Exercise 3

Molecular Statistics, Week 3

2014

1 Introduction and Theory

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

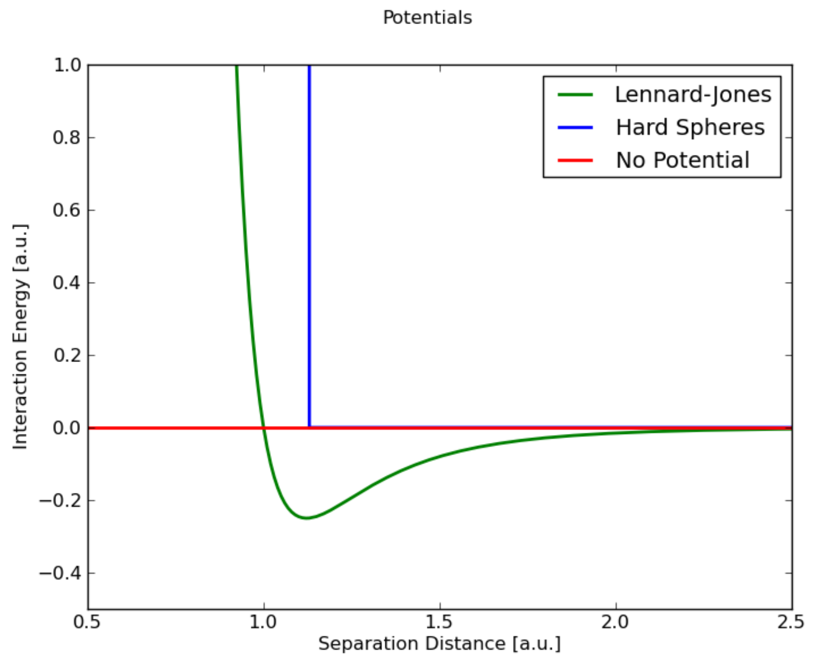

Figure 1.1: The three different potentials we've encountered so far.

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:

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

You should now be able to see how the summation $\sum_{i>j}$ requires a double loop over all particles in your code. These usually look something like:

```python
energy = 0.0
for i in range(n_particles):
    for j in range(n_particles):
        if i > j:
            energy = energy + ...
```

1.2 Inter-particle forces in the Lennard-Jones potential

In the past two exercises, the total momentum of the particles was conserved. In the Lennard-Jones potential, however, 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 **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, similar.

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 step. The Velo-Verlet equations for integrating the positions and velocities are (in the x-direction for one particle):

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

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

In the program you will be implementing today, we will set the mass of the particles to 1, so 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. 1.7
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 as input for the next Velo-Verlet integration step.

2 Exercises

Before you begin, inspect your code from last week. It should more or less contain the following functionality:

```python
import random
import matplotlib.pyplot as plt
import numpy as np
import video # 2D video creator

def distance(xi,yi,xj,yj):
    """ Calculate the distance between particle i and particle j """
    dx = xj-xi
    dy = yj-yi
    return np.sqrt(dx**2 + dy**2)


def initialize_particles(n_particles, box_width):
    """ initialize particles, positions and velocities for n_particles """
    return x_positions, y_positions, x_velocities, y_velocities


def take_step(x_positions, y_positions, x_velocities, y_velocities, dt):
    """ Simulate particle movement from their positions and velocities in a
    single time-step dt """
    return x_positions, y_positions, x_velocities, y_velocities


# Define simulation
box_with    = 10
n_particles = 20
n_steps     = 1000
dt          = 0.001

# Initialize particles
X, Y, Vx, Vy = initialize_particles(n_particles)

# Take steps
for n in range(n_steps):
    X, Y, Vx, Vy = take_step(X, Y, Vx, Vy, dt)

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

video.save('week3')
```

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

1. Save the solution from last week in a new file which you will be using today. *Note:* Remember to call it something new, e.g. `exercise_3.py`.

2. Write a function called `lennard_jones` that takes the lists of x- and y-coordinates as arguments and returns the Lennard-Jones energy. Use the code below as inspiration.

```python
def lennard_jones(x_positions, y_positions, n_particles):
    """ Calculates the energy of the LJ potential """
    energy = 0.0
    for i in range(n_particles):
        for j in range(n_particles):
            if (i > j):
                energy = energy + ...

    return energy
```

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:

```python
n_particles = 2
X_test = [0.0, 0.0]
Y_test = [0.0, 1.4]
```

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

Next, you will extend the Lennard-Jones 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 `x_force` and `y_force`, just like the velocities.

4. Extend your Lennard-Jones function to *also* return the forces. Use the code below for inspiration.

```python
def lennard_jones(x_positions, y_positions, n_particles):
    energy = 0.0

    x_force = [0.0 for i in range(n_particles)]
    y_force = [0.0 for i in range(n_particles)]

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

                x_force[i] = x_force[i] + ...
                y_force[i] = y_force[i] + ...

                x_force[j] = x_force[j] + ...
                y_force[j] = y_force[j] + ...

    return x_force, y_force, energy
```

5. Use the `X_test` and `Y_test` lists from the previous question and calculate the forces and energies. If your code returns `x_force[0]` = 0.0 and `y_force[0]` = 1.6720, your code might be working as it should.

