' in the formatting string are substituted by '("%fmt"):format(value)', where 'fmt' is one of the C-printf formatting codes. The method is added to the *string* type as *string.interp*.

Example:

print(("{name:s} is {val:7.2f}%"):interp({name = "concentration", val = 56.2795}))

–> "concentration is 56.28%"

5.6.11 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

SIX

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

SEVEN

VALIDATION

The simulation package is regularly run against various tests which reproduce results from physics literature.

7.1 Simple fluids

7.1.1 Thermodynamics

Lennard–Jones potential

values for the truncated and shifted Lennard–Jones potential in three dimensions:

	cutoff radius	density	temperature	pressure	potential energy per particle	isochoric specific heat	isothermal compressibility
	r_c^*	ρ^*	T^*	P^*	U^*	c_V/k_B	$\chi_T \rho k_B T$
[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)

7.1.2 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).

7.2 Binary mixtures

7.2.1 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

EIGHT

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
```

The Tesla GPUs had ECC *enabled*, no overclocking or other tweaking was done.

8.1 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 E5-2640	1.44 µs	10.8	double	GCC 4.7.2, -O3
NVIDIA Tesla S1070	57.0 ns	274	double-single	CUDA 5.5, -arch compute_12
	55.1 ns	284	single	CUDA 5.5, -arch compute_12
NVIDIA Tesla C2050	39.4 ns	397	double-single	CUDA 5.5, -arch compute_12
	34.3 ns	456	single	CUDA 5.5, -arch compute_12
NVIDIA Tesla K20m	22.2 ns	702	double-single	CUDA 5.5, -arch compute_12
	20.7 ns	756	single	CUDA 5.5, -arch compute_12
NVIDIA Tesla K20Xm	19.7 ns	792	double-single	CUDA 5.5, -arch compute_12
	18.4 ns	851	single	CUDA 5.5, -arch compute_12

Results were obtained from 1 independent measurement based on pre-release version 1.0-alpha1. Each

run consisted of NVT equilibration at $T^* = 1.2$ over $\Delta t^* = 100$ (10^4 steps), followed by benchmarking 10^4 NVE 5 times steps in a row.

8.2 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 E5-2640	2.03 µs	1.93	double	GCC 4.7.2, -O3
NVIDIA Tesla S1070	65.7 ns	59.4	double-single	CUDA 5.5, -arch compute_12
	66.3 ns	58.9	single	CUDA 5.5, -arch compute_12
NVIDIA Tesla C2050	39.2 ns	99.7	double-single	CUDA 5.5, -arch compute_12
	32.4 ns	120	single	CUDA 5.5, -arch compute_12
NVIDIA Tesla K20m	18.5 ns	211	double-single	CUDA 5.5, -arch compute_12
	17.0 ns	230	single	CUDA 5.5, -arch compute_12
NVIDIA Tesla K20Xm	16.2 ns	242	double-single	CUDA 5.5, -arch compute_12
	15.0 ns	261	single	CUDA 5.5, -arch compute_12

Results were obtained from 1 independent measurement and are based on pre-release version 1.0-alpha1. Each run consisted of NVT equilibration at $T^* = 0.7$ over $\Delta t^* = 100$ (2×10^4 steps), followed by benchmarking 10^4 NVE steps 5 times in a row.

CHAPTER

NINE

PUBLICATIONS

This section lists peer-reviewed publications related to *HAL's MD package*. Please send **suggestions** for papers that should be included to info@halmd.org along with a short description and a graphical illustration (if available).

9.1 Technical papers

MD simulation algorithms The paper describes the essential MD simulation algorithms and their implementation for the GPU. Please refer to it in all publications based on or linked to *HAL's MD package*.

P. H. Colberg and F. Höfling, Highly accelerated simulations of glassy dynamics using GPUs: Caveats on limited floating-point precision, Comput. Phys. Commun. **182**, 1120 (2011) [arXiv:0912.3824].

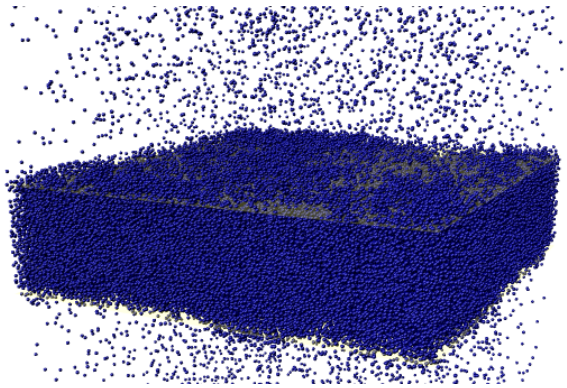
H5MD format of input and output files The paper introduces and justifies the H5MD file format and contains its detailed specification as of version 1.0.

