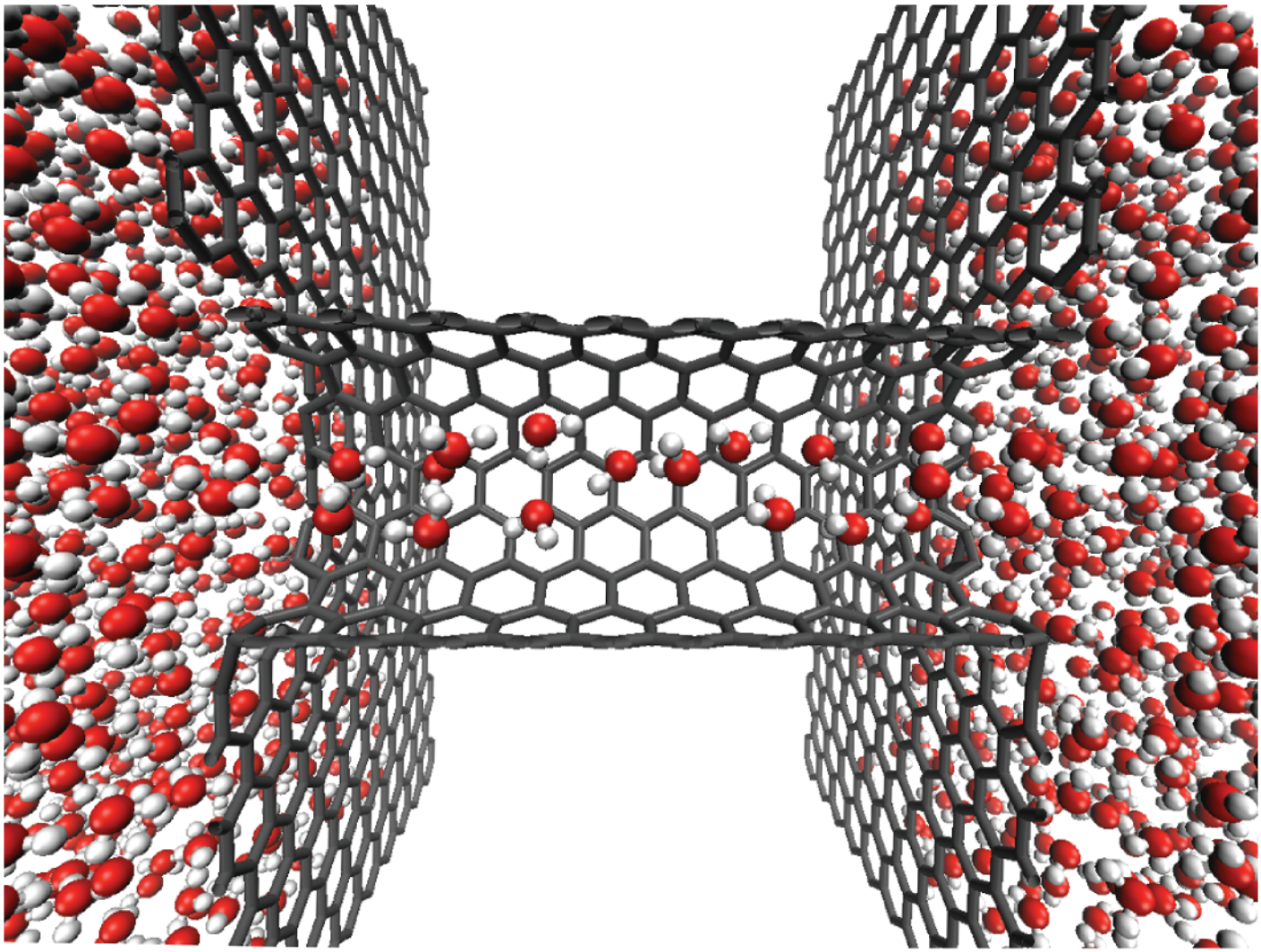

mdFOAM

Molecular Dynamics in OpenFOAM:
A user-guide for beginners

www.micronanoflows.ac.uk

Molecular Dynamics in OpenFOAM: A user-guide for beginners

Updated version on: April 27, 2016

Contents

1 Introduction to MD

There are many fluid dynamics problems where a computational fluid dynamics approach is invalid. Some examples include contact line dynamics, unknown fluid constitutive relations in rheological fluids, rarefied gas flows in micro channels, flow discontinuities and non-continuum flows in nanopores. Molecular dynamics (MD) is an accurate method for simulating these types and many other problems. MD is a discrete, deterministic method that simulates the temporal and spatial dynamics of matter (solid, liquid, gases) explicitly modelled by atoms or molecules. Molecules interact together through "*potentials*", which is turned into a force term thus determines the acceleration of that molecule. A potential can sometimes be thought of as being an imaginary "spring" connected between the midpoints of a pair of molecules. As the two molecules approach each other, the spring becomes stiffer (caused by the Pauli repulsion of electron clouds from both molecules interacting), and the molecules experience a pair-wise repulsive force. On the other hand when they move apart, the spring stretches so the molecules experience an inward attractive force. In a system of N molecules, each molecule interacts with all other remaining molecules in the system through $N-1$ springs.

The motion of all molecules in time and space is enabled using Newton's equations of motion:

$$\frac{d}{dt}\mathbf{r}_i(t) = \mathbf{v}_i(t), \tag{1}$$

$$m_i \frac{d}{dt}\mathbf{v}_i(t) = \mathbf{f}_i(t), \tag{2}$$

where m_i is the i^{th} molecule's mass, $\mathbf{v}_i$ is the "laboratory" velocity and $\mathbf{f}_i$ is the total force. The rationale of MD is the numerical implementation of equations (1) and (2) to get the discretisation of the molecules' trajectories $\mathbf{r}_i(t)$, velocities and forces in time. Time is discretised by a time-step Δt_m, typically around 5 fs for monatomics and 1 fs for water. The choice of the time-step is crucial in an MD simulation for two properties: (a) accuracy and (b) numerical stability.

1.1 Quick algorithm overview

Consider a cubic box of side length L filled with N molecules. Almost always MD requires an "initial condition" (at time $t = 0$) that essentially consists of specifying all positions $\mathbf{r}_i$, masses m_i, velocities $\mathbf{v}_i$ and accelerations $\mathbf{a}_i$ (and other properties), where i is an index running over all molecules in the system, $i = \{1, 2, \ldots, N\}$. The most common boundary conditions used in the literature of MD are called the periodic boundary conditions. Periodicity is an important feature in MD, since it allows modelling an infinite system using a finite (but smaller) number of molecules, while maintaining conservation of mass, momentum and energy at the boundaries.

One common technique for discretising equations (1) and (2) is using the Velocity Verlet algorithm, which proceeds as follows for one MD time-step $t \to t + \Delta t_m$ (refer to Figure 1):

Step 1 Update half the velocity of all N molecules:

$$\mathbf{v}_i(t + \Delta t_m/2) = \mathbf{v}_i(t) + 1/2\mathbf{a}_i(t)\Delta t_m. \tag{3}$$

Step 2 Move molecules to their new position:

$$\mathbf{r}_i(t + \Delta t_m) = \mathbf{r}_i(t) + \mathbf{v}_i(t + \Delta t_m/2)\Delta t_m. \tag{4}$$

Note if any molecule hits the periodic boundary it gets wrapped around to the other coupled boundary.

Step 3 Create periodic images of molecules external to the main cell.

Step 4 Compute the intermolecular forces of all molecules:

$$\mathbf{f}_i = \sum_{j=1(\neq i)}^{N} -\nabla U(r_{ij}), \tag{5}$$

where $-\nabla U(r_{ij}) = \mathbf{f}_{ij}$ is the intermolecular force potential between molecules i and j within the system of N molecules.

Step 5 Compute the new acceleration of molecules:

$$\mathbf{a}_i(t + \Delta t_m) = \mathbf{f}_i(t + \Delta t_m)/m_i. \tag{6}$$

Step 6 Update the second half of the molecules' velocity (due to the new acceleration),

$$\mathbf{v}_i(t + \Delta t_m) = \mathbf{v}_i(t + \Delta t_m/2) + 1/2\mathbf{a}_i(t + \Delta t_m)\Delta t_m. \tag{7}$$

Step 7 Repeat from **Step 1** until end of simulation.

1.2 Potentials

The choice of intermolecular potentials is key to the results obtained in any molecular dynamics simulation. The general potential energy equation between a set of N interacting particles can be expressed by external fields, pair-interactions and higher-order interactions (multi-body potentials). In some simulations, it is sufficient to use only pair-wise interactions (two-body) to capture the essential physics. The most commonly-used potential to describe van der Waals interactions between non-bonded molecules is the Lennard-Jones (LJ) 12-6 potential:

$$U_{12-6}(r_{ij}) = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right], \tag{8}$$

where $r_{ij} = |\mathbf{r}_{ij}|$ is the separation distance between two molecules (i, j), σ is the hard-sphere diameter and corresponds to where the potential energy is zero, and ϵ is the energy well-depth. The potential is repulsive at small molecular separations and is attractive at long separations; potential energy is zero for infinite separation lengths. For argon, the following potential energy and length scale characertistics can be used: $\epsilon = 1.6568 \times 10^{-21}$J and $\sigma = 0.34 \times 10^{-9}$m. The graph of this function is seen in Fig. 2. The corresponding force potential is obtained by differentiating the potential function with respect to the separation r_{ij}, see Eqn. (5). The advantage of the Lennard-Jones potential is that it combines a realistic description of a pair-wise intermolecular interaction with computational simplicity. The potential is typically truncated at a separation distance r_{cut} so to improve the computational speed of the MD simulation. Each molecule does not need to interact with all N molecules in the system, but only with those molecules that are located within a sphere of radius r_{cut}. Furthermore the potential is sometimes shifted so that the force and potential energy is zero at $r_{ij} = r_{cut}$.

1.3 Limitations

1. **Computational cost** - The major drawback of MD is that the intermolecular force computation (step 4 above) is very costly, making MD only applicable to simulate cases of very small lengthscales and timescales; typically of nanoscale dimensions. The computational cost is caused by many nested loops within the algorithm of total $N(N-1)/2$ pair-force calculations per time-step over many millions of time-steps ($\tau = t/\Delta t_m$). Although it is now common to reduce the number of pair-force computations per time-step (e.g. to $N\log(N)$), by considering a cut-off in the potential as well as avoiding double counting procedures, the method is overall still considerably computationally inefficient.
   Tip: Making your simulation domain as small as possible, can save you time in the long run.

