Worksheet 2: Statistical Mechanics and Molecular Dynamics

Kartik Jain Rudolf Weeber

12.11.2018
Institute for Computational Physics, University of Stuttgart

Contents

1 General Remarks **1**

2 Statistical Mechanics **2**

3 Molecular Dynamics: Lennard-Jones Fluid **3**
3.1 Lennard-Jones Potential 3
3.2 Reduced Units 4
3.3 Lennard-Jones Billards 5
3.4 Periodic Boundary Conditions 6
3.5 Lennard-Jones Fluid 9

1 General Remarks

- Deadline for the report is **Monday, November 26, 2018, noon**
- On this worksheet, you can achieve a maximum of 20 points.
- The report should be written as though it would be read by a fellow student who attends to the lecture, but does not do the tutorials.
- To hand in your report, send it to your tutor via email. Please ensure that you send it to the tutor who is supervising you during exercises i.e. Kartik on Thursday and Rudolf on Friday.
  - Rudolf (`weeber@icp.uni-stuttgart.de`)
  - Kartik (`kjain@icp.uni-stuttgart.de`)

- Please attach the report to the email. For the report itself, please use the PDF format (we will *not* accept MS Word doc/docx files!). Include graphs and images into the report.
- If the task is to write a program, please attach the source code of the program, so that we can test it ourselves.
- The report should be 5–10 pages long. We recommend using LaTeX. A good template for a report is available on the home page of the lecture at `https://www.icp.uni-stuttgart.de/~icp/Simulation_Methods_in_Physics_I_WS_2018/2019#LaTeX`.
- The worksheets are to be solved in groups of **two people**. We will not accept hand-in-exercises that only have a single name on it.

2 Statistical Mechanics

Task (2 points)

- Consider a system A that consists of subsystems A_1 and A_2, with numbers of possible configurations $\Omega_1 = 10^{31}$ and $\Omega_2 = 10^{28}$, respectively. What is the number of configurations available to the combined system? Also, compute the entropies S, S_1 and S_2.
- By what factor does the number of available configurations increase when $1\,\mathrm{m}^3$ of neon at 1 atm and 298 K is allowed to expand by 1% at constant temperature? (Neon should be treated as an ideal gas here.)
- By what factor does the number of available configurations increase when an energy of 100 kJ is added to a system containing 1.0 mol of particles at constant volume and $T = 400\,\mathrm{K}$?

(Adapted from: Frenkel, Smit: *Understanding Molecular Simulation*)

Task (2 points)

(**Thermodynamic Variables in the Canonical Ensemble**) Starting with an expression for the Helmholtz free energy F as a function of N, V, T

$$F = -k_B T \ln \left[Z(N, V, T) \right] \tag{1}$$

one can derive all thermodynamic properties. Show this by deriving equations for intrinsic energy U, pressure p, and entropy S.
(Taken from: Frenkel, Smit: *Understanding Molecular Simulation*)

Task (2 points)
(**Ideal Gas**) The canonical partition function of an ideal gas consisting of monoatomic particles is

$$Z(N,V,T) = \frac{1}{h^{3N}N!}\int \mathrm{d}\Gamma \exp\left[-\beta H\right] = \frac{V^N}{\lambda^{3N}N!} \tag{2}$$

in which $\beta = 1/(k_B T)$, $\lambda = h/\sqrt{2\pi m/\beta}$ and $\mathrm{d}\Gamma = \mathrm{d}q_1 \ldots \mathrm{d}q_N \mathrm{d}p_1 \ldots \mathrm{d}p_N$. Derive expressions for the following thermodynamic properties:

- $F(N,V,T)$ (hint: $\ln(N!) \approx N\ln(N) - N$)
- C_V (heat capacity at constant volume)

(Taken from: Frenkel, Smit: *Understanding Molecular Simulation*)

3 Molecular Dynamics: Lennard-Jones Fluid

On this worksheet, you will do Molecular Dynamics simulations as they are commonly used to simulate molecular systems. The system that is to be simulated is a fluid of particles at different densities interacting via the Lennard-Jones potential.

All files required for this tutorial can be found in the archive `templates.tar.gz` that can be downloaded from the lecture's homepage.

After you have implemented the Lennard-Jones potential in the task below, you will start with a program very similar to the program that simulated the solar system from the last worksheet, and successively improve the program until it is a fully-fledged and fast Molecular Dynamics simulation program. Note that it is recommended to use the C programming language as Python is too slow for this computationally demanding task. You are however free to chose Python, if you like to.