P. de Buyl, P. H. Colberg, and F. Höfling, H5MD: a structured, efficient, and portable file format for molecular data, Comput. Phys. Commun. **185**, 1546 (2014) [arXiv:1308.6382].

9.2 Scientific research

Mesoscopic structure of liquid–vapour interfaces At the molecular scale, liquid–vapour interfaces are broadened and roughened by thermally excited capillary waves. These fluctuations give rise to a divergence of the interfacial structure factor at small wave-numbers; the latter being accessible to grazing-incidence small-angle X-ray scattering (GISAXS) experiments. The paper discusses deviations from the classical theory and relies on extensive simulations of planar interfaces using up to 445,000 Lennard-Jones particles. GISAXS intensities are computed on the fly of the simulations.

F. Höfling and S. Dietrich, Enhanced wavelength-dependent surface tension of liquid-vapour interfaces, EPL (Europhys. Lett.) **109**, 46002 (2015) [arXiv:1412.0568].



Critical dynamics of symmetric binary fluids

S. Roy, S. Dietrich, and F. Höfling, Structure and dynamics of binary liquid mixtures near their continuous demixing transitions, J. Chem. Phys. **145**, 134505 (2016).

Cavitation in dense glassy liquids

P. Chaudhuri and J. Horbach, Structural inhomogeneities in glasses via cavitation, Phys. Rev. B **94**, 094203 (2016).

P. Chaudhuri and J. Horbach, Phase separation in dense glassy liquids: effect of quenching protocols, J. Stat. Mech. **8**, 084005 (2016).

Adsorption Kinetics in Metal-Organic Frameworks

N. Höft, PhD thesis, Universität Düsseldorf (2016), supervised by J. Horbach.

Non-equilibrium fluids

J. Bartnick, A. Kaiser, H. Löwen, and A. V. Ivlev, Emerging activity in bilayered dispersions with wake-mediated interactions, J. Chem. Phys. **144**, 224901 (2016).

J. Bartnick, M. Heinen, A. V. Ivlev, and H. Löwen, Structural correlations in diffusiophoretic colloidal mixtures with nonreciprocal interactions, J. Phys.: Condens. Matter **28**, 025102 (2016).

CHAPTER

TEN

FAQ

HAL's MD package allowed me to do simulations of breath-taking quality and I have obtained new scientific insight. How shall I give credit to your work?

Please acknowledge the use of *HAL's MD package* 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,* Comput. Phys. Commun. **182**, 1120 (2011).

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`.

Configuring emits a warning on missing Boost C++ dependencies, e.g., 'Imported targets not available for Boost version 106500'

The CMake version used needs to be newer than the employed Boost C++ release. It is a good idea to stick with the combination shipped by your Linux distribution. For details check the CMake Boost compatibility table.

How to enable the static CUDA runtime library?

By default the CUDA runtime is linked dynamically and the `HALMD_USE_STATIC_LIBS` flag doesn't affect that.

To use the static CUDA runtime use `CUDA_USE_STATIC_CUDA_RUNTIME`.

Why does HALMD abort with "Program hit cudaErrorInvalidDeviceFunction (error 8) due to "invalid device function" on CUDA API call to cudaLaunch"?

This error occurs when using the static CUDA runtime library together with CUDA 9.2. As far as we are concerned there is no way to fix this, other than using the shared library or another CUDA version. See commit `5f1801bca`.

CHAPTER

ELEVEN

DEVELOPER'S GUIDE

11.1 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's guide*.

11.2 Cloning the repository

HAL's MD package is maintained in a public Git repository. Git is a fast and efficient, distributed version control system. A copy of the repository is created by

```
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/halmd-org/halmd.git
```

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

11.2.1 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
```

