Exercise 3

Molecular Statistics, Week 3

2014

1 Introduction and Theory

In this exercise you are going to implement the Lennard-Jones potential in our 2D simulation. Figure 1.1 shows the three different potentials we've encountered so far.

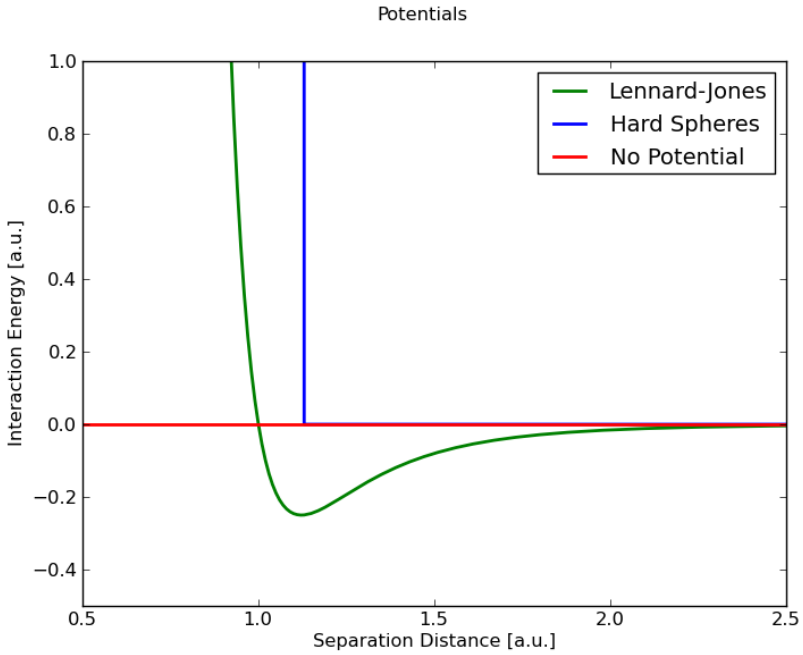

Figure 1.1: The three different potentials we've encountered so far.

1.1 The Lennard-Jones potential energy

The Lennard-Jones potential energy between two interacting particles i and j is given by:

$$U_{ij} = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{1.1}$$

Here r_{ij} is the distance between the two particles. For simplicity, we set $\epsilon = 1$ and $\sigma = 1$, in which case the energy is reduced to:

$$U_{ij} = 4 \left[\left(\frac{1}{r_{ij}} \right)^{12} - \left(\frac{1}{r_{ij}} \right)^{6} \right] \tag{1.2}$$

The total potential energy of the system is then;

$$U_{\text{Total}} = \sum_{i>j} U_{ij} = 4 \sum_{i>j} \left[\left(\frac{1}{r_{ij}} \right)^{12} - \left(\frac{1}{r_{ij}} \right)^{6} \right] \tag{1.3}$$

which computationally requires a double loop over each particle interaction.

1.2 Inter-particle forces in the Lennard-Jones potential

In the past two exercises, the total momentum of the particles was conserved. However in the Lennard-Jones potential, we must calculate the force exerted by the particles upon each other, and use the forces to update the positions and velocities of the particles. First, recall how we are able to calculate the force $\mathbf{F}$ using the energy gradient:

$$\mathbf{F} = -\nabla U = -\left(\frac{\partial U}{\partial x_1}, \frac{\partial U}{\partial y_1}, \frac{\partial U}{\partial x_2}, \frac{\partial U}{\partial y_2}, \ \cdots \ , \frac{\partial U}{\partial x_n}, \frac{\partial U}{\partial y_n} \right) \tag{1.4}$$

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

$$-\frac{\partial}{\partial x_i} U_{ij} = -48 \ \frac{x_j - x_i}{r_{ij}^2} \left[\left(\frac{1}{r_{ij}} \right)^{12} - 0.5 \left(\frac{1}{r_{ij}} \right)^{6} \right] \tag{1.5}$$

$$-\frac{\partial}{\partial x_j} U_{ij} = 48 \ \frac{x_j - x_i}{r_{ij}^2} \left[\left(\frac{1}{r_{ij}} \right)^{12} - 0.5 \left(\frac{1}{r_{ij}} \right)^{6} \right] \tag{1.6}$$

The y-components are, of course, defined the same way.