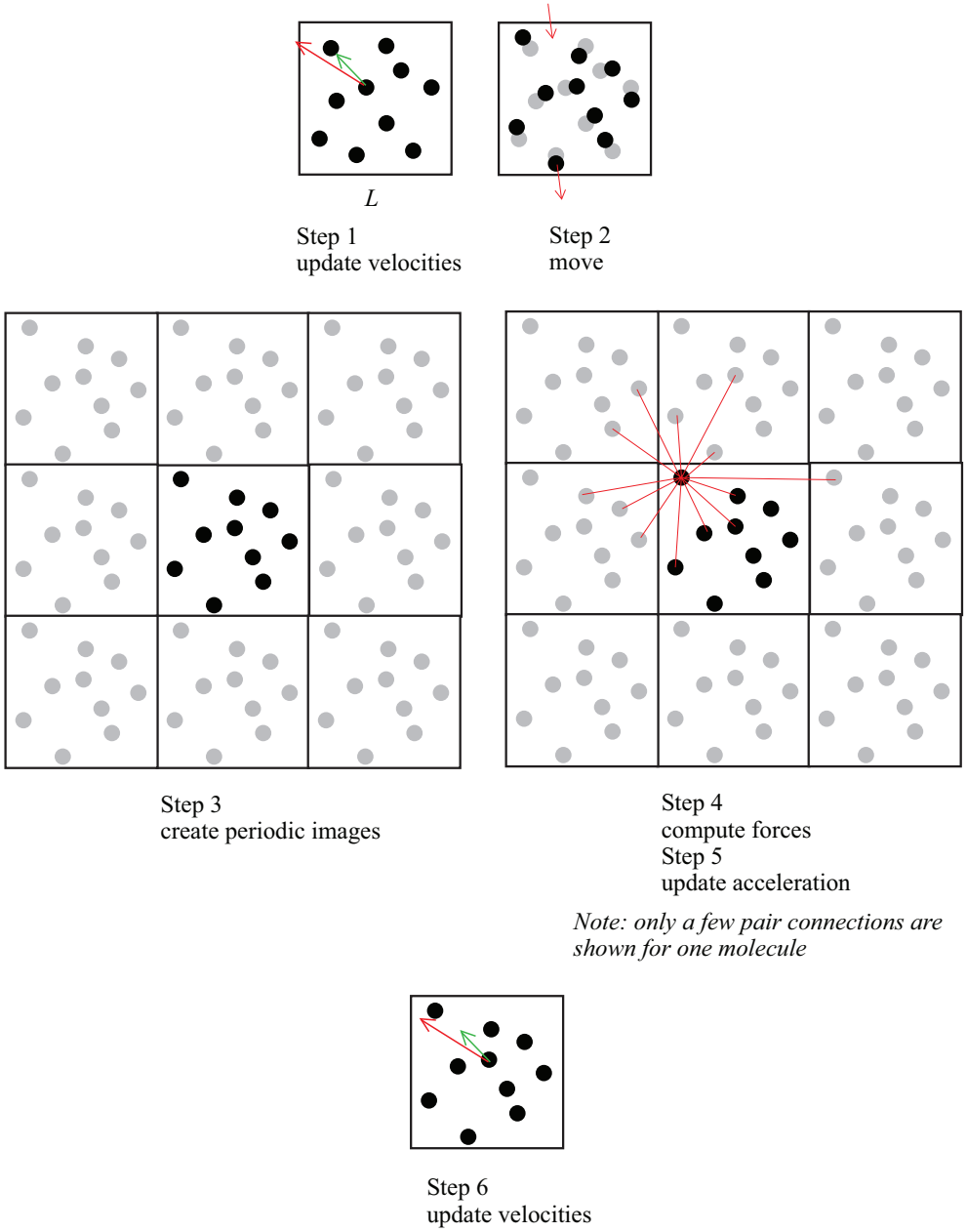

Figure 1: Step-by-step illustration of the basic MD time-integration scheme with periodic boundary conditions applied in all three cartesian co-ordinates.

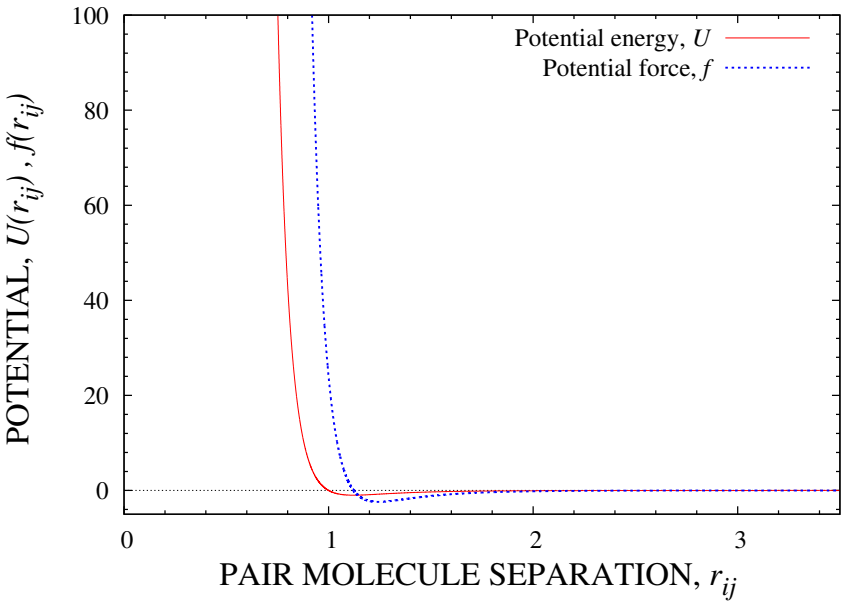

Figure 2: Lennard Jones truncated and shifted potential.

2. **Choice of Potential** - "You can get an MD simulation to give you results you are looking for simply by modifying the potentials." This can be a good thing or a bad thing. The choice of potentials is perhaps one of the most crucial aspects for accurate MD simulations. You will need to look well in the literature for the potentials of molecules you would like to simulate, use mixing rules for missing potentials or even perform your own calibration tests with experiments.

1.4 Reducing computational cost of MD

1. **Potential cut-off** - Some potentials have repulsive or attractive tails that tend to zero at large pair-molecule separations. Using a "cut-off" is common in MD that aims to minimise pair-force calculations only 'locally' (i.e. in the neighbourhood of a given molecule), instead of being a calculation over the entire system of molecules. What is required to enable this cut-off is the implementation of a neighbour list algorithm or a cell-list algorithm [1]. The latter is currently implemented in mdFOAM which requires the mesh to contain cells each of size r_{cut}. Thus all molecules in a given cell, will only need to interact with molecules within that cell, but also with all touching cells (in 3D, this amounts to 26 neighbour cells).

2. **Prevent double-counting** - Another way of halving the MD algorithm's computational cost is by realising that pair-potential calculations are symmetric. This means that the pair force between molecule i and j is equal to that between j and i, so a pair is sufficient to be calculated only once.

3. **Parallel-processing** - A mandatory approach to making simulations faster is by ensuring that the entire algorithm runs in parallel on many processors. In this code we use parallel domain decomposition.

4. **GPU processing** - GPU's can offer a 10-20 times algorithm speed-up over serial CPUs, but there may be limitations.

1.5 Stability of MD algorithm

MD simulation runs are generally stable provided a few rules are followed. MD will become unstable (generally referred to as 'blow up') as a result of a molecule moving into the deeply repulsive core of one or more molecules, leading to a cascading process of overlapping molecules. *Note:* you will notice that the simulation blows-up when the total energy of the system (typical observed in the output log of a MD simulation) become very large (close to infinity) over a few MD time-steps, after which the simulation just stops. The reason why blow-up happens can be perceived from the typical shape of the potential which is used: the potential becomes extremely repulsive over very short pair-molecule separations. This means a very slight adjustment of one molecule into the core of another molecule gives a repulsive force of magnitude which is very large (close to infinity), giving both molecules a huge acceleration. In the next time-step they move into two other molecule cores. The process repeats until the simulation blows up. If this scenario occurs, you will need to stop the simulation, understand what has caused that blow-up (perhaps you've initialised molecules too close to each other) and circumvent it. Blow-up can almost always be avoided provided the correct choice of time-step is made. The lower the time-step the lower the chance of blow up, of course at the expense of computational inefficiency. An optimum simulation is one when the correct time-step is chosen for the physics being modelled, see Fig 3. For example when a very large external force is applied to a system (e.g. to impose high strain rates), the time-step has to be lowered to accommodate the high molecular velocities. Another example is that a monatomic fluid with 3 degrees of freedom requires a less stringent time-step than a water molecule with 6 degrees of freedom.

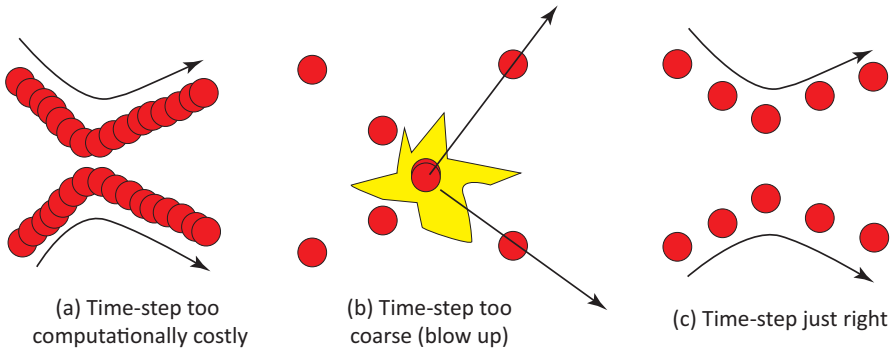

Figure 3: Figure showing the importance of selecting the correct time-step: (a) time-step too small, makes the dynamics of molecules slow and computationally inefficient, (b) time-step too large, high probability of molecules overlapping leading to simulation blow-up, and (c) correct choice of time-step.

1.6 Reduced Quantities

It is common practice in MD to convert all quantities into "reduced quantities" or "reduced units" (RU), mostly so that computers do not perform calculations with extremely small numbers. It also means the calculations and memory storage can sometimes be made cheaper by using variables with smaller bit sizes. As an example let's take the mass of one water molecule $m = 2.9 \times 10^{-26}$ kg. If we divide (or normalise) this value by a "reference" mass which we pick (arbitrarily) as $m_r = 3 \times 10^{-26}$ kg then this gives a value close to 1, more precisely 0.966 (RU).

In all our simulations we use only four fundamental normalisation/reference parameters for length l_r, mass m_r, time t_r and charge q_r. All other properties can be derived from these fundamental parameters. Here the subscript r indicates it is a normalisation/reference parameter.

Table 1 shows some examples for the four fundamental reference parameters we normally pick for water, and how other properties are derived from these. In the literature you will sometimes see that a reduced quantity is labelled by an asterisk (*). We'll use this notation here too. Switching between reduced units and SI units is an important task you will be expected to master. Divide an SI quantity by the reference property to get it into RU, or multiply an RU quantity by the reference property to obtain SI.

Table 1: Table of reduced units for properties of our water simulations. A property q in SI units is converted into a reduced unit q^* by its reference property q_r, given in the last column of the table.

Fundamental quantities		
length	$l^* = l/l_r$	$l_r = 3.154 \times 10^{-10}$ m
time	$t^* = t/t_r$	$t_r = 1.66 \times 10^{-12}$ s
mass	$m^* = m/m_r$	$m_r = 2.987 \times 10^{-26}$ kg
charge	$q^* = q/q_r$	$q_r = 1.602176487 \times 10^{-19}$ C
Derived quantities		
energy	$\epsilon^* = \epsilon/\epsilon_r$	$\epsilon_r = (m_r l_r^2/t_r^2) = 1.08 \times 10^{-21}$ J
force	$f^* = f/f_r$	$f_r = (\epsilon_r/l_r) = 3.42 \times 10^{-12}$ N
velocity	$u^* = u/u_r$	$u_r = (\sqrt{\epsilon_r/m_r}) = 190$ m s^{-1}
mass density	$\rho^* = \rho/\rho_r$	$\rho_r = (m_r/l_r^3) = 952.03$ kg/m^3
temperature	$T^* = T/T_r$	$T_r = (\epsilon_r/k_b) = 78.1$ K
pressure	$p^* = p/p_r$	$p_r = (\epsilon_r/l_r^3) = 34.368 \times 10^6$ N m^{-2}

1.7 Water simulations

In this document we will focus on simulating water, but other monatomic/polyatomic fluids can be modelled, provided the bonds between the intramolecular sites are rigid. The number of sites depends on the water model which is used (3, 4 and 5 sites have been used, see Ref. [2]). In this tutorial we use the TIP4P/2005 model [3] as shown in Figure 4, which has 4 sites: one oxygen atom (O), two positively charged hydrogen sites (H), and a negatively charged fictitious site (M). The consequence of a 4 sited molecule over a monatomic 1 site model is that there are more interactions per pair of molecules. To be exact between a pair of water molecules there is one LJ potential between oxygen sites and nine electrostatic potentials (combination of H's and M's) as shown in Fig. 5. In this work we will assume that all charged sites (H,M) adopt the Coulomb potential:

$$U_C(r_{ij}) = \frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}}, \tag{9}$$

where ϵ_0 is the electric constant and q_i and q_j are the charges of sites i and j across a pair of molecules (see Fig. 6). The Coulomb potential is truncated (and adjusted) at separations of approximately 1 nm for computational speed up:

$$U'_C(r_{ij}) = \frac{1}{4\pi\epsilon_0}\left(\frac{q_i q_j}{r_{ij}} - E_s(r_{ij})\right), \tag{10}$$

where $E_s(r_{ij})$ is a smoothing function of the form:

$$E_s(r_{ij}) = \frac{q_i q_j}{r_{cut}} - (r_{ij} - r_{cut})\frac{q_i q_j}{r_{cut}^2}. \tag{11}$$

The tail of the Coulomb potential is very long, because it decays slowly proportional to r^{-1}. There are many ways of modelling long range interactions (e.g. PPPM, PME), but we will not discuss these. A summary of all the properties of the TIP4P/2005 model can be found in Table 2.

