RMPCDMD Documentation

Release 1.1-dev

Pierre de Buyl and contributors

Nov 21, 2017

CONTENTS

CHAPTER

ONE

ABOUT RMPCDMD

Authors Pierre de Buyl, Peter Colberg, Laurens Deprez, Mu-Jie Huang

License BSD 3-clause

Website http://lab.pdebuyl.be/rmpcdmd/

Version 1.1-dev

RMPCDMD is a software for the simulation of colloids via Molecular Dynamics, embedded in a MPCD fluid. Ready-to-execute simulation programs are provided for the dimer nanomotor in Periodic Boundary Conditions (PBC), the forced Poiseuille flow or for N colloids in PBC. These programs only require the setting of parameters in the ad-hoc text file for execution.

Highlights:

- The simulation of dimer nanomotors, reproducing the pioneering work of Rückner and Kapral *[RK07]* is well tested.
- This code is a research code, so other features are probably *under development*. This should not prevent you from using it!
- We have a *Tutorial* on nanomotor simulations.
- If you use this code, the appropriate citation is *[dBHD17]* (the bibtex data is in `CITATION`). Consider citing the paper *also* if you use the present resources (documentation, algorithm, tutorial).

Features:

- MPCD collision rule for the solvent
- Chemical activity (either catalytic at a colloid or in the bulk)
- Rattle constrained dynamics for rigid bodies
- Quaternion-based rigid-body Molecular Dynamics
- Walls (specular, bounce-back, virtual particles)
- Hilbert curve based spatial sorting of solvent particles
- H5MD trajectory file output *[dBCH14]*
- Fortran 2008 codebase using modules and *no global variable*
- OpenMP multithreaded operation

Development and contact information:

- Development of the code takes place on GitHub.
- The contact for RMPCDMD is the main author, Pierre de Buyl.

- Bug reports are welcome either by email or via GitHub issues

The source code features inline comments, published with Doxygen: api.

The use of the Hilbert curve sorting and of the Threefry Random Number Generator (*[SMDS11]*) is inspired by Peter Colberg's code *nano-dimer [Col15]*.

CHAPTER

TWO

ALGORITHMS

2.1 Quaternion velocity Verlet

2.1.1 Intro and notation

Rigid-body dynamics can be performed either by applied distance constraints to the bonds in the body with RATTLE or by using a "quaternion integrator" *[AT87]*. In the quaternion approach, the rigid body is completely determined by its center-of-mass position and velocity, by its quaternion orientation and by its angular velocity.

Here, we follow the velocity Verlet form of the algorithm proposed by Rozmanov and Kusalik *[RK10]*.

For clarity, the notation for quaternions and the rigid body's coordinates is given explicitly.

- r is a vector denoting the position of a point or particle in Euclidean space.
- r_i is as r, but for the i^{th} particle of the rigid body.
- $r_{i,j}$ is the k^{th} component of r_i.
- v_i is the velocity of the i^{th} particle of the rigid body.
- ω is the angular velocity of the rigid body.
- I is the inertia tensor, with components $I_{i,j}$.
- L is the angular momentum, a vector.
- T is the torque on the rigid body. Elsewhere in this documentation, T is the temperature. The context is sufficient to avoid ambiguity.
- t is the time.
- h is the integrator time step.
- $\dot{\bullet}$ is the time derivative of $\bullet$.
- $\wedge$ is the cross product.
- $\cdot$ is the dot product.
- $\hat{x}$ is the unit vector parallel to x.
- q, q_1, etc. are quaternions.
- q^* is the conjugate of q.
- $|q|$ is the norm of q.
- The superscript ^{B} is for quantities defined in the frame of reference of the rigid body. Other quantities are in the laboratory reference frame.

Quaternions are defined as the sum of a scalar part s and a 3-dimensional vector part w as

$$q = s + w$$

The multiplication of the elementary vectors $w_i = (1, 0, 0)$, $w_j = (0, 1, 0)$ and $w_k = (0, 0, 1)$ is defined as

- $w_i w_j = w_k$
- $w_j w_k = w_i$
- $w_k w_i = w_j$
- $w_j w_i = -w_k$
- $w_k w_j = -w_i$
- $w_i w_k = -w_j$
- $w_i^2 = w_j^2 = w_k^2 = -1$

From these definitions, one can derive the product of quaternions:

$$q_1 q_2 = (s_1 s_2 - w_1 \cdot w_2) + (q_1 w_2 + q_2 w_1 + w_1 \wedge w_2)$$

where the first parenthesized term is the scalar part and the second one the vector part. The addition of quaternions is done separately for the scalar and vector parts, following conventional algebra. The conjugate of a quaternion is $q^* = s - w$, its norm is $|q| = \sqrt{s^2 + w \cdot w}$ and its inverse is $q^{-1} = q^*/|q|^2$.

The utility of quaternions arises from their ability to encode any solid rotation. The rotation operator about an axis n with an angle θ is given by $x' = qxq^*$ where q is a unit quaternion with scalar component $\cos\theta/2$ and vector component $\sin\theta/2\ \hat{n}$.

All quaternion operations in RMPCDMD are performed with the Fortran module fortran_quaternion.

2.1.2 Definition of rigid-body quantities

The position in the laboratory frame of a member of the rigid body is

$$r_i(t) = q(t) r_i^B q^*(t)$$

The equation of motion for the angular momentum in the laboratory frame is

$$\dot{L} = T,$$

and in the body frame

$$\dot{L}^B = T^B - \omega^B \wedge L^B.$$

The equation of motion for the quaternion is

$$\dot{q} = \frac{1}{2}\omega q$$

2.1.3 Algorithm

Here, we follow section IV.A of Ref. *[RK10]*.