11.2.2 Git tutorials

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

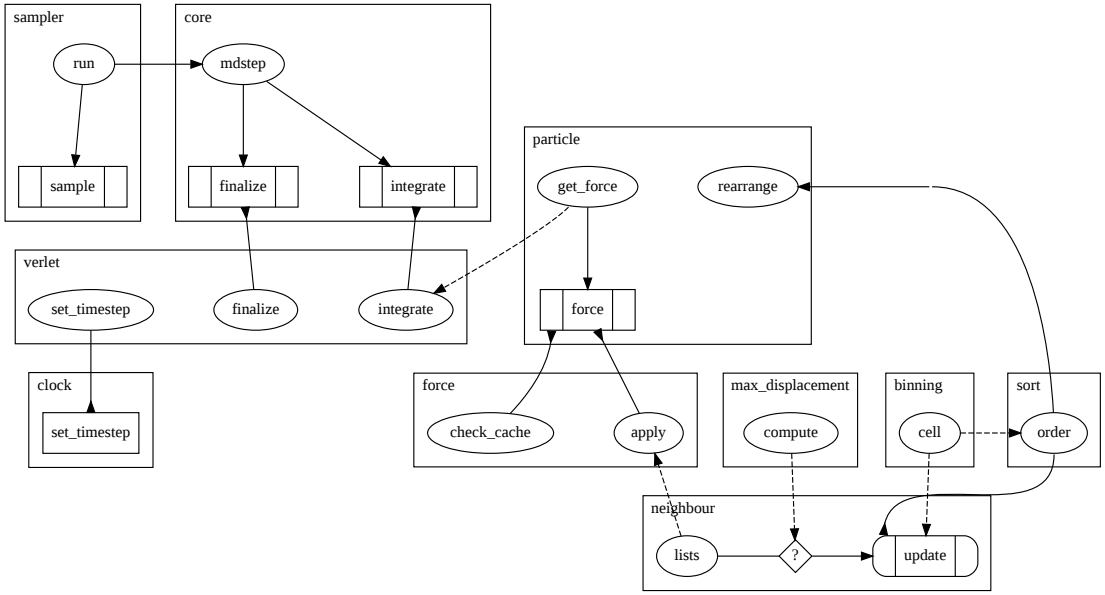
Former Subversion users may also read the Git SVN Crash Course.

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

11.3 Programme flow: signals and data caches

The design goals of modularity and scriptability of *HAL's MD package* lead us to a data-driven programme flow, which shares similarities with functional programming. For instance, the sampling of an observable requires information about the system state (particle positions, velocities, forces, ...) at the given time. This triggers the integrator, and reading the forces causes a recomputation of all force contributions defined. To avoid unnecessary recalculation of the same quantity, data caches and cache observes are used (see class `halmd::utility::cache`). Such strict dependencies are implemented in the C++ layer.

Second, the dependencies between modules are realised using signal/slot relations, which are set up in the Lua layer of each module. These connections are usually not explicit in the simulation script, but can be manipulated by advanced users. Upon emission of a signal, a number of previously connected slot functions is invoked. Slot functions are typically of signature `void ()`. A typical molecular dynamics simulation using one velocity Verlet integrator and one truncated pair force is depicted below. Signals are shown as rectangles within the emitting module, methods are shown as ellipses; dashed lines are hard-coded dependencies across modules. Observables would connect to the signal `sample` and have been omitted for clarity.



11.4 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.

11.5 How to write a HALMD module

11.5.1 Implementation

11.5.2 Testing

11.5.3 Documentation

11.6 Test suite

11.6.1 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.

11.6.2 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
)
```

11.7 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

11.8 Short guide on Luabind

The *Luabind* library is used to export C++ classes to the Lua scripting interface. *Luaponte* is a (temporary) fork from *Luabind*.

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>
```

(continues on next page)

(continued from previous page)

```
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>(dimension) + "_");
    // 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::timestep)
                    .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) + "_"));
```

(continues on next page)

(continued from previous page)

```
    // 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_str())
                            .def(constructor<
                                shared_ptr<particle_type>
                              , shared_ptr<box_type>
                              , double>()
                            )
                            .property("module_name", &module_name_wrapper
↪<dimension, float_type>)
                    ]
                ]
            ]
        ]
    ];
}

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()`?

11.8.1 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)
```