Next is to implement the Velocity Verlet solver (Velo-Verlet). 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. It's not a good idea, however, to use random X-coordinates since your initial gradient might be unphysically large.

Because of this we want to re-write the `initialize_particles`-function from last week to initialize particles in a grid, instead of random positions. *Note:* The initialize_particles function also initializes the forces.

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

    sqrt_npart = int(math.ceil(math.sqrt(n_particles)))

    X = []
    Y = []

    for j in range(sqrt_npart):
        X += [i for i in range(sqrt_npart)]
        Y += [j for i in range(sqrt_npart)]

    while len(X) > n_particles:
        X.pop()

    while len(Y) > n_particles:
        Y.pop()

    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)]

    Fx, Fy, energy = lennard_jones(X, Y)

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

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

6. Copy your simulate_step function to a new function called velo_verlet.
7. The Velo-Verlet algorithm takes the forces as input in addition to the positions, velocities and time-step (dt). Edit the velo_verlet function to also take x_forces and y_forces as arguments.
8. Last time our code checked for particle collisions and used this information to update the velocities and positions. This code does not belong in the velo_verlet function, so remove this garbage.

9. Implement the three steps of the Velo-Verlet integration algorithm. The following code might be useful:

```
# Step 1: Update the positions and
#         remember the old forces.
for i in range(n_particles):
    X[i] = X[i] + dt * Vx[i] + 0.5 * dt * dt * Fx[i]

Fx_old = copy.copy(Fx)

# Step 2: Calculate the new forces and energy
Fx, Fy, energy = lennard_jones(X, Y)

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

10. Because we need the forces calculated in step two of the Velo-Verlet algorithm, what should your velo_verlet function return? Hint: the answer is the forces, so add these to the return statement.

11. We want to remember the energy of each step as well. Add the energy to the list of returned values.

12. In your code you have something like:

```
for i in range(n_steps):
    X, Y, Vx, Vy = simulate_step(X, Y, Vx, Vy, dt)
```

    Replace the call to the take_step function with a call to your velo_verlet function (something like this):

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

13. Your code should now run a simulation of a Lennard-Jones 2D gas. Save a video of the particle movement and convince yourself that your code is working properly.

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 5th 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?

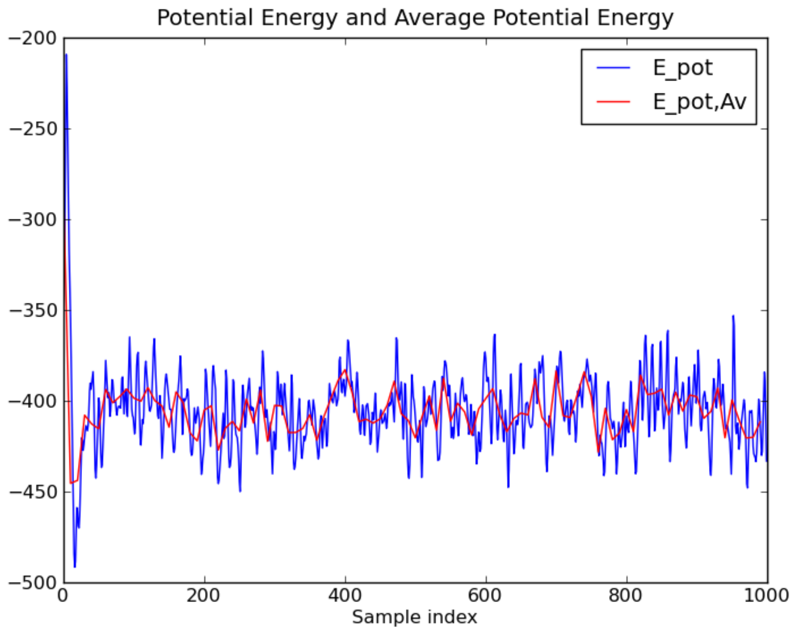

Figure 2.2: Potential energy during a "healthy" MD simulation in a Lennard-Jones potential.

If you finish early, do these exercises before you leave:

Multiply the random velocities in the initialize_particles function by a small scaling factor (i.e. a small number < 1). This lowers the average velocities of the particles and corresponds to running the simulation at a lower temperature. Recall the Maxwell-Boltzmann distribution of the mean speed of particles in a gas:

$$\langle v \rangle = \sqrt{\frac{8\ R\ T}{\pi\ M}} \tag{2.9}$$

Here R is the gas constant, T the temperature and M the molar mass of the gas.

Answer the following two questions:

1. Can you find a scaling factor so the particles behave more like a liquid rather than a gas?
2. What property of the Lennard-Jones is very important if you want to simulate aggregation of particles?