An important issue with water is that the Coulomb potential is repulsive at all pair-site separations unlike the LJ potential which is positive and converges to zero quicker at radial separation distances larger than 1σ, see Fig. 6. Furthermore water must use a shorter time-step in order to prevent blow-up, due to the additional rotational degrees of freedom. This means a further increase of the computational effort of a water MD simulation. E.g. a time-step of Δt_m of 1 - 2 fs is common.

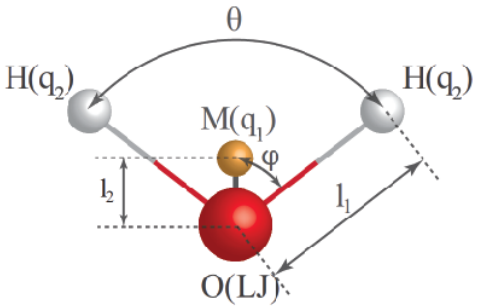

Figure 4: Schematic of a water molecule (Figure taken from William Nicholls' thesis, 2012). See Table 2 for values of different properties.

Table 2: TIP4P/2005 water model properties.

property	description	value	units
m_H	mass of hydrogen atom	1.67353255×10^{-27}	kg
m_O	mass of oxygen atom	2.6560176×10^{-26}	kg
m_M	mass of M site	0	kg
q_H	charge of hydrogen atom	$8.91450997367\times10^{-20}$	C
q_O	charge of oxygen atom	0	C
q_M	charge of M site	$-1.782901995\times10^{-19}$	C
θ	bond angle	104.52	°
ϕ	M angle	52.26	°
l_1	H-O bond length	0.9572	Å
l_2	O-M length	0.1546	Å
σ_{O-O}	O-O LJ length	0.31589×10^{9}	m
ϵ_{O-O}	O-O LJ energy	$1.286751503\times10^{-21}$	J

1.8 OpenFOAM

OpenFOAM is an acronym for "Open-source Field Operation and Manipulation". It consists of C++ libraries primarily used (but definitely not restricted) for computational fluid dynamics. "Open-source", means it is free to be downloaded, distributed and altered to suit the users, while "field operation" and "field manipulation" is the core of a very flexible toolbox for solving

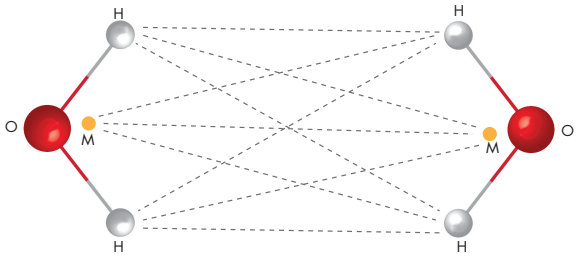

Figure 5: Nine electrostatic pair-interactions (Coulomb potential) between two water molecules using the TIP4P/2005 water model. There is one van der Waals pair-interaction (LJ potential) between oxygen *(red)* sites, not marked in the figure.

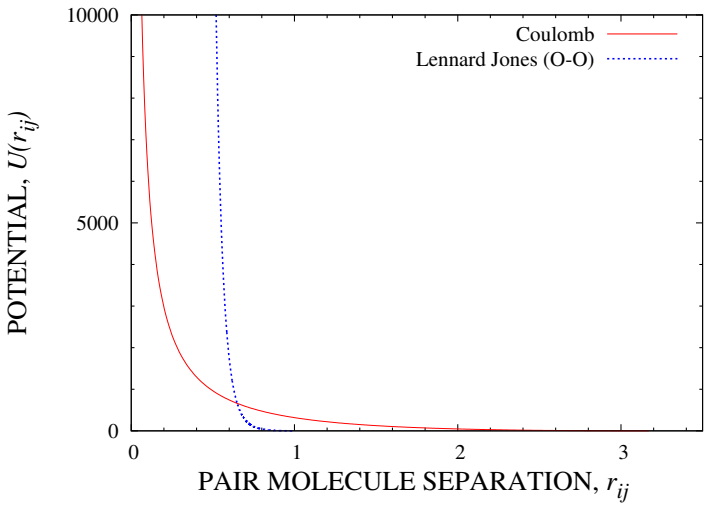

Figure 6: Coulomb potential to model electrostatics, and LJ potential to model van der Waals.

continuum formulations, where the description is described mainly by fields of partial differential equations.

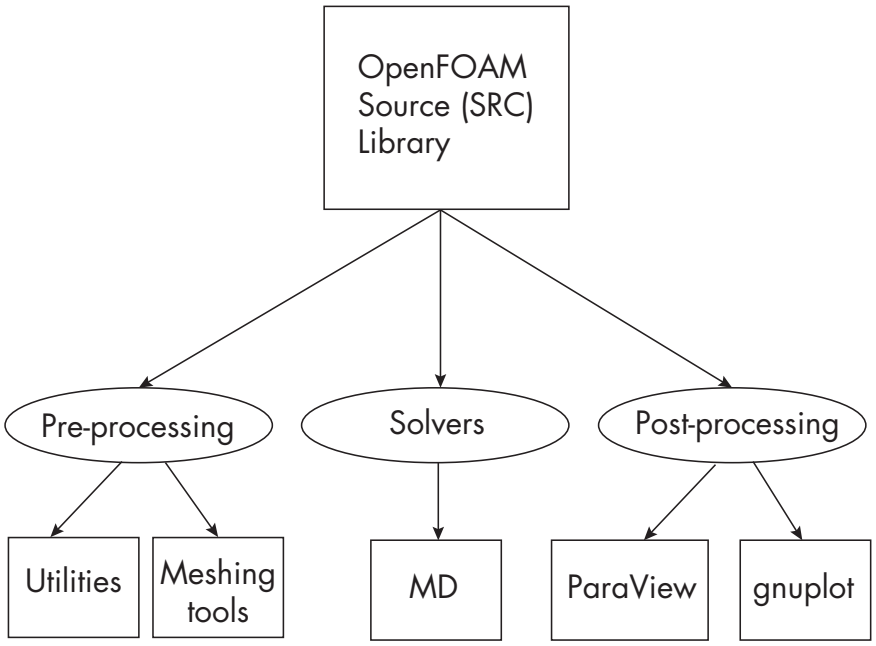

Figure 7: OpenFOAM structure.

OpenFOAM essentially is divided into two parts. First there is the *source* code, where the libraries (classes, templates etc) are written in C++ and stored. Note this is where most of the intensive coding happens. Then there are the executables, known as *applications* that are generally very short code in the form of a main function that handle creating and using objects from the stored classes in source. The applications fall into two categories: *solvers*, that are each designed to solve a specific problem in continuum mechanics; and *utilities*, that are designed to perform tasks that involve data manipulation (e.g. pre- and post-processing).

The OpenFOAM distribution contains numerous solvers and utilities covering a wide range of problems. In this document we are describing an enhanced MD solver which is a direct continuation of the OpenFOAM MD solver (mdFoam) and provides additional functionality compared to the standard application. We will also describe the important utilities for use with the MD solver such as pre- and post-processing utilities. Example of pre-processing are meshing and creating a lattice of molecules. An example of post-processing is visualisation of molecules in 3D (e.g. using ParaView or VMD). The structure of OpenFOAM is shown in Figure 7.

2 Installation and keeping up-to-date

The software base currently uses OpenFOAM version 2.4.0 (http://www.openfoam.org/), which has been extended to include the most up-to-date version of our molecular dynamics code. Our combined OpenFOAM-MD code is freely available to download from https://github.com/ where the name of the repository is OpenFOAM-2.4.0-MNF. If installing this software to your workstation, we recommend using the latest Long Term Service release of the Ubuntu distribution and all following instructions assume this environment. However installation within other environments that support the basic OpenFOAM 2.4.0 package should be possible as the MNF code adds no additional dependencies. To install OpenFOAM along with the MNF MD code base:

1. Subscribe for an account on https://github.com.
2. If you want to become a collaborator, please send your GitHub account name to the administrators of the MicroNanoFlows (MNF) repository. Please specify if you just want to use OpenFOAM (use of Master account), or if you want your own development branch. For the latter please also provide a brief description of the nature of the contribution you would like to make.
3. Prerequisites from OpenFOAM website (see git installation on http://www.openfoam.org for more information):
   - Install Git software packages: `apt-get install git-core`
   - Install the following packages: `apt-get install build-essential flex bison cmake zlib1g-dev libopenmpi-dev openmpi-bin gnuplot libreadline-dev libncurses-dev libxt-dev`
   - Install the following packages: `apt-get install qt4-dev-tools libqt4-dev libqt4-opengl-dev freeglut3-dev libqtwebkit-dev`
   - Install the OpenMPI, Scotch and CGAL: `apt-get install libscotch-dev libcgal-dev`
4. Change directory to home directory.
5. Create a new base OpenFOAM folder and switch to it: `mkdir OpenFOAM; cd OpenFOAM`
6. Create a clone of the OpenFOAM MNF repository:
   `git clone https://github.com/MicroNanoFlows/OpenFOAM-2.4.0-MNF.git`
7. Download 2.4.0 ThirdParty package (http://www.openfoam.org/archive/2.4.0/download/source.php) and extract to the OpenFOAM folder `tar xzf ThirdParty-2.4.0.tgz`
8. Rename ThirdParty directory to `ThirdParty-2.4.0-MNF`
9. Download an appropriate text editor if one is not already installed (i.e. nano, kate, kwrite, vim, emacs etc.) and if you are intending to contribute to the project then modify settings to apply 4 spaces for each tab if possible. This ensures consistency between contributions.
10. Detailed instructions for how to build and compile the repository on a typical workstation can then be found within the OpenFOAM-2.4.0-MNF documentation folder in the file `README-MNF`

2.1 Separate MD code compilation

While the `./Allwmake` in the highest level of OpenFOAM is used to install the entire OpenFOAM source library and applications, you may require to do some local compiling of just the MD source code and MD applications. These example include when you are doing some development work, or after downloading the latest version of the mdFOAM code from github. You can compile the MD

code manually or by using a script; the script is perhaps simpler. The following is a script, called `compile_mdFOAM_fromScratch` which can be copied into a text file and added to your `$HOME/bin`. Note: you will need to change permissions of that text file using `chmod a+rwx fileName`.

Following the instructions found in the `README-MNF` file will result in the use of the standard OpenFOAM build mechanism of running the `./Allwmake` script at the highest folder level within the source folder to install the entire OpenFOAM 2.4.0 library and associated applications. Once this is complete it may then be preferable to only run recompilation of individual elements of the MD code while developing new functionality. One way to achieve this is using a compilation script which calls each level of the MD code's wmake components or this can also be done manually. The following is a script, called `compile_mdFOAM_fromScratch`, which can be copied into a new text file and added to your `$HOME/bin` path for easy re-use. Note: you will need to change permissions of that text file once created so it is executable using `chmod a+rwx fileName`.

```
-------------------------------------------------------------------
Script: compile_mdFOAM_fromScratch
Use: fresh install of mdFOAM;
Use: when upgrading code from github using pull
-------------------------------------------------------------------
#!/bin/bash

currentDir=‘pwd‘

FOAM="$HOME/OpenFOAM/OpenFOAM-2.4.0-MNF"

# compile code

cd $FOAM/src/lagrangian/molecularDynamics

./Allwclean
./Allwmake

cd $FOAM/applications/solvers/discreteMethods/molecularDynamics

./wcleanAll
./wmakeAll

cd $FOAM/applications/utilities/preProcessing/molecularDynamics

./wcleanAll
./wmakeAll

cd $currentDir
-------------------------------------------------------------------
```

2.2 Installing OpenFOAM on ARCHER

On ARCHER, the installation needs to go into the work directory, rather than in the usual `$HOME/OpenFOAM` location. Find the following location:
`/work/eXXX/eXXX/username/OpenFOAM/OpenFOAM-2.4.0-MNF` and git clone OpenFOAM and put the ThirdParty folder here.

