Exercise 4

Molecular Statistics, Week 4

2014

1 Introduction

The goals of this exercise:

1. Extend your MD-program to three dimension using the supplied module.
2. Implement a 3D Velo-Verlet solver.
3. Extend your program with functions to calculate the following properties:
   (a) Kinetic energy
   (b) Total energy
   (c) Temperature
   (d) Pressure
4. Visualize your results in a video.

You should read the Numpy handout available on the course website.

1.1 Lennard-Jones potential and periodic boundary conditions

The potential we are using this week is almost the same Lennard-Jones potential as we saw last week. You may recall from your simulation last week, that the calculation of the particle pair-interactions took a long time to compute, even with only 10-20 particles. To speed up the calculation time we can make the approximation that only particles closer than a certain minimum distance r_{cut} are interacting. This is a justified approximation, since particles that are far apart have weak interactions. Here we set $r_{\text{cut}} = 2.5$. Using this approximation, we can define the the potential energy between two interacting particles i and j by:

$$U_{ij} = \begin{cases} 4\left[\left(\frac{1}{r_{ij}}\right)^{12} - \left(\frac{1}{r_{ij}}\right)^{6}\right] - E_{\text{cut}} & \text{if } r_{ij} < r_{\text{cut}} \\ 0 & \text{if } r_{ij} \geq r_{\text{cut}} \end{cases} \tag{1.1}$$

The derivatives of the potential energy are used to calculate the forces between particles:

$$\mathbf{F} = -\nabla U = -\left(\frac{\partial U}{\partial x_1}, \frac{\partial U}{\partial y_1}, \frac{\partial U}{\partial z_1}, \frac{\partial U}{\partial x_2}, \frac{\partial U}{\partial y_2}, \frac{\partial U}{\partial z_2}, \ \cdots\ , \frac{\partial U}{\partial x_n}, \frac{\partial U}{\partial y_n}, \frac{\partial U}{\partial z_n}\right) \tag{1.2}$$

The x-components of the force between two interacting particles i and j are:

$$-\frac{\partial}{\partial x_i}U_{ij} = \begin{cases} -48\ \frac{x_j - x_i}{r_{ij}^2}\left[\left(\frac{1}{r_{ij}}\right)^{12} - 0.5\left(\frac{1}{r_{ij}}\right)^{6}\right] & \text{if } r_{ij} < r_{\text{cut}} \\ 0 & \text{if } r_{ij} \geq r_{\text{cut}} \end{cases} \tag{1.3}$$

$$-\frac{\partial}{\partial x_j}U_{ij} = \begin{cases} 48\ \frac{x_j - x_i}{r_{ij}^2}\left[\left(\frac{1}{r_{ij}}\right)^{12} - 0.5\left(\frac{1}{r_{ij}}\right)^{6}\right] & \text{if } r_{ij} < r_{\text{cut}} \\ 0 & \text{if } r_{ij} \geq r_{\text{cut}} \end{cases} \tag{1.4}$$

The y- and z-components are of course similar.

1.2 Velo-Verlet integration and periodic boundary conditions

Last week your setup confined the particles in the simulation inside a box with hard walls. Today's exercise uses periodic boundary conditions in the minimum-image convention. You don't have to implement this code, since it's already implemented in the FORTRAN library you will be using. The Velo-Verlet integrator you have to implement is thus very simple (and identical to last week). The Velo-Verlet equations for integrating the positions and velocities are (in the x-direction for one particle):

$$x(t+dt) = x(t) + dt\ v_x(t) + 0.5\ dt^2\ a_x(t) \tag{1.5}$$
$$v_x(t+dt) = v_x(t) + 0.5\ dt\,[a_x(t) + a_x(t+dt)] \tag{1.6}$$

In the program you will be implementing today, we will again set the mass of the particles to 1, such that the value of the acceleration is equal to the value of the force.

Let's recap: The Velo-Verlet algorithm uses the forces, velocities and positions from the current position to calculate the forces, velocities and positions after the next time-step. In short it is a three-step procedure:

1. Calculate new positions using Eq. 1.5
2. Calculate new forces using Eq. 1.3-1.4
3. Calculate new velocities using Eq. 1.6

The resulting forces, velocities and particle positions are then saved as input for the next Velo-Verlet integration step.