Angular momentum	Torque	Quaternion	Angular velocity
$L(h/2) = L(0) + hT(0)/2$			
$L^B(0) = q^*(0)L(0)q(0)$	$T^B(0) = q^*(0)T(0)q(0)$		
			$\omega^B(0) = L^B(0)/I^B$
$\dot{L}^B(0) = T^B(0) - \omega^B(0) \wedge L^B(0)$			
$L^B(h/2) = L^B(0)+h\dot{L}^B(0)/2$			
			$\omega^B(h/2) = L^B(h/2)/I^B$
		${}^0\dot{q}(h/2) = q(0)\omega^B(h/2)/2$	
		${}^0q(h/2) = q(0) + h\,{}^0\dot{q}(h/2)/2$	
Start of the iterative procedure			
${}^{k+1}L^B(h/2) = {}^kq^*(h/2)L(h/2)\,{}^kq(h/2)$			
			${}^{k+1}\omega^B(h/2) = {}^{k+1}L^B(h/2)/I^B$
		${}^{k+1}\dot{q}(h/2) = {}^kq(h/2)\,{}^{k+1}\omega^B(h/2)/2$	
		${}^{k+1}q(h/2) = q(0) + h\,{}^{k+1}\dot{q}(h/2)/2$	
End of the iterative procedure			
		$q(h) = q(0) + h\dot{q}(h/2)$	
Update positions using $q(h)$			
Compute the updated forces and torques in the lab frame			
	$T^B(h) = q^*(h)T(h)q(h)$		
$L(h) = L(h/2) + hT(h)/2$			
Update the velocities using $\omega^B(h) = L^B(h)/I^B$.			

The steps until the update of $q(h)$ and of the positions is the first step of the velocity Verlet algorithm. These steps are implemented in rigid_body_vv1.

The update of $L(h)$ and of the velocities form the second part of the velocity Verlet algorithm and are implemented in rigid_body_vv2.

2.2 MPCD

The MPCD algorithm introduced in *[MK99]* is implemented in simple_mpcd_step for periodic boundary conditions.

The presence of walls is taken into account in wall_mpcd_step following *[LGIK01]*: ghost particles are placed in the boundary cells, during the collision, to reach the average density of the solvent. Optionally, a bulk Anderson thermostat can also be applied in this routine.

2.3 Molecular Dynamics

Molecular Dynamics (MD) is implemented using the Velocity verlet integration scheme *[MK00]*, where solvent particles and colloids evolve at the MD timestep dt.

The implementation in RMPCDMD is found in md_pos and md_vel. It is important to know that particles close to walls have their velocities updated in the *stream* routines and are skipped in md_vel. This is only correct if they are outside of the interaction range of all colloids, which is the case in all simulations here.

2.4 Boundary conditions

Bounce-back boundary conditions are used for the walls, in addition to the modified collision rule. The are presented in *[AG02]* or *[WL10]* and implemented in mpcd_stream_xforce_yzwall. There, the bounce-back collision is computer for the parabolic trajectories relevant for forced flows.

2.5 Convenience routines

2.5.1 Temperature computation

A general routine to compute the temperature compute_temperature computes cell-wise the kinetic energy relative to the cell's center-of-mass.

$$T = \frac{1}{3N_c} \sum_{\xi} \frac{1}{N_\xi - 1} \sum_i m_i \left(v_i - v_\xi\right)^2$$

where N_c is the number of cells, the variable ξ represents a cell, N_ξ the number of particles in a cell, v_ξ the center-of-mass velocity of the cell, and m_i and v_i the mass and velocity of particle.

CHAPTER

THREE

DESIGN

3.1 General principles

General purpose routines and computational algorithms are implemented as Fortran modules in *src/*. Parts of the code is tested with programs in *test/*.

Actual simulations are found in *programs/*, where every program corresponds to a given setup. The programs read parameters from a text file and store trajectories in H5MD files.

The aim of the code organization is that one can write a program that call trusted routines from the modules, giving a high-level view on the simulation. There is no global variable in the code so that a routine may only act on variables that are passed to it.

3.2 Storage of cell and particle data

The storage of particle and cell data is defined in the module *particle_system* and *cell_system*, respectively.

The derived type *particle_system_t* defines arrays for the position, velocity, etc of every particle.

The derived type *cell_system_t* defines a cell list with a one-dimensional index that is mapped to a three-dimensional compact Hilbert index *[Ham06]*. Particles are sorted according to this cell system, as in the nano-dimer software *[Col15]*.

Operations on particles can proceed in two different ways:

1. Particle-wise: The loop goes over every particle, following the particle index of the particle data in memory.
2. Cell-wise: The loop runs over successive cells, where an inner loop runs over all particles. Thanks to the sorting of particles, the indices of the particles in a given cell are contiguous.

3.3 Programming language

RMPCDMD is written in Fortran 2008 and uses derived types objects to store simulation data.

RMPCDMD is built using CMake, which facilitates the inclusion of external libraries (fortran_h5md, random_module, ParseText and fortran_tester). The HDF5 library, with the Fortran 2003 interface, is also necessary to build RMPCDMD.

3.4 Random numbers

A C library implements the Threefry Random Number Generator (RNG) *[SMDS11]*. This RNG can be used in separate independent threads, provided that the thread's numerical ID is part of the *key* of the RNG. The other part of the key, that is made of two 64-bit integers, is the seed that is provided by the user.

The RNG is implemented in random_module, that is fetched as a git submodule in RMPCDMD. A Fortran wrapper, using the `bind(c)` attribute, manages the Fortran/C interface.

CHAPTER

FOUR

INSTALL

4.1 Requirements

To compile RMPCDMD, the following software is required

- A Fortran 2008 compiler (e.g. gfortran $\geq$ 4.7 with support for OpenMP $\geq$ 3.1)
- A Fortran-2003 enabled HDF5 installation
- CMake
- GNU Make
- git

Other utilities, mostly Fortran code, will be downloaded automatically as part of the installation process.

RMPCDMD has been developed on Linux, using both the GFortran and Intel Fortran compilers. Compilation under OS X has been successfully achieved by users of RMPCMD. The Windows platform is currently not tested.

If you encounter a configuration issue, it is often useful to remove all CMake data by executing:

```
rm -r CMake*
```

in the build directory (*not in the source directory*).

After successfully building the program, copy the file *rmpcdmd* in a location where executables are found (i.e. `$HOME/.local/bin` or `$HOME/bin` for instance). This file provides a unique interface to RMPCDMD.

To analyze simulation data, RMPCDMD provides the command `rmpcdmd plot` (see *Run the code*) that requires Python and a number of packages (see below).

Note: Once CMake has been executed, the RMPCDMD directory cannot be moved. If it is moved, the complete configuration procedure has to be restarted and the *rmpcdmd* program (in `$HOME/.local/bin` or `$HOME/bin`) must be replaced by the newly created version.