`git checkout devel-dave`
Put the following text in your `~/.bashrc`:

```
export TMPDIR=$HOME/tmp
module swap PrgEnv-cray PrgEnv-gnu
export CRAYPE_LINK_TYPE=dynamic
module load zlib
source /work/eXXX/eXXX/username/OpenFOAM/OpenFOAM-2.4.0-MNF/etc/bashrc
export PATH=$PATH:$HOME/bin
export WM_64=ON
export WM_NCOMPPROCS=8;
```

We now suggest that you checkout your devel branch, if you are a developer:

```
git checkout devel-dave
```

In OpenFOAM, open the `etc/bashrc` file and edit the following:

```
line 49: foamInstall=/work/eXXX/eXXX/username/$WM_PROJECT
line 67: export WM_COMPILER=Gcc49
line 85: export WM_MPLIB=MPICH2
line 133: export WM_PROJECT_USER_DIR=$FOAM_INST_DIR/$USER-$WM_PROJECT_VERSION
```

In `etc/config/scotch.sh` modify to the following:

```
export SCOTCH_VERSION=scotch_6.0.3
export SCOTCH_ARCH_PATH=$WM_THIRD_PARTY_DIR/platforms/
$WM_ARCH$WM_COMPILER$WM_PRECISION_OPTION$WM_COMPILE_OPTION/$SCOTCH_VERSION
```

In the `etc/config/settings.sh` modify the following:

```
Line 398:    MPICH2)
    export FOAM_MPI=mpich2
    export MPI_HOME=/opt/cray/mpt/7.0.3/gni/mpich2-gnu/49
    export MPI_ARCH_PATH=/opt/cray/mpt/7.0.3/gni/mpich2-gnu/49
    ;;
```

In `wmake/rules/linux64GccMNF/` edit the `c` file

```
line 5: cc          = cc
line 15: LINKLIBSO   = $(cc)
```

Then edit the c++ file

```
line 5: CC          = CC
```

In `ThirdParty-2.4.0-MNF/etc/wmakeFiles/scotch/Makefile.inc.i686_pc_linux2.shlib-OpenFOAM-64` change the following:

```
line 12: CFLAGS          = -O3 -DCOMMON_FILE_COMPRESS_GZ
-DCOMMON_RANDOM_FIXED_SEED -DSCOTCH_RENAME -Drestrict=__restrict
```

2.3 Keeping up-to-date

We regularly update our code to include bug-fixes, new models and new functionality. There are two types of update that we do: (a) local changes to the current version of our code, these are pushed into the master release branch on our GitHub account, or (b) shifts to future versions of OpenFOAM, these types of updates are done less frequently and only as required. Subscribe to our mailing list for more information.

2.4 Information for developers

If you have your own development branch and you develop your own models, then we encourage you to keep in touch with the development team to ensure your new additions end up in the master release branch. Please follow these instructions:

1. Get training on how to use Git on your machine (see some quick tips in Section 2.5)
2. Request a devel-user branch from the MNF GitHub development team
3. When you develop your own models, these need to be stored in the following user-directory:
   - Source code in: `$HOME/OpenFOAM/username-x.x.x/src/`
   - Solver and utility applications: `$HOME/OpenFOAM/username-x.x.x/applications/`
   - Also it is important that when you write your own code, to follow the same formatting style used by OpenFOAM.
4. Avoid developing or changing code in the main locations:
   - Source code: `$HOME/OpenFOAM/OpenFOAM-x.x.x/src/lagrangian/molecularDynamics`
   - Solver and utility applications: `$HOME/OpenFOAM/ OpenFOAM-x.x.x/applications/`

   If you have no option but to do so (e.g. you find a bug, you add a new functionality, or you develop a better piece of code), then please contact the project leaders to ensure your source code is included in future versions.
5. When you are happy with some of the models that you have created and which you have extensively tested, then you can inform the project leaders about this to ensure these models are included in future versions. We will ask you for the code of your model and a working test case.

2.5 Using GIT for developers

Here are some of the commonly used instructions for using GIT (assuming the arbitrary name 'Dave').

- If you want to create a new branch (you'll do this once):
  `git checkout -b devel-dave origin/devel-dave`
  Check you're on the right branch:
  `git branch`
- You've installed OpenFOAM on another machine and want to get your devel branch installed:
  `git checkout devel-dave`
  `git pull origin devel-dave`
  Compile the code.
- Let's say you add new code to your development branch. Time to sync your branch with git online:
  1. Check the new stuff you added: `git status`
     This produces a list of edited files and new files you've added. This is useful for the next step.
  2. Add folders, files etc to your devel branch (use previous function call to give you the list you need to add):
     `git add <filename1>`
     `git add <filename2>`
     `git add <filenameN>`

3. Then commit and add some comments to identify what you've added (for future reference):
   `git commit -m  "added a new measurement class called ..."`
4. Push your development to git online:
   `git push origin devel-dave`

- Merging two development branches. Let's say you want to merge your `devel-dave` branch with another branch `devel` (the two exist but they are different, `devel-dave` is more advanced), then follow these steps:

1. Make sure `devel-dave` is up-to-date on your machine and has been synced properly to git (see above).
2. Change to `devel`:
   `git checkout devel`
3. Pull:
   `git pull origin devel`
4. Merge:
   `git merge devel-dave`
5. Look well for any dependency issues and solve these. Compile and check that code runs well. If code needs to be modified to handle dependencies, then after making the changes make sure you resync this devel in the same way as before (add, commit).
6. Push back to git:
   `git push origin devel`

2.6 Dealing with bugs

1. Identify first that this is actually a bug, and not an erroneous mistake related to an incorrect entry or incorrect set up of a case.
2. When you are positive this is a bug, identify the part of model/code where this occurs. We ask that you try and solve it yourself first before seeking further assistance!
3. If you do manage to solve the problem then please ensure you include this as a bug fix on GitHub (see below for more information of how to do this). You may wish to update any outstanding issues within GitHub to reflect this, defining a new meaningful milestone and if you are not going to push the fix to your development branch of the code then please also provide an explanation as to how the issue was fixed and where.
4. If you do not manage to solve it, submit your bug to GitHub. Include a short description and provide a case where the problem has occurred. In your case files, make sure there are the `0`, `constant`, and `system` directories, and if possible make the error appear as the simulation starts, so that debugging is quick. Allocate to project leaders on GitHub. This can be done within GitHub by:
   (a) Ensuring you have the correct branch of the code selected.
   (b) Click on the "Issues" tab on the right hand side of the page (denoted by an exclamation mark in a circle).
   (c) Give the issue a meaningful title and provide a detailed description of the bug.
   (d) If there are case files etc. to be associated with the bug, please also attach them to the bug report. GitHub limits the type of files that can be attached directly to an issue report (Image (jpeg/png); Office document (Word, Powerpoint, Excel) and text file), therefore it is suggested you attach any relevant files in raw text format (.txt) and should there be anything further of consequence to solving the issue, please email it directly to the person that has been assigned (see step 6).

(e) Define a label for the bug (GitHub provides some defaults such as “bug”, “enhancement” etc. please select the most appropriate value.)

(f) If you think a specific member of the development team is best suited too handling the issue then please also select an assignee.

3 Introduction to mdFOAM

To get started with mdFOAM and OpenFOAM, we have prepared a basic case tutorial called `equilibriumWaterCube`, which can be found in the following path (on GitHub):

`$HOME/OpenFOAM/OpenFOAM-2.4.0-MNF/tutorials/discreteMethods/molecularDynamics`

3.1 About the case

This tutorial describes how to initialise, run and post-process a simple equilibrium molecular dynamics simulation of water in an NVT ensemble (constant number of molecules N, constant volume V and constant temperature T). Figure 8 shows the setup of the *case*. The domain consists of a cubic box of side length L=5.05 nm (20 RU), filled with N=4298 water molecules (in order to set a mass density of ρ_M=1000 kg/m^3), and with a Berendsen thermostat applied in the entire domain (in order to set the temperature to T=300 K). Periodic boundary conditions are applied in all x, y and z directions. The aim of this simulation is to learn how to control temperature, and then measure basic properties such as density, temperature and pressure in an equilibrium simulation. More information on the water model can be found in Section 1.7.

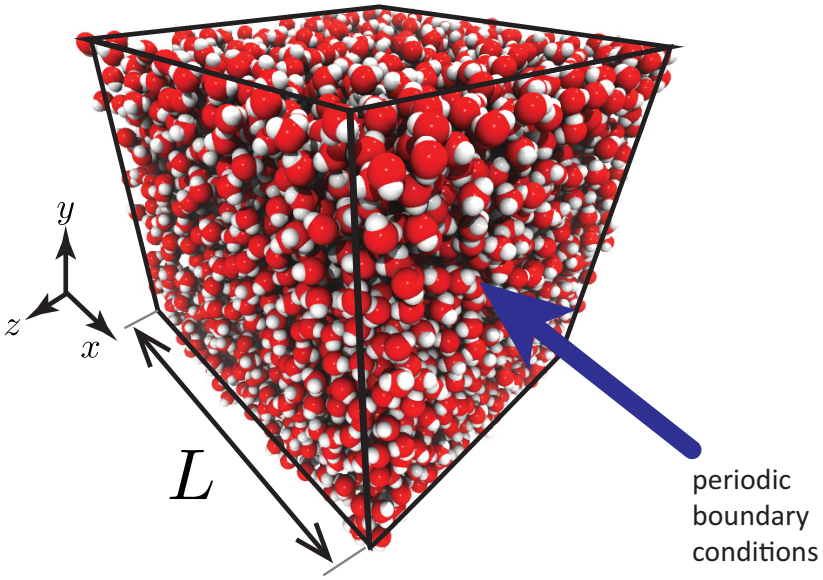

Figure 8: MD simulation of water in a cube of length L, with periodic boundary conditions applied in all x, y, z directions.

3.2 Mesh generation

All MD simulation cases are defined in a three-dimensional Cartesian coordinate system. We generate a mesh using the `blockMesh` utility that comes standard with OpenFOAM. A mesh is necessary in our simulations for a number of reasons, including for: (a) domain decomposition for parallel-processing, (b) the cell-list algorithm, (c) coupling with CFD, and (d) resolving macroscopic field properties. An example of the `blockMeshDict` dictionary can be found in the case-path:

`case/constant/polyMesh/blockMeshDict`

In OpenFOAM the word 'dictionary' is used to define an 'input file' - i.e. instructions which are input by the user in that file, which will then be read-in by the simulation. The following is an example of the `blockMeshDict` for generating a mesh:

```
1  /*--------------------------------------------------------------------------*\
2  | =========                 |                                                |
3  | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox          |
4  |  \\    /   O peration     | Version:  1.6                                  |
5  |   \\  /    A nd           | Web:      http://www.openfoam.org              |
6  |    \\/     M anipulation  |                                                |
7  \*--------------------------------------------------------------------------*/
8
9  FoamFile
10 {
11     version         2.0;
12     format          ascii;
13
14     root            "";
15     case            "";
16     instance        "";
17     local           "";
18
19     class           dictionary;
20     object          blockMeshDict;
21 }
22
23 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
24
25 convertToMeters 1;
26
27 vertices
28 (
29     (0 0 0)
30     (16 0 0)
31     (16 16 0)
32     (0 16 0)
33     (0 0 16)
34     (16 0 16)
35     (16 16 16)
36     (0 16 16)
37 );
38
39 blocks
40 (
41     hex (0 1 2 3 4 5 6 7) mdZone (5 5 5) simpleGrading (1 1 1)
42 );
43
44 boundary
45 (
46     periodicX_half0
47     {
48         type cyclic;
49         faces ((1 2 6 5));
50         neighbourPatch periodicX_half1;
51     }
52
53     periodicX_half1
54     {
55         type cyclic;
56         faces ((0 4 7 3));
57         neighbourPatch periodicX_half0;
58     }
59
60     periodicY_half0
61     {
62         type cyclic;
63         faces ((2 3 7 6));
64         neighbourPatch periodicY_half1;
65     }
66
67     periodicY_half1
68     {
69         type cyclic;
70         faces ((0 1 5 4));
71         neighbourPatch periodicY_half0;
72     }
73
```

```
74      periodicZ_half0
75      {
76          type cyclic;
77          faces ((4 5 6 7));
78          neighbourPatch periodicZ_half1;
79      }
80
81      periodicZ_half1
82      {
83          type cyclic;
84          faces ((0 3 2 1));
85          neighbourPatch periodicZ_half0;
86      }
87  );
88
89  mergePatchPairs
90  (
91  );
92
93
94  // ************************************************************************* //
```

Referring to the `blockMeshDict` example (above), the first lines 1 - 23 is a banner which exists in every OpenFOAM dictionary, not just the `blockMeshDict`; it is not important and will be ignored from now onwards. On lines 27-37, there are 8 vertices that define one block on the mesh (Note: these values are in reduced units). In this case we need just one block (8 vertices) to define the geometry boundary of a 3D box, as indicated in figure 9. Actually, in most MD simulations we only require to simulate cubic or cuboid domains so the `blockMeshDict` example above generally remains the same for most MD simulations, though complex-geometrical domains have also been produced in the past.