3.1 Lennard-Jones Potential

We are going to simulate a system consisting of particles interacting with the Lennard-Jones (LJ) potential V_{LJ}, which has been originally derived for a system of noble gas atoms (see Fig. 1 on the following page):

$$V_{\mathrm{LJ}}(r) = 4\epsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right) \tag{3}$$

Here, r denotes the distance between a pair of particles, while σ and ϵ are constants determining the spacial range and energy scale of the potential. On very short distances

of the order of the atom radius, the Pauli principle disallows the atoms to overlap, which results in a strongly repulsive potential that should be $\propto e^{-r}$. However, the actual functional form of the core does not really influence the results as long as it is strong enough, therefore the computationally simpler term $\propto r^{-12}$ is used. On intermediate distances, the interaction is slightly attractive, which corresponds to the van-der-Waals-interaction of induced dipoles. It can be shown that this interaction should be $\propto -r^{-6}$, which is the second term in Equation 3. For long distances, the interaction is negligible.

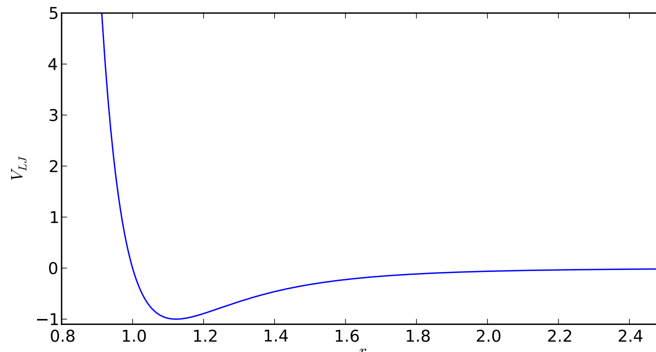

Figure 1: The (12,6)-Lennard-Jones potential ($\sigma = 1$, $\varepsilon = 1$)

3.2 Reduced Units

For simple liquids composed of point-like particles which interact with the same pairwise-additive potential of a general form:

$$U(r) = \varepsilon\phi(\sigma/r)\,, \tag{4}$$

where ε and σ are constants and ϕ is a smooth and differentiable funcion, it is possible to define a set of dimensionless reduced units such as $x^* = x/\sigma$, $V^* = V/\sigma^3$, $T^* = k_\mathrm{B}T/\varepsilon$ and $p^* = p\sigma^3/\varepsilon$. Within this set of reduced units, all systems interacting with potentials of the form given by equation (4) follow one single universal equation of state. This law is called the *theorem of corresponding states*. Therefore, results of simulations of simple fluids are usually given in these reduced units. One particular implication of the theorem of corresponding states is that we can perform all simulations with $\varepsilon = 1.0$ and $\sigma = 1.0$ and if desired, the results can be transfered to other values of interaction parameters. For computational precision, it is desirable to choose the reduced units in the natural units of the problem under consideration, which makes all measured quantitites on the order of unity.

Therefore, in all the following tasks, we will assume that $\varepsilon = 1$ and $\sigma = 1$.

Hint The archive contains a python script `plot_samples.py`, which demonstrates how to make nice-looking plots. For a more complete introduction to plotting, have a look at `http://matplotlib.org/users/pyplot_tutorial.html`.

Task (2 points)

- Implement a Python function `compute_lj_potential(r_ij)` that returns the Lennard-Jones potential (for $\varepsilon = 1$ and $\sigma = 1$) for two particles with the distance vector `r_ij` (which is a 3d-Numpy-array).
- Implement a Python function `compute_lj_force(r_ij)` that returns the Lennard-Jones force (as a 3D-Numpy-array) for two particles with the distance vector `r_ij`.
- Create a distance vector $(d, 0, 0)$ and let d attain 1000 values in the interval $[0.85, 2.5]$. Compute the LJ force and potential for two particles at these distances and create a plot of the potential and the first component of the force against d.

3.3 Lennard-Jones Billards

A large part of the Python program `ljbillards.py` from the archive is very similar to the sample solution of `solar_system.py` from the last worksheet. It is almost ready to play 3d-Lennard-Jones billards. It sets up five LJ particles ("billards balls") at certain positions and with certain velocities and simulates them for 20 time units. The only thing that is missing are the functions from the previous task.