Note: The *rmpcdmd* program does not contain the simulations programs themselves and relies on their presence at the location of compilation.

4.2 Building on Linux

On a Debian distribution (or derivative, e.g. Ubuntu), as root:

```
apt-get install gfortran libhdf5 cmake git
```

will install the dependencies. (See *install_hdf5* for Ubuntu 14.04 or problematic HDF5 installs).

At the command line, execute the following:

```
git clone https://github.com/pdebuyl-lab/RMPCDMD
cd RMPCDMD
git submodule init
git submodule update
mkdir build
cd build
cmake ..
make VERBOSE=1
```

This will first fetch RMPCDMD, then the related modules *ParseText*, *fortran_tester*, *fortran_h5md* and *random_module*. The compilation is prepared by the `cmake` command and executed by the `make` command.

If your HDF5 installation is properly setup, CMake should find it automatically. If this is not the case, you may define the environment variable `HDF5_ROOT=/path/to/hdf5` before invoking cmake (see the OS X installation notes for an example).

Note: When using compilers that are not the default one of the Linux distribution, such as the Intel compilers, the following settings must be used (change the compiler name if needed) *before* running cmake and *before* possibly building HDF5:

```
export CC=icc
export FC=ifort
```

4.3 Building on OS X

The first step is to install a development environment, and thus the XCode software from Apple (we have tested RMPCDMD with XCode 7.3.1 on OS X El Capitan).

We used MacPorts for gcc, git, cmake and make. Install MacPorts (a GUI installer is available at https://www.macports.org/install.php) and then the dependencies:

```
sudo port install gcc-5 cmake git
```

The rest is similar to Linux except that the name of the compiler has to be specified manually at the command-line to prevent the automatic selection of Apple's provided compiler. Also, HDF5 is built locally:

```
git clone https://github.com/pdebuyl-lab/RMPCDMD
cd RMPCDMD
git submodule init
git submodule update
export CC=gcc-mp-5
export FC=gfortran-mp-5
./scripts/download_and_build_hdf5.sh
mkdir build
cd build
HDF5_ROOT=../_hdf5-1.8.17 cmake ..
make
```

The compiler names given here may vary depending on your setup.

4.4 Building on Windows

Installation on Windows is not tested at this time. The major difficulty is likely related to a functional installation of HDF5 for Fortran under Windows, that is only provided for the Intel compiler. Having no Windows computer with the Intel Fortran compiler at our disposal, user feedback is welcome.

4.5 HDF5

On OS X or with older Linux systems (such as Ubuntu 14.04 still in wide usage), it is necessary to build HDF5 to enable the Fortran 2003 interface.

A script is provided to download and build HDF5 that has been tested on OS X and on Ubuntu. It must be run from the main RMPCDMD directory:

```
./scripts/download_and_build_hdf5.sh
```

with compiler definitions `CC` and `FC` set when using OS X or a compiler that is not the default one for the computer, such as `icc` and `ifort`. This script must be run only once and the environment variable `HDF5_ROOT` must be set when invoking cmake (also see the OS X installation notes).

The script downloads HDF5 1.8.17 and installs it under `_hdf5-1.8.17`. You can remove the directory `hdf5-1.8.17` (no leading underscore) after the execution of the script.

4.6 Python

Several analysis scripts in the `experiments/` directory and a command-line tool `rmpcdmd plot` are provided. They all rely on the Python programming language and the following Python packages:

- NumPy
- SciPy
- matplotlib
- h5py

Optionally, you may wish to install Mayavi for the 3D visualization of the dimer simulation.

Installing those tools under Linux is straightforward:

```
sudo apt-get install python-numpy python-scipy python-matplotlib python-h5py
```

for Debian-based systems or:

```
sudo yum install numpy scipy h5py python-matplotlib
```

for Red-Hat based systems.

On OS X, we recommend to use a Python "super package" such as Enthought Canopy or Anaconda from Continuum that bundle the required software.

CHAPTER

FIVE

RUN THE CODE

The execution of RMPCDMD is controlled via a single command-line program, `rmpcdmd`. RMPCDMD must be built before using this tool, see the *Install* documentation.

5.1 rmpcdmd run

Usage:

```
rmpcdmd run program input output seed
```

Arguments

- `program` one of the simulation program coming with RMPCMD (i.e. single_dimer_pbc)
- `input` text file with simulation parameters
- `output` filename for the output data
- `seed` a signed 64-bit integer value. `seed` can be set to the value `auto`, a seed will the be generated from the `/dev/urandom` device.

When run with no argument, `rmpcdmd run` will list the parameters and the possible values for `program`.

The simulation programs (see *Programs*) can also be executed directly at the command-line with the syntax:

```
./program input output seed
```

Where the arguments are the same as when using `rmpcdmd`, with the differences

- For `seed` the keyword `auto` cannot be used.
- The `./` must be replaced by the full path to the build directory when executing from another directory.
- Less information is available in the output (start and end times, value of `OMP_NUM_THREADS` environment variable.

5.2 rmpcdmd plot

Usage:

```
rmpcdmd plot [-h] datafile [--obs OBSERVABLE] [--traj GROUP/TRAJECTORY]
```

Arguments

- `-h` display the the full command-line syntax and exit

- `datafile` a datafile produced by one of the simulation programs

One of `[--obs OBSERVABLE]` or `[--traj GROUP/TRAJECTORY]` can be given to display an observable or a trajectory from the file.

5.3 rmpcdmd seeder

Usage:

```
rmpcdmd seeder
```

Returns a signed 64-bit integer seed.

5.4 rmpcdmd timers

Usage:

```
rmpcdmd timers [-h] datafile [--plot]
```

Arguments

- `-h` display the the full command-line syntax and exit
- `datafile` a datafile produced by one of the simulation programs
- `--plot` plots the timers data as a bargraph instead of printing to the terminal.

Prints (or plot in the `--plot` option is given) the value of the timers in the simulation file `datafile`.