1.3 Kinetic energy

The total kinetic energy of the system can be calculated as:

$$E_{\text{kin}} = \sum_i \frac{1}{2} m_i |\mathbf{v_i}|^2 \tag{1.7}$$

In our simulation m_i is set to 1, so the above reduces to:

$$E_{\text{kin}} = \frac{1}{2} \sum_i |\mathbf{v_i}| \cdot |\mathbf{v_i}| \tag{1.8}$$

Hint: If you do this in Numpy, the above summation is simply carried out as (something like):

```
e_kin = np.sum(V * V) * 0.5
```

Why is this true? What does V * V do and what does np.sum() do?

1.4 Conservation of energy

If your simulation is successful the total energy will be conserved. That is:

$$E_{\text{total}} = E_{\text{pot}} + E_{\text{kin}} \approx \text{constant} \tag{1.9}$$

Conservation of energy can sometimes fail during a MD simulation. Often this is due to the timestep dt being too large, which causes the numerical Velo-Verlet integration to fail.

1.5 Instantaneous temperature

The first thermodynamic property we will derive from our simulation is the instantaneous temperature, that is the temperature as a function of time, $T(t)$. You will sometimes see the temperature defined in terms of the average kinetic energy *per* degree of freedom in a system:

$$\langle E_\alpha \rangle = \frac{1}{2} k_\mathrm{B}\, T \tag{1.10}$$

where α denotes the degrees of freedom in the system. The above states that each degree of freedom gives a contribution to the temperature which is proportional to the kinetic energy of the individual particles. Instead of this formulation it is more practical for our purpose to write the temperature in its instantaneous form:

$$T(t) = 2\frac{\langle E_\alpha(t) \rangle}{k_\mathrm{B}\, N_\alpha} \tag{1.11}$$

$$= \sum_i \frac{m_i |\mathbf{v_i}| \cdot |\mathbf{v_i}|}{k_\mathrm{B}\, N_\alpha} \tag{1.12}$$

where N_α is the number of degrees of freedoms, i.e. $N_\alpha = N_\mathrm{dimensions} \cdot N_\mathrm{particles}$. Notice, that we use fundamental units and simple masses, so $k_\mathrm{B} = 1$ and $m_i = 1$.

Hint: If you do this in Numpy, this can be written easily as:

```
temperature = 2.0 * e_kin / np.size(V)

# or

temperature = np.mean(V * V)
```

Why this is true? What does np.size() calculate? What does np.mean() calculate? Why do both implementations the yield the same result?

1.6 Instantaneous Pressure

The instantaneous pressure in a Van der Waal gas simulation can be calculated as:

$$P(t) = \rho\, T(t) + \frac{1}{3V_\mathrm{box}} \sum_{i>j} \mathbf{F_{ij}} \cdot \mathbf{r_{ij}} \tag{1.13}$$

$$= \rho\, T(t) + \frac{1}{3V_\mathrm{box}} W_{vir} \tag{1.14}$$

where V_box is the volume of the box, ρ is the particle-density of the box and $W_{vir} = \sum_{i>j} \mathbf{F_{ij}} \cdot \mathbf{r_{ij}}$ is the Clausius virial function (or *virial* for short) of the system.

Since we were not using a standard Lennard-Jones potential (we use a small cut-off distance), we have to add a small correction (the *tail correction*) to Eq. 1.14 in order to account for the interaction we are neglecting:

$$\Delta P_\mathrm{tail} = \frac{16}{3}\pi\rho \left[\frac{2}{3} \left(\frac{\sigma}{r_\mathrm{cut}} \right)^9 - \left(\frac{\sigma}{r_\mathrm{cut}} \right)^3 \right] \tag{1.15}$$

Here we simply set $\sigma = 1$. Remember $r_\mathrm{cut} = 2.5$.