The program creates two plots: A projection of the trajectories onto the x-y-plane (remember: the system is 3d now!) and a plot of the total energy of the system over time. The latter plot can be used to verify that the LJ potential and forces are correct. If they are correct, the energy will be constant, except for a few low-amplitude wiggles when the balls hit each other.

Furthermore, the program creates a file `ljbillards.vtf` in the `VTF`-format. Open the file in a text editor so that you can see what it contains. The file can also be viewed with the 3D-visualization program VMD[1]. To load the file into VMD, either execute

```
vmd ljbillards.vtf
```

in the shell, or load it via the in-program dialog `VMD Main` → `File` → `New Molecule...`.

When you load the file into VMD, the LJ particles are displayed as points which are almost invisible on the screen. To display the particles as colored spheres, open the dialog

[1] `http://www.ks.uiuc.edu/Research/vmd/`

`VMD Main` → `Graphics` → `Representations...` and choose `Draw Style` → `Drawing Method` → `VDW` and `Draw Style` → `Coloring Method` → `Index`.

VMD loads the whole trajectory of the system. The slider in the `VMD Main` window can be used to watch the system evolve in time.

To create a high-quality image of the currently visible scene, open the dialog `VMD Main` → `File` → `Render`. Choose `Tachyon (internal, in-memory rendering)` and press the button `Start Rendering`. This will create a file `vmdscene.tga` that contains a raytraced image of the scene. To convert it to a PNG image file, use ImageMagick:

```
convert vmdscene.tga vmdscene.png
```

Task (2 points)

- Add the Python functions written in the previous task to the program `ljbillards.py`.
- Let the program run and visualize the system with VMD.
- Change the position of the fifth particle (index 4) so that it is hit by the third particle (index 2).
- Although the particles almost behave like billard balls, some of them do not. Which particles and why? Create a plot of the trajectory or a series of images that supports your hypothesis.

3.4 Periodic Boundary Conditions

Our goal on this worksheet is the simulation of a LJ fluid to determine macroscopic bulk properties of the fluid. In practice, however, even the largest modern computers are not able to simulate systems with a macroscopic size with $\approx 10^{23}$ particles, but use systems that are significantly smaller.

One of the problems of fluid simulations with a finite number of particles is how to handle the boundaries of the system. If we would use *open boundary conditions* (PBC) where the available space is infinite, the particles would simply escape into empty space. Therefore, we somehow have to confine the particles in a finite volume. We could create walls on the boundaries that push back the particles when they try to escape. However, this creates boundary effects, and the properties of the fluid close to the walls differ greatly from the bulk properties that we are actually interested in.

The common solution to this problem is to use *periodic boundary conditions*, where the system with a finite size is (virtually) multiplied infinitely into all directions. A particle on the right boundary of the system interacts with the particles close to the left boundary

of the system as though there would be a copy of the whole system (a *periodic image* of the system). The same goes for all directions. An alternative view on the situation is that the system lives on a three-dimensional torus. A particle that leaves the system to the right will turn up on the left. This principle is well-known from various computer games.

The advantage of periodic boundary conditions is, that the system is virtually infinite - the environment of the system is the system itself, so there are no boundary effects.

Note that by introducing the periodic boundary conditions, we have also introduced a new parameter of the system, namely the system size L, and we can define the volume of the system $V = L^3$, and the number density $\rho = \frac{N}{V}$, where N is the number of particles in the system.

To visualize a system with PBC, VMD has different functions. These functions can be accessed via the VMD console that you can open by choosing `VMD Main` → `Extensions` → `Tk Console`. This console allows you to type the following commands:

`pbc box` which will draw a box at the boundaries of the system.

`pbc wrap -all` which will wrap all coordinates of the system into the central image. This is only needed when the coordinates can also be outside the central image in the first place.

However, first of all, VMD needs to know about the size of the system. This can be easily done in a VTF file, by adding a line so that it looks like the following:

```
atom 0:1 radius 0.5
pbc 10.0 10.0 10.0
timestep
3.88000009704 2.98000008821 3.97900008821
6.11999990296 5.01999991179 6.01999991179
.
.
```

Long- and Short-Range Interactions

How can we perform simulations with PBC? The trouble is, that there are an infinite number of copies of each particle. If the range of the interactions is not finite, computing the force acting on a single particle becomes very tricky. Such interactions with an infinite range are also called *long-range interactions*, examples for this type of interaction are the Coulomb interaction or gravitation. In fact, there are methods to handle this kind of problem, which you will learn about in the second part of the lecture.

