Worksheet 3: Molecular Dynamics 2 and Observables

Olaf Lenz Fatemeh Tabatabaei Marcello Sega

November 25, 2015
Institute for Computational Physics, University of Stuttgart

Contents

1 General Remarks

- Deadline for the report is **Monday, 7th December 2015**
- In 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 the lecture, but doesn't do the tutorials.
- To hand in your report, send it to your tutor via email.

  - Georg (georg@icp.uni-stuttgart.de) (Thursday 11:30-13:00)
  - Johannes (zeman@icp.uni-stuttgart.de) (Friday 14:00-15:30)
- 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 online.
- The worksheets are to be solved in groups of two or three people.

2 Introduction

On this worksheet, you will continue to do the Molecular Dynamics simulations of the Lennard-Jones (LJ) system that you have started on the last worksheet. While on the last worksheet, the focus was on implementing an MD simulation of an LJ system, on this worksheet you will learn how to set up and analyze a system by measuring some observables and their mean values.

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

You will start with a program very similar to the final program of the last worksheet, *i.e.* an LJ simulation with Verlet and Cell lists, written in C and Python. Again, to compile the C-part of the program, use the following command:

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

As the simulation runs in this worksheet will typically take a few minutes, it is split into two programs. The Python program `ljsim.py` will perform the simulation itself and measure different observables. Compared to the program from the last worksheet, it has been extended by two features. First of all, it doesn't compute the energy and write out the coordinates to the VTF file in every time step, but only every `measurement_stride` time steps. This saves both memory as well as computation time. Furthermore, at the end of a simulation run, the program will write the measured energies into a file `ljsim.dat`. To do that, it uses the Python module `pickle`, which can be used to write almost any Python object into a file and read it from there.

The Python program `ljanalyze.py` can then read this data file and plot the data. Like this, it is not necessary to repeat the costly simulations over and over again, but you can plot the results in various ways by just modifying `ljanalyze.py`.

Throughout this worksheet, you will be asked to perform simulations of a system with the following parameters: $N = 1000$ particles at a density of $\rho = 0.316$ with a time step

$\Delta t = 0.01$. Initially, the particles are set up on a regular cubic lattice and given small random velocities. Later on, you will perform simulations at three different temperatures ($T \in \{0.3, 1.0, 2.0\}$).

3 Saving and Restarting the Simulation

Unfortunately, it is usually not clear beforehand how long a simulation actually needs to be run. Furthermore, as you will see in the following tasks, it might be necessary to turn on or off specific measurements or algorithms after some simulation time has elapsed.

Therefore, it is useful to be able to stop a simulation after some time and save the state of the system, so that you can later restart it from that state, possibly with other algorithms activated.

Task (3 points)

- Study the programs `ljsim.py` and `ljanalyze.py`. Focus on those parts of the program that you do not know yet, in particular reading from and writing to a file.
- Extend the program `ljsim.py` such that you can restart the simulation from the point where it left off in the last run, and simulate the system for some more time.

Hints

- Think about which variables need to be saved in a file to be able to restart the simulation (*i.e.* the *state* of the system) and which variables should/can stay in the Python program itself.
- To plan the program and to establish a workflow of how to perform a whole simulation, it might be a good idea to first read the rest of the worksheet.
- You can store the current state of the simulation in the same data file where the energies are stored, and also restart the simulation from that file. In fact, the measured energies should probably be part of the state, as you want to append the newly measured energies to the end of the old energies.

4 Simple Observables

Performing a simulation is futile if we don't at least measure some observables of the simulated system, so that we can watch how they change over time and compute their average values.

First of all, we want to measure the energy components of the system, *i.e.* the *kinetic energy* E_{kin} and the *potential energy* E_{pot}. The program already computes the total

energy, which is just a sum of both of these components. In a microcanonical ensemble, the total energy should remain mostly constant, but the energy components may vary.