Vertices on the block are numbered starting from 0 to 7 and proceed around the block in a very specific order, see Figure 9. More information on how to produce meshes of more than one block can be found from the online OpenFOAM user guides. The `convertToMeters 1;` on line 25 is an option of scaling the domain if required, which is done by multiplying this input value to each vertex in the `vertices` list (in this case it is 1, so there is no scaling).

The blocks can be defined in the entry header 'blocks' starting on line 39. Here we use just one block that comprises 8 vertices in a particular order. For the purpose of most cuboid-shaped MD domains you will only need to focus on the following entry:

```
mdZone (5 5 5)
```

The word `mdZone` does not need to change; it defines the name given to the zone in the block. The vector following `mdZone` defines the number of cells in the x, y and z directions respectively, which we write as $(N_{c,x}, N_{c,y}, N_{c,z})$. The total number of cells is then $N_{c,x} \times N_{c,y} \times N_{c,z}$. The cell size is then given by $\Delta L_i = L_i/N_{c,i}$, where $i = x, y, z$. The cell-size plays an important role in our MD simulations. We always recommend that the cell-size is as close as possible to the largest cut-off distance of the potentials. For clarity if r_{cut}=1 nm, then all cell sizes should be either equal to 1 nm or slightly larger, but never smaller than 1 nm. Although the code is capable of handling any mesh refinement, we find that having $\Delta L_i = r_{cut}$ is optimal and is highly encouraged to use in your meshes.

The boundary entries found between lines 44-87, indicate the 'patches' of the domain boundary. For example, this cubic box has 6 boundaries/patches, so there are 6 cyclic patches. An explanation of one patch is given here:

```
periodicX_half0   // a unique name you assign to one patch
{
       type cyclic;   // the type assigned to the patch (e.g. cyclic, patch)
       faces ((1 2 6 5));   // the mesh vertices of that patch
       neighbourPatch periodicX_half1;   // it's neighbourPatch if cyclic
}
```

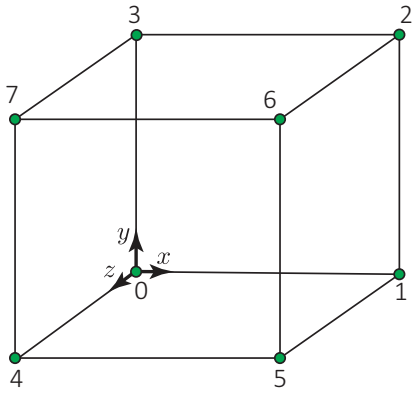

Figure 9: Block structure of the mesh for the case.

Furthermore, the 6 patches in the `blockMeshDict` need also to be defined (and linked) properly in the `system/boundariesDict` as follows:

```
25 polyPatchBoundaries
26 (
27
28
29 );
30
31 polyCyclicBoundaries
32 (
33     boundary
34     {
35         boundaryModel  polyStandardCyclic;
36         patchName      periodicX_half0;
37     }
38     boundary
39     {
40         boundaryModel  polyStandardCyclic;
41         patchName      periodicX_half1;
42     }
43     boundary
44     {
45         boundaryModel  polyStandardCyclic;
46         patchName        periodicY_half0;
47     }
48     boundary
49     {
50         boundaryModel  polyStandardCyclic;
51         patchName      periodicY_half1;
52     }
53     boundary
54     {
55         boundaryModel  polyStandardCyclic;
56         patchName        periodicZ_half0;
57     }
58     boundary
59     {
60         boundaryModel  polyStandardCyclic;
61         patchName      periodicZ_half1;
62     }
63 );
64
65 polyGeneralBoundaries
66 (
67
68 );
```

Here `polyStandardCyclic` is a model which our MD code identifies as the standard periodic boundary conditions (other models can be chosen or new ones developed that define modified

periodic boundary conditions; there could also be patch boundaries).

We now demonstrate how to generate the mesh and construct the cyclic boundary conditions from the mesh. This is done by the following two command lines:

```
blockMesh
createCyclicBoundaries
```

We find it best to create your own scripts to speed-up the initialisation of the mesh (you might find you will need to add some cleaning steps when recycling other cases), here is an example of one we normally use:

```
Script: setCase
Description: clean case, generate mesh, link cyclic boundaries
Usage: setCase
-----------------------------------------------------------------
#!/bin/bash

rm constant/polyMesh/*Zones
rm -rf storage
rm -rf 0
blockMesh
createCyclicBoundaries
-----------------------------------------------------------------
```

In this script we added a line that removes any of the cell- and face-zones that are created from other utilities. This has to be done *before* running `blockMesh`, as we found that these files do not get removed after the `blockMesh` utility is called.

3.3 Reduced units

The *fundamental* normalisation parameters used in this simulation (see Section 1.6) are defined in the `case/system/reducedUnitsDict`. For example entries for this case include:

```
25  reducedUnits yes;
26  outputSI yes;
27
28  refLength 3.154e-10;
29  refTime 1.66e-12;
30  refMass 2.987e-26;
31  refCharge 1.602176487e-19;
```

Refer back to Section 1.6 for learning how to convert between SI units and RU. For example, the cut-off radius $r_{cut} = 1 \times 10^{-9}$ m is changed into reduced units as follows: $r^*_{cut} = 1 \times 10^{-9}$ / `refLength` $= 1/0.3145 = 3.17$ RU.

Note: the normalisation parameters for the full list of properties (e.g. pressure, density, temperature etc) can be viewed in the first 10-20 lines in the log file when you run the mdFOAM solver (or one of the MD utilities).

3.4 Constant molecule properties

The `case/constant/moleculeProperties` dictionary stores the constant properties of all the molecule species that will be used in the simulation. The entries for this case are as follows:

```
19  idList          (water);
20
21  moleculeProperties
22  (
23      water
```

```
24      {
25          cloudType               polyMoleculeCloud;
26          siteIds                 (H H O M);
27          pairPotentials          (0 0 1 0);
28          siteReferencePositions
29          (
30              (7.56950327263661e-11 5.85882276618295e-11 0)
31              (-7.56950327263661e-11 5.85882276618295e-11 0)
32              (0 0 0)
33              (0 1.546e-11 0)
34          );
35          siteMasses
36          (
37              1.67353255e-27
38              1.67353255e-27
39              2.6560176e-26
40              0
41          );
42          siteCharges
43          (
44              8.91450997367e-20
45              8.91450997367e-20
46              0
47              -1.782901995e-19
48          );
49      }
50  );
```

The first entry is called the `idList` which stores the full list of molecules that will be used in a simulation. Each molecule species is given a unique 'id name' - in this case this is `water`. The index of the entries in the `idList` is the 'id number' and is used internally in the code to keep track of molecules. In this case, the id number for all water molecules is 0 (since it is the first and only entry in the `idList`). The molecule properties of all species in the `idList` then need to be defined **in the same order** in the `moleculeProperties` sub-dictionary. These properties do not expect to change in the simulation, (e.g. the mass of each site on the molecule in S.I. units, charges on sites, pair potentials etc).

After the molecules are initialised, you can check the id numbers of all molecules in the file:

`case/timeDir/lagrangian/polyMoleculeCloud/id`

3.5 Initialising molecules

The mesh can now be filled with molecules, using the `mdInitialiseField` utility. The utility, once run, will create a zero directory with populated lists of properties, such as positions, velocities, and id's (to name a few) of all the molecules in the system. The size of each list is equal to the number of molecules, N.

There are various models available for initialising molecules, and we encourage users to improve on existing ones, create new ones and share these with the development team - the process is relatively straightforward. All available molecular-initialising models can be found in:

`src/lagrangian/molecularDynamics/polyCloud/initialiseMolecules/derived/`

For this case we need to insert N water molecules that satisfies a target fluid density of ρ_M= 1000 kg/m^3. Using the equation of mass density:

$$\rho_M = \frac{M}{V} = \frac{Nm_i}{L^3} \tag{12}$$

where $m_i = 2.99{\times}10^{-26}$ kg is the mass of one water molecule (check: `case/constant/moleculeProperties`), and $L = 5.05$ nm (16 RU) is the cube's side length.

We use `polyBCC` insertion model, which allows the insertion of an exact number of molecules placed regularly in a BCC lattice. The input file for inserting molecules can be found in

`case/system/mdInitialiseDict`, as follows:

```
25  polyConfigurations
26  (
27
28      configuration
29      {
30          type                    polyBCC;
31          temperature             3.816;
32          bulkVelocity            (0.0 0.0 0.0);
33          molId                   water;
34          frozen                  no;
35
36          boundBox
37          {
38              startPoint          (0.9 0.9 0.9);
39              endPoint            (16 16 16);
40          }
41
42          unitCellSize            1.1;
43          N                       4298;
44      }
45  );
```

In the `mdInitialiseDict` (above) there is just one 'configuration entry' specified (i.e. which in this case will be building one lattice of water molecules). However for more complicated cases where you need to initialise various molecular configurations (e.g. graphene sheets, carbon nanotubes, solid regions, gas regions etc) the user can specify any number of entries one after the other in `polyConfigurations`. The utility will execute and initialise the molecules in each configuration in sequence as defined in the `polyConfigurations` subdictionary.

The cardinal rule of initialisation in molecular dynamics is that molecules can NEVER overlap. Any two molecules are said to be overlapping if they are less than a critical distance from each other $r_{ij} < r_{cr}$. This critical distance is dependent on the pair potential, but a general rule of thumb is around $r_{cr} = 0.3$ nm (0.9 RU).

The configuration above avoids overlapping by filling a box (called a `boundBox`) which allows for a slight gap at the periodic boundaries (note the offset in the `startPoint`). The `unitCellSize` indicates the size of the unit cell of a BCC lattice structure (typically we find that this should be between 1.0 - 1.2 for initialising water). By specifying $N = 4298$ we ensure that the initialising tool inserts *exactly* 4298 molecules. The initialising utility can be run in the command-line terminal by typing:

```
mdInitialiseField
```

The utility creates a zero time-directory that stores the initial particle data of the lattice structure, and which is required to be read in by the MD simulation later on. If you have problems when recycling from older, simple delete the zero directory, run `setCase` and `mdInitialiseField`.

3.6 Viewing the mesh and initialised molecules

3.6.1 ParaView

Before you run the MD simulation it is important to view and verify the mesh, and also to check that the molecules have been distributed properly within the mesh. We use two visualisation software: (a) paraView and (b) VMD, both have key advantages.

The post-processing utility paraView is supplied with OpenFOAM in the ThirdParty libraries, and is called by executing the following command-line from within the case directory:

```
paraFoam
```

There are various ways of viewing the molecules in paraView. Our preferred option is to execute the `foamToVTK` utility beforehand:

`foamToVTK`
`paraFoam`

The `foamToVTK` utility creates a `VTK` folder in the case directory, and the contents of which can be opened with paraView.