1.3 The Velo-Verlet solver

The Velocity Verlet (Velo-Verlet) solver is one of many ways to go about integrating the motion of particles. The Velo-Verlet algorithm updates the forces, velocities and positions at the same time. The Velo-Verlet equations for integrating the positions (r) and velocities (v) are given the acceleration (a), for the x-direction of one particle:

$$r_x(t + dt) = r_x(t) + dt \cdot v_x(t) + 0.5 \cdot dt^2 \cdot a_x(t) \tag{1.7}$$

$$v_x(t + dt) = v_x(t) + 0.5 \cdot dt \cdot [a_x(t) + a_x(t + dt)] \tag{1.8}$$

In out simultation, we will set the mass of the particles to 1, so the acceleration is equal to the force (Newtons 2nd law). Let's recap: The Velo-Verlet algorithm uses the forces, velocities and positions from the current time step to calculate the forces, velocities and positions for the next time-step. In short it is a three-step procedure:

1. Calculate new positions, using Eq. 1.7
2. Calculate new forces, using Eq. 1.4
3. Calculate new velocities, using Eq. 1.8

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

2 Exercise

Before you begin, inspect the solution from last week, keep it as a reference, but do not copy everything over. It should more or less contain the following functionality:

```
import numpy as np
import matplotlib.pyplot as plt
import random
import video

def distance(xi, yi, xj, yj):
    return np.sqrt((xi-xj)**2 + (yi-yj)**2)

def initialize_particles(n_particles):
    pos_x = [random.random() for i in range(n_particles)]
    pos_y = [2 * (random.random() - 0.5) for i in range(n_particles)]
    vel_x = [2 * (random.random() - 0.5) for i in range(n_particles)]
    vel_y = [2 * (random.random() - 0.5) for i in range(n_particles)]
    return pos_x, pos_y, vel_x, vel_y

def simulate_step(pos_x, pos_y, vel_x, vel_y, dt):
    for i in range(n_particles):
        if abs(pos_x[i] + vel_x[i]*dt) > 1.0:
            vel_x[i] = -vel_x[i]

        if abs(pos_y[i] + vel_y[i]*dt) > 1.0:
            vel_y[i] = -vel_y[i]

        pos_x[i] += vel_x[i]*dt
        pos_y[i] += vel_y[i]*dt

        for j in range(n_particles):
            if j > i:
                d = distance(pos_x[i], pos_y[i], pos_x[j], pos_y[j])
                if d < r_min:
                    x_temp = vel_x[i]
                    y_temp = vel_y[i]
                    vel_x[i] = vel_x[j]
                    vel_y[i] = vel_y[j]
                    vel_x[j] = x_temp
                    vel_y[j] = y_temp

    return pos_x, pos_y, vel_x, vel_y

n_particles = 40
dt = 0.001
n_step = 1000

X, Y, Vx, Vy = initialize_particles(n_particles)

for n in range(n_steps):

    X, Y, Vx, Vy = simulate_step(X, Y, Vx, Vy)

    if n % 10 == 0:
        video.add_frame(X, Y)

video.save('simulation')
```

Firstly, you will need to write the relevant code/functions to calculate the Lennard-Jones potential energy for a given system.

1. Create a new file called `week3.py`.
2. Define a function called `lennard_jones()`, that takes positions and number of particles as parameters.
3. Loop over all particle interaction (double loop over i and j particles) and calculate the total energy using eq. 1.3.

When you think you have written the Lennard-Jones energy function properly, it's time to test whether your function works correctly or not. Define two lists:

```
n_particles = 2
pos_x_test = [0.0, 0.0]
pos_y_test = [0.0, 1.4]
```

4. Now call the Lennard-Jones function with the two lists as arguments and print the result. If your energy function works, the resulting energy should be -0.4607.

Next, we will extend the Lennard-Jones (LJ) function to also calculate the forces. In our 2D system the force has x- and y-components for each particle. So for this it would be natural to extend your code to store the x- and y-components in two lists called `force_x` and `force_y`, just like the velocities.

5. In the LJ function, initialize two new lists for the forces, `force_x` and `force_x`, containing only zeros, with the length equal to the number of particles. *Hint:*

```
force_x = [0.0 for i in range(n_particles)]
```

6. In the loop when calculating the energy, insert code to calculate the force for each interaction and update the force lists with the corresponding interaction force, from eq. 1.4. Use the following code as inspiration:

```
for i in range(n_particles):
    for j in range(n_particles):
        if i > j:
            energy = energy + ...

            force_x[i] = force_x[i] + ...
            force_y[i] = force_y[i] + ...

            force_x[j] = force_x[j] + ...
            force_y[j] = force_y[j] + ...
```

7. Use the position test lists from the previous question and calculate the forces and energies. The function should return `force_x[0]` $= 0.0$ and `force_y[0]` $= 1.6720$.

Next is to implement the Velo-Verlet solver. The Velo-Verlet solver needs the current forces, velocities and particle positions as input. Your code already initializes the velocities and positions, but doesn't calculate the initial forces.

Last week we used random x- and y-coordinates, within a (-1, 1) box. However, when we are working with a potential it is not a good idea to use completely random coordinates, as the initial gradient might be unphysically large. And because we are working with a potential with $\sigma = 1$, we will need to expand the box by setting `box_width = 10.0` instead of 1 since if two particles end up starting with $r_{ij} < 1$ (see figure 2.2), the system would 'explode'.