Another interesting observable is the temperature T_m of the system. It can be measured via the equipartition theorem, which states that the kinetic energy per degree of freedom is $1/2\, k_\mathrm{B} T_m$. As point-like particles have 3 translational degrees of freedom each, the temperature can be computed as follows:

$$\frac{1}{2} k_\mathrm{B} T_m = \frac{E_\mathrm{kin}}{3N} \tag{1}$$

Finally, we want to measure the *pressure* P of the system. From statistical mechanics it can be shown that the pressure can be computed from two contributions. One of them is the ideal gas contribution, the other one comes from the interactions. Without giving details, we just state the formula:

$$P = \frac{1}{3V} \left(\sum_{i=0}^{N} m\mathbf{v}^2 + \sum_{i=0,\, j<i}^{N} \mathbf{F}_{ij} \cdot \mathbf{r}_{ij} \right) . \tag{2}$$

where V is the volume of the system and $\mathbf{F}_{ij}$ is the force between particle i and j.

Task (3 points)

- Extend `ljsim.py` such that the energy components are measured and returned from the function `compute_energy()`.
- Add a function `compute_temperature()` that computes the temperature of the system.
- Did you implement the function in the Python- or in the C-part of the program? Explain why you did it in the one and not the other part!
- Add a function `compute_pressure()` that computes the pressure of the system.
- Did you implement the function in the Python- or in the C-part of the program? Explain why you did it in the one and not the other part!

Hints

- All of the observables should be measured whenever the energy was measured in the original program.
- The time series of the various observables should be written to the data file.

5 Equilibration

Imagine you want to measure the density of water at ambient conditions and you have two samples: one stored in a freezer and one on a shelf. If you take the one from the freezer and do the measurement immediately, the result will differ from the one on the shelf. If you leave the frozen sample on a table for some time and let it come to equilibrium with the surroundings, the result will be the same as for the one from the shelf. In a simulation we are in a similar situation. Usually we do not know at the beginning what the system looks like under the simulation conditions. In fact, we often perform the simulation to answer this question. We prepare the system in an arbitrary initial configuration and let it equilibrate with a virtual heat bath.

Equilibration is manifested in a time-drift of all observables. Once the equilibrium state is reached, *all* observables fluctuate around a constant value. Some observables relax to equilibrium quickly while others may take much longer. How long the equilibration takes depends on system properties as well as on how far from equilibrium it is. A system cannot be considered as equilibrated if there exists one single observable which exhibits a time-drift. Note that it is even possible that an observable exhibits a non-monotonic behavior, and that the direction of the time-drift reverses after some time. When you want to compute statistical averages of the observables, you should do so only after the system is equilibrated.

Unfortunately, there is no recipe to determine whether a system has equilibrated or not. Instead, it is necessary to have a thorough look at the plots, and probably also to visualize the system to see what is going on. When the measured quantities do not drift anymore, most probably the system is equilibrated, but this is no guarantee! To decide if we can start collecting data, we have to make sure that the time over which the observables are fluctuating around a constant value exceeds the estimated duration of equilibration considerably. We may also need to apply some physical knowledge of the simulated system.

Often, plots of the raw data of the observables are fluctuating very strongly so that it is hard to see whether the observables still display a time-drift on a scale that is smaller than the fluctuations themselves. To make such a drift visible, one can smooth the curve of the observable. The simplest way to do so are *running averages*. If you have a time series of N measurements of the observable O_i, the running average time series $\hat{O}_i$ is defined by:

$$\hat{O}_i = \frac{1}{M} \sum_{i-M/2}^{i+M/2} O_i \tag{3}$$

where M is the *window size* of the running average, *i.e.* the number of points over which the average is performed. The larger the window size is chosen, the smoother the function becomes.

Task (3 points)

- Let the simulation run for 1000 time units and plot the different observables over time.
- Extend the program `ljanalyze.py` and add a function `compute_running_average(O,M)` that computes the running average, where `O` is a Numpy-Array containing the time series of the observable and `M` is the window size of the running average. The function should return a NumPy array of the time series of the running average.
- Plot the running averages of the pressure, temperature and energy components for window sizes of 10 and 100.
- Explain the behavior of the observables. What is happening in the system?
- At what time do you feel confident that the system is equilibrated? Provide plots or images supporting your claim.
- Extend the program so that it computes and outputs the mean values of the observables after a given time t_{eq}.

Hints

- Visualize the system with VMD to understand what is happening.
- The function for plotting and computing the running averages should be implemented in the program `ljanalyze.py`, so that you do not have to rerun the simulation for every plot.
- To simplify plotting of the running average, the function `compute_running_average` should return an array of the same size as the input array. The values of $\hat{O}_i$ for $0 < i < \frac{M}{2}$ and $N - \frac{m}{2} < i < N$ can be set to `NaN`.