5.5 experiments/ directory

The execution of some RMPCDMD simulations is illustrated in the directory `experiments/`, using makefiles for simplicity. An example simulation session is given below

```
user@pc$~$ cd /tmp/RMPCDMD/
user@pc$/tmp/RMPCDMD$ cd experiments/01-single-dimer/
user@pc$/tmp/RMPCDMD/experiments/01-single-dimer$ ls
dimer.parameters  Makefile  plot_histogram.py  plot_velocity.py
ruckner-kapral.parameters
user@pc$/tmp/RMPCDMD/experiments/01-single-dimer$ make simulation
/tmp/RMPCDMD/experiments/01-single-dimer/../../build/rmpcdmd run single_dimer_pbc
dimer.parameters dimer.h5 auto
RMPCDMD running single_dimer_pbc
OMP_NUM_THREADS not set
Start time -- Thu Jun 16 13:40:08 CEST 2016
single_dimer_pbc dimer.parameters dimer.h5 3589052620060159831

 Running for         100 loops
 mass   1130.9733867645264        1130.9733867645264
    5   10   15   20   25   30   35   40   45   50   55   60   65   70   75   80   85
   90   95  100
 n extra sorting         747

real    3m25.006s
```

```
user    10m13.496s
sys     0m0.988s
End time -- Thu Jun 16 13:43:33 CEST 2016
205s elapsed
user@pc$/tmp/RMPCDMD/experiments/01-single-dimer$
```

CHAPTER

SIX

PROGRAMS

6.1 Configuration files

The parameters listed for each program are read from text configuration files. The syntax is given by example for the arguments that follow.

- **T** Temperature
- **L** Box size
- **enable_thermostat** Activate the thermostat

```
T = 1.2
L = 32 32 32
enable_thermostat = F
```

Boolean values are input as T (True) or F (False). For vectors (such as the box size **L**), several components are listed, separated by a space. For more examples, the subdirectories in `experiments/` contain configuration files for some simulations.

6.2 `single_dimer_pbc`

6.2.1 Synopsis

Source code: single_dimer_pbc

Simulate a single dimer nanomotor

Consider a dimer nanomotor in a periodic simulation cell filled with A particles. After a collision with the catalytic sphere of the dimer, a A particle is converted to B.

6.2.2 Parameters

- **L** length of simulation box in the 3 dimensions
- **rho** fluid number density
- **T** Temperature. Used for setting initial velocities and (if enabled) bulk thermostatting.
- **tau** MPCD collision time
- **probability** probability to change A to B upon collision
- **bulk_rmpcd** use bulkd rmpcd reaction for B->A instead of resetting

- **bulk_rate** rate of B->A reaction
- **N_MD** number MD steps occuring in tau
- **N_loop** number of MPCD timesteps
- **colloid_sampling** interval (in MD steps) of sampling the colloid position and velocity
- **equilibration_loops** number of MPCD steps for equilibration
- **sigma_C** radius of C sphere
- **sigma_N** radius of N sphere
- **d** length of rigid link
- **epsilon_C** interaction parameter of C sphere with both solvent species (2 elements)
- **epsilon_N** interaction parameter of N sphere with both solvent species (2 elements)
- **epsilon_C_C** interaction parameter among C spheres
- **epsilon_N_C** interaction parameter among N and C spheres
- **epsilon_N_N** interaction parameter among N spheres

6.3 chemotactic_cell

6.3.1 Synopsis

Source code: chemotactic_cell

Model a chemotactic experiment in a microfluidic channel

In this simulation, an inlet (x=0) is fed with A and S fluid species in the lower and upper halves in the y direction, respectively. A constant accerelation is applied in the x direction and walls in the z direction confine the flow, leading to a Poiseuille velocity profile.

The colloid is a passive sphere, an active sphere or a dimer nanomotor.

6.3.2 Parameters

- **g** magnitude of acceleration
- **buffer_length** length of the inlet buffer
- **max_speed** maximum velocity of profile to initialize the velocities
- **probability** probability of reaction
- **alpha** angle of collision
- **store_rho_xy** store the xy density of solvent particles on a grid
- **store_rho_xy_z** bounds in z for the slice of rho_xy to store (2 elements)
- **dimer** simulate a dimer nanomotor (boolean, else it is a single sphere)
- **N_type** assign N species to the single sphere (boolean, else it is a C sphere)
- **L** length of simulation box in the 3 dimensions
- **rho** fluid number density

- **T** Temperature. Used for setting initial velocities and for wall thermostatting.
- **d** length of rigid link
- **N_in_front** place N sphere in front (higher x), for the dimer nanomotor
- **tau** MPCD collision time
- **N_MD** number MD steps occuring in tau
- **N_loop** number of MPCD timesteps
- **colloid_sampling** interval (in MD steps) of sampling the colloid position and velocity
- **steps_fixed** number of steps during which the colloid is fixed (only when buffer_length>0)
- **equilibration_loops** number of MPCD steps for equilibration (only when buffer_length=0)
- **sigma_C** radius of C sphere
- **sigma_N** radius of N sphere
- **track_y_shift** shift of the track in the y direction with respect to Ly/2
- **epsilon_C** interaction parameter of C sphere with both solvent species (2 elements)
- **epsilon_N** interaction parameter of N sphere with both solvent species (2 elements)

6.4 single_body

6.4.1 Synopsis

Source code: single_body

Simulate a single colloidal rigid-body particle

This simulation models a chemically active colloid particle in either a periodic simulation box or with confinement in the y direction.

The coordinates of the colloid particle's beads must be provided in a H5MD file, as a "fixed-in-time" dataset. The body of the particle can operate as a rigid-body (with either RATTLE or quaternion dynamics) or as an elastic network.

6.4.2 Parameters

- **L** length of simulation box in the 3 dimensions
- **rho** fluid number density
- **T** Temperature. Used for setting initial velocities and (if enabled) bulk thermostatting.
- **tau** MPCD collision time
- **alpha** angle of collision
- **probability** probability to change A to B upon collision
- **bulk_rate** rate of B->A reaction
- **N_MD** number MD steps occuring in tau
- **N_loop** number of MPCD timesteps
- **colloid_sampling** interval (in MD steps) of sampling the colloid position and velocity