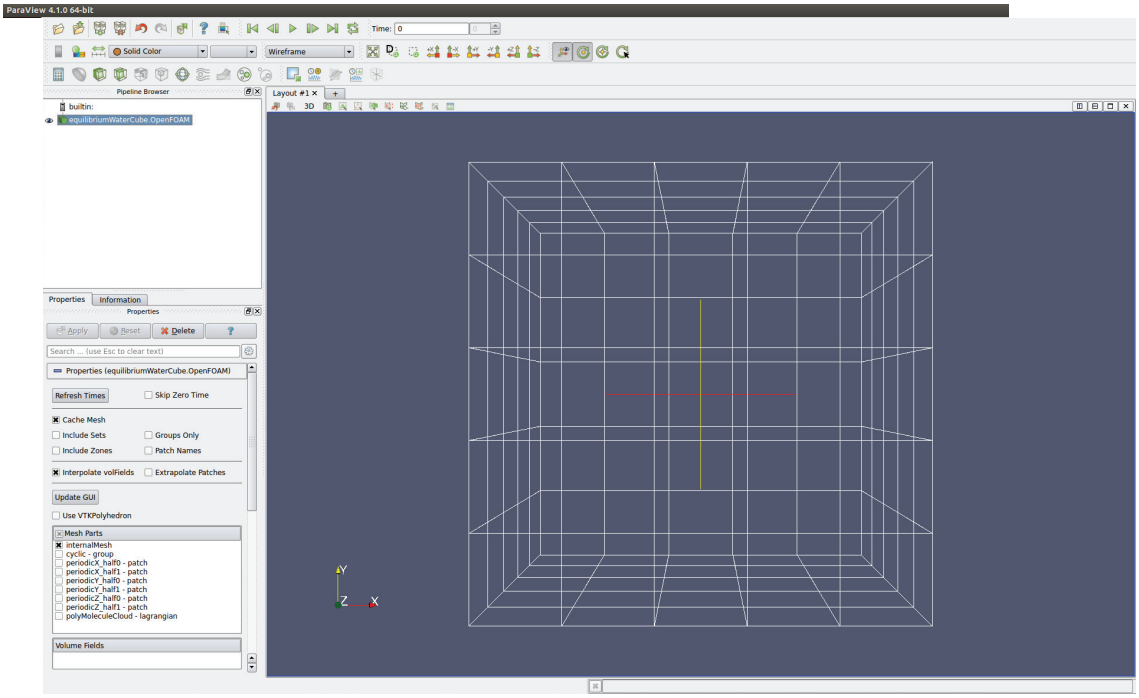

Figure 10: Viewing the mesh in paraView.

The `paraFoam` utility launches the paraView software. In the Pipeline Browser (on the left hand side), you can see that paraView has opened `equilibriumWaterCube.OpenFOAM`, the module for this case. You can click the Apply button. Then view the mesh using the"wireframe" option. There are various ways of changing the visual aspect of the mesh and background, perspective etc. We refer the reader to the user-guide of paraView for more information.

There are usually around two or three methods with which you can load and visualise the molecules within the mesh in paraView. We will be using the `VTK` files generated earlier by the `foamToVTK` utility. Click on File > Open, in the top menu panel. Open the file: `VTK` / `lagrangian` /`polyMoleculeCloud` / `polyMoleculeCloud0.vtk`. Here 0 implies this is the zero time-directory. This creates a new entry in the Pipeline Browser. While the new entry is selected, choose the glyph filter from the Filters > Alphabetical from the top menu, or else if you have a quick button in paraView, as shown in Figure 11. Within the Properties panel of the Object Inspector, modify the properties there as as shown in Figure 11: (1) choose glyph to be sphere; (2) uncheck Orient; (3) Scale Mode: off; (4) check the edit option and Set Scale Factor to 0.5; (5) uncheck mask points and random points. This approach has the advantage of hiding the mesh by clicking on the eye button next to the mesh located in the Pipeline Browser, leaving the molecules to be displayed. The main reasons for using paraFoam as opposed to VMD, is to check that the mesh (cell sizes in all three directions etc) is correct, to check that the molecules have been initialised to your satisfaction in relation to the mesh, and to check that there aren't any signs of overlapping molecules.

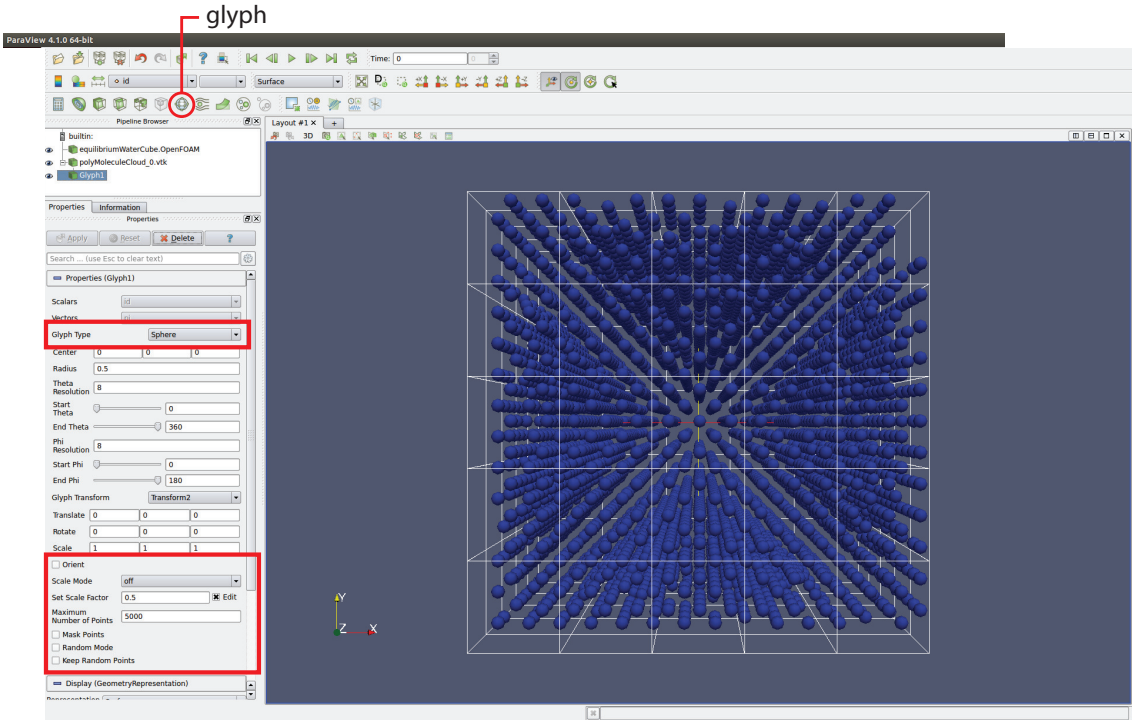

Figure 11: Viewing the molecules in paraView.

3.6.2 VMD

VMD is a molecular visualisation software which can be downloaded for free (on almost any operating system), and which has much better graphics than paraView. VMD is the best of the two for creating movies or rendering high quality snap shots for your presentations and journal papers. To display the initial molecular lattice structure of the case, launch the VMD from the terminal or application. From File > New Molecule load the `polyMoleculeCloud.xmol` file located in the `0/lagrangian` directory. For this particular case I prefer removing the M site from the 4-sited water model, to visualise the more familiar H20 molecule. To do this run:

```
extractXMOLDelete M polyMoleculeCloud.xmol polyMoleculeCloud_water.xmol
```

where `extractXMOLDelete` is a script found in Section 6.

The molecules are then represented in the Display window. Click on Graphics > Representations and choose VDW from Drawing Method. To change the colour schemes, click on Graphics > Colors. See settings in Figure 12. More information on creating images and videos can be found online.

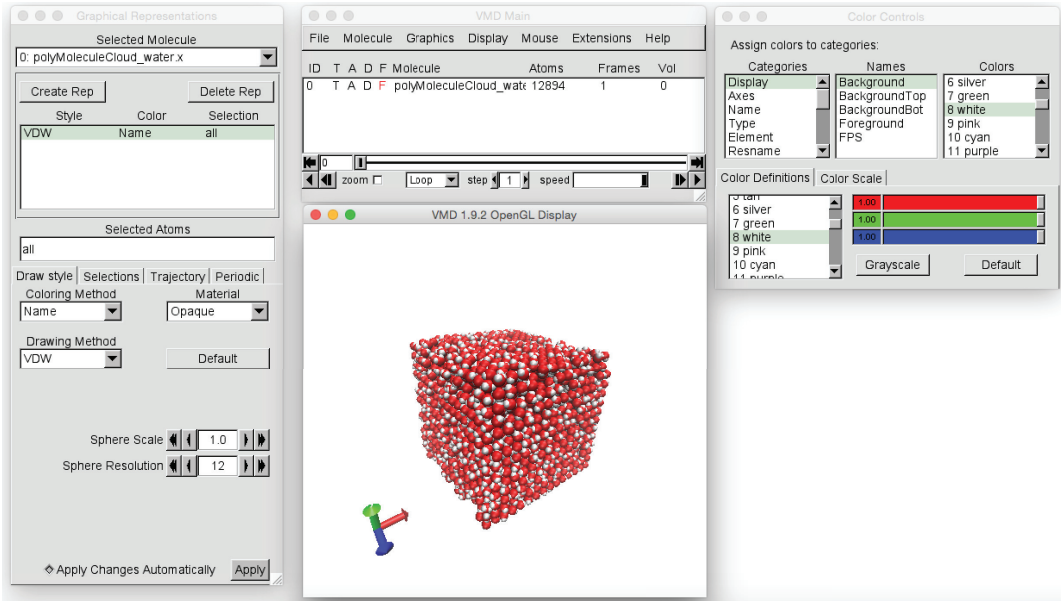

Figure 12: Viewing the molecules in VMD. Snap-shot of initial MD configuration.

3.7 Running mdFOAM

Like any UNIX/Linux executable, OpenFOAM applications can be run in two ways: (a) as a foreground process, i.e. one in which the shell waits until the command has finished before giving the command prompt back, or (b) as a background process, i.e. one which does not have to be completed before the shell accepts additional commands. I always prefer the latter, and redirect the output of the simulation into a log file so that you will check-up on the simulation later. In serial, the MD solver can be launched from the command-line terminal as follows:

```
mdFOAM > log &
```

This will create a log file in your case directory, which you can view during MD simulation:

```
tail -f log
```

This case has been setup to be simulated immediately. So up until now we have skipped out a number of important input dictionaries, which have to be looked at in more detail before running future MD simulations using `mdFOAM`. In the following sections we describe these dictionaries for completness.

3.8 Control of Time

Time (and the control of time) is central to simulations in OpenFOAM, including our MD solver. The input dictionary that controls time is found in `case/system/controlDict`. The following is the `controlDict` for this tutorial case:

```
17  application     mdFoam;
18
19  startFrom       startTime; // option to start simulation at startTime (also available: latestTime)
20
21  startTime       0.0; // start from 0 time-directory (can be changed to any time-directory, if available)
22
23  stopAt          endTime; // option to end at endTime (other options available)
24
25  endTime         3.6; // stops simulation when simulation time has reached this value
26
27  deltaT          0.0012; // size of MD time-step (important on stability)
28
29  writeControl    runTime;
30
31  writeInterval   0.012; // after ever write interval, a time-directory is written out
32
33  purgeWrite      3; // only latest 3 time-directories are stored (delete the rest); if 0, no deletion
34
35  writeFormat     ascii;
36
37  writePrecision  12;
38
39  writeCompression uncompressed;
40
41  timeFormat      general;
42
43  timePrecision   8;
44
45  runTimeModifiable yes;
46
47  adjustTimeStep  no;
```

We highlight above the important entries which you require to consider for most MD simulations. It is important to note that all units here are in reduced units of time. For example, to change the MD time-step $\Delta t^*_M = 0.0012$ to S.I. units, multiply by `refTime` (see reduced-units). OpenFOAM offers great flexibility with time control which is described in full in the OpenFOAM user-guide.