6 Molecular Dynamics at a Desired Temperature

Since we are using the energy-conserving velocity Verlet algorithm while keeping a constant number of particles in a box of constant volume, we are simulating the microcanonical NVE ensemble by construction. However, experiments are typically performed at a given constant temperature. To be able to compare simulations to experimental data, one would prefer to simulate the system at a predefined temperature, rather than at a predefined total energy, *i.e.* to simulate a canonical NVT ensemble.

Therefore, we need to be able to keep the temperature constant, *i.e.* we need an extension to the MD algorithm that mimic KS a heat bath the particles are coupled to. Such an

algorithm is called a *thermostat*. There are a number of different thermostat algorithms with different advantages and disadvantages.

The simplest thermostat is the *velocity rescaling* thermostat, which is based on the equipartition theorem (see eq. (1)). The idea of the velocity rescaling algorithm is to first measure the current temperature and then simply to scale all velocities by a factor so that the desired temperature is reached.

One important problem of the velocity rescaling thermostat is that it destroys the Maxwell-Boltzmann distribution of velocities in the system. This means that although a simulation keeps the temperature constant, it does not really produce the NVT ensemble. On the next worksheet, you will learn about more advanced thermostats that actually simulate the canonical NVT ensemble.

Here, we will first use the velocity rescaling thermostat to drive the system to the desired temperature, but then turn off the thermostat and measure the properties of the system in the microcanonical NVE ensemble with an average temperature close to the desired temperature. Note that the equilibration might still not have finished even when the temperature is stable.

Task (3 points)

- From equation (1), derive how to compute the factor with which to multiply the velocities to reach the desired temperature T.
- Extend `ljsim.py` such that the velocities are rescaled to yield the desired temperature directly after the observables are measured.
- Did you implement the velocity scaling in the Python- or in the C-part of the program? Explain why you did it in the one and not the other part!
- Run the simulation with the standard parameters (see above) at the desired temperatures $T \in \{0.3, 1.0, 2.0\}$ and plot the evolution of the various observables. Run the simulation long enough until you observe no more time drift in any of the observables.
- Explain the observed behavior.

Hints It should be possible to turn the thermostat on or off via a constant (or maybe even a command line parameter?), so that you can later measure the properties of the system without the thermostat.

7 Setting up and Warming up the System

At the moment, `ljsim.py` sets up the different LJ particles on a cubic lattice. This avoids excessively large energies due to overlapping particles. However, this is a very

artificial state of the system and only works in the particular case of an LJ system where all particles have the same size. Therefore, we want to learn how this can be done cleanly.

In general, it is very difficult to set up an arbitrary system that doesn't have such overlaps. Instead, in the most cases, it is sufficient to set up a system with overlapping particles in an arbitrary fashion, *e.g.* by placing the particles randomly, and then using a *warmup* to get rid of the overlaps.

There are different possibilities how to do a warmup. For example, one can try to minimize the energy via a gradient descent algorithm (or any other optimization algorithm), or one can slowly let the particles grow until they have reached their final size. Another frequently used procedure is called *force capping*, which works as follows: During the warmup phase, one performs a "normal" simulation of the system but limits the forces acting on a single particle to a maximum value. This way, overlapping particles feel a force pushing them apart, but these forces cannot become overly large. Once there are no very large forces left, the warmup procedure has to be turned off, as it modifies the physics of the system.

In practice, there are a few caveats with this procedure:

1. On the one hand, even with force capping, velocities may still become quite large when particles overlap, so that particles strongly bump into other particles, which in turn results in large forces. Therefore, when capping the forces, you should rescale the velocities regularly.
2. Furthermore, the forces that occur in a system depend on the density and kinetic energy of the system. Therefore, it is sensible to grow the value at which the forces are capped after a while when the desired capping cannot be achieved.

Task (2 points)

- Extend `ljsim.py` such that the components of the force acting on a particle can be limited to a maximum (absolute!) value $F_{\max}$.
- Did you implement the force capping in the Python- or in the C-part of the program? Explain why you did it in the one and not the other part!
- Initially, set $F_{\max} = 10$ and increase it by 10% whenever the measurements are done.
- Change the setup procedure of the simulation such that the particles are placed at random positions.
- Test whether the new setup with warmup works as intended.