2 Numpy Exercises

When working with Python "in the real world" you will most often use Numerical Python (NumPy) and vectorization. This is because it is faster and easier. Firstly we'll import the module `numpy`.

```
import numpy as np
```

Numpy is a fast library of functions and data types. It is mostly used for substituting the normal Python List for a new data type called `ndarray`.
As an example, generating an empty list of zeros with the two different methods:

```
list = [0.0 for i in range(5)] # Normal list
array = np.zeros(5)            # Numpy array
```

A huge advantage of using Numpy arrays is vectorization commands.
You can perform mathematical operations on every element at once in a Numpy array, just like you would on a single element. E.g.

```
V = np.zeros(5)
%V += 5
V = V + 5
```

This however doesn't work with normal Python Lists.

2. Execute and evaluate the above command. What is the result of V?

Note: When you create a Numpy array, you always specify a size. This also represents how much computer memory is allocated by Numpy. For this reason it is not possible to use the `append` method, as with the normal Python list.

If you are not sure how big your array is going to be, you can create a list, append elements, and then after all elements are appended, convert it too a Numpy array as following.

```
L = []
for i in range(10):
    L.append(i)
L = np.array(L)
```

3 Molecular Dynamics simulation

Before you start this weeks exercise, please make you you have downloaded the material from the course website.

Before you start, download the two files from absalon, `md_header.py` and `forces.so`. Put both files in the same folder as the Python program you are working on today. `md_header.py` contains functions initialize the particles, calculates the Lennard-Jones forces and energy as well as a function to visualize a 3D video of your simulation. "forces.so" implements the calculation of the Lennard-Jones forces in a very efficient FORTRAN routine You don't have to worry about the technical details of this, just know that the FORTRAN routine is about 300 times faster than the same code running in Python.
Make a new Python program, and name it `exercise_4.py`. In the first two lines import Numpy and the three functions from the downloaded module.

```
1 import numpy as np
2 import md_header as md
3 import video3d
```

Last week you wrote your own code to initialize the particle positions, velocities, box_width etc. The initialize_particles() function you imported is a bit more advanced. The new initialize_particles() function takes as arguments the number of particles, the given temperature and a desired particle density.

The temperature option is used to scale the particle velocities appropriately (see section 1.5) and the rho keyword is used to scale the width of the box according to the number of particles (we need to be able to specify a particle density in order to calculate the pressure correctly, see section 1.6)

1. Initialize your system with the following code:

   ```
   1 R, V, F, box_width = initialize_particles(n_particles, temperature, rho)
   ```

   Suggested values are temperature = 2.0 and rho = 0.1. Don't worry you will get a chance to play around with these values later

You will notice that initialize_particles() returns four values. `R` is the coordinate matrix. It is an Nx3 matrix, which holds for each of N particles the X, Y and Z coordinates. Likewise `V` and `F` holds the initial velocities and forces, respectively. `box_width` is the width of the scaled box, which today you cannot specify this yourself, because it is based on the density of the particles.

Now it's time to test the initialize_particles function.

2. What comes out of `initialize_particles(n_particles, temperature, rho)` if you set `n_particles = 10`, `temperature = 2.0` and `rho = 0.1`? What do `R`, `V` and `F` contain? Can you explain why?

The second new function you got from the md module is `lennard_jones()` - which can be used like this:

```
1 e_pot, F, w_vir = lennard_jones(R, box_width)
```

`lennard_jones` takes as arguments the position matrix `R` and `box_width`. Just like last week, the `lennard_jones` function returns the potential energy `e_pot`, a force matrix `F`, but now also the virial of the system, `w_vir` (see section 1.6).

3. Now it's time to test the `lennard_jones` function. Instead of giving R as input to lennard_jones, call the function with the following `R_test` matrix:

   ```
   1 R_test = np.array([[0.7, 2.3, 0.7], [0.7,  0.7, 2.3], [0.7, 0.7, 0.7]])
   ```

   and `box_width = 3.10`. That is:

   ```
   1 print lennard_jones(R_tests, 3.10)
   ```

The potential energy is -0.635, the virial -3.732. The forces matrix (rounded values):

```
array([[-1.16    1.24   -0.09]
       [-1.16   -0.09    1.24]
       [ 0.0     0.0     0.  ]])
```

The z-components of the force are correctly 0 (why?).

Now it is time to implement the Velo-Verlet solver from last week, but using the new module and Numpy.