Fortunately, most interactions have a finite range (so-called *short-range interactions*), or at least they converge to zero fast enough so that we can introduce an approximation and

truncate the potential at a certain distance (the *maximal interaction range*, or *cutoff*). In that case, interactions with particles that are further away are simply neglected.

In the case of the Lennard-Jones potential, we can define the *truncated LJ potential* as follows:

$$V'_{\mathrm{LJ}}(r) = \begin{cases} V_{\mathrm{LJ}}(r) - V_{\mathrm{LJ}}(r_{\mathrm{cutoff}}) & r \leq r_{\mathrm{cutoff}} \\ 0 & r > r_{\mathrm{cutoff}} \end{cases}. \tag{5}$$

Note that we also shift the potential slightly by $V_{\mathrm{LJ}}(r_{\mathrm{cutoff}})$, so that the potential is continuous at the cutoff. For $r < r_{\mathrm{cutoff}}$ the force due to the truncated LJ potential is the same as due to the full form, otherwise the force is 0. In practice, $r_{\mathrm{cutoff}} = 2.5\,\sigma$ is often used, which we will also do in the rest of this worksheet.

Minimum Image Convention

Now that we know that we use an interaction with a finite range, we can think of how to compute the forces on the particles. The so-called *minimum image convention* states that if the maximal interaction range is less than the system size L, to compute the force that a particle B exerts on particle A, you only have to compute the interaction with a single periodic image of particle B, the *minimum image* that is closest to the particle A. This minimum image of particle B is not necessarily in the same image of the system that particle A is in!

Knowing this, it becomes significantly easier to implement periodic boundary conditions. In particular, the code of the LJ billards from the previous task has to be changed only at a few places to implement periodic boundary conditions. There are several possibilities to implement PBC.

Note that it is *not* necessary to keep the particles in the central image, by folding them back when they leave the central image. Instead, one can also simply let the particles propagate outside of the central image and take into account the PBC only when computing the interactions. Doing so has a few advantages when it comes to computing observables. For visualization, coordinates that are folded back into the central image are more useful. Therefore, VMD supports the function `pbc wrap` that folds all coordinates into the central image.

Task (2 points)

- Modify the LJ potential and force such that it implements the trucated LJ potential (with $r_{\text{cutoff}} = 2.5$).
- Extend the program `ljbillards.py` such that it takes into account periodic boundary conditions.
- Which functions need to be modified to implement them?
- To test the PBC, set up two particles in a system with edge length $L = 10.0$, initial positions $\mathbf{x}_0 = (3.9, 3, 4)$ and $\mathbf{x}_1 = (6.1, 5, 6)$, and initial velocities $\mathbf{v}_0 = (-2, -2, -2)$ and $\mathbf{v}_1 = (2, 2, 2)$. Simulate them for 20 time units. With PBC, the particles should collide. Plot the trajectory.

Hints The following Python functions might be useful to implement periodic boundary conditions: `numpy.fmod()`, `numpy.rint()` and the modulo operator `%`.

3.5 Lennard-Jones Fluid

Now we are ready to simulate a Lennard-Jones fluid!

Pure Python

First, let's implement the simulation in pure Python. The Python program `ljfluid.py` from the archive is an almost complete simulation program that should be very similar to your LJ billards program with PBC.

To simulate a LJ fluid, we simply have to increase the number of particles and increase density. When the number of particles becomes large, setting up the system manually by defining their coordinates is unpractical. On the other hand, if we put in the particles randomly, chances are good that some of the particles overlap (in particular at high denisties), which would lead to extremely high forces possibly leading to an explosion of the system. In our case, we therefore simply set up the particles on a cubic lattice with n sites per dimension, resulting in a lattice with $N = n^3$ sites.

The parameters of this program therefore are not the number of particles N and the system size L, but the number of particles per dimension of the cubic lattice n and the density ρ.

To measure the runtime of your program, use the Unix command `time` as follows:

```
> time python ljfluid.py
.
.
.
real    0m2.866s
user    0m2.778s
sys     0m0.057s
```

In our case, the value of interest is user time.