- **do_solvent_io** if true (T), a snapshot of the solvent in the final step is dump to the datafile
- **equilibration_loops** number of MPCD steps for equilibration
- **epsilon_C** interaction parameter of C sphere with both solvent species (2 elements)
- **epsilon_N** interaction parameter of N sphere with both solvent species (2 elements)
- **data_filename** filename for input Janus coordinates
- **data_group** particles group in the input file
- **epsilon_colloid** interaction parameter for colloid-colloid interactions
- **reaction_radius** radius for the reaction around the Janus particle
- **link_treshold** distance criterion for finding rigid-body links
- **do_read_links** read link information from a file
- **links_file** filename for the links data
- **do_rattle** perform RATTLE
- **do_lennard_jones** compute colloid-colloid Lennard-Jones forces
- **do_elastic** compute colloid-colloid elastic network forces
- **elastic_k** elastic constant for the elastic network
- **rattle_pos_tolerance** absolute tolerance for Rattle (position part)
- **rattle_vel_tolerance** absolute tolerance for Rattle (velocity part)
- **do_quaternion** perform quaternion velocity Verlet
- **quaternion_treshold** treshold for the iterative procedure for the quaternion integrator
- **sigma** radius of the colloidal beads for colloid-solvent interactions
- **sigma_colloid** radius of the colloidal beads for colloid-colloid interactions
- **polar_r_max** maximal radius for the polar fields
- **do_ywall** use a confining potential in the y direction, 9-3 Lennard-Jones
- **wall_sigma** wall LJ sigma
- **wall_epsilon** wall LJ epsilon
- **wall_shift** wall shift
- **fluid_wall** boundary condition for the fluid

6.5 poiseuille_flow

6.5.1 Synopsis

Source code: poiseuille_flow

Simulate a forced flow between two plates

Consider a pure fluid under a constant acceleration in the x-direction. Confining plates, modeled as Bounce-back boundary conditions are used in the z-direction in addition to ghost cells for the collisions near the walls.

6.5.2 Parameters

- **L** length of simulation box in the 3 dimensions
- **g** strength of the constant acceleration in x
- **rho** fluid number density
- **T** Temperature. Used for setting initial velocities, for wall thermostatting and (if enabled) bulk thermostatting.
- **tau** MPCD collision time
- **alpha** MPCD collision angle
- **thermostat** whether to enable bulk thermostatting
- **N_therm** number of unsampled thermalization MPCD timesteps
- **N_loop** number of MPCD timesteps

6.6 `single_sphere_thermo_trap`

6.6.1 Synopsis

Source code: single_sphere_thermo_trap

Simulate a thermal gradient with an embedded colloidal sphere

Consider a pure fluid under a thermal gradient in the z-direction. The confining plates are modeled as bounce-back boundary conditions in the z-direction in addition to ghost cells for the collisions near the walls.

The x-z components of the fluid velocity field is stored at fixed intervals in the center-y layer of cells.

The sphere can be either fixed or held by an harmonic trap.

6.6.2 Parameters

- **L** length of simulation box in the 3 dimensions
- **rho** fluid number density
- **T** Temperature. Used for setting initial velocities.
- **wall_T** Temperature at the walls (two values).
- **k** trap stiffness
- **tau** MPCD collision time
- **alpha** MPCD collision angle
- **N_therm** number of unsampled thermalization MPCD timesteps
- **N_loop** number of MPCD timesteps
- **N_MD** number of MD timesteps per tau
- **vxz_interval** interval for storing the xz velocity field
- **sigma** size of spherical colloid
- **epsilon** interaction parameter of sphere with fluid
- **move** enable motion of the sphere (boolean)

6.7 n_colloids_pbc

6.7.1 Synopsis

Source code: n_colloids_pbc

Simulate an ensemble of spherical colloids

The periodic simulation box is filled with a number of spherical colloids, that interact with an attractive Lennard-Jones potential, and with solvent particles. The temperature is controlled with the MPCD Anderson thermostat.

6.7.2 Parameters

- **L** length of simulation box in the 3 dimensions
- **rho** fluid number density
- **T** Temperature. Used for setting initial velocities and for thermostatting.
- **T_final** Target temperature. Used for thermostatting with temperature program from T to T_final.
- **tau** MPCD collision time
- **N_MD** number MD steps occuring in tau
- **colloid_sampling** interval (in MD steps) of sampling the colloid position and velocity
- **N_loop** number of MPCD timesteps
- **N_colloids** number of colloids
- **epsilon** solvent-colloid epsilon
- **sigma** radius of the colloids
- **epsilon_colloids** colloid-colloid epsilon

CHAPTER

SEVEN

TUTORIAL

This tutorial introduces the reader to particle-based simulations of nanomotors. The main simulation method consists of a coupled scheme of Multiparticle Collision Dynamics (MPCD), for the fluid, and Molecular Dynamics (MD), for the motor. Chemical activity is introduced in the fluid via a surface-induced catalytic effect and bulk kinetics.

To start working with RMPCMD, visit the *Install* section first.

7.1 Introduction

Nano- to micro-meter scale devices that propel themselves in solution are built and studied since about a decade. They represent a promise of future applications at scales where usual control strategies reach their limits and, ideally, autonomous action replaces manual control.

It is possible to compute, via phoretic theory, the stationary regime of operation of nanomotors in simple geometries. Still, testing geometrical effects or including fluctuating behaviour is best done using numerical simulations. A successful modeling strategy was started by Rückner and Kapral in 2007 *[RK07]*. It builds on a particle-based fluid, explicitly the Multiparticle Collision Dynamics (MPCD) algorithm. The flexibility of particle-based simulations allowed for numerous extensions of their work to Janus particles, polymer nanomotors, various chemical kinetics, thermally active motors, among others.

The principle of the hybrid scheme is very close to full Molecular Dynamics (MD), with the major difference that solvent-solvent interactions are not explicitly computed and are replaced by cell-wise collisions at fixed time intervals. This saves computational time and renders otherwise untractable problems feasible.

In Ref. *[RK07]*, the authors introduce a computational model for a dimer nanomotor that is convenient thanks to its simple geometry. There are two spheres making up the dimer, linked by a rigid bond, one of which being chemically active and the other not. Solvent particles in contact with the chemically active sphere are converted from product to fuel. The active sphere thus acts as a sink for reagent particles (the "fuel") and a source for product particles.

7.2 Preliminary remarks

This tutorial does not intend to cover all *possible* manners to conduct nanomotor simulations. Rather, it aims at presenting one strategy for modeling chemically powered nanomotors, that is the combination of a chemically active MPCD fluid coupled to possibly catalytic colloidal beads.

While this tutorial relies on the RMPCDMD software, it is worth mentioning that Peter Colberg has also made his software for dimer nanomotor simulations available openly *[Col15]*. `nano-dimer` is based on OpenCL and can benefit from GPU acceleration.