4. Somewhere in your program file put the solver code in:

```
for n in range(n_steps):
    # Step 1: Calculate new positions
    R = R + ...

    # Step 2: Calculate new forces
    F_old = copy.copy(F)
    e_pot, F, w_vir = lennard_jones(R, box_width)

    #Step 3: Calculate new Velocities:
    V = V + ...
```

Hint: Look at the last exercise if you need inspiration on where to put the solver code.

5. Initialize your system with 40 particles, `rho = 1.1` and `temperature = 2.0`. Run your simulation for 1000 steps with $dt = 0.001$. Every 10 steps print the potential energy. The potential energy should stabilize around -20 if your solver works properly.

6. Now would be a good time to implement the video function. The new 3D video is used like the other by adding frames like;

```
# Add Frame
video3d.add_frame(R)

# Save movie
video3d.save(box_width, 'my_video')
```

And add frame every 10'th step in the Velo-Verlet step.

7. Run 500 steps and save a video.

You have now successfully implemented an MD simulation of a Lennard-Jones gas. The goal now is to use the equations from section 1 to calculate interesting physical properties of your system.

8. Use the equations from section 1.3 to calculate the kinetic energy. Print the kinetic energy, potential energy and total energy each 10 step.

9. Make three empty lists called Ekin_list, Epot_list and Etot_list. Append

```
Ekin_list = []
Epot_list = []
Etot_list = []

for n in range(n_steps):

    """ Your Velo-Verlet solver code
        is here ...
    """

if (n % 10):
```

```
12      # Calculate e_kin and e_tot
13      # here!
14
15      Ekin_list.append(e_kin)
16      Epot_list.append(e_pot)
17      Etot_list.append(e_tot)
```

10. Run your simulation for 10,000 steps. Do not save a video, since this will take several minutes to compile. Use Pylab to make a plot of the energies vs. time.:

```
pylab.clf()
pylab.plot(Ekin_list, 'r-', label='Kinetic Energy')
pylab.plot(Epot_list, 'b-', label='Potential Energy')
pylab.plot(Etot_list, 'g-', label='Total Energy')
pylab.legend() # Plot labels in figure
pylab.savefig("my_energy.png")
```

    How does the energies evolve if you run your simulation with 40 particles at T = 2.0? Now run your simulation at a lower temperature (maybe T = 0.5). How does the energy change ?

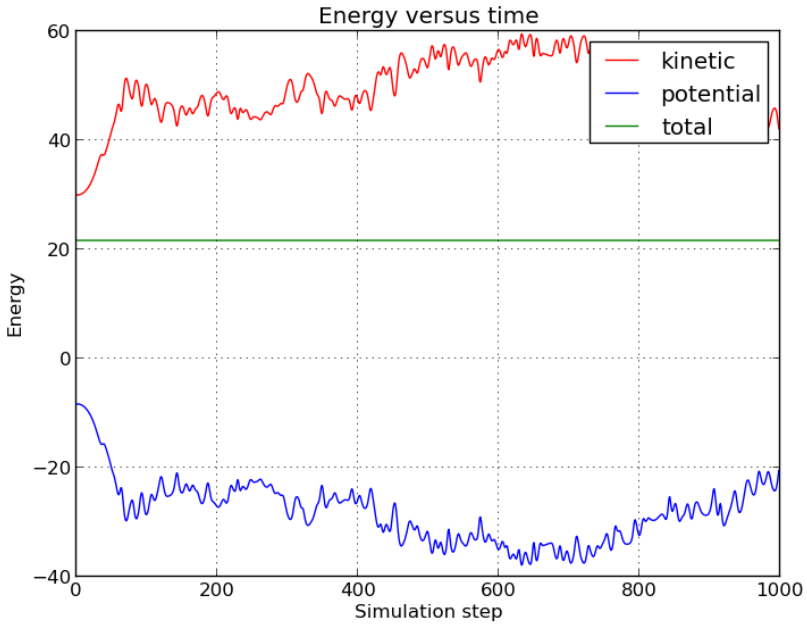

Figure 3.1: Evolution of energies during an MD simulation with temperature = 0.5, 40 particles, rho = 0.1, for 10,000 steps and saving every 10'th step.

11. Use the equations from section 1.5 to calculate the instantaneous temperatures during the simulation.

12. Like in the previous question, make an empty list called T_list and append the current temperature every 10 steps. Use Pylab to make a plot of this list. See Figure 3.2 for example.