Hints

- It should be possible to turn force capping on or off via a constant (or maybe even a command line parameter?) so that you can later measure the properties of the system without force capping.
- When running the warmup, you need to know whether or not the warmup is finished so that there are no overly large forces anymore. One possibility to solve this problem is to run the warmup not for a specified time, but simply until no force is capped anymore.
- When you measure observables during the warmup, their values will be bogus, so do not store the measurements while force capping is active. However, it might still be interesting to output the measurements to the screen to see how the warmup progresses.

8 Radial Distribution Function

An observable containing information about the structure of the system is the *radial distribution function* (RDF) (or pair correlation function) $g(r)$. The value of the RDF at a given distance r is the probability to find a particle in an infinitesimal distance interval $[r, r + dr]$ to another particle, divided by the probability to find a particle in that distance in an ideal gas. Formally, it is defined by

$$g(r) = \frac{1}{\rho 4\pi \mathbf{r}^2 \mathrm{d}r} \sum_{ij} \langle \delta(r - |\mathbf{r}_{ij}|) \rangle \ . \tag{4}$$

The RDF describes the structure of the system quantitatively. The nice thing about the RDF is that it can be measured experimentally from scattering experiments, and one can compare experimental and simulation results. From a theoretical point of view, it has the advantage that the RDF contains all information about the pair structure of the system. Although it cannot fully account for higher order contributions, all of the system's observables only dependent on pair correlations can be calculated when the RDF is known.

Figure 1 displays typical plots of the RDF in different phases. In an ideal gas, $g(r) = 1$ for all values of r. In a real gas, this is not really true: the particles do interact on short length scales, as they have an excluded volume interaction, *i.e.* they cannot overlap. Therefore, the function is zero for distances below the contact distance of two particles and jumps to a finite value at contact. The value of the function at contact increases with growing density of the system. In the gas phase, the function decays quickly at larger distances, and the RDF approaches 1, which means that from he point of view of the particle, the rest of the system is just a medium with the given density, but no further structure. In the liquid phase, the function decays in several oscillations from the first maximum at contact. In the fluid phase, the oscillations decay fast, while a clear structure at lower distances remains. In the solid phase, these oscillations look

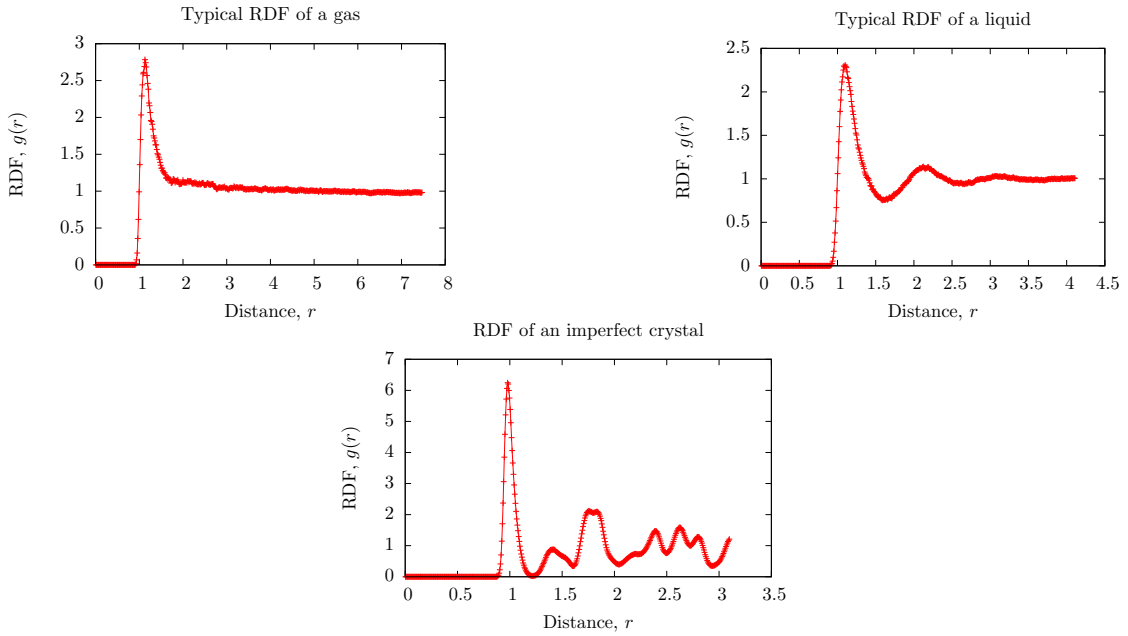

Figure 1: Examples of radial distribution functions obtained from simulations.

fundamentally different from the liquid phase. The amplitude of the oscillations decays only slowly with growing distance. Thus, the RDF gives evidence for the fact that liquids only display a short range order, whereas solids (*i.e.* crystal structures) also have a distinct long range order.