7.3 Literature

The MPCD algorithm was introduced in *[MK99]* and *[MK00]*. General reviews on the MPCD simulation method are available in the literature.

- Raymond Kapral, *Multiparticle collision dynamics: simulation of complex systems on mesoscales*, Adv. Chem. Phys. **140**, 89 (2008). *[Kap08]*
- G. Gompper, T. Ihle, D. M. Kroll and R. G. Winkler, *Multi-Particle Collision Dynamics: A Particle-Based Mesoscale Simulation Approach to the Hydrodynamics of Complex Fluids*, Adv. Polymer Sci. **221**, 1 (2008). *[GIKW08]*

An overview of chemically powered synthetic nanomotors has been published by Kapral *[Kap13]*.

There is no literature on the practical conduct of nanomotor simulations, however.

7.4 The MPCD fluid and Molecular Dynamics

A MPCD fluid consists of point particles with a mass (set to unity here for convenience), a position x and a velocity v. The particles evolve in two step: (i) indepedent streaming of the particles for a duration τ and (ii) cell-wise collision of the particles' velocities.

For particle i this results in the following equations:

$$x_i' = x_i + v_i\tau \tag{7.1}$$

and

$$v_i' = v_\xi + \omega_\xi(v_i - v_\xi) \tag{7.2}$$

where the prime denotes the quantities after the corresponding step, ξ is a cell, ω_ξ is a rotation operator and v_ξ is the center-of-mass velocity in the cell. The cell consists in a regular lattice of cubic cells in space. Equations (7.1) and (7.2) conserve mass, energy and linear momentum.

The viscosity for a MPCD fluid can be computed from its microscopic properties:

$$\eta = \frac{k_B T\tau\rho}{2m}\left(\frac{5\gamma - (\gamma - 1 + e^{-\gamma})(2 - \cos\alpha - \cos 2\alpha)}{(\gamma - 1 + e^{-\gamma})(2 - \cos\alpha - \cos 2\alpha)}\right) + \frac{m}{18a\tau}(\gamma - 1 + e^{-\gamma})(1 - \cos\alpha)$$

One can embed a body in a MPCD fluid by using a explicit potential energy. Then, the streaming step is replaced by the velocity-Verlet integration scheme. Collision involve only fluid particles and not the colloid.

7.5 The dimer nanomotor

7.5.1 Physical setup

In this section, we review the propulsion of the dimer nanomotor presented by Rückner and Kapral. The geometry of the motor and the chemical kinetics are presented in Fig. 7.1.

The solvent consists of particles of types A and B, initially all particles are set to A (the fuel). Fuel particles that enter the interaction range of the catalytic sphere are flagged for reaction but the actual change of A to B only occurs when the solvent particle is outside of any interaction range. Else, the change would generate a discontinuous jump the in the potential energy and disrupt the trajectory. This chemical activity generates an excess of product particles "B" around the catalytic sphere and a gradient of solvent concentration is established.

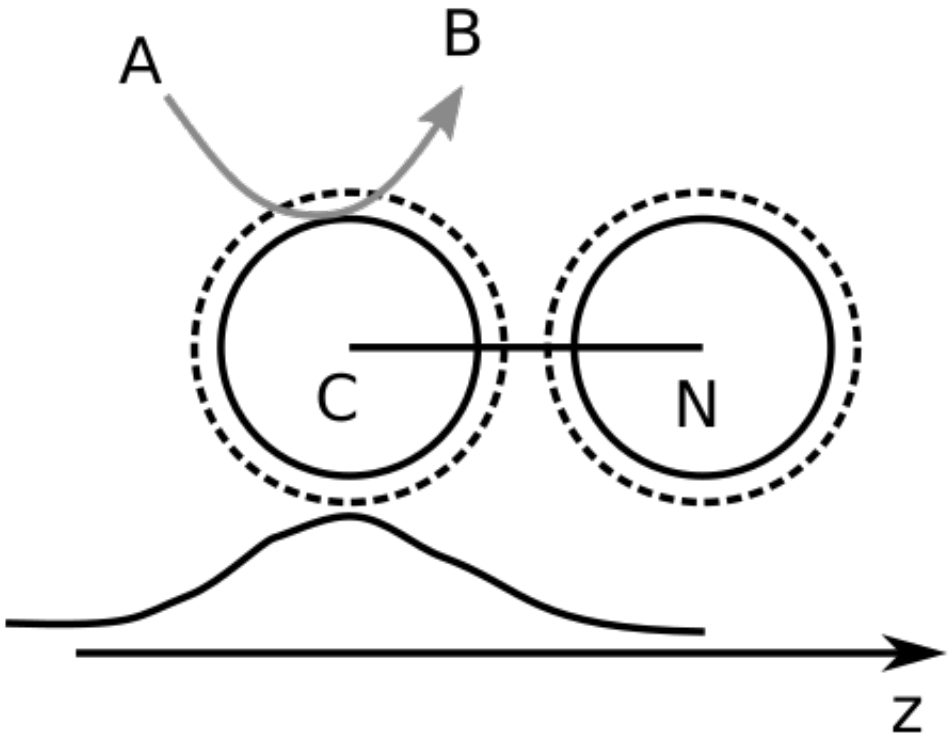

Fig. 7.1: Geometry and chemistry for the dimer nanomotor. The graph sketched below represents the local excess of "B" particles that is asymmetric for the "N" sphere. Many more "A" and "B" particles not shown.

In this type of simulation, the total energy is conserved but the system is maintained in nonequilibrium by *refueling*, that is by changing B particles to species A when they are far enough from the colloid.

The solvent and colloids interact via a purely repulsive Lennard-Jones potential of the form

$$V(r) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} - 1 \right)$$

where ϵ and σ can be different depending on the combination of solvent and colloid species.