Task (2 points)

- Extend the program `ljfluid.py` to set up the particles on a cubic lattice with a given number of particles per dimension n at a given density ρ.
- Perform simulations of the LJ fluid with $n = \{3, 4, 5\}$ particles per dimension of the cubic lattice (time step $\Delta t = 0.01$, end time $t_{\max} = 1.0$, density $\rho = 0.7$) and measure their runtimes.

Hints To measure the runtime, it is a good idea to comment out the code that plots the energy of the system!

Pure C/C++

The performance of the simulation in the previous example is not very satisfactory. With such runtimes, we will not be able to perform simulations of several thousands of particles.

Therefore, let's try to implement the simulation in the programming language in C. The program `ljfluid.cpp` is a C program (that uses a few C++ things to simplify it) doing pretty much the same as the Python code in the previous task. However, it is missing the function to measure the total energy of the system.

Study the code! Now that we have already seen the Python code, implementing the program in C is not very hard anymore.

To compile the code, execute

```
g++ -O3 -o ljfluid ljfluid.cpp
```

Once it has successfully compiled, you can execute the program as follows

```
./ljfluid
```

Task (2 points)

- Given the Python code from the previous task, implement the function `compute_energy()` in the C-program `ljfluid.cpp` and let the program output the total energy of the system.
- Run the program and measure the runtime for $n = 5$. Compare the runtime with the pure Python program from the previous task.
- Run the program and measure runtimes for $n = \{5, 6, 7, 8, 9, 10, 11, 12\}$. Plot the runtime t against the number of particles N in the system.
- How does the runtime scale with n?

Mixing C/C++ and Python: Cython

Now, we will mix the C-code and the Python-code to see that it is possible to gain the benefits of both worlds.

The subdirectory `cython1` in the archive contains some files that are necessary for this.

`c_lj.cpp` contains some C-code from the previous task, namely the time-critical functions of the LJ simulation. They are copied from `ljfluid.cpp` almost unmodified. The file `lj.pyx` is written in the language CYTHON[2] and provides the “glue” between C and Python. `setup.py` contains a (very short) python script that helps to compile `lj.pyx` and `c_lj.cpp` into the Python module `lj`. To compile the files, execute the following command

```
> python setup.py build_ext -fi
```

When you have done this, you have generated a Python module `lj.so` that can now be loaded into a Python program. `ljfluid.py` loads this module and executes the simulation, calling the C-functions from Python. Compare this program to the pure Python program and look for the differences!

Task (2 points)

- Copy the function `c_compute_energy()` from the previous task into `c_lj.cpp`, uncomment the corresponding lines in `lj.pyx` and let `ljfluid.py` measure the energy.
- Run the program and measure the runtime for $n = 5$ and $n = 10$. Compare the runtimes with the pure C program.

[2] `http://cython.org`

Cell Lists and Verlet Lists

In the previous tasks, we have noticed that even though we have gained a lot of speed by implementing the core functions in C, the scaling behavior is still very unfavorable and would not allow large systems to be simulated. To do that, it is necessary to modify the actual algorithm. In the lecture, you learned about cell lists and Verlet lists.

Implementing these is out of scope of this worksheet, still we provide a program in the subdirectory `cython2` that adds corresponding functions. The program files are the same as in the previous task, the program can be compiled and run as before. The following modifications have been done compared to the program from the previous task:

- A new global variable `verlet_list` has been added to the C-code. It stores the Verlet list, *i.e.* the particle pairs that are within the cutoff distance plus the skin size.
- The function `rebuild_neighbour_lists(x, size)` has been implemented in the C-code and can be called from Python. The function will recompute the Verlet list and store it in the global variable `verlet_list`. To speed up computation of the Verlet list, it creates cell lists, but these are never stored, but only exist while rebuilding the Verlet list.
- The main loops in the functions `c_compute_forces()` and `c_compute_energy()` are modified to employ the `verlet_list`.
- In the Python function `step_vv()`, it is tested whether the Verlet list has to be rebuilt.

Task (2 points)

- Study the program and the modifications that have been done for the inclusion of Verlet and Cell lists.
- Copy the function `c_compute_energy()` from the previous task into `c_lj.cpp`, uncomment the corresponding lines in `lj.pyx` and let `ljfluid.py` measure the energy. Adapt the function to use the Verlet list like the function `c_compute_forces()`!
- Run the program and measure the runtime for $n = \{5, 10, 15, 20\}$. Compare the obtained runtimes with the programs from the previous tasks.