From Eq. (4), it is not immediately obvious how to compute the RDF from a given set of particle coordinates. The solution to this problem is to create histograms of the distance vectors $\mathbf{r}_{ij}$ to approximate $\frac{1}{\mathrm{dr}} \sum_{ij} \langle \delta(r - |\mathbf{r}_{ij}|) \rangle$.

Task (3 points)

- Extend your simulation program `ljsim.py` to measure the radial distribution function of the current state for distances $0.8 < r < 5.0$. The histogram should use 100 bins.
- Extend the analysis program `ljanalyze.py` to compute the averaged equilibrium RDF.

Hints

- Implementing the RDF completely in Python has the disadvantage that it is relatively slow, as it needs to compute all distances.

- Implementing the RDF completely in C has the disadvantage that you need to implement histograms in C and return them to Python, which is tedious.
- Therefore, it might be a good idea to implement the RDF partly in C and partly in Python. For example, one can implement a function `c_compute_distances()` in C, that computes and returns an array of the particle distances, and then use the Python function `numpy.histogram()` to compute the histogram.

9 Measuring Equilibrium Mean Values of the Observables

Task (2 points)

- Run simulations of the LJ fluid from a random initial configuration with the standard parameters for the desired temperatures $T \in \{0.3, 1.0, 2.0\}$. Equilibrate the system and document the equilibration with suitable plots. Measure the equilibrium mean pressure, energy components, and averaged RDF for at least 1000 time units.
- Describe the different steps you used to perform the simulations: which programs did you call, which constants did you change for the different steps, what was the outcome?

One way to calculate the pressure in computer simulations is via the virial theorem. The virial of a system is defined as

$$G = \sum_i^N \mathbf{p}_i \cdot \mathbf{r}_i,$$

Where $\mathbf{p}_i$ are the momenta and $\mathbf{r}_i$ the positions of the particles.

The virial theorem states that if $\langle \frac{dG}{dt} \rangle = 0$, then

$$3Nk_BT = 2\left\langle \sum_i^N \frac{\mathbf{p}_i^2}{2m_i} \right\rangle = -\sum_i^N \langle \mathbf{F}_i \cdot \mathbf{r}_i \rangle, \tag{5}$$

Here m_i are the masses of and F_i are the forces acting on the particles. $\langle \cdot \rangle$ denotes the time average $\langle \cdot \rangle = \lim_{\tau \to \infty} \frac{1}{\tau} \int_0^\tau \cdot dt$. The condition $\langle \frac{dG}{dt} \rangle = 0$ holds for systems where the particles are bound or confined to a finite volume. This is fulfilled for typical molecular dynamics simulations as well as for other interesting systems, e.g. stars where the particles are gravitationally bound.

For an ideal gas, where there are no interactions between the particles, it follows from the equation of state $PV = Nk_BT$ that

$$\sum_i^N \langle \mathbf{F}_i \cdot \mathbf{r}_i \rangle = 3PV.$$

Basically this states the only forces on the particles are due to the pressure of the container. For an interacting gas there will be deviations because of the interactions between the particles.

Task (1 point)

Suppose we have a system with only pair interactions $\mathbf{F}_{ij} \equiv \mathbf{F}_{ij}(\mathbf{r}_{ij})$, where $\mathbf{r}_{ij}$ is the distance between particles i and j. Show that:

$$P = \frac{1}{3V}\left(\left\langle \sum_{i=0}^{N} \frac{\mathbf{p}_i^2}{m_i} \right\rangle + \left\langle \sum_{i=0,\, j<i}^{N} \mathbf{F}_{ij} \cdot \mathbf{r}_{ij} \right\rangle\right) . \tag{6}$$

Hint: Split $\mathbf{F}_i$ in (5) into an interaction part and an ideal gas part.