7.5.2 Simulation setup

Within RMPCDMD, the simulation program for the dimer is called `single_dimer_pbc`. This program requires a configuration file that contains the physical parameters, an example of which is given in the listing below.

```
# physical parameters
T = .16666666
L = 32 32 32
rho = 9
tau = 1.0
probability = 1.0
bulk_rmpcd = F
bulk_rate = 0.001

# simulation parameters
N_MD = 200
N_loop = 50
equilibration_loops = 50
colloid_sampling = 50

# interaction parameters
sigma_N = 4.0
sigma_C = 2.0

d = 6.8
epsilon_N = 1.0 0.1
epsilon_C = 1.0 1.0

epsilon_N_N = 1.0
epsilon_N_C = 1.0
epsilon_C_C = 1.0
```

The configuration allows one to set the size of both spheres in the dimer as well as the interaction parameters. The setting `epsilon_N` contain the prefactor to the Lennard-Jones potential for the "N" sphere and all solvent species on a single line. In this example, all the interaction parameters are set to 1 except for the interaction between the "N" sphere and "B" solvent particles, as was done in *[RK07]*.

7.5.3 Running the simulations

Note: Make sure that you have built the code properly (see *Install*) and that the command-line tool `rmpcdmd` is available at your command-line prompt. You will also need a working scientific Python environment (see *Python*).

An example simulation setup is provided in the directory `experiments` of RMPCDMD. There, the sub-directory `01-single-dimer` contains a parameter file.

Review the parameters in the file `dimer.parameters` then execute the code

```
make dimer.h5
```

The actual commands that are executed will be shown in the terminal.

7.5.4 Analyzing the data

The output of the simulation is stored in the file `dimer.h5`, that follows the H5MD convention for storing molecular data *[dBCH14]*. H5MD files are regular HDF5 files and can be inspected using the programs distributed by the HDF Group. Issue the following command and observe the output:

```
h5ls dimer.h5
```

HDF5 files have an internal directory-like structure. In `dimer.h5` you should find

```
fields                   Group
h5md                     Group
observables              Group
parameters               Group
particles                Group
timers                   Group
```

The elements are called "groups" in HDF5 terminology. Here, there is data about the particles (positions, velocities, etc), observables (e.g. temperature) and fields (here, the histogram of "B" particles). The `h5md` group contains metadata (simulation creator, H5MD version, etc.), the `timers` group contains timing data that is collected during the simulation and `parameters` contains all the parameters with which the simulation was run.

The command

```
h5ls -r dimer.h5
```

will visit all groups recursively. The output is then rather large. Let us focus first on the velocity of the dimer, it is located at `/particles/dimer/velocity`, where it is stored in `value` and the time step information of the dataset is stored in `step` and `time`. In the present case, the velocity is sampled at regular time interval of 100 timesteps or equivalently 1 in units of τ.

All the data analysis in this tutorial is done using the Python language and a set of libraries: NumPy for storing and computing with array data, h5py for reading HDF5 files, matplotlib for plotting and SciPy for some numerical routines. For installation, see appendix [install-py]. Some generic programs are provided with as an introduction to reading the files, such as `h5md_plot.py`. Its usage is

```
rmpcdmd plot dimer.h5 --obs temperature
```

(the `obs` option is preceded by two dashes) to display the temperature in the course of time. This program can also display the trajectory of the dimer

```
rmpcdmd plot dimer.h5 --traj dimer/position
```

More specific information on the dimer nanomotor can be obtained via Python programs located in the `experiments/01-single-dimer` directory.

The directed velocity, that is the velocity in the direction of the motor's propulsion axis, can be obtained via

```
python plot_velocity.py dimer.h5 --directed
```

A further option `--histogram` show an histogram instead of the time-dependent value.

Finally, an important quantity to assess both for theory and experiments is the effective diffusion that results from both thermal fluctuations and the combination of self-propelled motion and random reorientation.

```
python plot_msd.py dimer.h5
```

This latter program can take several simulation files as input to obtain better statistics. It is also important to use a much simulation time (`N_loop`) than the default one to produce meaningful results.

7.6 The Janus nanomotor

A Janus motor comprises two hemispheres with one chemically active surface and one inactive surface as shown in Fig. 7.2 (a). Because chemical reactions happen asymmetrically on the Janus motor surface, a concentration gradient of product particles is generated giving rise to self-propulsion. In 2013, de Buyl and Kapral introduced a composite model for Janus motor *[dBK13]*, see Fig. 7.2 (b). The active (blue, C) and inactive (red, N) parts are composed of spheres linked by rigid bonds. These spheres have the same radius, and interact with the surrounding solvent particles through repulsive Lennard-Jones potentials $V_{\alpha C}$ and $V_{\alpha N}$, where $\alpha = A, B$ is the type of solvent species.

An example simulation setup for self-proplusive Janus motor is provided in the directory `experiments/03-single-janus`. The parameter file is show below:

```
# physical parameters
T = .333333333
L = 32 32 32
rho = 9
tau = 1.0
probability = 1

# simulation parameters
N_MD = 100
N_loop = 512
colloid_sampling = 50
equilibration_loops = 20
data_filename = janus_structure.h5
link_treshold = 2.5
do_read_links = F
polar_r_max = 10
bulk_rate = 0.1

# interaction parameters
sigma_colloid = 2
epsilon_colloid = 2
do_lennard_jones = T
do_elastic = F
do_rattle = T
rattle_pos_tolerance = 1d-8
rattle_vel_tolerance = 1d-8

sigma = 3
epsilon_N = 1.0 5.0
epsilon_C = 1.0 1.0
```

To run the simulation, use `make simulation`, and check the propulsion speed V_z with

```
python plot_velocity.py janus.h5 --directed
```

