# Exercise 2

Molecular Statistic, Week 2

2014

## 1 Introduction

This is the second exercise in the course Molecular Statistics. The exercises in this course are split in two parts. The first part of each exercise is a general introduction to new concepts and programming techniques in the python programming language, which covers and extends what is written in the curriculum. Just like last week.

## 2 Functions

Last week, the computer program you wrote was executed line by line in a top to bottom approach, i.e. the programs way of execution is easily followed by just reading the source file as you would read a book. It turns out, however, that when the programs become large enough, it is no longer a clever way to organize your program since you'll eventually loose track of what is going on. What one usually does is to break up the program into logical parts, such as "Initialize particles", "perturb the particles" and "calculate properties". Today's exercise is an exercise in restructuring the code from last week into logical parts which we can easily distinguish. Before we get to the exercise, however, we shall have a look at functions, which are essential in this process.

As in mathematics, functions in any programming language can be seen as a means to get a predefined output based on some input. Also, just as in mathematics, functions can have several variables, but in programming languages such as python, you can essentially parse anything into these functions, i.e. lists, floats, text and even your own data types. We'll start with some simple functions that arise in mathematics and continually extend upon them until you are ready to start with the exercise.

They function $f(x) = x^2$ is a good example of how you can turn mathematics into code in python without much hassle.

```
1 def f(x):
2     y = x * x
3     return y
```

1. Insert the above code into your text file, save and execute. What happens? Can you explain why? *Hint:* Nothing happens right now. Why?

In programming languages you provide the *definition* of a function, and just like on your calculator you need to *call* those functions before they are actually executed.

2. Below the code you wrote above, insert the following lines of code and explain what happens when you run the script

```
1 print f(-1.0)
2 print f(0.0)
3 print f(1.0)
4 print f(2.0)
```

3. Instead of printing directly as you did in (2), assign the values of the functions to variables and print them afterwards, i.e. `a = f(6.0)`.

The reality is that our custom functions actually works like the built in functions in python (remember the `len(arg)` function from last week for instance) ,so once you know how to make one, you can essentially create as many you like.

To test functions we would need a list of $x$-values as input. We could use the `range()` function from last week, but that would limit us to integers. Instead we use a very useful tool call NumPy which is imported as all other tools;

```
1 import numpy
```

We can then use the `arange(start, end, step)` function. This will produce a list just like the `range()` function, but this time the `start`, `end`, and `step` can be floats instead of only integers.

```
1 x_list = numpy.arange(0, 10, 0.1)
```

4. Define the following function using `numpy.exp()`.

$$h(x) = \frac{1}{x} + e^{5x} \tag{2.1}$$

5. Define a list of $x$-values, ranging from -1 to 1 with a step size of 0.1, using the `arange()` function.
6. Plot the result using pylab.

Next let us plot something more complex. Use `numpy.sin()` and `numpy.cos()`.

7. Define the following two functions

$$q(t) = 13\sin(t)^3 \qquad \text{and} \qquad p(t) = 13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t) \tag{2.2}$$

8. Define a list of $t$-values from $-2\pi$ to $2\pi$. *Hint:* Use `numpy.pi` to get the values of $\pi$.
9. For each value of $t$, plot $p(t)$ vs. $q(t)$, with $p(t)$ as the $y$-value and $q(t)$ as the $x$-value.

# 3 Interacting particles - The Hard Sphere Model

Goal of today

- Restructure the program from last week using functions.
- Allow the particles to interact by measuring the distance from all the other particles and if two particles are close make a reflection.
- Count particles and estimate when an equilibrium condition has been established.

As a starting point for your code, you should copy over your solution to last weeks exercise. If you didn't finish the exercises last week, download the file `week1_solution.py` from Absalon. The following code should be seen as inspiration for how the code should be structured after this exercise is done.

```
import random
import pylab
import video # 2d Video

def distance(xi, yi, xj, yj):
    """ Calculate the distance between particle i and particle j
    """
    return d


def initialize_particles(n_particles):
    """ Initialze particles, positions and velocities
    """
    return pos_x, pos_y, vel_x, vel_y


def simulate_step(pos_x, pos_y, vel_x, vel_y, dt):
    """ Here we simulate a step dt for all particles.
    """
    return pos_x, pos_y, vel_x, vel_y


def plot_particles(X, Y, filename):
    """Insert code to save a picture
    of the particle positions
    """


# Run the simulation
n_particles = 40
n_steps = 10000
dt = 0.001
X, Y, Vx, Vy = initialize_particles(n_particles)

plot_particles(X, Y, 'coordinates_start.png')

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

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

plot_particles(X, Y, 'coordinates_end.png')
video.save('exercise_2')
```

Important Note. Notice how the bottom part of the script is the code that actually calls the appropriate functions to make the simulation. This makes reading the overall structure of the program easier because you can now see how the program is executed and then inspect the appropriate functions. Also, it is not important in what order your functions are defined. What matters is the order in which you call them. This is.different than i.e. variables, which have to be defined and have a value assigned before you use them.