The start/stop times and the time step are perhaps the most important and need to be defined first. In the above example we use a `startTime` of $t_{start} = 0$. This means that you need to have a molecular dynamics state in the 0 time-directory of the case. In this case the initial time-directory has been created by the `mdInitialisePolyField` utility, but if you want to restart a simulation at a later time, you will just need to change the `startTime` to the required time-directory.

The `endTime` is when the simulation will stop. Most probably you would want to choose this as large as possible, large enough for example to reach a steady-state, or to measure certain properties. (Remember that MD simulations can take several days of processing).

As we described in the MD introduction, the time-step will determine the stability as well as the computational efficiency of the algorithm. In this case we choose a time-step `deltaT`, $\Delta t^*_M = 0.0012$, which is around 2 fs.

At regular intervals during the simulation, time-directories will be written out in the case, each directory describing the state of the system at that particular point in time. Each directory would be identical in format to the currently available zero time-directory, but the new time-directories

would contain different positions, velocities etc which have been generated by the MD simulation. The `writeInterval` keyword is the interval between successive write outs, and in this case has been specified as $\Delta t^*_W = 0.012$, which means that the simulation is going to write out results after *every* 10 MD time-steps (calculated from $\Delta t^*_W / \Delta t^*_M$). It is important that you choose the write interval to be an exact *integer* of the MD time-step size (in this case, 10 is a good choice). The choice of `writeInterval` will depend on a number of factors and will become more apparent as you start running simulations and understanding what parameters depend on it. We suggest you make the `writeInterval` at least 1.2 - 4.8 for other simulations. If the `writeInterval` is equal to the MD time-step size `deltaT` then the simulation would need to write out every time-step, which could be extremely costly on processing time, input-output time as well as hard disc space.

You can calculate a priori how many time-directories will be written during the simulation using $t_{end} - t_{start} / \Delta t_W$. Multiplying this by the memory requirement of one time-directory (check using command `du -sh *`) you can start to understand how much free memory you will need for that particular simulation. An analysis of how much memory is required becomes an important step when running these simulations on high performance computers, where memory allocation is very restrictive (typically a few hundred gigabytes). It is important to ensure that your simulations do not have a large memory foot print, especially if you want to run many of these simulations in tandem. For example, we normally do not allow the MD simulation to write out all the time-directories, in particular for simulations that are run for millions of time-steps. Instead we find it more hard-disc friendly to use the `purgeWrite` option with a value greater than 0. If it is 0, it means the simulation will store all the write-interval time-directories; this is useful perhaps to make video files or for measuring an important and very specific property. In the case above we use a value of 3, which means it will store only the latest 3 directories.

There are various ways of stopping a simulation which is currently running. If it is running on a high performance computer, usually running the following command: `qdel job-id-number` would stop the simulation. If running on your PC, you can bring the simulation to the foreground first, and then type `ctrl-C` to stop the simulation. You can also stop the job using the `top` utility that comes with linux. OpenFOAM also offers this functionality as a means of terminating the job properly. In the `controlDict` edit the `stopAt` entry to `writeNow` (if you want the simulation to stop immediately), or `nextWrite` (if you want the simulation to stop at the next write-interval). Don't forget to change it back to `endTime` once the simulation stops otherwise the simulation may never restart again.

3.9 Potentials

We are constantly incorporating new force fields and pair-potentials in our code. For this case we model short-range electrostatic and Lennard Jones pair-potentials. All information associated with the potentials are included in the `case/system/potentialDict` as follows:

```
51 pairs
52 (
53     O-O
54     {
55         pairPotential   lennardJones;
56         rCut            1.0e-9;
57         rMin            1e-15;
58         dr              1e-13;
59         lennardJonesCoeffs
60         {
61             sigma       3.1589e-10;
62             epsilon     1.286751503e-21;
63         }
64         energyScalingFunction   noScaling;
65         writeTables     yes;
66     }
67 );
68
69 electrostatic
```

```
70  {
71      pairPotential   coulomb;
72      rCut            1e-9;
73      rMin            1e-15;
74      dr              2e-12;
75
76      energyScalingFunction   shiftedForce;
77      writeTables     yes;
78  }
```

Note that in this case there is only one specie, (water), which uses the Lennard Jones shifted potential for O-O interactions and short range electrostatic interactions for the charged sites.

3.10 Controlling MD

Another important part of MD simulations is to control Lagrangian properties, e.g. thermostats to control temperature, or molecular insertion/deletion protocols to control density. To implement this in a more flexible and user-friendly approach, we have created our own controller architecture [4]. Controllers can be added to the simulation in the `case/system/controllersDict` input file. For this case we just have a Berendsen thermostat:

```
18  polyStateControllers
19  (
20      // thermostat
21
22      controller
23      {
24        zoneName            mdZone;
25
26          stateControllerModel        polyTemperatureBerendsenBinsNew;
27
28          polyTemperatureBerendsenBinsNewProperties
29          {
30              temperature             3.816;
31              tauT                    0.04;
32              molIds                  (water);
33              peculiar                yes;
34
35              binModel                uniformBins;
36
37              uniformBinsProperties
38              {
39                  startPoint                  (0 0 0);
40                  endPoint                    (16 0 0);
41                  nBins                       4;
42                  area                        100; // dummy variable
43              }
44          }
45      }
46  );
47
48  polyFluxControllers
49  (
50
51  );
```

The temperature of this case is controlled using the Berendsen thermostat in 4 bins, which controls temperature through molecular velocity re-scaling, hence adjusting the system's kinetic energy ($1/2 m_i v_i^2$). Molecules undergo re-scaling using the following modified factor:

$$\chi = \left[1 + \frac{\Delta t_m}{\tau_T}\left(\frac{T_t}{\langle T \rangle} - 1\right)\right]^{1/2}, \tag{13}$$

where τ_T is a time-relaxation constant that defines the coupling strength between the measured system temperature $\langle T \rangle$ and the target temperature T_t. In this case we choose a target temperature of $T_t = 3.816$ in reduced units ($T_t = 298$ K in S.I. units) and a time-relation a few times larger than the MD time-step, e.g. $\tau_T = 0.04$ in reduced units.

3.11 Measurements

A typical molecular dynamics simulation outputs Lagrangian positions, velocities, etc of all molecules in the time-directories. A central task in MD is to extract from these simulations more meaningful measurements. To do this we have created our own measurement architecture. In this example we will measure only the simplest of properties by specifying measurement entries in the `case/system/fieldPropertiesDict`. The input file for the `waterCubeEquilibrium` case looks like:

```
19  polyFields
20  (
21      // - standard
22      field
23      {
24          fieldModel              polyOutputProperties;
25      }
26
27
28      // REGION
29
30      field
31      {
32          fieldModel              polyPropertiesZoneBounded;
33
34          polyPropertiesZoneBoundedProperties
35          {
36              fieldName               cumul_region;
37              molIds                  (water);
38              resetAtOutput           off;
39              boxes
40              (
41                  box
42                  {
43                      startPoint          (0 0 0);
44                      endPoint            (16 16 16);
45                  }
46              );
47          }
48      }
49
50      field
51      {
52          fieldModel              polyPropertiesZoneBounded;
53
54          polyPropertiesZoneBoundedProperties
55          {
56              fieldName               instant_region;
57              molIds                  (water);
58              resetAtOutput           on;
59              boxes
60              (
61                  box
62                  {
63                      startPoint          (0 0 0);
64                      endPoint            (16 16 16);
65                  }
66              );
67          }
68      }
69
70      // poly PDB (videos)
71
72      field
73      {
74          fieldModel              polyPDB;
75
76          polyPDBProperties
77          {
78              fieldName               water;
79              molIds                  (water);
80              sitesToExclude          (M);
81              numberOfOutputSteps     50;
82              molOption               water;
83          }
84      }
85
```

```
86 );
```

The first entry, `polyOutputProperties` is a field-model which could be included in every MD simulation (exclude on long simulation runs to spare memory). This measurement model handles output to a log file of the basic information, such as total number of molecules in the system, density, average momentum and energies per molecule. This is a very useful measurement tool for the user to quickly check-up on a running simulation.

The next two entries use the same measurement field-model, `polyPropertiesZoneBounded`, which is the measurement of time-varying continuum-type fields such as pressure, density, temperature, velocity, etc within a particular zone. Here the zone is defined by the `boundBox` and is chosen here to occupy the entire domain; it is an equilibrium simulation, so the properties are spatially homogeneous. The two models use different inputs in order to show different time-variations of properties. The first shows variations of properties averaged over the entire simulation, while the second one performs averaging over the write interval. Results of these two measurements are shown in the results section. The last entry generates a PDB file for creating videos in VMD.

3.12 Viewing the results

MD outputs a variety of data for you to visualise and review about a particular simulation. There are two locations in a typical case directory where MD stores data. First and foremost there are the time-directories which are created during run-time, i.e. at every write interval. In each time directory you will find the `lagrangian` and `uniform` directories. The `timeDir/lagrangian/polyMoleculeCloud` directory stores the Lagrangian properties of the molecules: think of it as a 'snap-shot' of the simulation run. This means it stores the *instantaneous* positions, velocities, acceleration etc of all molecules at the time-instance corresponding to the write-time. These can be opened and viewed with paraView or VMD as explained above. You can also make videos of the system dynamics by assembling many of these Lagrangian snap-shots together (there are better ways of making videos). The time-directories also store spatial continuum field measurements (having lower degrees of freedom than the molecules). In this case we did not output any field sampling because the system is spatially homogeneous. Most of these measurements are stored in the `uniform/poly` directory.

The second type of measurements are those that vary in time. These properties are stored in the `fieldMeasurements/poly/` directory of the case. Unlike spatial measurements, temporal measurements accumulate during the simulation. This means that properties can be plotted even during run-time; it can be used also as a 'check' of the progress of the MD simulation. To do this, we use `gnuplot`, a command-line driven graphing utility for Linux. In the case directory, type in the terminal:

```
gnuplot
```

Let us say we want to plot the instantaneous temperature of the system in SI units. In the `gnuplot` environment type:

```
plot "fieldMeasurements/poly/zone_bb_instant_region_T_SI.xy" w l
```

To present your results in a report or publication, gnuplot can be used also using a script, using the following terminal command:

```
gnuplot <scriptName>
```

Here is an example for a script, called `temperature_gnuplot` for creating a plot of temperature against time (the output from this script is seen in Figure 13):

```
set xlabel "Time, t (fs)" font "Helvetica,24"
set ylabel "Temperature, T (K)" font "Helvetica,24"

set key left top

set terminal postscript eps enhanced color 24 "Helvetica"

set linetype 1 dt 1 lc rgb "red" lw 4 pt 7
set linetype 2 dt 2 lc rgb "blue" lw 4 pt 9

f(x) = 300

set pointsize 2

plot "fieldMeasurements/poly/zone_bb_instant_region_T_SI.xy" u ($1/1e-15):2 w l ls 1 title "instant temperature", \
f(x) w l ls 2 title "target temperature"


set size 1.4,1.0

set output "temperature.eps"
replot
```

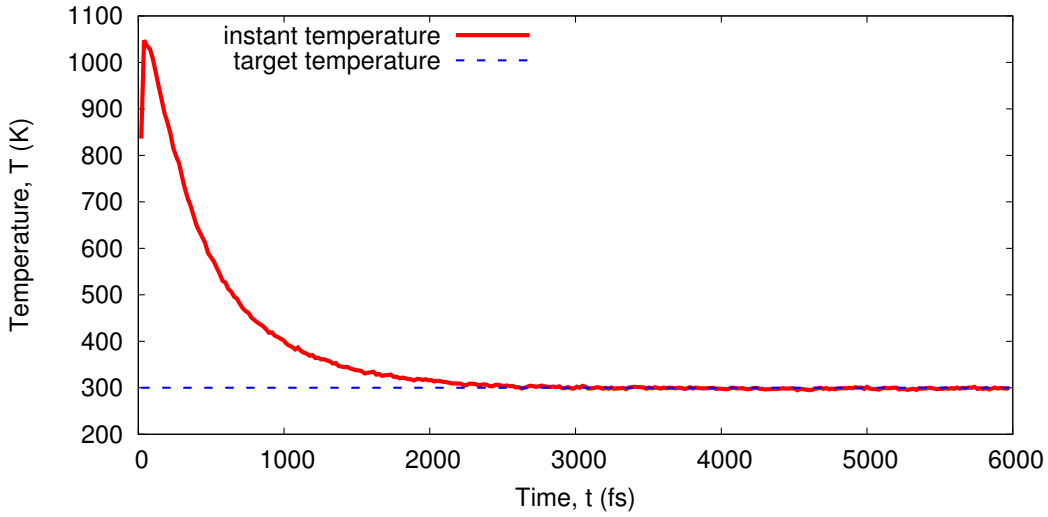

Figure 13: Results for variation of temperature in the MD simulation run. Graph is an *eps* file output from gnuplot, which is an ideal format for including it as a graphics in this Latex document.