Because of this we want to re-write the `initialize_particles`-function from last week to initialize the particles in a nice grid, instead of random positions. We have completed the following code for your convenience.

```python
def initialize_particles(n_particles, box_width):
    """ Initialize particles in a grid
    """

    # Get the smallest grid that all particles will fit in.
    sqrt_npart = int(np.ceil(np.sqrt(n_particles)))

    X = []
    Y = []

    # Arrange sqrt_npart**2 particles in a grid
    for j in range(sqrt_npart):
        X += [i for i in range(sqrt_npart)]
        Y += [j for i in range(sqrt_npart)]

    # Remove excess particles created
    X = X[:n_particles]
    Y = Y[:n_particles]

    # Rescale particle positions to fit in the box.
    for i in range(n_particles):
        X[i[ = (X[i] - 0.5 * sqrt_npart) * 1.0/sqrt_npart * box_width * 1.8
        Y[i] = (Y[i]) * 1.0/sqrt_npart * box_width * 0.8

    # Initialize particle velocities
    Vx = [2 * (random.random() - 0.5) for i in range(n_particles)]
    Vy = [2 * (random.random() - 0.5) for i in range(n_particles)]

    # Inititalize particle forces
    Fx, Fy, energy = lennard_jones(X, Y)

    return X, Y, Vx, Vy, Fx, Fy
```

8. Include the above `initialize_particles` function in your code, set `box_width` to 10.0 and initialize the positions, velocities and forces. *Note:* You don't need to understand everything in detail in the initializing code.

Now it's time to implement the Velo-Verlet solver.

9. Create a new function and name it `velo_verlet`. Have the function take positions, velocities, forces, number of particles and dt as arguments.
10. The new function should return the positions, velocities, forces and the energy of the system.
11. Implement the three steps of the Velo-Verlet integration algorithm. Which means, implement the following code in your `velo_verlet`-function.

```python
# Step 1: Update the positions
for i in range(n_particles):
    pos_x[i] = pos_x[i] + dt * vel_x[i] + 0.5 * dt * dt * force_x[i]

# Remmeber the previous forces
force_x_old = copy.copy(force_x)

# Step 2: Calculate the new forces and energy
force_x, force_y, energy = lennard_jones(pos_x, pos_y)

# Step 3: Calculate the new velocities.
for i in range(n_particles):
    vel_x[i] = vel_x[i] + 0.5 * dt * (force_x_old[i] + force_x[i])
```

12. Correct the particles if they collide with the wall (just like week 1 and 2).

13. After the initialization of the particles, insert the loop over `range(n_steps)`, this time with the `velo_verlet` function, instead of `simulate_step`. This should look something like;

```
for n in range(n_steps):
    X, Y, Vx, Vy, Fx, Fy, energy = velo_verlet(X, Y, Vx, Vy, Fx, Fy, dt)
```

14. Your code should now run a simulation of a Lennard-Jones 2D gas. Save a video of the particle movements using the following constants, and convince yourself that your code is working properly *Note:* Use `video.save('simulation',box_width)` to save a video with a box width differing from 1.

```
box_width = 10.0
n_particles = 42
n_steps = 5000
dt = 0.001
```

Let's try and see how your simulation works.

15. Make a list called `energy_list` and use it to store the energy during the simulation. Append the energy to the list every 5'th step.

16. Plot `energy_list` and see if the potential energy is roughly conserved over time. If your time-step is too large or too small, this might not be the case. A healthy simulation looks something like Figure 2.2

17. Try and run the simulation with a small and a large dt value. Can you make the simulation break down this way? And how does dt influence the number of steps it takes to equilibrate the potential energy?

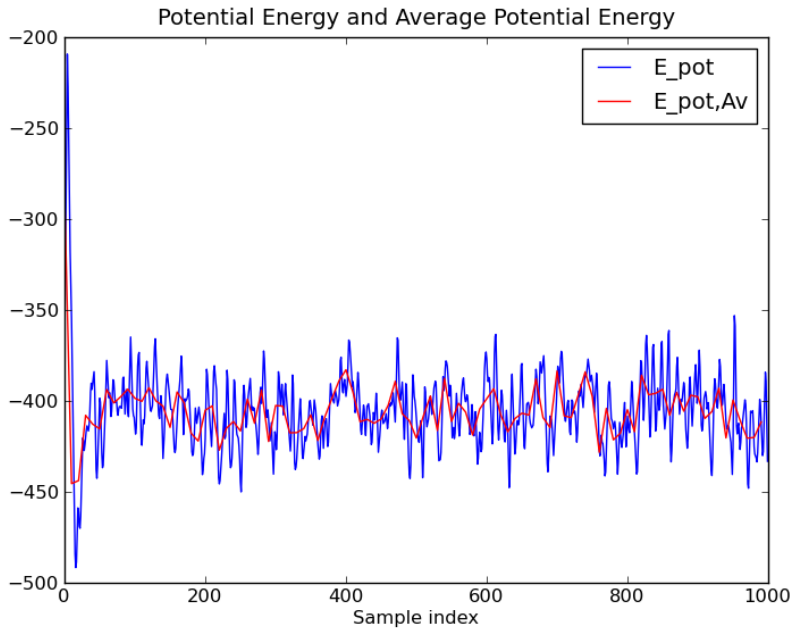

Figure 2.2: Potential energy during a "healthy" MD simulation in a Lennard-Jones potential.