11.8.2 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> ()>
```

11.9 Debugging

11.9.1 Debugging C++ exceptions

When debugging exceptions, instruct the GNU debugger to catch exceptions by executing "catch throw" before executing the program with "run".

11.9.2 Invalid device function

If you get a CUDA error "invalid device function" upon kernel launch, check whether the kernel function has a __global__ attribute.

11.10 Redmine

11.10.1 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

CHAPTER

TWELVE

CHANGELOG

12.1 Version 1.0.0

Breaking changes

- fix misspelled keyword `acquire` in Lua API, it was `aquire` before *(Felix Höfling)*
- rename particle group `from_range` to *`id_range`* *(Roya Ebrahimi Viand)*

Bug fixes

- correctly support GPUs of compute capability ≥ 7.0 (Volta). Since the Volta architecture, Nvidia dropped the CUDA paradigm of executing threads within a warp in lock-step fashion, which led to wrong results in some algorithms. *(Felix Höfling)*
- fix wrong output of particle data to a file, which happened if several *`phase space`* samplers, e.g., for different groups, were operating on the same *`particle`* instance. *(Roya Ebrahimi Viand)*
- fix wrong output of wavenumbers to a file in case of discarded values, e.g., if a non-cubic box and a wavevector filter were passed to *`halmd.observables.utility.wavevector`* *(Felix Höfling)*
- various small fixes in example scripts, corrected and more robust unit tests *(Jaslo Ziska, Felix Höfling)*

New features

- select particles in a region of the simulation box through the new particle group *`region`*. The selection can be inside or outside of a cuboid or a sphere. Further, the new particle group *`region_species`* restricts this selection to a certain particle species. *(Nicolas Höft, Roya Ebrahimi Viand)*
- support fluctuating particle number in file writer of *`halmd.observables.thermodynamics`*. It needs the new property `fluctuating` of the `particle group` to be set. *(Felix Höfling)*
- support for *`external, one-body potentials`*. The list of potential functions is easily extensible, currently we have a harmonic trap and a set of planar Lennard-Jones walls to form, e.g., a slit pore or a wedge *(Felix Höfling, Sutapa Roy)*
- signals `prepend_apply` and `append_apply` of the force modules, which can be used e.g., to read out the force partially when adding up contributions from different potentials *(Sutapa Roy)*

- wavevectors can be restricted to an axis or plane aligned with the coordinate frame, using the new `filter` keyword of *halmd.observables.utility.wavevector*. In addition, a `dense` grid of wavevectors can be generated from the reciprocal lattice. *(Felix Höfling)*

Improvements

- usage documentation: add *tutorial* on diffusion in a simple liquid *(Jake Atwell)*
- examples: simplify scripts for basic simulations of a one-component Lennard-Jones fluid *(Felix Höfling)*
- developer's guide: describe *programme flow*, data dependencies between modules, and the signal/slot mechanism *(Felix Höfling)*
- support multiple calls to *sampler:run()*. With the default simulation engine, the signals `on_start` and `on_finish` are triggered only upon first invocation of `run()` and upon return from `main()` of the simulation script, respectively. The *profiler* output function is connected to the `on_finish` signal, removing the need to call it explicitly. *(Felix Höfling)*
- forward non-default neighbour list parameters to *forces.pair_trunc*, which in many cases removes the need to construct a custom neighbour list module *(Felix Höfling)*
- simple substitution of environment variables in filename pattern using a new argument parser action from *halmd.utility.program_options* *(Felix Höfling)*
- *halmd.mdsim.clock* can be advanced explicitly in the simulation script (for experienced users only) *(Felix Höfling)*
- support the CUDA SDK up to version 10.2, the HDF5 library up to version 1.10.6, and the Boost C++ library up to version 1.72.0 *(Jaslo Ziska)*

12.2 Version 1.0-alpha6

Breaking changes

- simplifications of the simulation script: essential setup (e.g. logger) is no longer the user's responsibility, merely routines `main()` and optionally `define_args()` are needed. Provide argument parser actions *(Daniel Kirchner)*
- pair potentials are truncated or modified by generic potential adapters *(Daniel Kirchner)*

Bug fixes

- remove velocity rescaling from *halmd.mdsim.velocities.boltzmann*, shifting of the velocities to zero centre of mass is retained. This fixes also the integrator *halmd.mdsim.integrators.verlet_nvt_boltzmann*, which now samples correctly from a Maxwell-Boltzmann distribution (except for the mentioned constraint). *(Felix Höfling, Daniel Kirchner)*
- host backend compiles with single floating-point precision again *(Daniel Kirchner)*
- support very large particle numbers on the GPU (more than 10^7) by improved choice of the CUDA configuration dimensions *(Daniel Kirchner)*

Improvements

- more flexible interfaces of *halmd.mdsim.particle* and *halmd.observables.phase_space* using string-based identifiers of data arrays *(Daniel Kirchner)*

- specify floating-point precision in the simulation script (see construction of `halmd.mdsim.particle`). Use CMake flags to control for which precisions GPU and host backends are built. *(Daniel Kirchner)*