    Does the temperature you calculate correspond to the temperature you specified when you initialized your particles? Run your simulation at T = 0.05. What could easily go wrong now?

13. Answer this question before you proceed: Why is the gas a Van der Waals gas and not an ideal gas? I.e. why isn't the pressure simply $p = \frac{NT}{V}$.

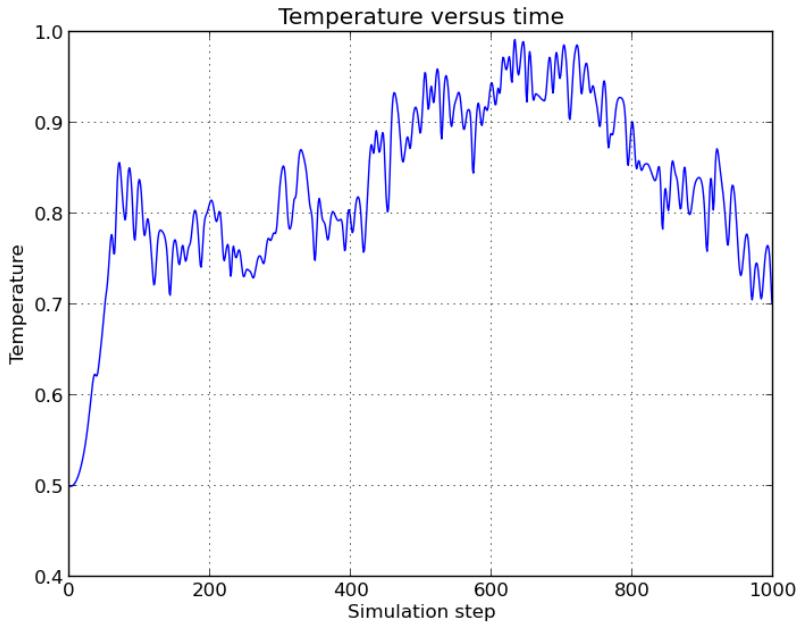

Figure 3.2: Evolution of temperature during an MD simulation with T = 0.5, 40 particles, rho = 0.1.

Use the equations from section 1.6 to calculate the instantaneous pressure during the simulation.

14. Use Pylab to plot the pressure during the simulation, as done in the previous exercises. How does the temperature and particle density affect the pressure? See Figure 3.3 for example.

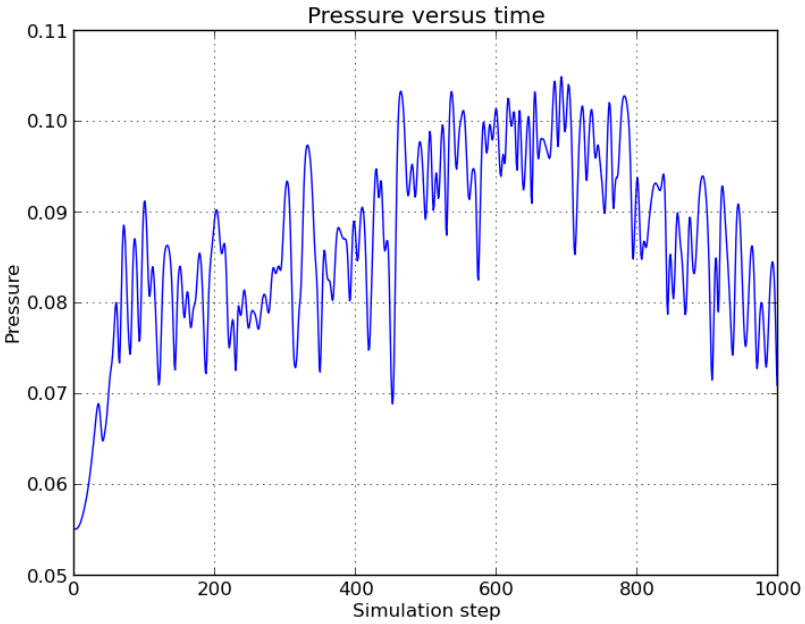

Figure 3.3: Evolution of pressure during an MD simulation with T = 0.5, 40 particles, rho = 0.1.