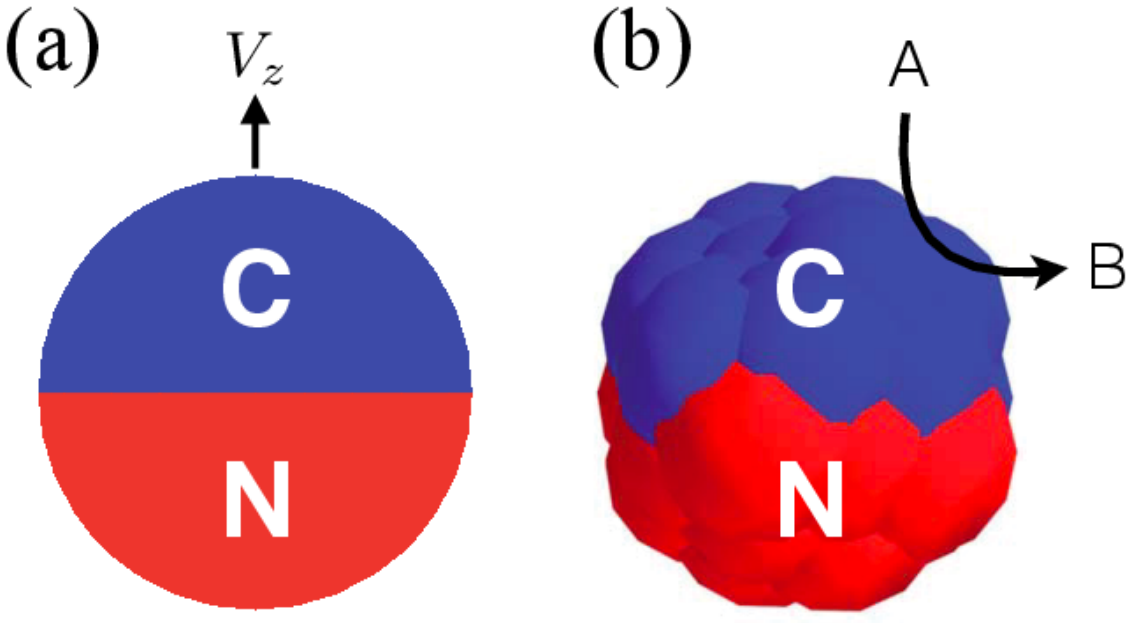

Fig. 7.2: (a) The sketch of a Janus particle with active (C) and inactive (N) hemispherical surface moving along particle axis with velocity V_z. (b) The composite model for Janus particle. Chemical reactions, $A \rightarrow B$, take place at the active surface.

7.6.1 Controls of motor speed

For Janus particles, one can obtain from phoretic theory that the propulsion speed is determined by factors such as the system temperature, fluid properties (viscosity and density) but also the chemical kinetics and the specific solvent-colloid interactions *[Kap13]*.

In this section, we explore the effects of the interaction parameters and of the reaction rate k_2 on the direction and strength of the propulsion. By modifying only the lines below in the parameter file for the janus simulation, it is possible to observe a reversal of propulsion and the effect of the reaction rate k_2.

Example 1, forward moving Janus motor.

```
epsilon_N = 1.0 0.5
epsilon_C = 1.0 0.5
bulk_rate = 0.001
```

Example 2, backward moving Janus motor.

```
epsilon_N = 0.5 1.0
epsilon_C = 0.5 1.0
bulk_rate = 0.001
```

Example 3, changing the bulk reaction rate.

```
epsilon_N = 1.0 0.5
epsilon_C = 1.0 0.5
bulk_rate = 0.0001
```

CHAPTER

EIGHT

BIBLIOGRAPHY

[AG02] E Allahyarov and G Gompper. Mesoscopic solvent simulations: multiparticle-collision dynamics of three-dimensional flows. *Phys. Rev. E*, 66:36702, 2002. doi:10.1103/PhysRevE.66.036702.

[AT87] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Clarendon Press, Oxford, 1987.

[Col15] Peter H. Colberg. Nano-dimer. 2013-2015. URL: https://colberg.org/nano-dimer/.

[dBCH14] Pierre de Buyl, Peter H. Colberg, and Felix Höfling. H5MD: a structured, efficient, and portable file format for molecular data. *Comp. Phys. Commun.*, 185:1546–1553, 2014. doi:10.1016/j.cpc.2014.01.018.

[dBHD17] Pierre de Buyl, Mu-Jie Huang, and Laurens Deprez. RMPCDMD: Simulations of colloids with coarse-grained hydrodynamics, chemical reactions and external fields. *Journal of Open Research Software*, 5:3, 2017. doi:10.5334/jors.142.

[dBK13] Pierre de Buyl and Raymond Kapral. Phoretic self-propulsion: a mesoscopic description of reaction dynamics that powers motion. *Nanoscale*, 5:1337–1344, 2013. doi:10.1039/c2nr33711h.

[GIKW08] G. Gompper, T. Ihle, D. M. Kroll, and R. G. Winkler. Multi-particle collision dynamics: a particle-based mesoscale simulation approach to the hydrodynamics of complex fluids. *Advances in Polymer Science*, 221:1–87, Jan 2008. doi:10.1007/978-3-540-87706-6_1.

[Ham06] C. H. Hamilton. Compact hilbert indices. Technical Report CS-2006-07, Dalhousie University, 2006. URL: https://www.cs.dal.ca/research/techreports/cs-2006-07.

[Kap08] Raymond Kapral. Multiparticle collision dynamics: simulation of complex systems on mesoscales. *Adv. Chem. Phys.*, 140:89, 2008. doi:10.1002/9780470371572.ch2.

[Kap13] Raymond Kapral. Perspective: nanomotors without moving parts that propel themselves in solution. *J. Chem. Phys.*, 138(2):020901, 2013. doi:10.1063/1.4773981.

[LGIK01] A. Lamura, G. Gompper, T. Ihle, and D. M. Kroll. Multi-particle collision dynamics: flow around a circular and a square cylinder. *EPL (Europhysics Letters)*, 56:319, 2001. doi:10.1209/epl/i2001-00522-9.

[MK99] Anatoly Malevanets and Raymond Kapral. Mesoscopic model for solvent dynamics. *J. Chem. Phys.*, 110:8605, 1999. doi:10.1063/1.478857.

[MK00] Anatoly Malevanets and Raymond Kapral. Solute molecular dynamics in a mesoscale solvent. *J. Chem. Phys.*, 112:7260, 2000. doi:10.1063/1.481289.

[RK10] Dmitri Rozmanov and Peter G. Kusalik. Robust rotational-velocity-verlet integration methods. *Phys. Rev. E*, 81:056706, 2010. doi:10.1103/PhysRevE.81.056706.

[RK07] Gunnar Rückner and Raymond Kapral. Chemically powered nanodimers. *Phys. Rev. Lett.*, 98:150603, Apr 2007. doi:10.1103/PhysRevLett.98.150603.

[SMDS11] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Com-*

puting, Networking, Storage and Analysis, SC '11, 16:1–16:12. New York, NY, USA, 2011. ACM. doi:10.1145/2063384.2063405.

[WL10] Jonathan K Whitmer and Erik Luijten. Fluid–solid boundary conditions for multiparticle collision dynamics. *J. Phys. Condens. Matt.*, 22(10):104106, 2010. doi:10.1088/0953-8984/22/10/104106.