- overwrite output files only if forced to do so, add command line flag `--overwrite`. (*Daniel Kirchner*)

Internal changes

- switch to extensible and generic data arrays in `halmd.mdsim.particle`, unified implementation of single and mixed precision arrays *(Daniel Kirchner)*

12.3 Version 1.0-alpha5

Breaking changes

- increase minimal required version of the CUDA SDK to 5.0 *(Nicolas Höft)*

Bug fixes

- fix build with up-to-date versions of build tools and third-party libraries: *(Daniel Kirchner, Nicolas Höft)*
- minor fixes in exemplary simulation scripts *(Felix Höfling, Nicolas Höft)*

New features

- release the software under the terms of the LGPL-3+ license *(Felix Höfling)*
- find additional Lua scripts in the directory of the simulation script, which allows factoring out common functions or definitions, e.g., of interaction potentials *(Felix Höfling)*
- add function `to_particle()` to particle groups, which convert the selected particles to an instance of `halmd.mdsim.particle` *(Nicolas Höft)*

Improvements

- drop restriction on minimal number of Verlet neighbour cells (fall back to no binning upon neighbour list construction) *(Nicolas Höft)*
- builds with standard CMake $\geq$ 2.8.12, the patch for native CUDA support is not needed anymore *(Daniel Kirchner)*
- support version 7.5 of the CUDA SDK *(Daniel Kirchner)*
- increase default CUDA compute capability to 2.0 (older hardware is still supported with CUDA SDK $\leq$ 6.0) *(Daniel Kirchner)*

12.4 Version 1.0-alpha4

Breaking changes

- Lua scripts in `examples/liquid`: rename option "–trajectory" to "–input" *(Felix Höfling)*

Bug fixes

- fix native build for Fermi and Kepler GPUs (compute capability $\geq$ 2.0) *(Nicolas Höft)*
- fix build with Boost $\geq$ 1.57.0 *(Nicolas Höft)*

- compatibility with HDF5 $\geq$ 1.8.14 *(Nicolas Höft, Felix Höfling)*

New features

- add function `halmd.random.shuffle()`, which allows one, e.g., to setup well-stirred fluid mixtures *(Felix Höfling)*

Improvements

- documentation: new sections "Recipes" and "Publications" *(Felix Höfling)*
- support version 6.0 of the CUDA SDK *(Nicolas Höft)*
- support both Lua 5.1 and Lua 5.2 (built without compatibility flags) *(Nicolas Höft)*

12.5 Version 1.0-alpha3

Breaking changes

- mdsim/particle: make space dimension mandatory *(Felix Höfling)*
- mdsim/potentials: move to sub-namespace "pair" *(Felix Höfling)*

Bug fixes

- potentials: fix uninitialised memory for energy shift (host only) *(Nicolas Höft)*
- integrators: make Nosé–Hoover thermostat working again *(Felix Höfling)*
- compile and build against Boost C++ 1.56 *(Felix Höfling)*

New features

- tensor-valued custom dynamic correlation functions *(Felix Höfling)*

Improvements

- packages.mk: more robust installation of prerequisites *(Felix Höfling)*
- documentation: installation instructions, minor fixes *(Felix Höfling)*

Internal changes

- move from boost::{array,tuple} to std::{array,tuple} *(Nicolas Höft)*
- mdsim/box: fix failing unit test *(Nicolas Höft)*

12.6 Version 1.0-alpha2

Improvements

- drop submodule Boost.Log and use library shipped with Boost $\geq$ 1.55 *(Nicolas Höft)*
- integrators: reduce memory access gives a 10-20% performance gain (GPU only) *(Felix Höfling)*
- documentation: new benchmark results, minor fixes and updates *(Felix Höfling, Nicolas Höft)*

Bug fixes

- integrators: fix missing update of box image data (GPU only) *(Felix Höfling)*

12.7 Version 1.0-alpha1

Substantial development (1225 commits!) happened since version 0.2.1 with contributions from Peter Colberg, Felix Höfling, and Nicolas Höft.

Most important changes

- completed the switch from a monolithic to a modular design
- modules are exposed through a Lua API
- simulations are defined and steered by either custom or shipped Lua scripts
- dynamic correlation functions can be customised
- H5MD format (version 1.0) for input and output files
- face lift of the website

12.8 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

12.9 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.

12.10 Version 0.1.3

Improvements

- fully support mobility filters for the VACF

12.11 Version 0.1.2

Improvements

- revise documentation

Bug fixes

- fix build failure with Boost C++ 1.46

12.12 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

12.13 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

THIRTEEN

DOCUMENTATION ARCHIVE

- cutting edge snapshot
- latest testing release
- latest stable release
- version 0.2.1
- version 0.2.0
- version 0.1.3

CHAPTER

FOURTEEN

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.

This flag does not apply to the CUDA runtime library. To use the CUDA runtime statically see CUDA_USE_STATIC_CUDA_RUNTIME.

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`.

CUDA_USE_STATIC_CUDA_RUNTIME Link to the CUDA runtime library statically.

Note, that linking statically could lead to kernel launch failures when using CUDA 9.2.

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

FIFTEEN

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/hdf5-1.8.8:$HOME/opt/rhel6-x86_64/boost_1_49_0
```

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.

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

W