1. Inspect the above code and restructure your own code from last week to have the same structure and functions.

Remember that this is only done because it is easier to read and in the end, it is up to yourself to decide how you want to structure your programs.

2. Finish the function `distance()` and make it return the distance `d` between particle $i$ and $j$.
3. Extend your program to create a new variable $r_{min}$ and set its value to 0.03. This is the distance that the two particles $i$ and $j$ have to be within in order to make a reflection.

We need to get the interaction between all the possible particle pairs. By looping over each particle twice, we can get all the possible interactions.

```
for i in range(n_particles):
    for j in range(n_particles):
        print "Interacting particles", i, "and", j
```

Since the particles are interchangable, this will count each interaction twice. We also want to skip the interaction if particle $i$ is the same as $j$. All this can be solved by a simple `if`-statement.

```
if i > j:
    print "Interacting particles", i, "and", j
```

4. Before implementing the solution, check that the above loop and if-statement works as expected, and checks all unique interactions only once. *Hint: Use a print statement.*
5. Modify your code to loop over all particle interactions. Print the distance between the two particles interacting.

To let the particles interact, we will use a simple model. The particles are considered as being small disks with a diameter of $r_{min}$ and they have perfect elastic scattering when particle $i$ comes into contact with particle $j$. (Think a game of billiards)
When two particles collide elastically they will exchange velocities which is done by simply setting

$$\mathbf{v}_i = \mathbf{v}_j \qquad \text{and} \qquad \mathbf{v}_j = \mathbf{v}_i \tag{3.3}$$

6. Implement in your code this exchange of velocities for particles upon collision. Use the following if-statement as inspiration. Hint: Save the velocity of particle $i$ in a temporary variable.

```
if d < r_min:
    # Collision
```

As a final part of todays exercise, we can extract some useful information from the simulation. Since the system we are investigating is not the most physical correct one and particle positions do tend to be a bit boring by themselves, we shall instead sample a property, which shows that the system is indeed random.

7. For each 5 steps in your simulation, calculate the number of particles that are present in the interval $x \in [0, 1]$. Append that number to a list named `partdist`. To get proper results, you need to change `n_steps` to 20000 and `n_particles` to 40.

   *Note:* Remember to disable saving of the video when running with large steps!

We want a histogram of how often $N$ particles occurred in the above interval. So along the $x$-axis should give you the number of particles we have encountered in the interval, and along the $y$-axis how oftern that number has occurred.

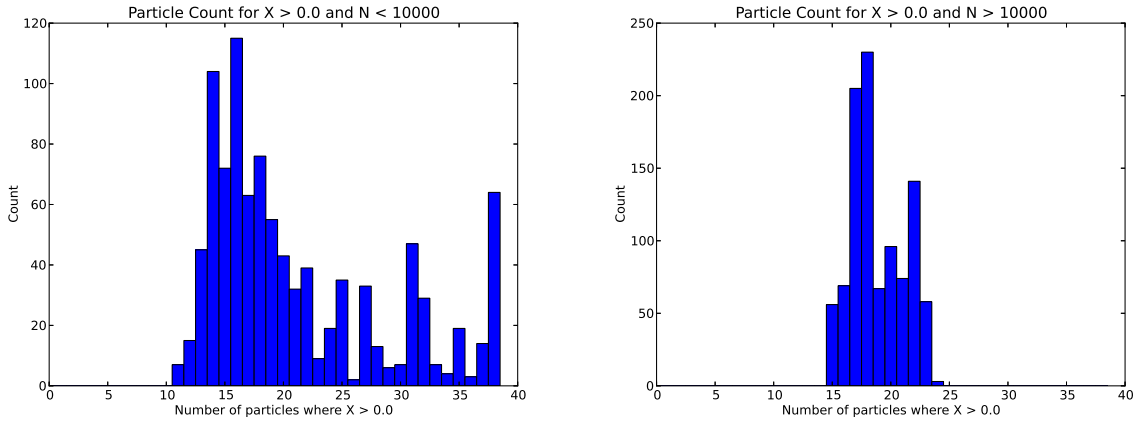
To plot the information from the list we can use the pylab histogram as follows

```
pylab.hist(partdist, bins=range(0, n_particles), align='left')
pylab.savefig('dist_histogram.png')
```

which is inserted after the main loop over `n_steps`.

8. Plot the histogram and comment on the results, i.e. how is the distribution of particles in interval?

9. Make another list named `partdisteq`. Here you should store the number of particles in the same interval as above, but only **after** 10000 calibration steps have passed. Plot it on a new figure and comment on the result.




**(a)** Before equilibrium steps

**(b)** After equilibrium steps

**Figure 3.1:** Count of the number of particles placed to the right in a container in the before and after equilibrium steps.