4 Parallel-Processing

The major issue with MD is its computational cost scaling with number of molecules in the system as well as simulation time. One way of attaining speed up is by using a domain-decomposition over several processors. This scheme is used in OpenFOAM as the primary method for parallel-processing. The general idea is to use an algorithm that partitions the main mesh along its internal faces into smaller *sub-meshes*. Each sub-mesh is then assigned to a different processor, and a "replica" of the MD code is used by each processor to solve the MD dynamics on its own sub-mesh. The overall simulation speeds up because each processor performs the algorithm over fewer number of molecules in the sub-mesh. At well-defined times of the MD run, all processors communicate with each other the specific data to ensure correct parallelisation.

4.0.1 Processor-decomposition

We will now run the `equilibriumWaterCube` case on 4 processors. Domain-decomposition properties are defined in the `decomposeParDict` input file located in the `system/` directory of the case. Here is the example for this case:

```
26      numberOfSubdomains 4;
27
28      method              simple;
29
30      simpleCoeffs
31      {
32          n               (2 2 1);
33          delta           0.001;
34      }
```

There are many methods for mesh-decomposition. The `simple` method is sufficient in this case. It uses Cartesian coordinate weightings, given by the entry `n`. The mesh will be partitioned in half, for both the $x-$ and $y-$directions, and none in the z-directions. See Figure 14.

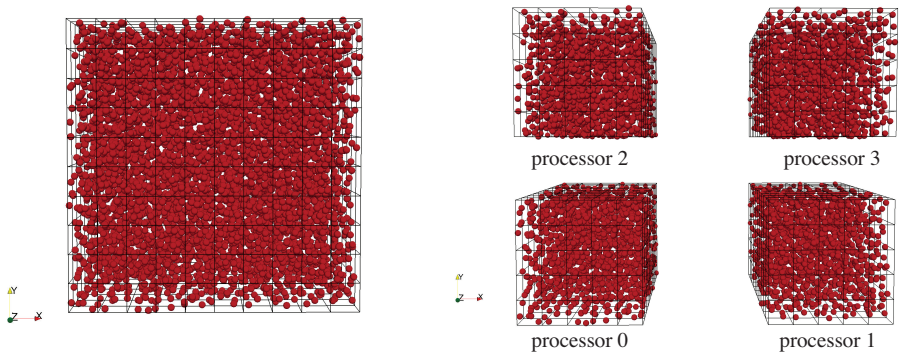

Figure 14: Comparison of equilibrium case: (left) case is run on one processor mesh; (right) case is decomposed and run on 4 processor sub-meshes.

Before executing the MD solver, the following pre-processing step needs to be performed. The mesh and Lagrangian fields is partioned on separate processors using the `decomposePar` utility, after editing correctly the `decomposeParDict`, as shown above. In the terminal type:

```
decomposePar
```

The case now consists of four separate directories labelled by: `processor0`, `processor1`, `processor2`, `processor3`. In each processor the Lagrangian fields and the mesh are segmented as discussed above.

4.0.2 Run and view results

The MD simulation can finally be run in parallel using:

```
mpirun -np 4 mdFOAM -parallel > log &
```

To view the decomposed mesh and Lagrangian field per sub-mesh, you can change into each processor directory in tandem and run the `foamToVTK` utility. You can do this automatically using a script:

```
#!/bin/sh

printUsage () {
cat << EOF
   Usage: $0
EOF
}

if ! [ $# -eq 0 ] ; then
    printUsage
    exit 1
fi

cases=`echo processor*`

for case in $cases
do
    cd $case
    foamToVTK
    cd ../
done
```

To view them individually in paraView, run `paraFoam` as normal, but load files from the `processor0/VTK/` directory.

Imp. Note: To save on hard-disc memory space, results are output only in the `processor0` directory, which also known as the *master processor*. The temperature profile can be viewed in `gnuplot` as follows:

```
plot "processor0/fieldMeasurements/poly/zone_bb_instant_region_T_SI.xy" w l
```

To reconstruct the simulation to a single time-directory, run:

```
reconstructPar -latestTime
```

To reconstruct the *.xmol* file, you will also need to run:

```
reconstructXmol -latestTime
```

5 Troubleshooting

A well-written code is one which reports understandable error messages to an *execution-level* type of user such that to prevent the code from outputting incorrect results, or simulation to behave abnormally. An *execution level* user is someone who runs MD cases only, and knows little about the programming or implementation of the code a few levels down the hierarchy. Here I will try to list some of the common errors that arise either in the initial part of the simulation or else midway through a simulation. By learning how to read an error you will be able to solve it quickly or even report the bug to a programmer, if this is the case. We will try to include as many error types as possible. Of course we ask for you help to include the missing ones.

5.1 Missing/incorrect input keyword in dictionary

This type of error is common in the following circumstances:

- There is a change in the version of the code:
  - the keyword name is changed to better represent its meaning (e.g. `density` instead of `rho`)
  - a new entry is included which is missing in the user's dictionary.
- The user makes a typo in the keyword (e.g. `denisty` instead of `density`)
- The user forgets to include the keyword.

The following is an example of an error message for this case:

```
--> FOAM FATAL IO ERROR:
keyword zoneName is undefined in dictionary "::controller::controllerProperties"

file: ::controller::controllerProperties from line 26 to line 35.

    From function dictionary::lookupEntry(const word&, bool, bool) const
    in file db/dictionary/dictionary.C at line 395.

FOAM exiting
```

5.2 Incorrect input type in dictionary

An example of an incorrect type of input is when a user specifies a scalar instead of a vector.

User writes scalar component:

```
force                0.01;
```

User meant to write vector:

```
force                (0.01 0 0);
```

This is the error message:

```
--> FOAM FATAL IO ERROR:
Expected a '(' while reading VectorSpace<Form, Cmpt, nCmpt>, found on line 160 the doubleScalar 0.01

file: ::controller::atomisticForceProperties::uniformForceProperties::force at line 160.

    From function Istream::readBegin(const char*)
    in file db/IOstreams/IOstreams/Istream.C at line 86.
```

5.3 Blow-up

We have already discussed *blow-up* in the introduction part of the this manual, and how to tackle it. You notice that a simulation blows up because the output log file quantities increase to infinity, such as the average kinetic energy (KE) per molecule.

Example of a standard time-step:

```
Time = 0.025
Calculate fields
Number of molecules in system = 5832
Overall number density = 0.729
Overall mass density = 0.729
Average linear momentum per molecule = (0.0350475226926 -0.000776640615697 0.0132470825267) 0.0374755548725
Maximum |velocity| = 6.35464409376
Average linear KE per molecule = 2.94844523348
Average PE per molecule = -4.12084427898
Average TE per molecule = -1.17239904551
ExecutionTime = 5.44 s  ClockTime = 6 s
```

Example of a time-step that indicates blow-up:

```
Time = 0.25
Calculate fields
Number of molecules in system = 5832
Overall number density = 0.729
Overall mass density = 0.729
Average linear momentum per molecule = (0.0382265676207 -0.0342201303606 0.0137561785413) 0.0531179841675
Maximum |velocity| = 3.90942020126e+16
Average linear KE per molecule = 2.6333131865e+29
Average PE per molecule = 302668938841
Average TE per molecule = 2.6333131865e+29
ExecutionTime = 19.27 s  ClockTime = 20 s
```

When an MD simulation blows up, the code remains at a standstill at this last time-step. The user needs to kill the simulation, do necessary adjustments to the input parameters to prevent blow-up again, and restart the simulation.

5.4 Floating Point Exception

A floating point exception is a division of a number by zero. If you encounter such a problem please report it as a bug.

5.5 Segmentation Fault / Print Stack

A segmentation fault or print stack error is also a bug in the programming part of the code. It is associated with an incorrect construction and use of lists or arrays. For example when the code attempts to access an entry in an array which is larger than its size. Please report this bug if you encounter it.

6 Scripts

Here we include a few of our favourite scripts. Copy the text between dotted lines into a text file, and save to your `$HOME/bin`. Note: you will need to change permissions of that text file using `chmod a+rwx fileName`. **IMPORTANT: Also please be careful when copying these scripts. We've noticed that the verbatim environment has messed the format of some of the punctuation symbols such as ~, /, etc**

```
Script: clean
Description: removes any temporary files (~fileName) inside nested directories
Usage:
cd into a directory that requires removal of temporary files
and then run: clean
-------------------------------------------------------------------
#!/bin/bash

find . -name "*~" -exec rm {} \;
-------------------------------------------------------------------
```

```
Script: changeFileFoam
Description: to create quick copies of classes, but renamed to your new class name
Usage: changeFileFoam className.C newClassName.C
Usage: changeFileFoam className.H newClassName.H
-------------------------------------------------------------------
#!/bin/bash

cp $1 $2

echo "$1" > tempFile1.$$
origName=`cut -f 1 -d "." tempFile1.$$`
rm tempFile1.$$

echo "$2" > tempFile1.$$
newName=`cut -f 1 -d "." tempFile1.$$`
rm tempFile1.$$

#replaces ALL instances in a line
sed -e "s/$origName/$newName/g" $2 > tempFile1.$$

mv tempFile1.$$ $2
-------------------------------------------------------------------
```

```
Script: extractXMOL
Description: to extract certain molecule species from a .xmol file
Usage: extractXMOL 0 polyMoleculeCloud.xmol polyMoleculeCloud_0.xmol
-------------------------------------------------------------------
#!/bin/bash
```

```
printUsage () {
cat << EOF
   Usage: $0 moleculeName inputFileName.xmol newFileName.xmol
EOF
}

if ! [ $# -eq 3 ] ; then
    printUsage
    exit 1
fi


sed -n "/$1 /p" $2 > $3

n=‘grep -c "" $3‘

touch tempFile.$$

echo "$n" > tempFile.$$

sed -n ’2p’ $2 >> tempFile.$$

cat $3 >> tempFile.$$

mv tempFile.$$ $3
-----------------------------------------------------------------



Script: extractXMOLDelete
Description: to delete a certain molecule species from a .xmol file
Usage: extractXMOLDelete M polyMoleculeCloud.xmol polyMoleculeCloud_water.xmol
-----------------------------------------------------------------
#!/bin/bash

printUsage () {
cat << EOF
   Usage: $0 moleculeName inputFileName.xmol newFileName.xmol
EOF
}

if ! [ $# -eq 3 ] ; then
    printUsage
    exit 1
fi


sed ’1,2d’ $2 > tempFile1.$$

sed -n "/$1 /!p" tempFile1.$$ > tempFile2.$$


n=‘grep -c "" tempFile2.$$‘
```

```
echo "$n" > $3

sed -n '2p' $2 >> $3

cat tempFile2.$$ >> $3

rm tempFile1.$$ tempFile2.$$
```

7 References

[1] D. C. Rapaport. *The Art of Molecular Dynamics Simulation.* Cambridge University Press, 2nd edition, 2004.

[2] Martin Chaplin. Water structure and science. Online: http://www.lsbu.ac.uk/water/models.html.

[3] J. L. F. Abascal and C. Vega. A general purpose model for the condensed phases of water: TIP4P/2005. *Journal of Chemical Physics*, 123(23):234505, 2005.

[4] M. K. Borg, G. B. Macpherson, and J. M. Reese. Controllers for imposing continuum-to-molecular boundary conditions in arbitrary fluid flow geometries. *Molecular Simulation*, 36(10):745–757, 2010.